**RICARDO STEGH CAMATI**

# Novos Algoritmos para Alocação de Máquinas Virtuais e um Novo Método de Avaliação de Desempenho

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná, como requisito para obtenção do título de Mestre em Informática.

Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Alcides Calsavara

**Curitiba**

**2013**

# DEDICATÓRIA

Dedico esta dissertação a minha família, e de maneira especial a minha avó materna Maria Stegh (*in memoriam*) que faleceu no início do primeiro ano deste trabalho. Ela foi uma companheira inseparável na minha infância, me criou como se fosse seu próprio filho, cuidou de mim e de minha irmã quando meus pais estavam trabalhando, nos alimentou e educou sem nunca pedir nada em troca. Sua alegria era servir as pessoas para que se sentissem bem, adorava cozinhar e sua comida que era deliciosa.

# AGRADECIMENTOS

Agradeço a Deus, a força superior responsável pela criação da energia, matéria e por dedução, da vida. Acredito que toda nova descoberta nada mais é do que uma pequena equação da grandiosidade e complexidade incalculável por ele arquitetada.

Agradeço a segura orientação do Prof. Dr. Alcides Calsavara, cujo profissionalismo e competência foram fundamentais para o crescimento e a conclusão da minha dissertação. Suas aulas sobre Sistemas Distribuídos juntamente com as leituras de artigos recomendadas e seu excelente desempenho como professor me ajudaram a compreender muitos conceitos de vital importância. O tempo todo ele me incentivou a continuar e cedeu seu precioso tempo para me ajudar, foram dezenas de reuniões e muitas horas que passamos juntos trabalhando para possibilitar a finalização dessa dissertação, me sinto honrado e privilegiado pela oportunidade de trabalhar ao seu lado .

Agradeço, também, ao Prof. Dr. Carlos Alberto Maziero, que foi meu orientador durante o primeiro ano do mestrado, suas excelentes aulas e apostilas me ajudaram a entender o funcionamento de Sistemas Operacionais e Máquinas Virtuais, revelando um novo mundo a ser explorado. Expresso minha gratidão também aos professores Dr. Luiz Lima Junior e Dr. Altair Santin, do grupo de Sistemas Distribuídos que sempre estiveram presentes em nossas reuniões do grupo, promovendo a troca de conhecimento entre professores e alunos. Meus agradecimentos e respeito a todos os professores do PPGIA, podem ter certeza que todo conhecimento absorvido em suas aulas foram de grande valia para este trabalho.

Agradeço de forma muito especial a minha mãe, Prof. Dr. Anna Stegh Camati, e ao meu pai, Dr. Rocha Vitor Camati, seus incentivos e amor incondicionais me deram forças para terminar esta jornada, tenho muita sorte de ter pais maravilhosos como eles. À minha esposa Denalzira Mariano, expresso minha gratidão e reconhecimento, por ser uma companheira amorosa, e pelo seu apoio constante. À meu avô materno (*in memoriam*) minha homenagem por sua força de vontade, não se deixando abater pelos revezes da vida.

Não poderia deixar de agradecer também ao mestrando Fábio Pandolfo, que me ajudou a montar um pequeno cluster de máquinas virtuais. Apesar de ter usado um simulador nesta dissertação, o cluster que montamos serviu para avaliar as dificuldades do mundo real.

# EPÍGRAFE

*"Corrija um sábio, e o fará mais sábio.*

*Corrija um tolo, e o fará seu inimigo."*

**RESUMO**

A tecnologia de máquinas virtuais tem sido fortemente empregada nos últimos anos, visando, principalmente, a economia de recursos computacionais e o isolamento entre usuários e aplicações, para fins de segurança. O principal contexto de emprego de máquinas virtuais é na arquitetura de sistemas operacionais, inclusive como base para a construção de plataformas para a computação nuvem. A alocação eficiente de máquinas virtuais em um cluster de máquinas físicas é essencial para a otimização do uso de recursos computacionais e para reduzir a probabilidade de realocações. Entretanto, tipicamente, os algoritmos de alocação de máquinas virtuais empregam abordagens simples, tais como *First Fit* e suas variantes, restringindo-se a apenas uma dimensão de recurso computacional. Além disso, os métodos de avaliação empregados não contemplam integralmente a heterogeneidade entre as máquinas físicas de um cluster com relação à capacidade dos diferentes recursos computacionais e nem permitem analisar o impacto da variação de densidade de alocação. Esta dissertação propõe novos algoritmos de alocação de máquinas virtuais, os quais consideram múltiplas dimensões de recursos computacionais, e define um novo método de avaliação que contempla integralmente a heterogeneidade de recursos computacionais em múltiplas dimensões e permite analisar a variação de densidade de alocação. Os novos algoritmos foram comparados com outros tradicionais através de simulação, empregando o novo método definido. Os experimentos realizados permitiram observar que, de forma geral, a heterogeneidade de recursos entre máquinas físicas prejudica o desempenho dos algoritmos, enquanto que a heterogeneidade de recursos entre máquinas virtuais o beneficia. Também foi possível observar que o aumento na densidade de alocação até certo ponto beneficia o desempenho dos algoritmos. Finalmente, um dos algoritmos teve desempenho até 13% superior.



**Palavras-chave:** Computação em Nuvem, Alocação de Máquinas Virtuais, Método de Avaliação de Desempenho.

## ABSTRACT

Virtual machine technology has been more employed recently, mainly aiming at the economy of computational resources and the separation of users and applications for the sake of security. The main context for the use of virtual machines is the architecture of operating systems, inclusively as a basis for the construction of platforms for cloud computing. The effective allocation of virtual machines in a cluster of physical machines is essential for the optimization of the use of computational resources to reduce the probability of reallocations. However, typically, the allocation algorithms of virtual machines employ simple approaches, such as *First Fit* and its variants, restricted to only one dimension of computational resources. Furthermore, the evaluation methods do not contemplate integrally either the heterogeneity of physical machines of a cluster with respect to the capacity of the different computational resources, nor do they permit analyzing the variation impact of the density of allocation. This dissertation proposes new allocation algorithms for virtual machines, and defines a new method of evaluation that contemplates integrally the hetereogeneity of computational resources in multiple dimensions and permits analyzing the variation of density allocation. The algorithms have been compared to other traditional ones by means of simulation, employing the newly defined method. The experiments realized permitted observing that, generally, the heterogeneity of resources among physical machines impairs the performance of algorithms, while the heterogeneity of resources among virtual machines benefits it. It was also possible to observe that the increase of allocation density up to a point benefits the performance of all algorithms. Finally, one of the algorithms reached a superior performance of 13%.

**Keywords:** Cloud Computing, Virtual Machine Allocation, Method of Performance Evaluation.

## SUMÁRIO


**1 INTRODUÇÃO ..... 14**
1.1 MOTIVAÇÃO ..... 14
1.2 OBJETIVOS ..... 16
1.3 ESTRUTURA DA DISSERTAÇÃO ..... 18
**2 FUNDAMENTOS DE MÁQUINAS VIRTUAIS E COMPUTAÇÃO EM NUVEM ..... 19**
2.1 HISTÓRICO DE MÁQUINAS VIRTUAIS ..... 19
2.2 ARQUITETURA DE MÁQUINAS VIRTUAIS ..... 24
2.3 PROPRIEDADES DE MÁQUINAS VIRTUAIS ..... 28
2.4 MIGRAÇÕES DE MÁQUINAS VIRTUAIS ..... 31
2.5 COMPUTAÇÃO EM NUVEM ..... 33
2.6 DISCUSSÃO SOBRE COMPUTAÇÃO EM NUVEM ..... 40
**3 ALOCAÇÃO DE MÁQUINAS VIRTUAIS EM *DATA CENTERS* ..... 42**
3.1 DETERMINANTES DE VARIABILIDADE DO PROBLEMA ..... 42
*3.1.1 O conjunto de hosts ..... 43*
*3.1.2 Modo de chegada de máquinas virtuais ..... 45*
*3.1.3 Dimensões das máquinas ..... 46*
3.2 A ESCOLHA DO PROBLEMA ..... 47
3.3 TÉCNICAS DE ALOCAÇÃO DE MÁQUINAS VIRTUAIS ..... 48
*3.3.1 Classificação de Anjana Shankar ..... 48*
*3.3.2 Técnica baseada no Problema da Mochila ..... 50*
3.4 HEURÍSTICAS TRADICIONAIS PARA ALOCAÇÃO DE MÁQUINAS VIRTUAIS ..... 53
*3.4.1 First Fit ..... 53*
*3.4.2 First Fit Decreasing ..... 53*
*3.4.3 Volume ..... 54*
*3.4.4 Dot Product ..... 54*
*3.5* HEURÍSTICAS PROPOSTAS PARA ALOCAÇÃO DE MÁQUINAS VIRTUAIS ..... 55
*3.5.1 Best Dimension ..... 56*
*3.5.2 Osmosis ..... 57*
3.6 TRABALHOS RELACIONADOS ..... 58
3.7 CONCLUSÃO ..... 61
**4 MÉTODO DE AVALIÇÃO DE DESEMPENHO ..... 62**
4.1 CONSIDERAÇÕES SOBRE A HETEROGENEIDADE ..... 62
4.2 CONSIDERAÇÕES SOBRE A DENSIDADE DE ALOCAÇÃO ..... 66
4.3 TAXA DE ALOCAÇÃO ..... 67
4.3.1 O CENÁRIO IDEAL ..... 68
4.3.2 O CENÁRIO REAL ..... 68
4.4 CONCLUSÃO ..... 69
**5 EXPERIMENTOS ..... 70**
5.1 DELIMITAÇÃO DO ESCOPO DOS EXPERIMENTOS ..... 70
5.2 RESULTADOS OBTIDOS ..... 73
*5.2.1 Cenário I ..... 73*
*5.2.2 Cenário II ..... 75*
*5.2.3 Cenário III ..... 76*
*5.2.4 Cenário IV ..... 77*
*5.2.5 Cenário V ..... 78*
*5.2.6 Cenário VI ..... 80*
*5.2.7 Cenário VII ..... 81*
5.3 O SIMULADOR *PYCLOUD* ..... 83

5.4 DISCUSSÃO ..... 88
**6 CONCLUSÃO ..... 90**
6.1 CONTRIBUIÇÕES ..... 90
6.2 TRABALHOS FUTUROS ..... 91
**REFERÊNCIAS BIBLIOGRÁFICAS ..... 93**
**ANEXO I ARQUITETURA DE *HARDWARE* PARA MÁQUINAS VIRTUAIS ..... 99**
**ANEXO II CLASSIFICAÇÃO DE MÁQUINAS VIRTUAIS ..... 114**
**ANEXO III TÉCNICAS DE MINIMIZAÇÃO DO *OVERHEAD* PARA MÁQUINAS VIRTUAIS ..... 126**
**ANEXO IV O AMBIENTE DE TESTES *OPEN CIRRUS* ..... 128**

**LISTA DE FIGURAS**

Figura 1. O *System/360* da IBM [Google Imagens]. ........................................21
Figura 2. Acoplamento entre interfaces [Laureano e Maziero, 2008]. ..........................24
Figura 3. Pilha de *software* num sistema [Laureano, 2006]. .....................................25
Figura 4. Interfaces entre camadas num sistema[Laureno e Maziero, 2008]. ..................26
Figura 5. Transparência provida pela camada de virtualização [Laureano e Maziero, 2008]..27
Figura 6. Tipos de Migração [Hines, 2009]. ...........................................................32
Figura 7. Camadas do paradigma da *Cloud Computing* [Zhang et al., 2010]. ..................37
Figura 8. Variabilidade do problema de alocação de VMs. ........................................43
Figura 9. Exemplo de alocação de VM. ..................................................................48
Figura 10. Classificação de Anjana Shankar [Shankar, 2010]. ....................................49
Figura 11. Exemplo de amplitudes de dimensões de uma máquina. ..............................64
Figura 12. Gráfico para a taxa de alocação obtido no Cenário I. .................................74
Figura 13 Gráfico para a taxa de alocação obtido no Cenário II. .................................75
Figura 14. Gráfico para a taxa de alocação obtido no Cenário III. ...............................76
Figura 15. Gráfico para a taxa de alocação obtido no Cenário IV. ...............................77
Figura 16. Gráfico para a taxa de alocação obtido no Cenário V. ................................79
Figura 17. Gráfico para a taxa de alocação obtido no Cenário VI. ...............................80
Figura 18. Taxa de alocação no Cenário VII para o algoritmo *First Fit*. .......................81
Figura 19. Taxa de alocação no Cenário VII para o algoritmo *Volume*. ........................81
Figura 20. Taxa de alocação no cenário VII para o algoritmo *Osmosis*. ........................82
Figura 21. Taxa de alocação no Cenário VII para o algoritmo *Best Dimension*. ..............82
Figura 22. Taxa de alocação no Cenário VII para o algoritmo *Dot Product*. ..................83
Figura 23. A arquitetura do *CloudSim* [Calheiros et al., 2010]. ...................................84
Figura 24. A arquitetura do *PyCloud*. .....................................................................86
Figura 25. Arquitetura Von Newmann e uma interrupção. [Maziero, 2011] ..................100
Figura 26. Mecanismo de interrupção [Toscani, 2011]. ............................................103
Figura 27. Funcionamento da MMU e do registrador de relocação de endereços [Maziero, 2011]. ............104
Figura 28. Funcionamento da MMU com segmentação de memória [Maziero, 2009]. .........106
Figura 29. MMU com paginação de memória [Maziero, 2009]. ...................................107
Figura 30. Virtualização utilizando os níveis de privilégios da arquitetura X86 [Maziero, 2009]. ............111

Figura 31. Instruções de um processador Intel com suporte a virtualização [Torres e Lima, 2005]. ....... 112

Figura 32. Modo *root* e não *root*, virtualização em hardware [Maziero, 2009] ....... 113

Figura 33. Classificação de máquinas virtuais de acordo com a aplicação convidada. [Laureano e Maziero, 2008]. ....... 114

Figura 34. Máquinas virtuais classificadas de acordo com a interface respeitam e que exportam [Laureano e Maziero, 2008]. ....... 115

Figura 35. Uso da abstração facilitando a virtualização [Smith and Nair, 2005]. ....... 115

Figura 36. Emulação de *hardware* [Jones, 2006]. ....... 116

Figura 37. O processador e emulador Transmeta Crusoe e suas camadas [Harrison, 2005].. 118

Figura 38. Comparação entre componentes do x86 e do Transmeta Crusoe [Harrisson, 2005]. ....... 119

Figura 39. A tecnologia do VLIW (molécula do Transmeta Crusoe). ....... 120

Figura 40. Virtualização no nível de bibliotecas : Wine [Laureano e Maziero, 2008]. ....... 121

Figura 41. Virtualização total com hipervisor nativo [Jones, 2006]. ....... 122

Figura 42. Diferenças entre virtualização total e para-virtualização. ....... 123

Figura 43. Melhorias de desempenho em hipervisores nativos e convidadados. [Laureano e Maziero, 2008] ....... 127

Figura 44. Federações de *datacenters* no globo terrestre do projeto *Open Cirrus* 2009 [Campbell et al., 2010]. ....... 128

Figura 45. Os serviços de alto nível do *Open Cirrus* [Campbell et al., 2010]. ....... 130

## LISTA DE TABELAS

Tabela 1. Tuplas de amplitudes selecionadas para $\delta$ e $\pi$ ....................................................70
Tabela 2. Variações de $\rho$ ....................................................................................................71
Tabela 3. Valores típicos para amplitudes e densidade ..........................................................71
Tabela 4. Valores utilizados para cada densidade experimentada...........................................73

## LISTA DE SÍMBOLOS

| | |
|---|---|
| H | Um conjunto de *hosts*. |
| V | Um conjunto de VMs. |
| V' | Um conjunto de VMs alocadas. |
| \|V\| | Número total de VMs. |
| \|H\| | Número total de *hosts*. |
| \|A\| | Número de VMs alocadas. |
| P | A dimensão do processador. |
| R | A dimensão da memória RAM. |
| D | A dimensão do disco. |
| B | A dimensão da banda. |
| M | Um conjunto de máquinas (*hosts* ou VMs) |
| $V_{min}$ | Valor mínimo de uma dimensão para um conjunto de máquinas. |
| $V_{max}$ | Valor máximo de uma dimensão para um conjunto de máquinas. |
| $V_{médio}$ | Valor médio de uma dimensão para um conjunto de máquinas. |
| $\alpha$ | Tupla de valores médios das dimensões para um conjunto de VMs. |
| $\beta$ | Tupla de valores médios das dimensões para um conjunto de *hosts*. |
| $\Delta_{M,x}$ | Amplitude de uma dimensão x para um conjunto de máquinas M. |
| $\theta_M$ | Tupla de amplitudes para um conjunto de máquinas M. |
| $\pi$ | Tupla de amplitudes de um conjunto de VMs. |

| | |
|---|---|
| $\delta$ | Tupla de amplitudes de um conjunto de *hosts.* |
| $\rho$ | Densidade de alocação para um conjunto de máquinas. |
| $\tau$ | Taxa de alocação. |
| T | Taxa de alocação média. |
| k | Tamanho de cada dimensão em um conjunto de hosts. |
| t | Tamanho de cada dimensão em um conjunto de VMs. |
| C | Um conjunto de cenários. |
| S | Um conjunto de pontos de observação. |
| E | Um conjunto de experimentos. |
| $T_j$ | O instante de chegada da VM j. |
| $x_{i,j}$ | A VM j esta alocada no *host* i. |
| $w_{j,k}$ | O valor da dimensão k para a VM j. |
| $c_{i,k}$ | A capacidade que o *host* i tem para dimensão k. |
| $L_{i,k}(T_j)$ | A parte livre da capacidade da dimensão k no *host* no instante da chegada da VM j. |

# 1 Introdução

## 1.1 Motivação

Desde que o modelo cliente-servidor foi inventado, é muito difícil prever com precisão o tanto que uma aplicação irá consumir de recursos ao longo de seu tempo de vida. Para exemplificar o desperdício financeiro, podemos analisar o investimento em infraestrutura em três casos possíveis:

a) A aplicação pode cair em desuso, não sendo mais acessada pelos clientes de forma intensa. Com picos de acessos menores, o *hardware* ficará ocioso e não terá uma demanda que justifique o investimento feito. Uma solução possível para evitar o desperdício poderia ser migrar outras aplicações para esta máquina ociosa, mas isso poderia gerar novos problemas de gargalos e segurança.

b) A aplicação se comporta como o esperado na análise, saindo tudo como foi planejado. O *hardware* foi perfeitamente estipulado para o pico de carga da aplicação. Este é o melhor caso, mas ainda existe o desperdício de dinheiro no investimento; a máquina foi estipulada para o pico de uso e o *hardware* ficará ocioso no tempo em que a carga estiver habitual ou baixa.

c) A aplicação começa a crescer de forma não controlada. Neste caso, o analista erra em prever o tanto que a aplicação poderia crescer ao longo do tempo e a máquina é subestimada, tendo um desempenho insuficiente para a aplicação. Este é o pior caso possível, pois o investimento em infraestrutura é precário. Como consequência os clientes são penalizados com um serviço de péssima qualidade (com lentidão, travamentos e baixa disponibilidade). Para remediar o problema, o serviço pode ser limitado, ficando parcialmente disponível para alguns clientes até que uma solução definitiva possa ser encontrada (a qual pode demorar semanas ou até mesmo meses). A única saída possível para este caso é uma nova análise que será feita levando em conta o trauma da experiência anterior. Nesta análise os recursos provavelmente vão ser superestimados, induzindo a um novo erro. Enquanto a nova análise é feita, o investimento primário já sofreu desvalorização, os clientes não têm a mesma credibilidade no sistema e começam a procurar concorrentes mais confiáveis

Analisando estes casos típicos de um ambiente não elástico, conclui-se que em todos eles aconteceu o desperdício de dinheiro no investimento. Mesmo no melhor caso citado, o *hardware* ficou ocioso, significando desperdício de recursos. Como os exemplos ilustram na melhor, na pior e num caso intermediário de situações, com as quais todo analista de suporte se depara na hora de quantificar a necessidade do *hardware*, podemos intuitivamente deduzir que a única solução possível para este problema é a elasticidade dos recursos.

A computação em nuvem é de extrema importância para o melhor aproveitamento do investimento em TI em todas as áreas. As empresas estão cada vez mais aderindo à computação em nuvem, não só por opção, mas também por uma questão de sobrevivência. Para as pequenas empresas este paradigma pode ser o fator determinante para alavancar o negócio, pois utilizando a nuvem pública não existe o investimento inicial de infraestrutura, investimento este que pode endividar a empresa e até mesmo provocar sua falência, no caso de não obter o esperado retorno. Além da ausência do custo inicial, na nuvem pública o início de produção acontece de forma instantânea, em poucos segundos o sistema operacional é instanciado, com os recursos necessários para executar a aplicação. Analisando um cronograma tradicional de infraestrutura, a construção de um *data center* particular poderia demorar meses e até mesmo anos, criando uma dependência de fornecedores, fretes e mão de obra que irão deixar a estimativa de tempo para ter um ambiente de produção impreciso, agravando ainda mais o problema. Com o modelo *pay-per-use* acaba o problema da necessidade de superestimar os recursos para atender picos de acessos, pois a nuvem cresce e encolhe de acordo com a carga de cada aplicação. As médias e grandes empresas normalmente possuem *data centers* particulares que podem ser remodelados para permitir a computação em nuvem flexibilizando a própria *intranet* empresarial, transformando-a em uma *private cloud.* O modelo de nuvem híbrida pode ainda ser explorado visando diminuir custos.

A motivação para este trabalho parte do princípio que a computação em nuvem é essencial para que empresas de todos os portes melhorem seus investimentos em TI, acabando com o desperdício e fornecendo sempre um serviço de qualidade para seus clientes, podendo prosperar e minimizar gastos. Algumas análises indicam que os modelos de computação em nuvem atuais possuem limitações. Outro agravante é que, na computação em nuvem, grande parte das aplicações usa o paradigma *web*, no qual uma grande porcentagem de aplicativos no sistema da *cloud* tem um comportamento de consumo de recursos instável [Liu e Wee, 2009].

O mecanismo de alocação de VMs é essencial para o uso eficiente dos recursos de hardware, em um *data* center. Esse mecanismo consiste em decidir em qual *host* deve ser alocada uma máquina virtual solicitada por um cliente, entre todos os *hosts* que comportam a máquina virtual. Caso não haja um *host* adequado, a máquina virtual é rejeitada. A adequação de um *host* como candidato para a alocação de uma máquina virtual deve levar em consideração os vários tipos de recursos exigidos, incluindo processador, memória, disco e banda. Assim, o mecanismo deve encontrar o melhor casamento entre as necessidades da máquina virtual e a disponibilidade em cada *host*, tal que a probabilidade de sucesso na alocação de futuras máquinas virtuais seja maximizada. Um benefício de uma boa alocação de máquinas virtuais é favorecer a elasticidade de recursos, pois os *hosts* tendem a ter seus recursos consumidos de forma balanceada.

Na computação em nuvem, a maior parte das aplicações usa o paradigma *web*, no qual uma grande porcentagem de aplicativos no sistema da *cloud* tem um comportamento de consumo de recursos instável [Liu e Wee, 2009], assim uma VM pode aumentar ou diminuir o uso de recursos computacionais abruptamente. Para endereçar este problema, a solução mais elegante é o monitoramento das VMs e o uso da computação autonômica para aumentar ou diminuir a quantidade de recursos consumidos por ela de forma automática, liberando recursos a outras VMs quando não estiverem em uso ou fazendo o uso de migrações quando o servidor não tiver recursos suficientes disponíveis para satisfazer suas necessidades. Neste trabalho decidimos aprofundar os estudos na alocação de máquinas virtuais e deixar o aspecto de migração de VMs para trabalhos futuros, acreditamos que se a alocação inicial de VMs for feita de forma ideal e balanceada é possível evitar futuras migrações. Apesar do mecanismo de migração ser muito útil e possuir soluções avançadas como o *live migration*, existem limitações devido ao overhead imposto que podem provocar quebras de SLA, fazendo gargalos na nuvem.

## 1.2 Objetivos

Esta dissertação tem por objetivo investigar soluções para o problema de alocação de máquinas virtuais em *datacenters*, seguindo uma abordagem realista. Para tanto, os seguintes objetivos específicos são almejados:

a) Aprofundar os conhecimentos sobre a área de computação em nuvem e a tecnologia de máquinas virtuais.

b) Identificar todas as variáveis que influenciam no problema e formalizá-lo de acordo.

c) Identificar as soluções documentadas na literatura para o problema e escolher algumas para fins de validação.

d) Definir um método de avaliação de desempenho das soluções escolhidas.

e) Realizar o experimentos de avaliação e avaliar os resultados obtidos.

## 1.3 Estrutura da dissertação

Esta dissertação está organizada da seguinte forma: o Capítulo 2 descreve os principais conceitos e faz um histórico sobre máquinas virtuais e computação em nuvem; o Capítulo 3 discute as diferentes abordagens do problema documentado na literatura, define um problema realista de maneira formal e estuda as soluções típicas para o problema, bem como, propõe novas soluções; o Capítulo 4 define um método de avaliação de desempenho para as soluções do problema, considerando variáveis realistas; o Capítulo 5 descreve os experimentos realizados a fim de comparar as diversas soluções; finalmente, o Capítulo 6 apresenta algumas conclusões e discute trabalhos futuros.

# 2 Fundamentos de máquinas virtuais e computação em nuvem

O conceito de computação revolucionou a *Internet*, hoje a maior parte dos serviços da Internet estão na nuvem. Para entender a computação em nuvem é necessário estudar a ferramenta usada para a construção de seu alicerce, as máquinas virtuais. Através da computação em nuvem, é possível que vários usuários possam utilizar uma mesma máquina física, executando suas respectivas aplicações em sistemas operacionais exclusivos.

Este capítulo foi dividido em três partes:

- histórico de máquinas virtuais.
- arquitetura de máquinas virtuais
- computação em nuvem

## 2.1 Histórico de máquinas virtuais

A ideia da virtualização veio da necessidade de ter sistemas operacionais mais eficientes e seguros. Apesar das máquinas virtuais ganharem muita atenção da mídia nos últimos anos, principalmente devido à linguagem de programação Java e a computação em nuvem, elas são produto de um conceito antigo. Para podermos analisar o histórico de máquinas virtuais é importante conhecer como funcionavam os computadores da década de 50. Nesta época, cada programa de computador necessitava de uma máquina real exclusiva a sua disposição e os programas eram lentos devido às limitações tecnológicas da segunda geração de computadores. Nesta época, quando um *software* estava em execução, o usuário era obrigado a ficar fiscalizando a máquina, e se acontecesse um erro, ele deveria efetuar a correção imediatamente, consequentemente, erros de programação eram muito custosos (deixava a CPU em IDLE anulando sua produtividade), uma correção poderia demandar várias horas ou até mesmo dias de trabalho [Creasy, 1981].

Na década de 60, surge uma nova gama de sistemas operacionais em que o uso da CPU foi melhor aproveitada com duas técnicas que se mostraram muito eficientes, a multiprogramação e o *spooling*, estas técnicas, aliadas ao avanço tecnológico dos componentes eletrônicos, marcaram o início da terceira geração de computadores. A multiprogramação era usada quando uma operação de E/S acontecia, fazendo com que a CPU

continuasse trabalhando com alguma outra tarefa ao invés de ficar ociosa e, para isso, ela dividia a memória em partes menores, onde cada parte representava os dados referentes a um dos *jobs* que seriam executados. Para fazer esta divisão da memória, de forma eficiente, foi criado um mecanismo de proteção de dados no *hardware* [Tanenbaum, 2001].

Além da multiprogramação, o *spooling* era outra técnica que tinha como objetivo capacitar um computador a controlar diversos periféricos de forma mais inteligente. Por estes periféricos serem mais lentos que a CPU, eles atrasavam muito o processamento de dados. A técnica do *spooling* fazia o uso de *buffers* e gerava uma interrupção de *hardware* quando um dispositivo terminava uma leitura ou escrita de dados. Um exemplo de melhoria foi possibilitar aos sistemas lerem cartões perfurados, transferindo os dados para os discos magnéticos, sem restringir o funcionamento da *CPU* [Tanenbaum, 2001].

A partir do ano de 1959, a comunidade científica começou a discutir a ideia de um sistema operacional bem diferente dos sistemas *batch* dominantes. Este novo conceito de sistema operacional foi chamado de *time-sharing,* evidenciando o usuário *on-line*. A ideia principal de um sistema *time-sharing* era permitir que programas fossem *debugados* de forma iterativa por um ou mais usuários, e enquanto isso, uma sequência de *batch jobs* executava em *background* [Creasy, 1981].

Nesta epoca, um projeto concreto de sistema operacional *time-sharing* nasceu no MIT, foi chamado de CTSS (*Compatible Time Sharing Operating System*) e liderado pelo professor Fernando Corbató, ficando pronto em novembro de 1961. A escolha da plataforma para este sistema operacional naturalmente foi a IBM, pois nos anos 50 o presidente desta empresa decidiu doar um *mainframe* para pesquisas ao MIT (IBM 704), e toda vez que a IBM construía um novo processador, a máquina do MIT recebia uma atualização (a cada atualização seu nome mudava: IBM 709, IBM 7090, IBM 7094). O CTSS entrou em produção em 1963 e foi o primeiro sistema operacional que tinha a capacidade de executar uma tarefa em *background* e, ainda, podia capturar interrupções, isolando os usuários de forma parecida como os sistemas operacionais fazem nos dias de hoje, causando exceções de proteção e chamando os serviços do *kernel* [Melinda, 1997]. O CTSS ajudou a IBM a evoluir os processadores, fazendo com que suportassem de forma mais eficiente os sistemas *time-sharing*, um exemplo é o computador IBM 7090 do MIT que durante o tempo de desenvolvimento do CTSS necessitou receber um registrador especial para proteger a

memória. Quando o MIT passou a utilizar os serviços do CTSS para seus usuários e de outras universidades, os sistemas de *time-sharing* começaram a ganhar uma maior importância.

Um pouco depois do CTSS ficar pronto,  a IBM lança no mercado uma nova família de computadores *mainstream* chamada de *System/360* (Figura 1). Esta série de máquinas foi a primeira que utilizou os circuitos integrados, característica fundamental da terceira geração de computadores, fazendo com que elas tivessem um custo menor do que as de segunda geração. Estas máquinas ainda suportavam *upgrades*, com a vantagem de que um *software* desenvolvido para a versão anterior era compatível com a versão que recebeu o *upgrade*, ou seja, a máquina com o *upgrade* continha o subconjunto das instruções de máquina anteriores. A versatilidade do *System/360* era também seu defeito, pois sendo uma plataforma *mainstream*, desenvolver um sistema operacional para ele era muito complicado, este sistema deveria atender programas científicos e comerciais (IO *bound* e CPU *bound*), em pequena e larga escala [Tanenbaum, 2001].

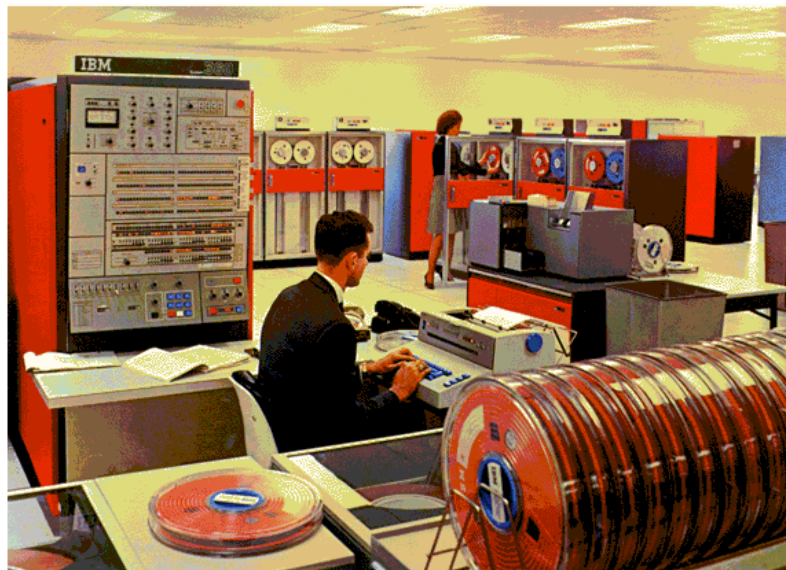



**Figura 1.** O *System/360* da IBM [Google Imagens].

Apesar de toda esta versatilidade, quando o *System/360* foi projetado, os engenheiros da IBM acreditavam que a necessidade do mundo era ter melhores sistemas de *batch* deixando de lado os sistemas de *time-sharing*. O pessoal do MIT e outros entusiastas de sistemas *time-sharing* ficaram desapontados quando a IBM anunciou que o *System/360* não teria registrador de realocação de endereços. Fazia muito pouco tempo que o MIT tinha

anunciado um novo projeto de um sistema operacional de *time-sharing* baseado no CTSS, chamado *Project MAC* (*Mathematics and Computation*), liderado pelo professor Corbató, que pretendia criar o sistema operacional *time-sharing* MULTICS (abreviação de "*Multiplexing Information and Computer Science*"). Para remediar a situação, em fevereiro de 1964, a IBM solicita que Norm Rasmussen vá para Cambridge com a missão de gerenciar o "*Cambridge Scientific Center*" (CSC). Inicialmente, Rasmussen teve que lidar com a reação negativa do MIT em relação ao novo *System/360* e propôs a fazer tudo que fosse possível para que o *System/360* atendesse ao MIT, sugerindo uma nova máquina baseada no *System/360* com modificações para favorecer sistemas de *time-sharing*. Apesar dos esforços de Rasmussen, esta máquina nunca foi construída, a ideia do seu projeto foi rejeitada pelo MIT, que resolveu escrever o MULTICS para um GE 645, implicando em graves consequências para a IBM. O *System/360* ficou ainda mais fragilizado, mesmo depois que o professor Corbató publicou um artigo analisando os pontos falhos desta máquina em relação à construção de sistemas *time-sharing*.

A resposta Rasmussen ao artigo de Corbató foi imediata, dando início a um projeto para a construção de um sistema operacional *time-sharing* para o *System/360* que foi chamado de "*Time Sharing System*" (TSS). Robert Creasy, um programador de grande importância do projeto MAC, estava insatisfeito com as tomadas de decisões do professor Corbató, pois achava que o MULTICS deveria ser um projeto para os compudores *mainstream* da IBM, mas seus colegas de projeto se mostraram inflexíveis em relação a este assunto. Robert Creasy soube que Norm Rasmussen estava interessado em fazer um sistema *time-sharing* para o *System/360* e deixou o MIT, inserindo-se no projeto da IBM. Sabendo que desta vez não podia errar, a IBM reuniu uma força tarefa com os melhores profissionais conhecedores de sistemas operacionais *time-sharing* disponíveis. Andy Kinslow, autor de um sistema experimental de compartilhamento de tempo chamado BOSS (*Big Old Supervisory System*) ficou na liderança de um projeto chamado de TSS (*The Time-Sharing System*), que seria desenvolvido em Nova York e suportaria memória virtual. Nesta época, uma nova máquina foi especificada para servir de base para a construção do TSS, chamada de *System/360-67*, que seria apropriada a sistemas de *time-sharing* [Melinda, 1997].

Não totalmente satisfeito com o TSS, Norm Rasmussen queria um sistema de *time-sharing* não apenas com memória virtual, ele imaginava um sistema integralmente virtual (uma verdadeira máquina virtual). Seguindo esta linha de raciocínio, foi iniciado um novo projeto em Cambridge, liderado por Robert Creasy, este projeto foi chamado de CP-40

(*Control Program 40*). A especificação do CP-40 não era apenas memória virtual, mas uma máquina virtual, pois achavam que era a melhor forma de proteger os usuários de um sistema *time-sharing* uns dos outros. Como o CP-40 seria uma camada de virtualização completa, um sistema operacional customizado precisava ser projetado para executar no topo das máquinas virtuais. Este sistema operacional foi chamado de *Cambridge Monitor System* (CMS) e o teve John Harmor na liderança do projeto. A ideia do CMS era que cada usuário tivesse a sua própria máquina virtual (*single-user)*. Por fim a combinação CP-40 com o CMS resultou no CP/CMS, concluído em 1967. O ponto chave do CP/CMS era a divisão da gerência dos recursos do computador e o suporte ao usuário.

Em 1972, uma versão mais refinada do CP/CMS foi disponibilizada, chamada de "*Virtual Machine Facility/370*" ou simplesmente VM/370 [Melinda, 1997]. O sistema operacional do VM/370, na verdade, era constituído de três sistemas operacionais diferentes. Estes sistemas eram respectivamente o "*control program*" (CP-67), o "*conversational monitor system*" (CMS) e o "*remote spooling and comunications subsystem*" (RSCS). O CP-67 era usado para levantar máquinas virtuais, particionando o *hardware* da máquina real; estas máquinas executavam seus próprios sistemas operacionais e eram independentes. O CMS era o sistema operacional exclusivo de cada usuário. O RGCS era um sistema usado para transferir informações entre as máquinas virtuais. Após o VM/370 surgem outras famílias de computadores como o VM/390, fazendo com que outros sistemas operacionais aparecessem, como, por exemplo: OS/90, VSE, Linux/390 [Creasy, 1881].

Nos anos 80, devido à miniaturização dos componentes eletrônicos e a popularização do uso do computador pessoal, a virtualização perdeu a importância, pois era economicamente viável que cada usuário tivesse seu próprio microcomputador pessoal [VMware, 2011].

No final da década de 90, e no início do novo milênio, a computação distribuída e aplicações do tipo cliente-servidor passam a ser dominantes. *Datacenters* com diversos servidores começaram a substituir *mainframes* e se torna interessante utilizar a virtualização para isolar serviços e centralizar gerenciamentos. Surge a linguagem *java*, que faz com que os olhos do mundo se voltem à virtualização, mostrando como ela pode ser uma elegante maneira de conseguir universalizar aplicações em um mundo heterogêneo. A virtualização passa a ser utilizada para promover a segurança, permitindo monitorar e prevenir ataques em um nível mais privilegiado que o atacante, sendo incorporada a *anti-virus.* Analisadores de

comportamento das aplicações baseados em máquinas virtuais começam a ser utilizados por *debuggers*. Não demorou muito para que máquinas virtuais também fossem utilizadas novamente em sistemas operacionais, notou-se que além de promover isolamento, poderiam fazer tolerância à faltas ou balanceamento de carga através de simples *clonagens* [Kin et al., 2006]. [Rosenblum, 2004] faz uma analogia em seu artigo sobre a volta do interesse da comunidade em máquinas virtuais usando o termo "reencarnação".

Nos últimos anos a virtualização vem sendo utilizada para fazer a elasticidade de recursos do *hardware*, resolvendo um dos maiores paradigmas da ciência da computação. A elasticidade de recursos é um conceito complexo que engloba a virtualização juntamente com algoritmos de computação autonômica.

## 2.2 Arquitetura de máquinas virtuais

Para estudarmos as máquinas virtuais, em primeiro lugar, é necessário conhecer interfaces, e como estas são usadas na pilha de *software* de um computador. Assim como qualquer *software*, uma máquina virtual respeita a interface sob a qual ela foi construída, e tem como objetivo, fornecer outra interface fazendo um acoplamento entre camadas, conforme pode ser observado na Figura 2.

**Figura 2.** Acoplamento entre interfaces [Laureano e Maziero, 2008].

As diversas interfaces em um computador formam o que chamamos de pilha de *software*. Como, em teoria, não existem limites para estas camadas, é possível construir uma pilha de *software*, produzindo quantas camadas quisermos. A Figura 3 mostra como sistemas operacionais e máquinas virtuais podem ser usados para construir uma pilha de *software*.

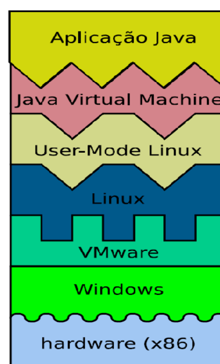



**Figura 3.** Pilha de *software* num sistema [Laureano, 2006].

A única limitação prática que temos em uma pilha de *software* é o *overhead*, pois a cada nova camada de *software* é gerado um consumo de recursos adicional. O *overhead* que a pilha de *software* impõe pode ser muito significativo dependendo do número de interfaces e da forma com que elas foram construídas.

Para entender como funciona uma máquina virtual, em primeiro lugar precisamos conhecer um pouco as principais interfaces de um sistema . A primeira interface (mais baixa) controla todo computador, ela é um conjunto de instruções fornecidas pelo *hardware, que* chamamos de *instruction set architeture* (ISA). A ISA ainda é subdividida em *System* ISA e *User* ISA [Laureno e Maziero, 2008].

a) *System* ISA: São as instruções privilegiadas, as quais podem ser executadas apenas com o *bit* de supervisor do processador ativado, quem usa estas instruções diretamente é apenas a primeira camada da pilha de *software*.

b) *User* ISA: São as instruções que podem ser executadas diretamente por qualquer programa de computador; estão sempre disponíveis, mesmo que o *bit* de supervisor do processador esteja desativado. Todos os programas de um computador poderão executar livremente estas instruções sem nenhuma restrição.

Um estudo mais detalhado sobre o bit de supervisor do processador e a arquitetura de *hardware* para máquinas virtuais pode ser encontrados no Anexo I.

Apesar das aplicações comuns não terem acesso direto a *System* ISA, elas também necessitam de algumas instruções privilegiadas para funcionarem corretamente, mas, se isso fosse permitido diretamente, seria uma grave falha de segurança. Para resolver este problema, a primeira camada de *software* (geralmente um sistema operacional) fornece o que chamamos

de *syscalls*, que é uma forma indireta e controlada de aplicações não privilegiadas usarem instruções da *System* ISA. Como as aplicações comuns de um sistema operacional usam ambas interfaces (*syscalls* e *user* ISA), estas interfaces recebem o nome especial *application binary interface* (ABI). Fazer programas diretamente para a ABI pode ser complexo, pois ela utiliza muitos conceitos de baixo nível. Para diminuir um pouco esta complexidade, uma nova interface, na forma de um acervo de bibliotecas, geralmente é usada pelos programas de usuários, facilitando a programação. Esta interface é chamada de *libcalls*. A Figura 4 mostra exemplos de interfaces.

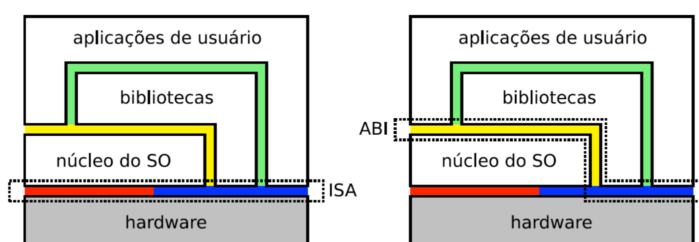



**Figura 4**. Interfaces entre camadas num sistema[Laureno e Maziero, 2008].

Máquinas Virtuais usam algumas taxonomias especiais. Neste trabalho, usamos a abreviação VM (*virtual machine*) para referenciar o programa que simula a interface virtual da ISA. A abreviação VMM (*virtual machine monitor*) [Rosenblum and Garfinkel, 2005] será usada para se referenciar ao hipervisor, que é o programa de controle da máquina virtual. A principal função do hipervisor é criar, excluir, modificar e monitorar as máquinas virtuais. No topo de uma máquina virtual podem ser instanciadas aplicações, sistemas operacionais ou ainda outras máquinas virtuais, caracterizando máquinas virtuais recursivas (RVM – *recursive virtual machines*). Quando uma aplicação está executando na interface de uma máquina virtual, dizemos que a aplicação é convidada, e se for um sistema operacional, o chamamos de sistema operacional convidado.

As máquinas virtuais são classificadas de forma diferente de acordo com a interface que respeitam, a interface que exportam e o sistema convidado que executam. Neste trabalho, um tipo especial de máquina virtual, classificado de *máquina virtual de sistemas,* é o objeto de nosso estudo. A classificação completa de máquinas virtuais e um estudo aprofundado sobre o assunto pode ser encontrados no Anexo II. Como o sistema convidado de uma *máquina virtual sistemas* é um sistema operacional, usaremos a abreviação SO para

referenciar a um sistema operacional qualquer. Existem casos que um sistema operacional pode servir como sistema base para um VMM, caracterizando um tipo especial de monitor de *máquina virtual de sistemas* chamado de tipo 2. Um SO pode ser do tipo monolítico ou *microkernel* [Firmino et al., 2007], e ambos podem ser executados como sistemas convidados em uma VM ou serem usados de base uma VM tipo 2. Um SO monolítico se caracteriza por ter um núcleo denso, incluindo além de primitivas básicas, todo o controle de dispositivos e *drivers*. *Microkernels* são sistemas operacionais enxutos, possuindo um conjunto mínimo de primitivas em seu núcleo. Um exemplo de máquina virtual utilizando SO monolítico pode ser dado pelo XEN [Xen, 2011], e outro exemplo usando S.O. microkernel pode ser dado pelo trabalho de [Peter et al., 2009] em *Virtual Machines Jailed.*

Para entender como as máquinas virtuais podem flexibilizar o uso do *hardware*, podemos observar a Figura 5. Na figura da esquerda, a primeira camada de *software* em contato com o *hardware* (em verde vivo), é a camada de virtualização. O *windows* é um SO que foi projetado para executar em um *hardware* que exporta uma interface ISA do tipo X86. Como não é possível executar o *windows* em uma máquina *spark*, a camada de virtualização é usada para traduzir  as instruções da máquina *spark* para X86, assim, o sistema operacional *windows* pode entender estas instruções. A virtualização é a única maneira de poder executar um sistema operacional em um processador para qual ele não foi projetado. O *windows* nada sabe sobre a camada de virtualização, a visão dele sobre as interfaces mais baixas é representada pela figura 5 (parte da direita), ou seja, ele continua achando que esta sendo executado em cima da ISA X86 de um *hardware* real, mas na verdade,  ele esta sendo executado no topo de uma interface X86 virtual.

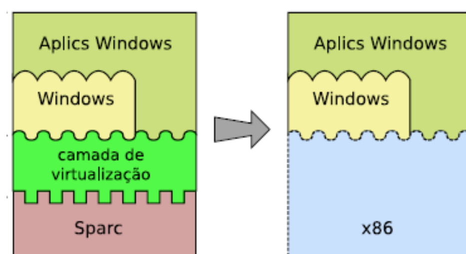



**Figura 5**. Transparência provida pela camada de virtualização [Laureano e Maziero, 2008]

## 2.3 Propriedades de máquinas virtuais

Existem três propriedades clássicas descritas por [Popek e Goldberg, 1974] para máquinas virtuais. São elas:

a) Eficiência: Todas as instruções não sensíveis poderão ser executadas diretamente no *hardware* pelas máquinas virtuais, sem intervenção do hipervisor. Isso garante agilidade na execução de códigos na máquina virtual sem comprometer a segurança.

b) Controle de recursos: Um programa não pode alterar, delegar, remover recursos globais diretamente. Quando algo nesse sentido for necessário, o hipervisor deve intervir.

c) Equivalência: As instruções são processadas de forma que o programa que está executando na plataforma virtualizada não percebe que o programa de controle está monitorando o mesmo e intervindo quando necessário. Existem duas exceções que podem acontecer para esta propriedade, fazendo com que um programa executado em máquina virtual venha a perceber sutis diferenças em relação a um programa executado na máquina real, que são respectivamente tempo de resposta e disponibilidade de recursos. O tempo de resposta modificado é devido ao *overhead* imposto pela virtualização. A disponibilidade de recursos pode ser alterada porque outras máquinas virtuais e o próprio programa de controle consomem recursos.

A partir dessas propriedades clássicas é possível definir outras derivadas, que se tornaram mais evidentes nos anos 90, quando a virtualização voltou a ganhar importância. Para melhorar nossa análise, propriedades adicionais intuitivas foram colocadas além das citadas por [Rosenblum, 2004].

a) Compatibilidade de *software*: A virtualização tem como característica suportar nativamente as aplicações desenvolvidas para o sistema real.

b) Encapsulamento: Como o hipervisor encapsula toda pilha de *software* e todo *hardware* das máquinas virtuais, ele tem poder irrestrito sobre as instâncias virtualizadas.

c) Isolamento: O hipervisor isola as máquinas virtuais umas das outras com pouco código e de maneira muito eficiente. Programas de uma determinada máquina virtual não tem acesso à área de memória de outra máquina virtualizada. A propriedade do encapsulamento dá ao hipervisor o controle total dos recursos de *hardware* e somente

ele pode designar ou revogar recursos a uma máquina virtual. Ainda sobre o isolamento.

d) Monitoramento: Os hipervisores possuem o controle total de toda pilha de *software* que está sendo executada em seu topo. Toda máquina virtual pode ser monitorada de forma transparente, e com base no monitoramento, ações ser tomadas tanto pelo usuário administrador, como pelo próprio hipervisor.

e) Migração: Máquinas virtuais podem ser migradas, permitindo a separação entre *hardware* e *software* de forma rápida e eficiente, muito útil no caso de manutenção e também na replicação e balanceamento de carga.

f) Capacidade de gerenciamento facilitada: Um hipervisor é um sistema centralizado que tem o controle de toda a pilha de *software*. É muito fácil administrar as diversas máquinas virtuais, executando sob seu domínio. Sistemas de virtualização possuem ferramentas poderosas, fazendo uma interface direta com o administrador do sistema, o qual gerencia as máquinas virtuais com um nível de privilégio superior.

g) Possibilidade de *checkpointing*: Devido à propriedade do monitoramento é possível salvar os estados das máquinas virtuais, utilizando-se uma tecnologia chamada de *checkpointing*. Esta tecnologia permite serem guardados diversos *snapshots* do sistema para uma restauração posterior, caso algo não ocorra dentro do esperado.

h) Tolerância a faltas: Máquinas virtuais podem ser usadas para replicar sistemas reais ou virtuais, promovendo o aumento de disponibilidade dos serviços com grande versatilidade. Em caso de uma pane identificada ou até mesmo prevista, podem ser usadas estratégias de migração de máquinas virtuais para disponibilizar o serviço novamente. Um exemplo pode ser dado pelo R-Xen [Jansen et al. 2008] que usa a replicação de VMs para promover tolerância a faltas.

i) Balanceamento de carga: O hipervisor pode aproveitar sua capacidade de monitoramento e identificar gargalos entre as diversas instâncias virtualizadas. Utilizando um algoritmo de balanceamento de carga é possível direcionar requisições para outras máquinas virtuais mais ociosas. Caso não houver máquinas virtuais disponíveis, em pouco tempo, poderá ser feita uma cópia de replicação da máquina virtual para o balanceamento de carga poder ser rapidamente estabelecido.

j) Recursividade: Hipervisores podem ser construídos recursivamente em cima de outros hipervisores. Isso traz uma grande flexibilidade ao sistema, pois cada hipervisor poderá ter controle sob outras máquinas virtuais com hereditariedade direta (filhos) ou indireta (netos, bisnetos, etc.). O principal problema desta abordagem é o *overhead* que cada camada de virtualização impõe, mas isso pode ser contornado com técnicas de credencias, fazendo uma espécie de curto circuito entre as camadas [Ford et al., 1996].

k) *Debug*: Máquinas virtuais são extremamente úteis para *debugar* aplicações, pois elas conseguem analisar os registradores internos e o comportamento da memória da máquina de forma totalmente transparente, possibilitando inclusive a engenharia reversa de programas.

l) Segurança: Muitos trabalhos afirmam que máquinas virtuais podem ser consideradas seguras devido a propriedade do isolamento , por outro lado ataques maliciosos no VMM são muito perigosos, pois são muito difíceis de serem detectados pelo sistema operacional convidado, comprometendo todas as instâncias virtuais. Visando segurança, [Garfinkel and Warfield, 2007] descrevem em seu trabalho o que as máquinas virtuais podem fazer pela segurança e [Jansen et al. 2008] desenvolveram o R-Xen, que é uma máquina virtual com detecção de intrusos. No ponto de vista do atacante, [Ferrie, 2006] alerta que ataques podem ser feitos de forma direta no monitor de máquina virtual e [King et al., 2006] desenvolve uma espécie de *malware* para máquinas virtuais chamado de *subvirt*.

Apesar de ter inúmeras propriedades vantajosas, o principal problema de utilizar máquinas virtuais é o *overhead*. É sempre importante avaliar quais os benefícios que a virtualização irá proporcionar, mensurando-se o *overhead* imposto pela camada de virtual. Nos últimos anos, diversas técnicas especiais foram criadas para diminuir o *overhead* de máquinas virtuais, estas técnicas são explicadas em detalhes no Anexo III. Hoje em dia temos máquinas virtuais com *overhead* aceitável, seu desempenho pode ser muito próximo ao de sistemas executando em máquinas reais.

## 2.4 Migrações de máquinas virtuais

As migrações de máquinas virtuais são de grande importância para datacenters, pois permitem transferir uma máquina virtual de um servidor à outro, reorganizando as máquinas virtuais nas máquinas físicas. As principais vantagens da migração são:

a) Desligar máquinas físicas para fins de economia de energia, concentrando as VMs em um número menor de máquinas.

b) Eliminar gargalos de servidores, fazendo um balanceamento de carga entre as máquinas ligadas.

c) Permitem aumentar a demanda de recursos de hardware de uma VM, não se limitando aos recursos físicos da máquina na qual a VM está instanciada.

d) É umas das ferramentas mais importantes para a computação autonômica na nuvem.

Apesar da migração de máquinas virtuais ser um recurso muito importante para a construção de uma nuvem de computadores de qualidade, existe uma limitação no seu uso indiscriminado devido ao *overhead*. Ao migrar uma VM é necessário copiar todas as informações relacionadas a ela do servidor de origem para o servidor de destino, a forma mais simples de fazer isso é desligar a VM para que seu estado seja persistido no disco, este modelo de migração é chamado de *stop and copy*. Existem ainda técnicas que tornam possível fazer a migração de forma transparente para o usuário não necessitando desligar a VM origem, estas técnicas são classificadas como *live migration*, e de acordo com a heurística empregada, as técnicas podem receber nomes específicos como *pré-copy*, *pós-copy* ou *modelo híbrido*.

As técnicas de migração de máquinas são detalhadas abaixo [Magalhães et al., 2011] :

a) Migração *stop and copy*: Para a máquina virtual de origem poder copiá-la. É um modelo com *downtime* muito grande, chegando a ser 5x maior que o *pré-copy*, porem possui um tempo total de migração melhor. Deve ser utilizado apenas quando a aplicação não necessitar de disponibilidade.

*b)* Migração *pré-copy*: A VM origem pode continuar sendo executada e respondendo às requisições no tempo de migração, diminuindo o seu tempo de *downtime*. A memória é reconstruída no servidor destino através de várias iterações. Ao final a VM origem é suspensa, o estado do processador é transferido e as páginas “sujas”

atualizadas; após a sincronização, as VM origem e destino podem ficar *online* (ou então a VM origem pode ser parada e apagada, se não for mais utilizada).

*c)* Migração *pós-copy* [Hines, 2009]: No *pós-copy*, a VM no hospedeiro destino é executada antes da transferência das páginas de memória, diminuindo o *downtime* da VM destino e também da VM origem. Para isso, logo após a preparação da migração, a VM origem é parada e as páginas sujas são copiadas com o estado do processador, terminando o processo ambas as VM são resumidas. Nesta abordagem, existe um aumento grande de *overhead* da aplicação na VM destino, pois ela começa a buscar por páginas na VM fonte toda vez que uma *page fault* é gerada. Uma tentativa de otimizar este processo pode ser feita, identificando padrões de acesso da memória na aplicação, antecipando o processo *paging*, reduzindo o número de *page faults*.

*d)* Modelo híbrido [Hines, 2009]: É a mistura do *pré-paging* e *pós-copy*, na fase de preparação, as páginas de memória são copiadas para a VM destino, e a VM origem continua recebendo requisições. Depois da primeira iteração, a VM origem é suspensa e o estado do processador é transferido. Depois ambas as VM, origem e destino, são habilitadas e o modelo segue funcionando com faltas de páginas como no modelo *pós-copy*.

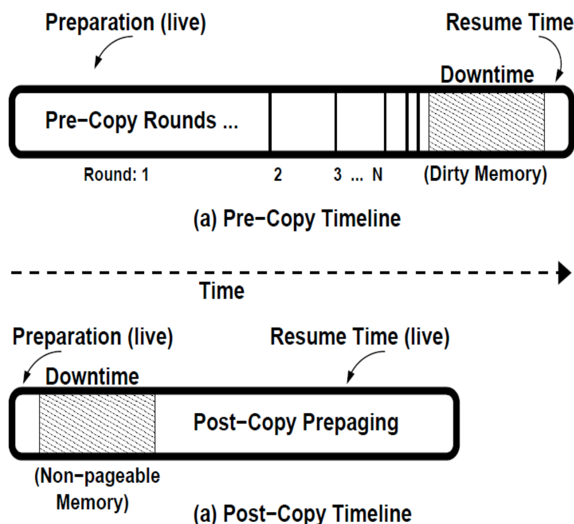



**Figura 6**. Tipos de Migração [Hines, 2009].

## 2.5 Computação em nuvem

Apesar do paradigma de computação em nuvem ser recente, algo parecido já foi pensado em 1960 com John Carthy, sua ideia era a construção de *datacenters* onde computadores poderiam ser utilizados pelo público em geral. O termo *cloud* já havia sido utilizado na década de 90 para se referenciar as grandes redes ATM. Em 2006, Eric Schmidt (CEO do Google) usa o termo *cloud computing* pela primeira vez para descrever um novo modelo de negócios através de serviços disponíveis na *Internet* [Zhang et al., 2010]. O modelo *pay-per-view* (pague por assistir) esteve presente em provedores de TV a cabo por mais de uma década, onde se paga toda vez que se deseja assistir a algum jogo de futebol ou um filme de lançamento. A computação em nuvem usa o mesmo conceito para servidores virtualizados com o paradigma do *pay-per-use* (pague por usar).

Além do conceito de computação ter raízes antigas, as tecnologias utilizadas em sua construção são conhecidas há vários anos. A base do *cloud computing* é a virtualização, a qual existe desde 1960 quando começaram os primeiros projetos de máquinas virtuais, juntamente com sistemas de *time-sharing.* A computação distribuída é um paradigma utilizado desde os anos 80 e 90. A computação em *grid*, onde vários computadores distribuídos executam uma parte de uma tarefa particionada, com o objetivo de executar a tarefa em comum, é tão antiga quanto os sistemas distribuídos. A computação autonômica é um conceito criado em 2001 pela IBM, neste tipo de computação, um computador pode ser capaz de se auto-gerenciar, alocando e liberando recursos quando necessário, analisando de forma inteligente o seu próprio estado interno. Podemos concluir que, apesar da computação em nuvem ser uma tecnologia bem recente, a tecnologia que possibilita sua construção é antiga.

Nos últimos anos, várias matérias sobre a computação em nuvem saíram na mídia (principalmente em veículos de comunicação de tecnologia), mas por falta de definições claras, algumas empresas pensam estarem oferecendo um ambiente de computação em nuvem, inclusive afirmam isso em suas páginas *web*, mas não passam provedores, alugam máquinas virtuais e muitas vezes demoram dias para colocar *online* um o ambiente virtual contratado. Um ambiente em nuvem é um pouco mais complexo do que simplesmente virtualizar o *hardware.*

O NIST (*The National Institute of Standards and Technology*) define a computação em nuvem como: "Um modelo para permitir uma forma conveniente de um acesso através da

rede de computadores (*Internet* ou *intranet*) a um conjunto de recursos computacionais configuráveis (redes, servidores, armazenamento, aplicações e serviços), demandados pelo cliente, os quais podem ser alocados rapidamente e descartados com um mínimo esforço gerencial ou interferência do provedor do serviço". Com esta possível definição, podemos enumerar cinco características que são fundamentais para um ambiente em nuvem [Mell e Grance, 2010]:

a) *Pool* de recursos : Os recursos físicos e lógicos do provedor de serviços são colocados à disposição dos clientes com independência de localização. O cliente não precisa saber exatamente onde o recurso está, mas alguma noção de localidade deve ser fornecida, como, por exemplo, informando em qual cidade se encontra o *datacenter* daquela máquina virtual. O cliente poderá usar estas informações de localização geográfica para fazer um plano de redundância (alta disponibilidade) com servidores fisica e geograficamente separados. Quanto mais *datacenters* espalhados pelo mundo, melhor a resiliência da nuvem em relação a desastres.

b) Automação de serviços: À medida que o cliente achar necessário, ele pode contratar novos serviços de forma automática, sem a intervenção de seres humanos. Para isso, é necessário que o provedor da *cloud* possua uma interface na qual seja possível o cliente escolher novos serviços, os quais serão colocados automaticamente no ar.

c) Elasticidade: Os recursos são reconfiguráveis com ou sem a necessidade de intervenção humana. Recursos adicionais podem ser comprados e automaticamente acrescentados ou podem ser limitados e, até mesmo, extintos a qualquer momento. Em outras palavras, o usuário pode contratar recursos adicionais de acordo com os picos de uso em seu sistema, e ao mesmo tempo pode redimensionar recursos que estão superdimensionados, evitando o desperdício. Além do controle de demanda de recurso manual, o usuário pode escolher a opção do sistema se auto monitorar e gerenciar os recursos de forma automática.

d) Acesso e controle por rede: Qualquer dispositivo com conexão à *Internet* pode acessar os serviços oferecidos pela nuvem através de um sistema de credenciais. Uma vez verificada a credencial do usuário, este poderá utilizar e gerenciar remotamente todos serviços oferecidos pela nuvem, da forma que mais lhe convier (Ex: celulares, pda, *notebooks*, computadores *desktop*).

e) Transparência no uso de recursos: O ambiente em nuvem deve monitorar e controlar os recursos utilizados pelo cliente, estipulando o valor que o cliente deve pagar. O cliente deve ter acesso ao sistema de monitoramento para saber por quais recursos e quantidades esta pagando, para isso ferramentas devem ser fornecidas pela nuvem, orientando o cliente a aumentar, diminuir ou continuar com a mesma quantidade de recursos contratados.

Baseado no conjunto de tecnologias utilizadas para a construção de *clouds*, uma outra definição, complementar a do NIST, é possível: “A computação em nuvem usa a tecnologia de virtualização com o objetivo de prover recursos computacionais. Ela usa os conceitos de computação em *grid* e autonômica, mas difere delas em alguns aspectos, oferecendo benefícios e impondo desafios para viabilizar suas necessidades ”. Com base nesta definição, [Zhang et al., 2010] apresentam as seguintes características:

a) Não é necessário um investimento inicial: O modelo *pay-per-use* dispensa qualquer investimento inicial. A nuvem esta sempre pronta para ser usada. A empresa que contrata um serviço da *cloud* não precisa esperar vários meses (que é o tempo que esperaria se fosse comprar todos os equipamentos necessários, montar estrutura de *datacenter* e servidores) para começar a produzir, pode fazer isso quase que de forma instantânea. Além da enorme economia em infraestrutura oferecida pelo modelo, todos os riscos envolvendo sistemas operacionais e servidores são transferidos para o provedor de recursos.

b) Baixos custos de operação: Recursos são rapidamente alocados e revogados sob demanda, minimizando, de forma muito inteligente, os custos operacionais.

c) Altamente escalável: A elasticidade fornecida pela computação em nuvem permite uma escalabilidade jamais vista em outros ambientes computacionais.

d) Redução de riscos: Todo risco referente à infraestrutura é assumido pelo *cloud provider.* Serviços adicionais de monitoramento, *backup*, balanceamento de carga e tolerância a faltas podem ser oferecidos ao cliente para aumentar a *performance* e confiabilidade do sistema hospedado, agregando valor ao ambiente computacional alugado.

e) Redução de despesas de manutenção: Gastos com funcionários especialistas, energia, cabeamento, refrigeração e *hardware* são diluídos entre todos os usuários da *cloud*, possibilitando o uso de um ambiente de alta disponibilidade a um preço justo.

[Calheiros et al., 2011] definem a computação em nuvem como um contrato entre o cliente e o servidor: "Um sistema paralelo é distribuído que é formado de uma coleção de computadores virtualizados interconectados que são dinamicamente alocados e dispõe de um ou mais recursos computacionais unificados baseados em contratos estabelecidos entre o provedor e o consumidor."

As empresas mais conhecidas que oferecem o serviço da computação em nuvem são: IBM (*Blue Cloud*), Amazon (*Elastic Compute Cloud*), Google (*Google App Engine*), Microsoft (*Windows Azure Plataform*), Salesforce (*Force.com*). Muitas dessas empresas oferecem ambientes flexíveis, onde o desenvolvedor tem a possibilidade de escolha entre paradigmas, linguagens de programação e plataformas, escolhendo seus sistemas operacionais e configurando-os com serviços da maneira que mais convém, como é o caso da *Amazon Elastic Compute Cloud*. Outras empresas restringem mais o acesso dos clientes forçando o uso de uma linguagem de programação específica ou dando suporte apenas a um paradigma, como é o caso do *Google App*, que restringe o uso da *cloud* ao paradigma da web e algumas linguagens de programação empregadas neste contexto. Uma tática comum de todas estas empresas é a de não revelar detalhes profundos de como a *cloud* funciona, por isso a pesquisa sobre computação em nuvem utilizando soluções *open source* será a melhor forma de popularizar e promover o uso desta tecnologia, possibilitando o avanço da tecnologia através da comunidade científica, desenvolvendo e discutindo possíveis soluções para os problemas enfrentados pelo paradigma da nuvem que parece desafiar todas as limitações de um ambiente computacional [Endo et al., 2010].

A tecnologia fornecida pela computação em nuvem pode ser dividida em alguns modelos. Estes modelos são classificados de acordo com o tipo de serviço fornecido: IaaS (*Infrastructre as a service*), PaaS (*Platform as a Service*) e SaaS (*Software as a Service*) [Mell e Grace, 2010]. O modelo em camadas esta ilustrado na Figura 7.

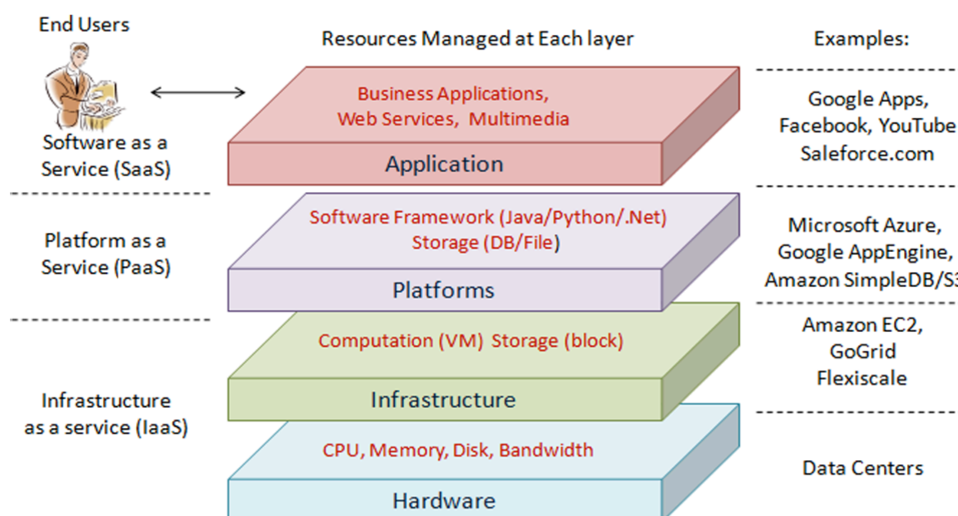



**Figura 7**. Camadas do paradigma da *Cloud Computing* [Zhang et al., 2010].

**Infraestrutura como um Serviço**

O primeiro modelo de serviço que vamos analisar é o IaaS. Hipervisores como o Xen, KVM e VMware são o ponto chave fundamental para ser possível o uso dinâmico do *hardware*, promovendo a elasticidade de recursos fornecida por este modelo de negócio. Os provedores de IaaS possuem *datacenters* especializados e interfaces (geralmente *web*) para que os clientes possam escolher a quantidade de recursos virtuais (processadores, *link* de *Internet*, discos, localidade geográfica, etc.) e também qual sistema operacional será instanciado. No IaaS, a nuvem fornece a infraestrutura básica para a instalação do sistema operacional do cliente, mas sempre de uma forma dinâmica. O cliente não controla nem gerencia o *hardware* e também não tem nenhum acesso aos hipervisores, mas tem controle total de seu sistema operacional virtualizado. Geralmente o cliente escolhe o sistema operacional que deseja instanciar através de uma interface *web* por um *menu* que possui inúmeras imagens de sistemas operacionais possíveis a serem utilizados. Existem *cloud providers* que permitem ainda a opção de fazer *upload* de uma imagem de uma distribuição de sistema operacional específica, desde que respeite os tipos de sistemas operacionais passíveis de serem virtualizados por aquele hipervisor. No caso de sistemas operacionais pagos, o cliente deverá entrar com a chave comprada antes de instanciar sua máquina virtual (Ex: chave do *Windows server*) ou o *cloud provider* pode fornecer uma chave de forma automática caso ele esteja disposto a pagar por ela. Como exemplo de provedores do modelo IaaS temos: Amazon EC2, GoGrid e Flexiscale.

*Datacenters* é o nome dado a estruturas físicas especializadas que acomodam toda infraestrutura física necessária para que o ambiente da nuvem funcione de forma correta. A responsabilidade do *datacenter* não se limita a fornecer os recursos físicos, mas também deve garantir o bom funcionamento de toda estrutura, primando pela segurança e integridade física do local. Os recursos físicos são: energia elétrica, baterias (*no breaks*), geradores de energia, *links* de *Internet*, condicionadores de ar, *racks* (para servidores), piso elevado (para organização de cabeamento e fluxo de ar em baixo dos *racks*), cabeamento de energia e rede, sistema de detecção e prevenção automática de incêndios (geralmente sensores espalhados pela sala com ductos que transportam um produto que fica armazenado em enormes cilindros, quando o sensor detecta fogo ou fumaça libera o produto armazenado no cilindro na atmosfera, impedindo que o fogo continue se alastrando), portas corta fogo, entre outros. O *hardware* também faz parte dos recursos físicos: os servidores (máquinas completas com processadores, memória, controladoras avançadas de disco, discos rígidos, memória RAM e ssd (*solid state drives*), interfaces de rede, *storages* (uma espécie de servidor dedicado ao armazenamento de informações), *switches* e roteadores.

Os *datacenters* fornecem alguns recursos de tolerância a faltas no nível de infraestrutura. Para isso, o datacenter deve possuir: alimentação de energia de corrente alternada proveniente de fontes diferentes, sistemas de *no-breaks* redundantes (caso um deles falhe, outro em *stand by* deve assumir), pelo menos dois geradores de energia elétrica que atendam individualmente todo parque de máquinas, *links* de Internet redundantes, discos em *raid* configurados em espelhamento ou gravação de paridade, a controladora de disco deve ter baterias para não corromper dados em caso de falta de energia, a ventilação dentro dos gabinetes dos servidores deve ser redundante, bem como as fontes de alimentação que convertem a energia alternada proveniente da rede elétrica (ou *no-break*) em energia contínua (cada fonte de alimentação de um servidor deve ser ligada a uma rede de *no-brakes* separada).

Nos últimos anos surgiu uma nova tecnologia barateando os custos, economizando energia, espaço e gerando menos calor nos *datacenters*. Esta tecnologia é chamada de *blade servers*. Um servidor *blade* é muito semelhante a um servidor normal, mas ele possui um módulo principal com fontes de alimentação e ventilação redundantes. A este módulo podem ser acopladas diversas lâminas, onde cada uma corresponde a um servidor ou a um *storage*. Todas as lâminas compartilham a alimentação e a ventilação redundante do módulo principal. Em um rápido exemplo, com esta tecnologia, é possível compartilhar 8 fontes de alimentação entre 16 lâminas (um servidor por lâmina), economizando nos custos de fabricação do

*hardware*, e aumentando, ainda, a tolerância a faltas. Em um servidor *blade* é possível ter o módulo principal de vários tamanhos (Ex: 8, 16 lâminas). Este módulo geralmente é colocado dentro de um *rack* de 42 U que geralmente pode comportar vários módulos. Financeiramente, um servidor *blade* usado com poucas lâminas é mais caro que servidores convencionais, pois existe o custo inicial das lâminas adicionado do módulo principal. Como o preço da lâmina é um pouco menor que o de um servidor convencional, as lâminas que vão sendo adicionadas representam uma economia significativa no custo final.

**Plataformas como um serviço**

Neste modelo de computação em nuvem, chamado de PaaS (*Platform as a Service*) é fornecido para o cliente uma plataforma que torna possível desenvolver e executar aplicativos desenvolvidos por linguagens de programação e ferramentas específicas devidamente suportadas pelo provedor. Neste modelo de serviço o cliente não controla os hipervisores nem os sistemas operacionais virtualizados, no entanto ele consegue fazer *deploy* de aplicações e executá-las [Mell e Grace, 2010].

Os *frameworks* que podem ser utilizados neste tipo de serviço são os mais diversos, como as plataformas *Java, Phython, Net, Ruby, Perl*. Sistemas gerenciadores de banco de dados também fazem parte da plataforma, tais como *MySql, PostgreeSQL, IBM DB2, Oracle* e *Caché*. O *Microsoft Azure*, O *GoogleAppEngine* e o *AmazonSimpleDb/S3* são exemplos de provedores que fornecem plataforma como serviço.

**Software como um serviço:**

O último modelo que vamos analisar é o SaaS (*Software as a Service*). Neste modelo, o cliente, executando na nuvem, usa aplicações fornecidas pelo provedor. As aplicações são acessíveis de diversos dispositivos, desde que tenham uma conexão com a *Internet* ou com a *intranet*. O cliente não tem controle algum sobre os sistemas operacionais, executando na nuvem, e não pode fazer *deploy* de aplicações, o único acesso que o cliente pode ter é algum menu de configuração pré-definido pelo provedor para fazer algum ajuste no aplicativo, mas via de regra o cliente apenas usa a aplicação e não possui nenhum privilégio adicional [Mell e Grace, 2010].

Alguns exemplos de provedores de Software como serviço: *Google Apps*, *Facebook*, *YouTube*, *SalesForce*.

**Maneiras de publicar serviços em nuvem:**

[Mell e Grace, 2010] expõe quatro maneiras diferentes de publicar serviços na nuvem, a saber: Privada, Comunitária, Pública e Híbrida:

a) Privada: Toda infraestrutura da nuvem é gerenciada pela própria empresa ou organização que a criou. Os serviços não são publicados fora da *intranet* da empresa e a nuvem é protegida por um *firewall*. A organização arca com todos custos da nuvem.

b) Comunitária: Modelo menos comum, quando os recursos de uma nuvem são compartilhados entre algumas organizações. O conjunto de organizações, que participa da nuvem, arca com os custos. Um exemplo deste tipo pode ser dado pela plataforma de testes *OpenCirrus*.

c) Pública: Modelo mais comum, todos recursos da nuvem ficam disponíveis publicamente. Os recursos podem ser contratados pelo modelo *pay-per-use*, não existe investimento do cliente, e o *cloud provider* arca com todos custos iniciais. Exemplo: *Amazon EC2*.

d) Híbrida: A organização tem uma nuvem privada, mas também utiliza e tem acesso a recursos da nuvem pública ou comunitária.

## 2.6 Discussão sobre computação em nuvem

Em [Chong, 2011], afirma-se que não são todas aplicações que são adequadas para colocar em um ambiente de computação em nuvem e [Lui and Wee, 2009] avalia a performance e limitação dos componentes em uma nuvem. Cargas massivas de dados transacionais, dados que necessitam de máxima segurança e privacidade (como movimentações bancárias) e aplicações complexas que necessitam de requerimentos muitos específicos não se enquadram no paradigma da computação em nuvem. Aplicações complexas podem necessitar de servidores e *arrays* de *Raid* exclusivos para extrair o máximo de desempenho do *hardware*, um ambiente virtualizado por si só poderia ser prejudicial à aplicação. A questão da segurança e privacidade é obvia, se esta for a prioridade, até mesmo

funcionários que administram os hipervisores em uma nuvem pública poderão comprometer o nível de segurança desejado, pois  o hipervisor tem acesso a todos discos virtualizados. Quanto a cargas massivas transacionais, elas requerem *lock* de escrita e demandam muita E/S de disco, o que não pode ser apropriado para um ambiente compartilhado. Ambientes de desenvolvimento e testes de sistemas empresariais, aplicações com cargas variáveis, aplicações sazonais são ideais para o ambiente em nuvem.

Além todas as características e mecanismos para computação em nuvem descritos neste capítulo, existe a necessidade de um mecanismo eficiente para alocação de máquinas virtuais em *data centers*. Como esse é o assunto de estudo desta dissertação, será abordado em detalhe no Capítulo 3.

# 3 Alocação de máquinas virtuais em *data centers*

Este Capítulo explora a variabilidade do problema de alocação de VMs em *data centers* e suas possíveis soluções. Dizemos que uma VM é alocada quando está consumindo recursos em um *host*, assim, podemos formalizar o conceito de alocação de VMs como segue.

**Formalização do conjunto de VMs alocadas:**

*Sejam:*

- $H = \{ h_1, h_2, ..., h_m\}$ *um conjunto de hosts.*
- $V = \{ v_1, v_2, ..., v_n\}$ *um conjunto de VMs a serem alocadas em H.*

*Define-se* $V' \subseteq V$ *um conjunto de VMs alocadas em H.*

## 3.1 Determinantes de variabilidade do problema

Os determinantes da variabilidade do problema de alocação de VMs são:

- O conjunto de *hosts*.
- O modo de chegada das VMs.
- O número de dimensões (tipos de recursos computacionais) consideradas nas máquinas.

A Figura 8 classifica as possíveis variações para o problema de alocação de VMs.

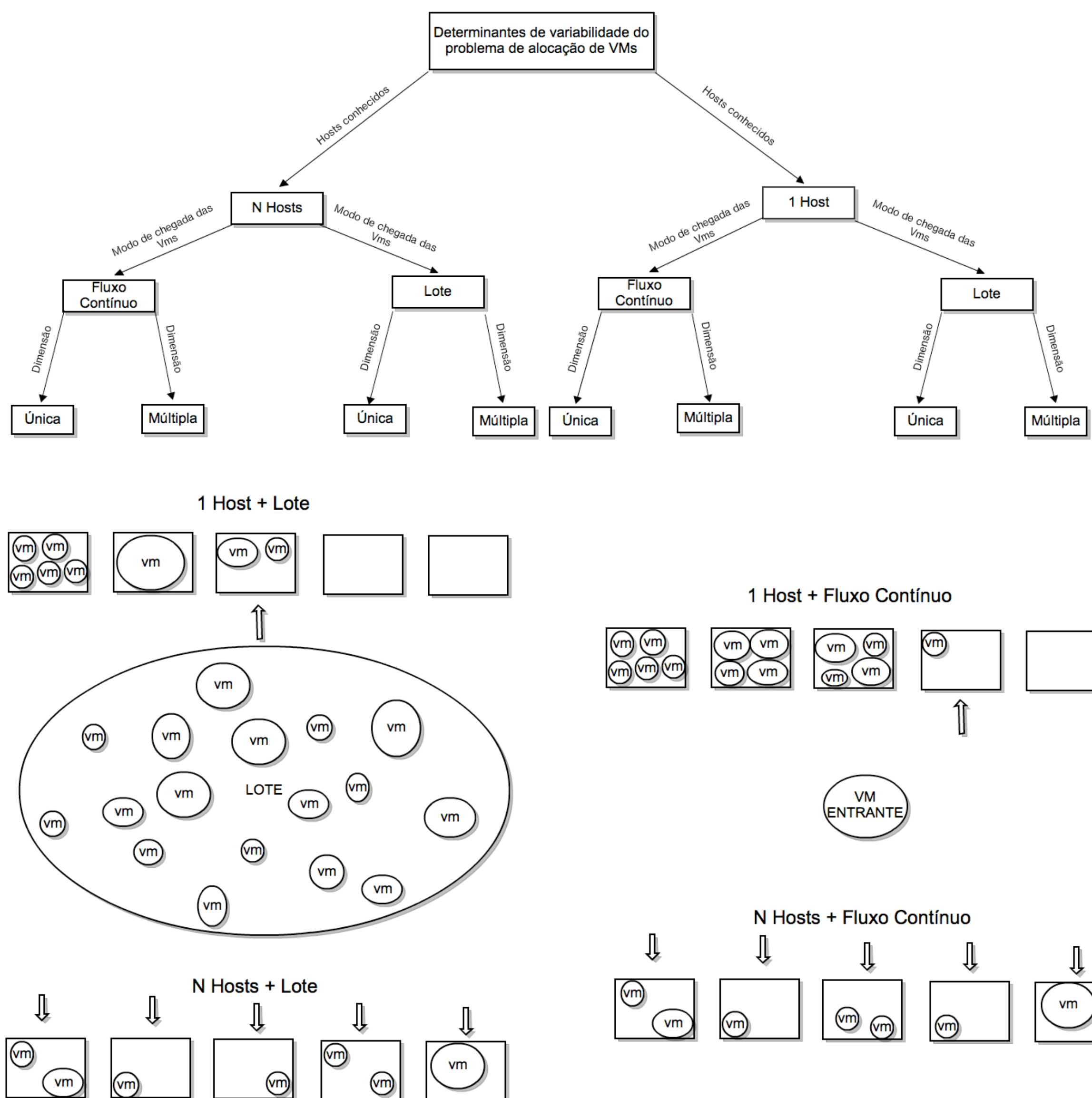



**Figura 8**. Variabilidade do problema de alocação de VMs.

### 3.1.1 O conjunto de *hosts*

A quantidade de *hosts* considerada no tempo de alocação pode ser 1 ou N, ambos modelos são exemplificados com detalhes abaixo:

- No modelo *1 Host,* apenas um host por vez é considerado pelo algoritmo. A abordagem utilizada para o problema é colocar o maior número possível de VMs no host considerado até que não seja mais possível colocar nenhuma VM, então passamos

ao próximo *Host*. O modelo de alocação utilizando *1 host* geralmente é usado para economizar energia, pois é ligada uma máquina nova apenas quando o número de VMs na máquina anterior já foi maximizado. A Figura 8 exemplifica um host sendo avaliado por vez (indicado pela flecha) no modelo "1 host + lote" e "1 host + fluxo contínuo".

- No modelo *N Hosts* as VMs são distribuídas em um conjunto de hosts de forma a maximizar o número de VMs alocadas de maneira balanceada, permitindo um número maior de VMs alocadas devido ao uso da informação sobre os recursos disponíveis de diversos servidores. Também é possível minimizar a quantidade migrações, pois em um estado balanceado, é muito menos provável que uma VM não tenha recursos físicos disponíveis de acordo com sua necessidade de elasticidade. Este tipo de abordagem geralmente é utilizado em *QoS* no entanto achamos que esta abordagem também pode ser usada para economia de energia e deixamos este último aspecto para trabalhos futuros. A Figura 8 exemplifica um conjunto de hosts (indicados pelas flechas) sendo considerados como candidatos para a alocação de VMs no modelo "*N Hosts* + lote" e "*N Hosts* + fluxo contínuo".

Neste trabalho consideramos a abordagem do problema de alocação de VMs com a abordagem *N Hosts* mais realista, pois é possível usar o conhecimento do conjunto de hosts para aumentar a qualidade de serviço para o usuário e maximizar o lucro do *cloud provider, devido a* :

- Facilitar a elasticidade de recursos das VMs na própria máquina minimizando a quantidade migrações, pois a alocação de VMs é balanceada.
- Maximizar o número de VMs que podem ser alocadas no conjunto total de *hosts*, de acordo com uma função ou critério.

### 3.1.2 Modo de chegada de máquinas virtuais

As VMs podem chegar em fluxo contínuo ou em lotes. Quando chegam em fluxo contínuo são tratadas imediatamente à medida que chegam. No caso da chegada em lotes, é necessário esperar para que um lote seja formado através do acúmulo de VMs em um determinado tempo.

Uma vantagem de ter VMs chegando em lote é a possibilidade de usar o conhecimento prévio do conjunto das VMs para alterar a ordem de alocação, privilegiando VMs maiores ou menores de acordo com uma função que maximiza a quantidade de alocações. Esta abordagem pode ser muito útil para o problema do *bin packing*, principalmente com o objetivo de economia de energia.

Este trabalho considerou a chegada de VMs em fluxo contínuo, uma abordagem mais realista. Na computação em nuvem, quando uma alocação de máquina virtual é solicitada, é esperado que ela seja colocada *on-line* o quanto antes para atender o cliente, ou seja, a espera da formação de lotes de VMs interfere negativamente no tempo de disponibilidade da VM. Além disso, a formação de lotes com ordenação não respeita a ordem de chegada das requisições de VMs, sendo injusta com clientes que solicitaram as VMs por primeiro.

Existem casos que não é possível tratar instantemente todas requisições de VMs, então admitimos que uma abordagem realista para este tipo de situação é formar lotes de VMs que sejam organizados na forma de fila, onde a primeira VM a chegar será a primeira VM a ser tratada e nenhuma ordenação de VMs é feita, no entanto a informação sobre o conjunto total de recursos que o lote de VMs irá consumir é conhecida e pode ser utilizada para melhorar o processo.

Apesar do assunto de paralelismo nas requisições de alocação de VMs ser importante, deixamos esta abordagem para trabalhos futuros pois o paralelismo de requisições poderia fazer com que surgisse uma situação de concorrência nos servidores e desvirtuaria do foco deste trabalho, que é testar heurísticas para maximizar o número de VMs alocadas em um datacenter.

### 3.1.3 Dimensões das máquinas

Quanto às dimensões das máquinas, o problema de alocação de VMs pode considerar apenas uma dimensão (por exemplo, processador) ou múltiplas dimensões (por exemplo, processador, memória, disco e banda).

Considerar apenas uma dimensão tem a vantagem das soluções para o problema serem muito mais simples, com um *overhead* menor, e a desvantagem de ser uma abordagem pouco precisa que só leva em conta uma pequena parte de todas variáveis.

Neste trabalho, consideramos várias dimensões por ser um modelo mais realista, já que uma VM e um *host* possui quatro dimensões básicas: processador, memória, disco e banda. Todas estas dimensões tem o mesmo "peso", pois qualquer uma delas pode barrar a possibilidade de uma VM ser alocada em uma máquina física, sendo assim as dimensões podem ser usadas para fazer cálculos matemáticos para maximizar a alocação de VMs sem necessidade de diferenciação do tipo da dimensão.

## 3.2 A escolha do problema

Estudando o problema de alocação de VMs, decidimos adotar uma abordagem realista, tal que:

- O fluxo de VMs é contínuo, sendo respeitada sua ordem de chegada. Escolhemos esta abordagem, porque queremos atender o cliente que solicitou a VM o quanto antes, não impondo atrasos para formar lotes, além disso, queremos ser justos respeitando a ordem de chegada das VMs.
- Caso não seja possível atender uma VM assim que ela chegou, assumimos a formação de uma fila de VMs em que a VM que chegou primeiro será tratada antes.
- As máquinas possuem quatro dimensões: processador, memória, disco e banda. Em um ambiente real, elas também possuem estas mesmas dimensões. Para fins de simplificação usaremos a notação (P, R, D, B) para referenciar essas dimensões, respectivamente. (A escolha do "R" para representar a dimensão memória refere-se à memória do tipo RAM; o uso da letra "M" geraria ambiguidade, pois será usado para representar um conjunto de máquinas.) .
- Escolhemos o modelo de *N Hosts*, pois conhecer o conjunto de hosts é o elemento chave para maximizar a quantidade de VMs alocadas.

A Figura 9 ilustra a abordagem escolhida para o problema, de forma simplificada, com *hosts* homogêneos e usando apenas duas dimensões: processador e memória. O host 1 possui 70% da capacidade de processador ocupada e 20% da capacidade de memória ocupada. O host 2 possui 50% da capacidade de processador ocupada e 40% da capacidade de memória ocupada. A VM entrante necessita de uma quantidade equivalente a 20% da capacidade de processador e 30% da capacidade de memória de um dos *hosts*. A questão a ser respondida é em qual *host* deve ser alocada a VM entrante.

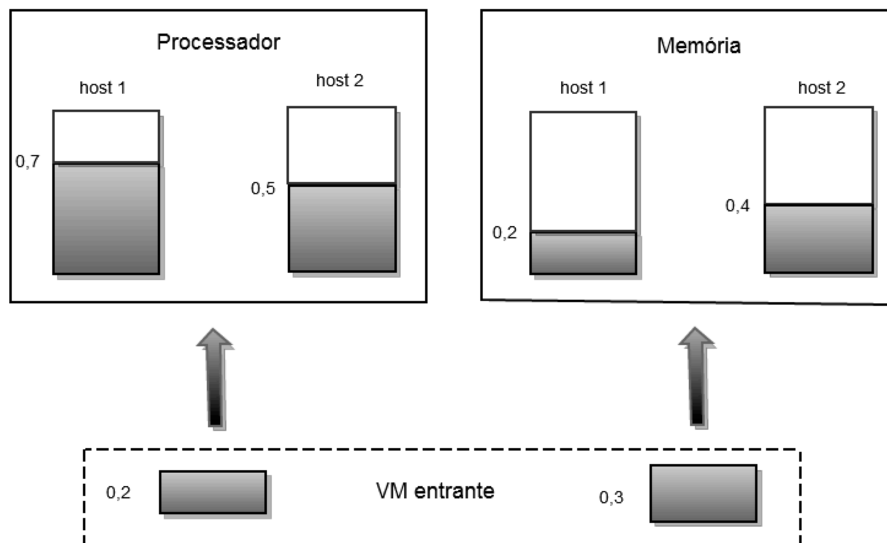



**Figura 9**. Exemplo de alocação de VM.

## 3.3 Técnicas de alocação de máquinas virtuais

Como a alocação de máquinas virtuais é um problema combinatório NP-*hard* (pois equivale ao problema da mochila), os algoritmos estudados e experimentados neste trabalho procuram resolver este problema com soluções mais realistas, isto é, baseadas em heurísticas que propiciam uma solução aproximada.

### 3.3.1 Classificação de Anjana Shankar

Anjana Shankar [Shankar, 2010] classifica as técnicas de alocação de máquinas virtuais de acordo com a forma de resolver o problema de alocação de alocação de VMs. A classificação é representada pela Figura 10.

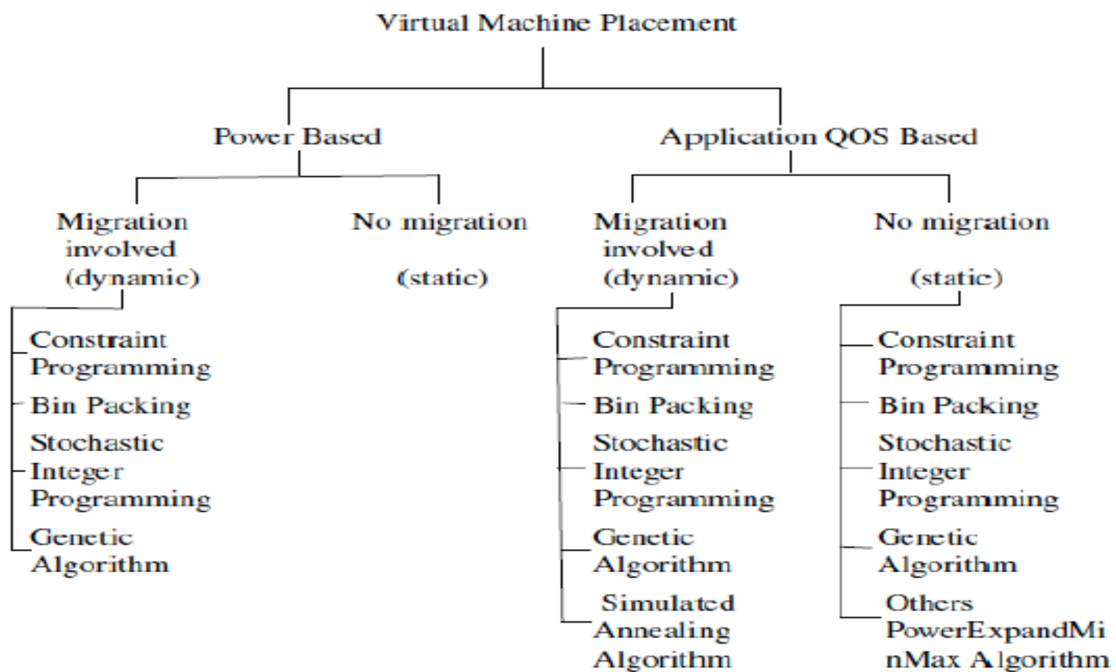



**Figura 10**. Classificação de Anjana Shankar [Shankar, 2010].

Para o autor, as técnicas de alocação de máquinas virtuais procuram resolver dois principais tipos de problemas: economia de energia (*power based*) e qualidade de serviço (QOS). Os principais tipos algoritmos por ele enumerados são:

- *Constraint Programming* – Dado um conjunto de variáveis e um conjunto de restrições com relação à satisfação de todas variáveis, tenta-se achar uma solução ótima que satisfaça todas as restrições.
- *Bin Packing* – Usando a analogia de objetos e recipientes, o problema do *bin packing* procura colocar objetos de diferentes volumes em cada recipiente, até que finalmente todos objetos sejam colocados e o número de recipientes usados seja minimizado. Este problema usa a analogia de peso (*weight*) para verificar se o recipiente poderá conter o objeto e de lucro (*profit*), caso o objeto seja colocado efetivamente no recipiente.
- *Stochastic Integer Programming* – Este tipo de programação tenta otimizar a solução de problemas que possuem incertezas e parâmetros desconhecidos através de probabilidades, quando estas probabilidades podem ser estimadas.
- *Genetic Algorithms* – O problema é representado por um modelo genético e uma função de uma solução escolhida. Usando conceitos da biologia evolucionária, tais

como herança, mutação, seleção e re-combinação, são achadas outras soluções que representam uma evolução genética da solução do problema.

- *Simulated Anneling Algorithm* – É uma técnica baseada na termodinâmica. O *anneling* é um termo herdado do processo de aquecer metais a 1100 graus Celsius e resfriando-os de maneira lenta e controlada em seguida, até que o metal se solidifique. O motivo do resfriamento ser controlado é para que os átomos ganhem energia para se movimentarem livremente, reduzindo os defeitos do material. Analogamente, esta técnica pode ser usada na computação para resolver diversos problemas. Uma solução qualquer pode ser escolhida para resolver um problema em função de uma variável T (temperatura). Incialmente, o valor de T é elevado e testa-se soluções com diferentes valores para T (temperaturas), diminuindo-se o valor de T gradativamente a cada iteração. Os resultados são avaliados encontrando-se o valor de T ideal.

### 3.3.2 Técnica baseada no Problema da Mochila

Quando estávamos estudando a classificação de Anjana Shankar e as heurísticas de *bin packing*, notamos que esta era a técnica mais citada na literatura. Por esse motivo, resolvemos aprofundar nossos estudos para que fosse possível fazer nossos experimentos, utilizando a técnica de *bin packing*.

Aconteceu que, quando estávamos analisando os determinantes do problema de alocação de VMs, achamos que a abordagem mais realista seria o modelo *N Hosts* multidimensionais, com VMs (também multidimensionais) chegando em fluxo contínuo. A técnica do *bin packing* parecia ser uma boa escolha, no entanto seu uso é mais adequado quando se deseja alocar um lote de VMs. Entretanto queremos alocar apenas uma VM. Além disso, o *bin packing* é adequado *somente* para abordagem de *1 host*, enquanto nosso objetivo é considerar a alocação da VM entrante em *N hosts*.

Por outro lado, o problema da mochila consiste em colocar o maior número possível de um conjunto de objetos (de tipos iguais ou diferentes) em uma mochila. Aprofundando um pouco os estudos, achamos três especializações do problema da mochila:

- A primeira delas considerava todos os objetos como sendo do mesmo tipo, mas com volumes diferentes. Esta especialização é chamada de *0-1 knapsack problem*.

- A segunda especialização considerava diversas mochilas e todas eram avaliadas na hora de escolher em qual mochila o objeto seria colocado. O nome desta especialização é *multiple knapsack problem.*

- A terceira especialização considerava que cada objeto e cada mochila poderia ter mais de uma dimensão. Esta especialização é chamada de *m-dimensional knapsack problem.*

A combinação dessas três especializações satisfazia perfeitamente o modelo *N Hosts* proposto na figura 8. Continuando nossos estudos vimos que casos de *multiple m-dimensional 0-1 knpasacks* também eram citados na literatura [Mohammadi et al., 2011]. Formalizamos a adaptação do problema de *multiple m-dimensional 0-1 knpasacks para o problema de alocação de VMs*, isto é, para maximizar a quantidade de VMs alocadas, conforme segue.

**Formalização do Problema de Alocação de VMs**

*Sejam:*

- *$H = \{ h_1, h_2, ..., h_m\}$ um conjunto de hosts.*
- *$T_j$ o instante de chegada da $VM_j$*
- *$V = \{ v_1, v_2, ..., v_n\}$ uma sequência de VMs, tal que, se $r < s$, então $T_r < T_s$*
- *$D =\{d_1, d_2, ..., d_g\}$ as dimensões das máquinas.*
- $x_{i,j} \in \{0, 1\}$, i = 1, ..., m, j = 1, ..., n
    - $x_{i,j} = 1 \Leftrightarrow$ *$v_j$ está alocada em $h_i$*
    - $x_{i,j} = 1 \Rightarrow x_{k,j} = 0 \ \forall \ k \neq i$
- *$w_{j,k}$ o valor da dimensão k para a VM j.*
- *$c_{i,k}$ a capacidade que o host i tem para a dimensão k.*
- $L_{i,k}$ *($T_j$) a parte livre da capacidade da dimensão k no host i no instante de chegada de VMj*

*Maximizar:*

$$\sum_{i=1}^{m} \sum_{j=1}^{n} x_{i,j}$$

*Sujeito a:*

*(i)* $\sum_{i=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{g} x_{i,j} . w_{j,k} \le c_{i,j}$

*(ii)* $v_s$ *alocada* $\wedge \; \exists i, \; 1 \le i \le m \mid L_{i,k}(T_r) \ge w_{r,k}, \mathrm{T_r} < \mathrm{T_s} \quad \forall_k \quad k = 1, ..., g \Rightarrow v_r$ *alocada*

## 3.4 Heurísticas tradicionais para alocação de máquinas virtuais

De todas as técnicas estudadas, as heurísticas do problema do *bin packing* foram as mais citadas na literatura. Estas heurísticas são, na maioria, variantes do *First Fit*. Não achamos trabalhos de alocação de VMs, usando heurísticas próprias para o problema do *multiple m-dimensional 0-1 knapsacks*. Sendo assim, resolvemos aplicar as mesmas heurísticas usadas no problema do *bin-packing*, adaptando-as para o problema do *multiple m-dimensional 0-1 knapsacks*.

A heurística mais tradicional para o problema de alocação de VMs é o *first fit,* que segue o modelo do *bin packing* (classificamos como *1 host* + *fluxo contínuo*). Se a ordem de chegada das VMs for ignorada é possível formar lotes de VMs ordenados de acordo com algum critério escolhido, dando origem a variantes do *first fit.* Estas variantes podem maximizar o número de VMs alocadas no menor número possível de hosts de forma mais eficiente. Estas variantes levam o nome de *first fit decrasing*, *volume* e *dot product* (são do tipo *1 host* + *lote*) . Em nosso trabalho adaptamos as heurísticas volume e dot product para o problema do *multiple m-dimensional 0-1 knapsacks* e também as adaptamos para fluxo contínuo de VMs (classificamos como *N Hosts* + *fluxo contínuo*). Adicionalmente criamos duas heurísticas novas, a *best dimension* ( *N Hosts* + *fluxo contínuo*) e *osmosis* (para *N Hosts* + *lote*, mas respeitando a ordem de chegada das VMs, ou seja, não tem ordenação de VMs).

### 3.4.1 *First Fit*

O *First Fit* é a heurística clássica mais utilizada no problema de alocação de VMs. Consiste em alocar uma VM no primeiro *host* que atenda todos seus requisitos dimensionais; para isso, não faz ordenação de elementos.

### 3.4.2 *First Fit Decreasing*

Como no problema do *bin packing,* as VMs chegam em lote e o objetivo é usar o menor número possível de hosts, geralmente é usada uma variante da heurística *First Fit*, chamada de *First Fit Decreasing* (FFD). Esta variante tenta alocar as VMs em ordem

decrescente de tamanho no primeiro *host* que atenda todos os requisitos da VM, isto é, o primeiro *host* que comporte a VM com respeito às quatro dimensões.

Esta heurística também pode ser aplicada no problema *multi m-dimensional 0-1 knapsacks*, apesar do fluxo de VMs ser contínuo. A heurística pode ser utilizada nos hosts, ordenando-os em ordem decrescente de acordo com uma função que calcule seu tamanho.

### 3.4.3 Volume

O Volume é uma variante geométrica do *First Fit Decreasing*. A adaptação desta heurística para o modelo *N Hosts* (problema da mochila) está justamente em fazer o processo de ordenação e cálculo de volume nos hosts ao invés de fazer o processo no lote de VMs. Os *hosts* são ordenados decrescentemente de acordo com seus volumes livres. Uma VM entrante é alocada no primeiro *host* que a comporte de acordo com a sequência ordenada. A heurística Volume, juntamente com a heurística *First Fit*, são as únicas que não tiram proveito do conhecimento das dimensões da VM entrante.

Vamos supor uma VM entrante com as seguintes dimensões:

| VM | P | R | D | B |
|---|---|---|---|---|
| 1 | 500 | 20 | 200 | 100 |

O volume de cada *host* é calculado como o produto das quatro dimensões. A ordenação dos hosts é feita pelo volume em ordem descrescente.

| Hosts | P | R | D | B | Volume | Ordem |
|---|---|---|---|---|---|---|
| 1 | 1000 | 90 | 800 | 700 | 50400000000 | 1 |
| 2 | 500 | 80 | 400 | 500 | 8000000000 | 3 |
| 3 | 700 | 200 | 300 | 800 | 33600000000 | 2 |

Caso exista alguma restrição e a VM não possa ser alocada no primeiro host ordenado, é feita uma tentiva de alocação no segundo host e assim por diante.

### 3.4.4 *Dot Product*

Esta variante geométrica tradicional do modelo *1 Host* ( *bin packing*) faz o produto de cada dimensão do host aberto com as respectivas dimensões de um lote de VMs, gerando um

produto para cada dimensão de todas as VMs do lote. Logo após, os produtos de cada VM por *Host* são somados e as VMs são ordenadas decrescentemente de acordo com esta somatória. Em nossa adaptação da heurística *Dot Product* para o modelo da mochila (*N Hosts+ fluxo contínuo*), cada dimensão da VM entrante é multiplicada pela parte livre da dimensão correspondente de cada *host*, gerando um produto para cada dimensão de todos os *hosts*. Estes produtos são somados para cada *host*, obtendo-se um valor por *host*. Assim, os *hosts* são ordenados decrescentemente de acordo com este valor.

Sendo a VM entrante em fluxo contínuo:

| VMs | P | R | D | B |
|---|---|---|---|---|
| 1 | 500 | 20 | 200 | 100 |

Sendo o conjunto de hosts:

| Hosts | P | R | D | B |
|---|---|---|---|---|
| 1 | 1000 | 90 | 800 | 700 |
| 2 | 500 | 80 | 400 | 500 |
| 3 | 700 | 200 | 300 | 800 |

É feito o produto VM por Host para cada parte livre da dimensão, em seguida os valores do produto são somados e os hosts ordenados decrescentemente:

| Host | P | R | D | B | Soma | Ordem |
|---|---|---|---|---|---|---|
| 1 | 500000 | 1800 | 160000 | 70000 | 731800 | 1 |
| 2 | 250000 | 1600 | 80000 | 50000 | 381600 | 3 |
| 3 | 350000 | 4000 | 60000 | 80000 | 494000 | 2 |

Uma VM entrante é alocada no primeiro *host* na ordem que a comporte (host 1), caso não seja possível, é feita uma tentativa no segundo host ordenado (host 3) e assim por diante.

## *3.5* Heurísticas propostas para alocação de máquinas virtuais

Durante este trabalho, após fazer o estudo no campo de máquinas virtuais e computação em nuvem, intuitivamente elaboramos duas novas heurísticas, descritas a seguir. Apesar de que à época, ainda não tínhamos feito a delimitação completa do problema, já tínhamos consciência de que deveríamos fazer uma abordagem mais realista, considerando a ordem de chegada das VMs sem alteração e o modelo *N Hosts*.

Delimitamos um problema como sendo equivalente ao *multiple m-dimensional 0-1 knapsacks,* com a chegada das VMs em fluxo contínuo (*N hosts + fluxo contínuo*) e o

chamamos de *best dimension*, e outro com a chegada em lotes (*N hosts* + *lote*) e o chamamos de *osmosis*, sendo que em ambas heurísticas respeitamos a ordem de chegada das VMs.

### 3.5.1 *Best Dimension*

Nesta heurística, da mesma forma que a versão adaptada do *dot product* para *N Hosts*, cada dimensão da VM que está sendo alocada é multiplicada pela parte livre da dimensão correspondente de cada *host*, gerando um valor para cada dimensão de cada *host*. Escolhe-se a dimensão que possui o maior valor gerado pelo produto. Os *hosts* são ordenados decrescentemente de acordo com o valor escolhido para cada um.

Sendo a VM entrante:

| VMs | P | R | D | B |
|---|---|---|---|---|
| 1 | 500 | 20 | 200 | 100 |

Tem-se o seguinte conjunto de Hosts:

| Hosts | P | R | D | B |
|---|---|---|---|---|
| 1 | 1000 | 90 | 800 | 700 |
| 2 | 500 | 80 | 400 | 500 |
| 3 | 700 | 200 | 300 | 800 |

Calcula-se o produto da dimensão da VM pela parte livre da dimensão de cada host, em seguida escolhe-se o maior valor para cada host e ordena-se os hosts decrescentemente.

| Host | P | M | D | B | Maior | Ordem |
|---|---|---|---|---|---|---|
| 1 | 500000 | 1800 | 160000 | 70000 | 500000 | 1 |
| 2 | 250000 | 1600 | 80000 | 50000 | 250000 | 3 |
| 3 | 350000 | 4000 | 60000 | 80000 | 350000 | 2 |

Por fim é feita a tentativa de alocar a VM entrante no primeiro host da ordenado (host 1), caso não seja possível é feita a tentativa no segundo host da ordem (host 2), sucessivamente.

### 3.5.2 *Osmosis*

A heurística *Osmosis* procura alocar uma VM entrante de forma a privilegiar o *host* que tem mais oferta relativa do recurso mais escasso globalmente, isto é, considerando todo o conjunto de *hosts.* Assim, para cada VM entrante, são executados os seguintes passos:

1. Aproveitamos o conhecimento do lote de VMs e do conjunto de *hosts* para calcular taxas que serão usadas como pesos. Para isso, somamos separadamente cada dimensão do lote de VMs, e em seguida, fazemos o mesmo procedimento nos *hosts*, somando as capacidades livres de cada dimensão.

| VMs | P | R | D | B |
|---|---|---|---|---|
| 1 | 500 | 20 | 200 | 100 |
| 2 | 400 | 80 | 100 | 50 |
| 3 | 200 | 200 | 400 | 220 |
| 4 | 400 | 10 | 900 | 20 |
| 5 | 450 | 70 | 300 | 130 |
| 6 | 100 | 45 | 450 | 150 |
| 7 | 50 | 800 | 500 | 300 |
| 8 | 300 | 1000 | 1000 | 100 |
| soma | 2400 | 2225 | 3850 | 1070 |

| Hosts | P | R | D | B |
|---|---|---|---|---|
| 1 | 1000 | 90 | 800 | 700 |
| 2 | 500 | 80 | 400 | 500 |
| 3 | 700 | 200 | 300 | 800 |
| soma | 2200 | 370 | 1500 | 2000 |

2. Depois, para cada dimensão, dividimos a soma obtida para o lote de VMs pela respectiva soma obtida para o conjunto de hosts, encontrando o *peso da dimensão*. Em outras palavras, quanto mais escasso é um recurso, maior o seu peso.

| Peso | 1,090909 | 6,013514 | 2,566667 | 0,535 |
|---|---|---|---|---|

3. Considerando a VM que chegou por primeiro como a VM entrante, calculamos a *relação de oferta* de cada dimensão de cada *host,* dividindo a capacidade livre de cada dimensão de cada host pela correspondente dimensão da VM entrante.

| Host | P | R | D | B |
|---|---|---|---|---|
| 1 | 2 | 4,5 | 4 | 7 |
| 2 | 1 | 4 | 2 | 5 |
| 3 | 1,4 | 10 | 1,5 | 8 |

4. Em seguida, para cada dimensão de cada host, achamos a *relação de oferta ponderada,* multiplicando a respectiva *relação de oferta* pelo *peso da dimensão*. Finalmente, para cada host, somamos as suas dimensões de ofertas ponderadas e ordenamos os *hosts* crescentemente de acordo com o resultado dessas somas.

| | | taxas | poderadas | VM1 | | |
|---|---|---|---|---|---|---|
| Host | P | R | D | B | Soma | Ordem |
| 1 | 2,181818 | 27,06081 | 10,26667 | 3,745 | 43,2543 | 2 |
| 2 | 1,090909 | 24,05405 | 5,133333 | 2,675 | 32,9533 | 1 |
| 3 | 1,527273 | 60,13514 | 3,85 | 4,28 | 69,79241 | 3 |

5. Após a ordenação dos *hosts* é feita uma tentativa de alocar a VM entrante em cada *host*, seguindo a ordem, na esperança que um deles a comporte.

## 3.6 Trabalhos relacionados

A maior parte dos trabalhos que encontramos considerava o problema de alocação de VMs como uma variante do problema do *bin packing*, em outras palavras, consideravam a como o modelo que classificamos como *1 host + lote*. Os trabalhos mais relevantes que estudamos foram os seguintes:

a) *Heuristics for Vector Bin Packing* [Panigrahy et al., 2011] É um estudo da Microsoft sobre as variantes das heuristicas *First Fit Decreasing* (FFD) para o problema do *bin packing,* objetivando minimizar o conjunto de *hosts* usados. Para isso, um *host* é aberto por vez até ser saturado de VMs. Através da simulação, chegaram a conclusão que as variantes geométricas do FFD são muito eficientes, e que considerar várias dimensões é melhor do que apenas considerar uma. Os autores fazem ainda a classificação das variantes do FFD como *Item Centric e Bin Centric.* O modelo *Item Centric* é caracterizado por ordenar as VMs utilizando uma função qualquer decrescentemente, e seguindo esta ordem, aloca-as no primeiro *host* disponível que atenda todos requisitos

dimensionais da VM. O modelo *Bin Centric* tem o diferencial que ordena as VMs através de uma função que faz um cálculo adicional com a capacidade livre do *host* aberto.

b) *Multi-Objective Virtual Machine Placement in Virtualized Datacenters Enviroments* [Xu and Fortes, 2010] Objetivando a economia de energia, propuseram o uso de algoritmos genéticos para minimizar o consumo energético e a dissipação de calor das máquinas. Dados de consumo de energia e dissipação de calor foram obtidos de um ambiente real (*IBM BladeCenter).* Através da simulação, testaram performance, escalabilidade e robustez de diversas heurísticas. Os resultados mostraram que usar funções que considerem vários objetivos ao invés de um isolado minimizam o número de servidores usados e custos adicionais com migrações.

c) *Techniques for Virtual Machine Placement in Clouds* [Bonde, 2010] Tem como principal objetivo aplicar a técnica do *bin packing* minimizando o número de *hosts* utilizados e avaliando os resultados através da simulação. Em seus testes, usa o conceito de heterogeneidade apenas nas VMs. Avalia as heurísticas: *First Fit* com várias dimensões, *First Fit* com uma dimensão, *Volume* e *Dot Product*. Nos testes realizados, as heurísticas geométricas se sairam melhor (*Volume* e *Dot Product*), com pequena vantagem ao *Volume*.

Quando percebemos que o modelo bin packing (*1 host*) não era uma abordagem realista, começamos a estudar o Problema da Mochila. Os trabalhos mais relevantes estudados foram:

a) *Heuristics for Multiple Knapsack Problem* [Fidanova, 2005] Este trabalho estuda de forma generalizada o problema de Multiple Knapsacks (MKP) utilizando a técnica de Otimização da Colônia de Formigas (Ant Colony Optimization - ACO). Neste trabalho é que encontramos pela primeira vez a definição de uma variante do Problema da Mochila chamada de *Multiple M-dimensional Knapsacks Problem* e foi exatamente este o tipo de delimitação do problema que queríamos aplicar a alocação de VMs. Apesar de o artigo ser muito útil para delimitarmos nossos problemas as Heurísticas ACO não se sairam melhor nas simulações do que heurísticas estáticas.

b) *Multiple Mutidimensional Knapsack Problem and its Applications in Cognitive Radio Networks* [Song et al., 2008] Este trabalho explora o Problema da Mochila fazendo uma anologia com redes cognitivas de radio, propondo uma solução ideal e uma nova heurística. Apesar do trabalho citar o problema de *Multiple Multidimensinal Knapsack Problem* como uma nova variante, encontramos indícios de que o problema já tinha sido descoberto anteriormente [Fidanova, 2005]. Aproveitamos a definição do problema que estava formalizada neste artigo para adaptar e formalizar o problema de alocação de VMs.

c) *Server-Storage Virtualization: Integration and Load Balacing in Data* Centers [Singh et al., 2008] O trabalho propõe um novo algoritmo de balanceamento de carga chamado de *VectorDot*. Este algoritmo tenta eliminar os gargalos de recursos dos servidores, switches e nós de armazenamento em datacenter que utiliza *Storage Area Network* (SAN). A performance do algoritmo é avaliada usando dados sintéticos e também com dados reais. Os autores afirmam que este é o primeiro algoritmo a resolver o problema do *multiple m-dimensional knapsacks* considerando restrições multidimensionais e hierárquicas.

d) *A Novel Virtual Machine Placement in Cloud* Computing [Mohammadi et al., 2011] É uma abordagem de alocação de máquinas virtuais que tenta minimizar o tempo de transferência de dados, melhorando o I/O de disco e posicionando as VMs de forma ótima, isto é, perto de seus respectivos *storages*.

Um estudo chamado *Virtual Machine Placement in Computing Clouds* [Shankar, 2010] foi o único artigo que encontramos alguma classificação das técnicas usadas para construção de algoritmos que tratem do problema de alocação de VMs. Embora esta foi a classificação mais completa que achamos, notamos que ela é incompleta e provavelmente qualquer classificação será incompleta com o passar do tempo devido a grande quantidade de trabalhos nesta área. Um exemplo pode ser dado pelo trabalho *A Simulated Anneling Algorithm for Energy Efficient Virtual Machine Placement* [Wu et al., 2012], que usa um algoritmo com a heurística de *anneling* para alocação de VMs, maximizando a economia de energia.

## 3.7 Conclusão

O trabalho de revisão da literatura e a reflexão sobre o problema a ser resolvido levaram às seguintes contribuições:

a) Definição de uma classificação de abordagens para o problema: a classificação permitiu diferenciar os diversos problemas de alocação de VMs e pode servir como guia no estudo de novas soluções.

b) Definição formal do problema segundo uma abordagem realista: o formalismo definido pode ser usado na verificação de novas propostas de solução para o problema.

c) Proposta de duas novas heurísticas de alocação de máquina virtual: as novas heurísticas podem ser verificadas para o problema de alocação de VM, como também podem servir de inspiração para a solução de outros problemas.

# 4 Método de Avalição de Desempenho

Neste Capítulo serão apresentados os resultados obtidos com os experimentos realizados neste trabalho.

Este capítulo é divido em quatro partes: a primeira parte discute a metodologia escolhida, a segunda aborda a delimitação do escopo dos experimentos, a terceira descreve o simulador utilizado, a quarta mostra os resultados dos experimentos e a quinta faz considerações finais sobre os resultados.

O método de avaliação de desempenho dos algoritmos de alocação de VMs, definido para os experimentos, utilizou os conceitos de heterogeneidade, densidade e taxa de alocação, no contexto de computação em nuvem.

## 4.1 Considerações sobre a heterogeneidade

A heterogeneidade dos *hosts* em um *datacenter* aumenta progressivamente com o passar dos anos. Gordon Earle Moore, em meados da década de 60, já havia previsto que o número de transistores em um processador iria dobrar a cada 18 meses, máxima que ficou conhecida com "A Lei de Moore". Esta lei é muito usada, até os dias de hoje, para se referir a rápida natureza evolutiva da tecnologia dos componentes eletrônicos dos computadores [Moore, 1965], mesmo que sua equação inicial não seja uma verdade absoluta. Por dedução, podemos facilmente concluir que a natureza de um *datacenter* é essencialmente heterogênea.

Para exemplificar como o avanço da tecnologia afeta um *datacenter*, podemos supor que uma empresa que atua com venda de eletrônicos apenas de forma *online*, e para esta finalidade ela construiu um pequeno *datacenter*. Os itens de 1 a 5 ilustram uma situação típica de previsão de crescimento que uma empresa pode ter em 2 anos (2013 a 2015):

1. (Ano: 2013) Um primeiro lote de 10 máquinas é comprado para suprir a necessidade da empresa.
2. (Ano: 2014) Os acessos nos sites da empresa dobram em relação ao ano anterior.

3. (Ano: 2014) Compram-se mais 9 máquinas para suprir a nova necessidade da empresa. Estas máquinas possuem algumas melhorias em relação às anteriores devido ao processo natural de refinação da mesma arquitetura de *hardware* do ano anterior (ex: aumento do *clock* do processador, devido a *batches* mais estáveis na linha de produção, barateamento do *gigabyte* nos discos rígidos e nas memórias RAM, barateamento do *link* de acesso a *Internet*). Assim, em relação a 2013, os *Hosts,* adquiridos em 2014, tiveram um aumento de 10%, 15%, 20% e 25% para as dimensões (P, R, D, B), respectivamente.
4. (Ano: 2015) Os acessos nos sites da empresa dobram em relação ao ano anterior.
5. (Ano: 2015) Compram-se mais 6 máquinas para atender a nova necessidade da empresa. No ano de 2015, houve um salto revolucionário na tecnologia. (Ex: Aumento dos barramentos internos na placa mãe, com uma nova arquitetura, discos de estados sólidos diminuíram o tempo de acesso a dados críticos e ao mesmo tempo os discos rígidos aumentaram muito suas capacidades, e houve um grande barateamento dos *links* de *Internet* devido a novos cabos de fibra ótica). Assim, em relação a 2014, os *Hosts* adquiridos em 2015 tiveram um aumento de 40%, 40%, 40% e 40% em suas dimensões (P, R, D, B), respectivamente.

No exemplo acima, a partir de uma projeção temporal maior, haveria um aumento ainda maior na heterogeneidade desse *datacenter.*

No modelo de Computação em Nuvem não são apenas os *hosts* que podem ser heterogêneos, as VMs também estão sujeitas à heterogeneidade devido a diversificada natureza das aplicações que são executadas no topo dos sistemas operacionais virtualizados. Podemos classificar a natureza destas aplicações nos seguintes grupos:

- Aplicações *CPU-bound*: Utilizam um grande percentual do processamento da CPU (P) e/ou memória Ram (R).
- Aplicações *I/O-bound*: São aquelas que usam intensamente disco (D) e/ou o *link* de dados (B).
- Aplicações híbridas: Fazem uso intenso de todos recursos (P, R, D, B) de maneira alternada ou simultânea.

Como precisávamos avaliar a heterogeneidade nos experimentos, criamos uma fórmula para quantificá-la através do conceito de amplitude. Sendo um conjunto de

máquinas (*hosts* ou VMs) que seja heterogêneo em relação às dimensões (P, R, D, B), define-se como a amplitude de uma dimensão,  a variação percentual entre todos os valores dessa dimensão em relação ao correspondente valor médio. A Figura 11 ilustra o conceito de amplitudes em uma máquina e, logo em seguida, ele é formalizado.

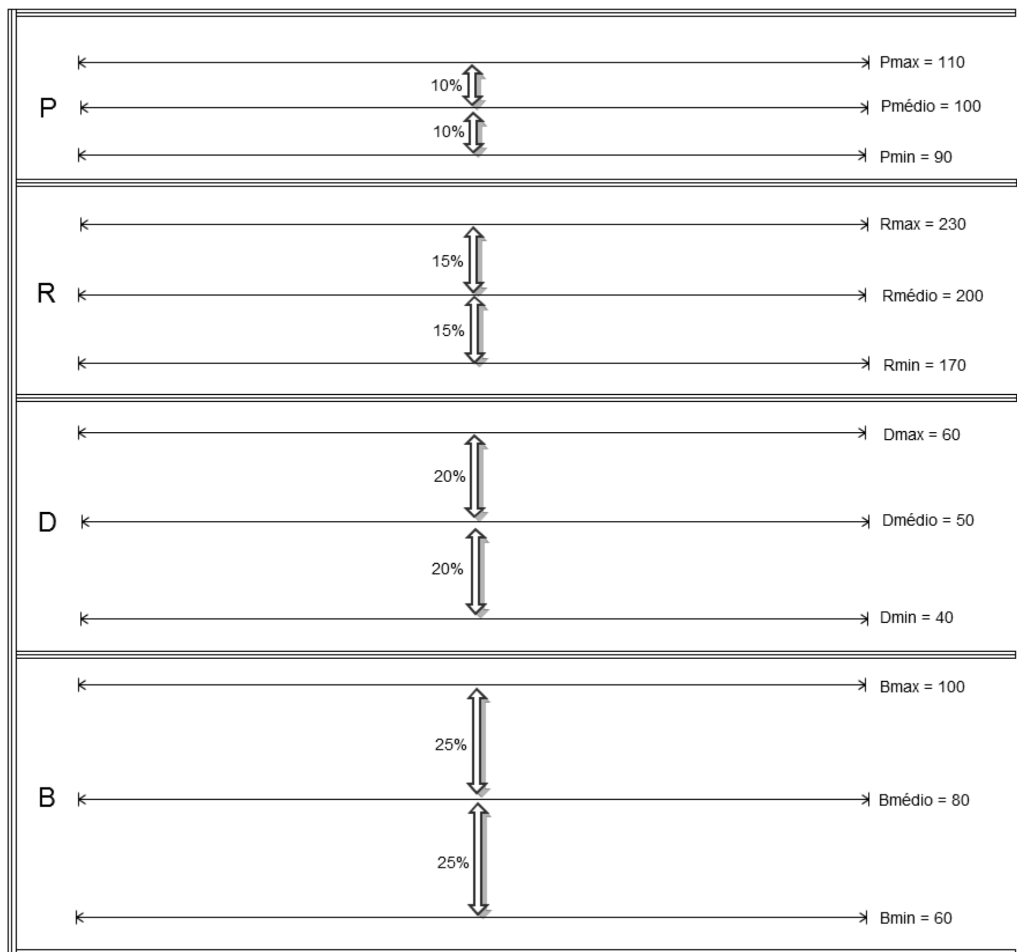



**Figura 11.** Exemplo de amplitudes de dimensões de uma máquina.

## Formalização da amplitude de uma dimensão:

*Seja $M = \{m_1, m_2, ..., m_n\}$ um conjunto de n máquinas.*

*Seja $V_{M,x} = \{v_1, v_2, ..., v_n\}$ o conjunto de valores da dimensão x em M.*

*Define-se a amplitude de x em M, denotado por $\Delta_{M,x}$, como:*

$$\Delta_{M,x} = \frac{v_{\max} - v_{médio}}{v_{médio}} = \frac{v_{médio} - v_{\min}}{v_{médio}}$$

*onde:*

$$\begin{cases} v_{\min} \mid v_{\min} \leq v_i, \ \forall_i \ 1 \leq i \leq n \\ v_{\max} \mid v_{\max} \geq v_i, \ \forall_i \ 1 \leq i \leq n \\ v_{médio} = \frac{\sum_i v_i}{n} \end{cases}$$

**Formalização da tupla de valores médios:**

*Considerando uma máquina, sejam:*

- *p o valor médio da dimensão processador*
- *r o valor médio da dimensão memória RAM*
- *d o valor médio da dimensão disco*
- *b o valor médio da dimensão banda*

*Define-se a tupla de valores médios de uma máquina como (p, r, d, b). No caso de host, denota-se essa tupla por* $\beta$*, enquanto que, no caso de VM, denota-se por* $\alpha$*.*

**Formalização da tupla de amplitudes:**

*Seja M =* $\{m_1, m_2, ..., m_n\}$ *um conjunto de n máquinas.*

*Sejam:*

- *P a dimensão processador*
- *R a dimensão memória RAM*
- *D a dimensão disco*
- *B a dimensão banda*

*Define-se a tupla de amplitudes para M, denotado por* $\theta_M$*, como:*

$$\theta_M = (\Delta_{M,P}, \quad \Delta_{M,R}, \quad \Delta_{M,D}, \quad \Delta_{M,B})$$

**Simplificação da notação:**

*Considerando um conjunto H de hosts e um conjunto V de VMs, as respectivas tuplas de amplitudes são, simplificadamente, representadas por* $\delta$ *e* $\pi$ *, ou seja:*

$$\delta = \theta_H$$

$$\pi = \theta_V$$

## 4.2 Considerações sobre a densidade de alocação

O uso da virtualização na computação em nuvem promove a elasticidade dos recursos dos *hosts*, permitindo um consumo de recursos específico para cada VM de acordo com a necessidade de cada aplicação executada em seu topo. Tendo em vista a heterogeneidade das VMs, conclui-se que mais ou menos VMs podem ser alocadas nos *hosts*, dependendo dos valores médios de suas tuplas de dimensões. Chamaremos a razão entre o número de VMs a serem alocadas e o número de *hosts* de densidade e a representaremos por $\rho$ .

Na computação em nuvem, o predomínio é de sistemas operacionais monolíticos, pois a maior parte dos *datacenters* usam alguma variante monolítica de Unix, Linux ou Windows. Os sistemas operacionais monolíticos requerem bastante recursos de *hardware*, pois seu *Kernel* é um enorme programa que contempla memória virtual , sistemas de arquivos e uma grande diversidade de *drivers*. Neste modelo, tipicamente, os *hosts* possuem uma densidade pequena.

Apesar dos Sistemas Operacionais monolíticos dominarem a computação em nuvem, existe um grande interesse da comunidade científica sobre os sistemas operacionais do tipo *microkernel*, cujas vantagens são muitas. Podemos enumerar as seguintes vantagens no contexto de computação em nuvem:

1. Menos sujeitos a *bugs* (devido ao código do *kernel* ser mais enxuto).
2. Consomem menos recursos de *hardware*.

3. Por consumirem menos recursos de *hardware*, um mesmo *host* pode abrigar centenas de sistemas operacionais ou até mesmo milhares.

Um exemplo de projeto científico, feito em cima de sistemas operacionais monolíticos, utilizando máquinas virtuais para promover o isolamento entre as aplicações é o Denali [Whitaker et al., 2002].

Sendo o consumo de recursos de *hardware* em um sistema operacional monolítico muito diferente do de um sistema operacional *microkernel*, segue a formalização do conceito de densidade com os intervalos escolhidos para sistemas operacionais monolíticos e *microkernel*.

**Formalização de densidade ($\rho$):**

*Sejam*

- *H um conjunto de hosts*
- *V um conjunto de VMs*

*Define-se a densidade de alocação de V em H, denotada por* $\rho$ *, como:*

$$\rho = \frac{|V|}{|H|}$$

## 4.3 Taxa de alocação

Uma métrica de avaliação de desempenho de um algoritmo de alocação é a sua taxa de alocação, denotada por $\tau$ e calculada como a razão entre o número de VMs alocadas e o número total de VMs. Formalmente:

$$\tau = \frac{|A|}{|V|}$$

Entretanto, para chegarmos a uma taxa de alocação precisa é necessário repetir os experimentos várias vezes, assim podemos formalizar o conceito de taxa de alocação média. Partindo do princípio que temos vários cenários que desejamos analisar, com vários pontos de

observação em cada cenário, onde realizamos os experimentos, devemos repetir estes experimentos algumas vezes e encontrar a taxa de alocação média de cada um.

*Sejam*

- $C = \{c_1, c_2, ..., c_z\}$ *um conjunto de z cenários de simulação.*
- $S = \{s_1, s_2, ..., s_x\}$ *um conjunto de x pontos de observação de* $c \in C$.
- $E = \{e_1, e_2, ..., e_n\}$ *um conjunto de n experimentos referentes a* $s \in S$.
- $\tau = \frac{|A|}{|V|}$ *a taxa de alocação de um algoritmo para* $e \in E$

*Define-se taxa de alocação média em um ponto de observação, denotada por* $\mathrm{T}$, *como:*

$$\mathrm{T} = \frac{\sum_{i=1}^{n} \tau_i}{n}$$

### 4.3.1 O Cenário ideal

Um cenário ideal é aquele onde não existe desperdício de recursos nos *hosts* e a totalidade das VMs (é) alocada. Ocorre sob as seguintes condições:

(i) A amplitude é zero tanto para *hosts* quanto para VMs, isto é, quando $\pi = (0, 0, 0, 0)$ e $\delta = (0, 0, 0, 0)$.

(ii) Se $k$ é o tamanho de cada dimensão dos *hosts* e $t$ é o tamanho de cada dimensão das VMs, isto é, $\beta = (k, k, k, k)$ e $\alpha = (t, t, t, t)$, então:

(ii.a) $k \bmod t = 0$

(ii.b) Se H é o conjunto de *hosts* e V é o conjunto de VMs, como não há sobra de recursos, então $|H| \cdot k = |V| \cdot t$

(ii.c) Se A é o conjunto de VMs alocadas, então $A = V$

### 4.3.2 O Cenário real

No entanto, em um cenário real ocorre amplitude maior que zero em uma ou mais dimensões das máquinas. Além disso, as dimensões das máquinas podem ser totalmente variáveis. A consequência é que a alocação de um conjunto de VMs, que seria 100% alocado no cenário ideal, é apenas parcial em um cenário real, para todo e qualquer algoritmo de alocação baseado em heurística conhecido, pois trata-se de uma instância do clássico Problema da Mochila.

Supondo um conjunto V de VMs, onde A = V seria o conjunto de VMs alocadas em um cenário ideal, em um cenário real esta alocação seria apenas parcial, ou seja, o conjunto de VMs alocadas é $A \subset V$ .

## 4.4 Conclusão

Os métodos de avaliação de desempenho encontrados na literatura possuem limitações, tais como:

a) Geralmente trabalham com apenas uma dimensão.

b) Não abordam heterogeneidade de hosts.

c) Os trabalhos que abordam a heterogeneidade de VMs em várias dimensões não formalizam o conceito.

d) Não contemplam variações de densidade de alocação.

Neste Capítulo, definimos um novo método de avaliação baseado nos seguintes conceitos: amplitude de dimensões, densidade de alocação e taxa de alocação. Este método permitiu eliminar as limitações listadas acima. As contribuições deste Capítulo estão no próprio método e também nos conceitos fundamentais formalizados, pois estes podem ser usados na concepção de novos métodos. A eficácia do método é verificada através dos experimentos relatados no Capítulo 5.

# 5 Experimentos

Este Capítulo descreve os experimentos realizados comparar o desempenho das heurísticas descritas na Seção 3.4 e na Seção 3.5 na solução do problema de alocação de VMs, conforme descrito na Seção 3.2. Os experimentos seguem o método de avaliação de desempenho definido no Capítulo 4. Os resultados foram obtidos através de simulação de diversos cenários concebidos considerando situações realistas de sistemas.

## 5.1 Delimitação do escopo dos experimentos

Como todas as possíveis combinações de $\delta$ , $\pi$ e $\rho$ seriam muitas, e para cada nova combinação entre eles seria necessária uma nova simulação, tivemos que limitar o escopo do experimento, fixando valores.

Para contemplar um grande espectro de amplitudes, decidimos utilizar amplitudes iguais e diferentes nos experimentos. A Tabela 1 mostra os valores selecionados para $\delta$ e $\pi$ .

**Tabela 1. Tuplas de amplitudes selecionadas para $\delta$ e $\pi$**

| *Amplitudes iguais* | *Amplitudes diferentes* |
|---|---|
| (10,10,10,10) | (10,15,20,25) |
| (20,20,20,20) | (10,20,30,40) |
| (40,40,40,40) | (10,30,50,70) |
| (80,80,80,80) | (10,40,70,100) |

Restringimos também as densidades que consideramos importantes para observar o comportamento dos algoritmos testados. A Tabela 2 mostra os valores escolhidos para $\rho$ .

**Tabela 2.** **Variações de $\rho$**

| $\rho$ para sistemas operacionais monolíticos | | | | | |
|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 10 | 20 |

| $\rho$ para sistemas operacionais *microkernel* | | | |
|---|---|---|---|
| 100 | 500 | 1000 | 2000 |

Como as restrições propostas pelas Tabelas 1 e 2 ainda permitiam 384 combinações para sistemas operacionais monolíticos e 256 combinações para sistemas operacionais *microkernel*, fixamos valores em algumas situações conforme a Tabela 3. Procuramos fixar os valores em situações típicas da computação em nuvem.

**Tabela 3.** **Valores típicos para amplitudes e densidade**

| | |
|---|---|
| $\pi$ | (40,40,40,40) |
| $\delta$ | (20,20,20,20) |
| $\rho$ | 10 |

O objetivo dos algoritmos experimentados é de alocar o maior número possível de máquinas virtuais. Para melhor visualizar os resultados dos experimentos, foram montados gráficos que avaliam a eficiência de cada algoritmo mostrando as porcentagens de VMs alocadas em relação a um conjunto de VMs segundo o cenário ideal (Seção 4.3).

Valores de $\delta$ foram fixados em 20% para toda dimensão da tupla, pois acreditamos que na maior parte das situações práticas temos um *hardware* de 0 a 3 anos de defasagem. Da mesma forma $\rho$ foi fixado em 10, simulando uma situação característica de micro e medias instâncias de VMs. Quando necessário $\pi$ é fixado em 40 para todas as tuplas, simulando um espectro de diversidade do consumo de recursos entre as VMs típico.

Para validar estes cenários através da simulação, foi preciso estipular valores para o número total de *hosts* e VMs, também foi necessário calcular os valores médios para suas respectivas dimensões (de acordo com a densidade a ser testada).

*Sejam:*

- *H: um conjunto de hosts.*
- *V: um conjunto de VMs que devem ser alocadas em H.*

*Assumindo:*

- $\beta = (k, k, k, k)$
- $\alpha = (t, t, t, t)$

*Queremos:*

$$|H| \, . \, k = |V| \, . \, t$$

*Assim:*

$$\frac{|V|}{|H|} = \frac{k}{t}$$

*Como:*

$$\rho = \frac{|V|}{|H|}$$

*Então:*

$$\rho = \frac{k}{t}$$

*Portanto:*

$$t = \frac{k}{\rho}$$

Respeitando as equações acima, montamos a Tabela 4, calculando os valores médios das tuplas de dimensões relativas às densidades usadas nos experimentos.

**Tabela 4.** Valores utilizados para cada densidade experimentada

| \|H\| | k | $\rho$ | t | \|V\| |
|---|---|---|---|---|
| 100 | 1000 | 2 | 500 | 200 |
| 100 | 1000 | 3 | 333 | 300 |
| 100 | 1000 | 4 | 250 | 400 |
| 100 | 1000 | 5 | 200 | 500 |
| 100 | 1000 | 10 | 100 | 1000 |
| 100 | 1000 | 20 | 50 | 2000 |
| 10 | 10000 | 100 | 100 | 1000 |
| 10 | 10000 | 500 | 20 | 5000 |
| 10 | 10000 | 1000 | 10 | 10000 |
| 10 | 10000 | 2000 | 5 | 20000 |

## 5.2 Resultados obtidos

Esta Seção mostra os resultados dos experimentos definidos por cada cenário da Tabela 4 e faz uma análise de cada um deles.

A partir das Tabelas 1, 2 e 3, montamos 7 cenários para experimentos. Os experimentos dos cenários I a VI resultam em um gráfico bidimensional cada um, mostrando o comportamento de alocação de VMs para todos algoritmos. O experimento do cenário VII gera um gráfico tridimensional para cada algoritmo.

### 5.2.1 Cenário I

O Cenário I experimenta a influência da variação das amplitudes das dimensões das VMs na taxa de alocação, isto é, os pontos de observação são tuplas ($\pi$) de amplitudes das dimensões das VMs, supondo:

- Amplitudes **idênticas** para cada dimensão de **VM**, em cada tupla.
- $\delta$ = (20,20,20,20)
- $\rho$ = 10

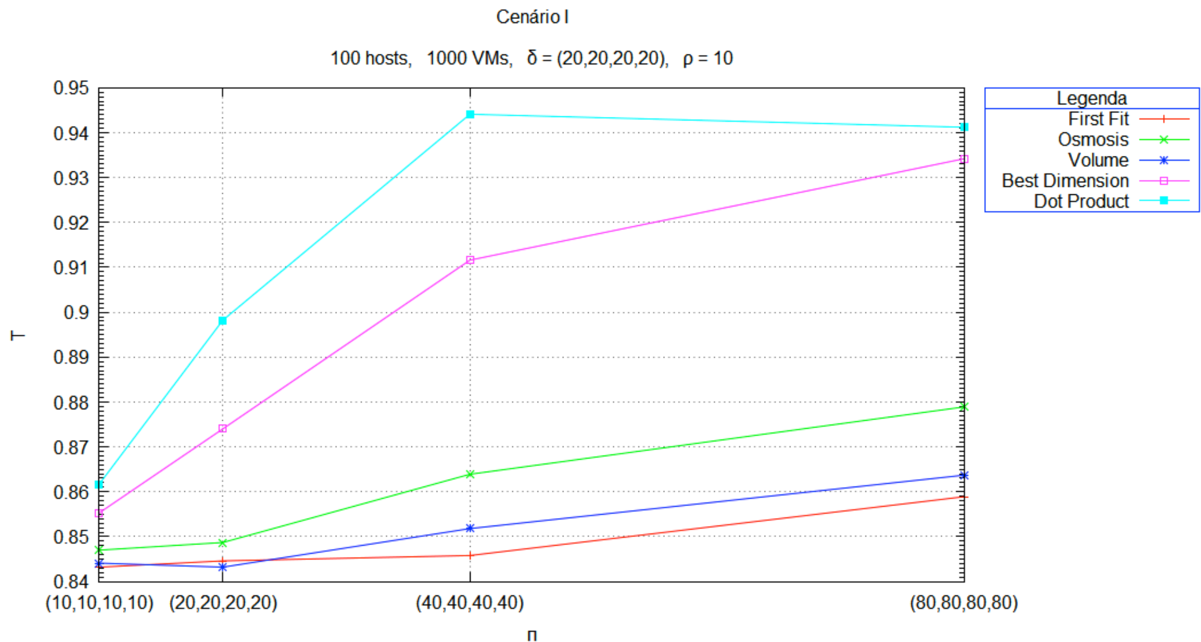



**Figura 12**. Gráfico para a taxa de alocação obtido no Cenário I.

De acordo com a Figura 12, pode-se observar que:

- O aumento de amplitude das dimensões das VMs favorece todos os algoritmos.
- O algoritmo que teve o desempenho mais baixo foi o *First Fit* que é o mais utilizado nas plataformas atuais.
- O algoritmo que teve o desempenho mais alto foi o *Dot Product*, chegando a ser aproximadamente 10% melhor que o *First Fit* e aproximadamente 3% melhor que o *Best Dimension*.
- Os algoritmos *Osmosis* e *Volume* tiveram desempenhos intermediários, com pequena vantagem para o *Osmosis*.

### 5.2.2 Cenário II

O Cenário II experimenta a influência da variação das amplitudes das dimensões dos *hosts* na taxa de alocação, isto é, os pontos de observação são as tuplas ($\delta$) de amplitudes das dimensões dos *hosts*, supondo:

- Amplitudes **idênticas** para cada dimensão de ***host***, em cada tupla.
- $\pi$ = (40,40,40,40)
- $\rho$ = 10

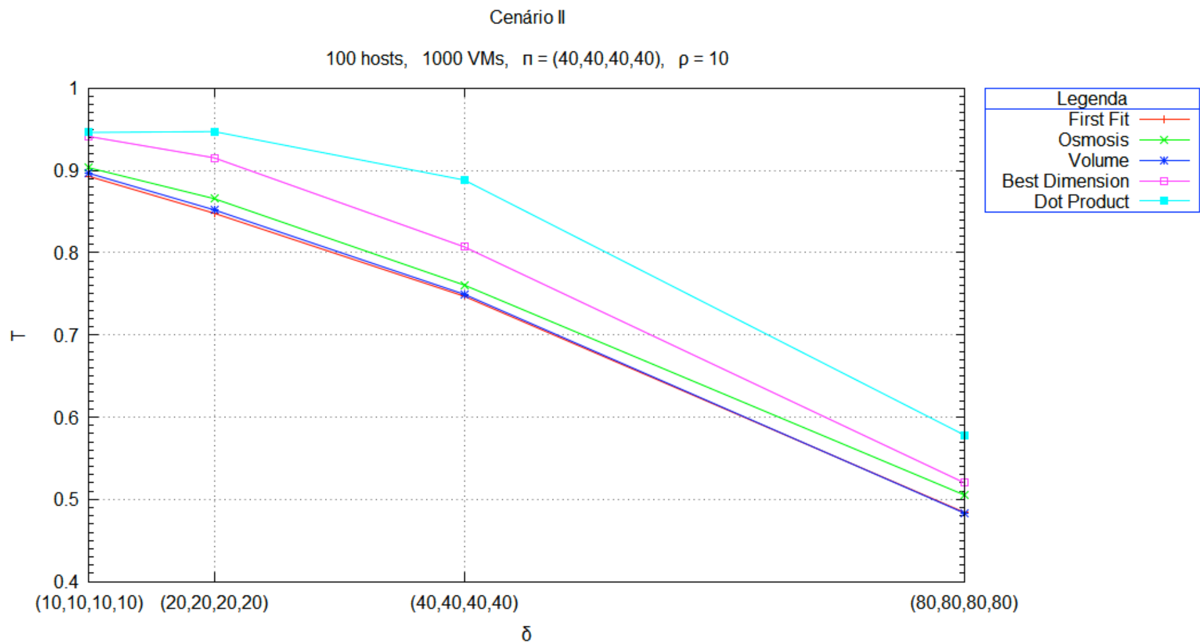



**Figura 13** Gráfico para a taxa de alocação obtido no Cenário II.

De acordo com a Figura 13, pode-se observar que:

- O aumento de amplitude das dimensões dos *hosts* prejudica o desempenho de todos os algoritmos.
- O algoritmo que teve o desempenho mais baixo foi o *First Fit*, que é o mais utilizado nas plataformas atuais.
- O algoritmo *Volume* teve um desempenho praticamente igual ao do *First Fit*.

- O algoritmo que teve o desempenho mais alto foi o *Dot Product*, chegando a ser aproximadamente 13% melhor que o *First Fit* e aproximadamente 7% melhor que o *Best Dimension*.
- O algoritmo *Osmosis* teve desempenho intermediário.

### 5.2.3 Cenário III

O Cenário III experimenta a influência da variação das amplitudes das dimensões das VMs na taxa de alocação, isto é, os pontos de observação são tuplas ($\pi$) de amplitudes das dimensões das VMs, supondo:

- Amplitudes **diferentes** para cada dimensão de **VM**, em cada tupla.
- $\delta$ = (20,20,20,20)
- $\rho$ = 10

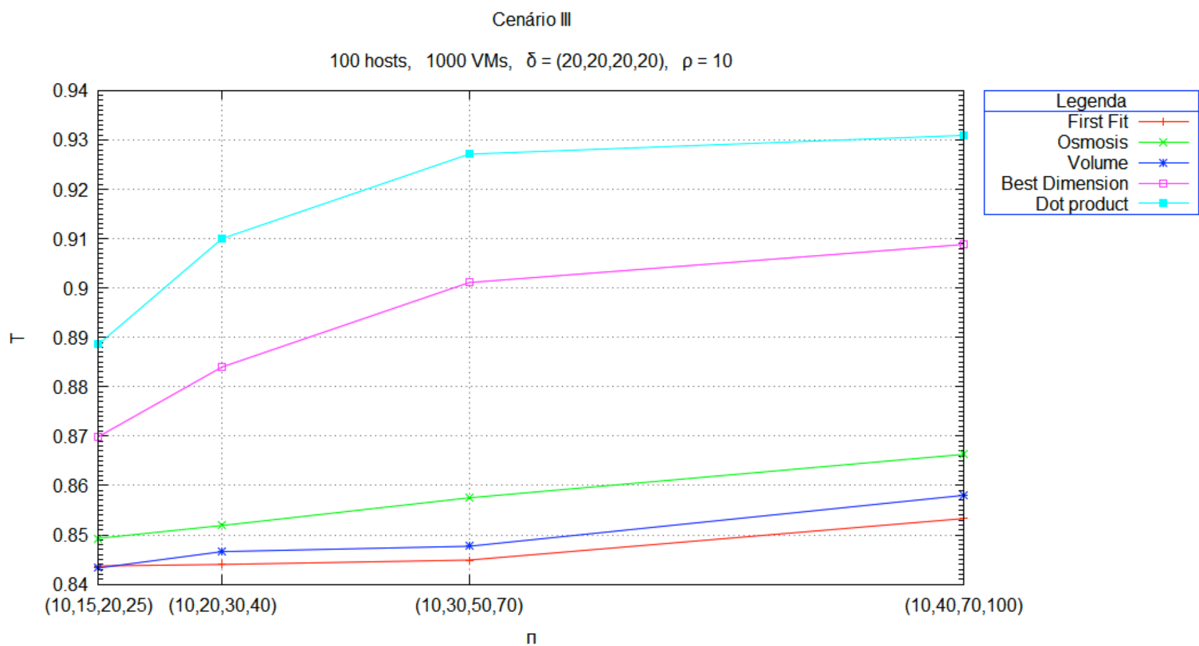



**Figura 14**. Gráfico para a taxa de alocação obtido no Cenário III.

Os pontos de observação foram definidos de forma que a diferença entre duas amplitudes em uma mesma tupla aumenta progressivamente; os valores escolhidos para essa diferença foram 5, 10, 20 e 30. De acordo com a Figura 14, pode-se observar que:

- O aumento de amplitude das dimensões das VMs favorece todos os algoritmos.
- O algoritmo que teve o desempenho mais baixo foi o *First Fit*.
- O algoritmo que teve o desempenho mais alto foi o *Dot Product*, chegando a ser aproximadamente 9% melhor que o *First Fit* e aproximadamente 3% melhor que o *Best Dimension*.
- Os algoritmos *Osmosis* e *Volume* tiveram desempenhos intermediários, com pequena vantagem para o *Osmosis*.

### 5.2.4 Cenário IV

O Cenário IV experimenta a influência da variação das amplitudes das dimensões dos *hosts* na taxa de alocação, isto é, os pontos de observação são as tuplas ($\delta$) de amplitudes das dimensões dos hosts, supondo:

- Amplitudes **diferentes** para cada dimensão de ***hosts***, em cada tupla.
- $\pi$ = (40,40,40,40)
- $\rho$ **=** 10

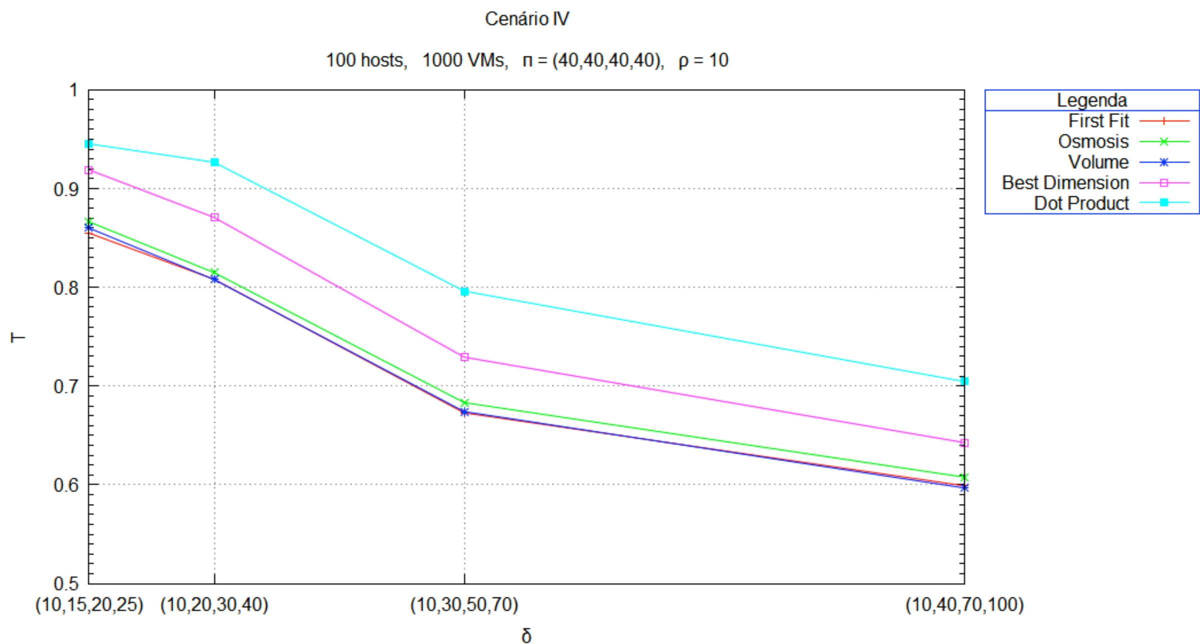



**Figura 15.** Gráfico para a taxa de alocação obtido no Cenário IV.

Os pontos de observação foram definidos de forma que a diferença entre duas amplitudes em uma mesma tupla aumenta progressivamente; os valores escolhidos para essa diferença foram 5, 10, 20 e 30. De acordo com a Figura 15, pode-se observar que:

- O aumento de amplitude das dimensões dos *hosts* prejudica o desempenho de todos os algoritmos.
- Os algoritmos que tiveram o desempenho mais baixo foram o *First Fit* e o *Volume*.
- O algoritmo que teve o desempenho mais alto foi o *Dot Product*, chegando a ser aproximadamente 13% melhor que o *First Fit* e aproximadamente 7% melhor que o *Best Dimension*.
- Os algoritmo *Osmosis* teve um desempenho intermediário.

### 5.2.5 Cenário V

O Cenário V experimenta a influência da variação da **densidade de alocação** na taxa de alocação, isto é, os pontos de observação são valores de $\rho$, supondo:

- $\pi$ = (40,40,40,40)
- $\delta$ = (20,20,20,20)
- A $\rho$ são atribuídos valores compatíveis com densidades típicas da computação em nuvem, utilizando sistemas operacionais monolíticos nas VMs.

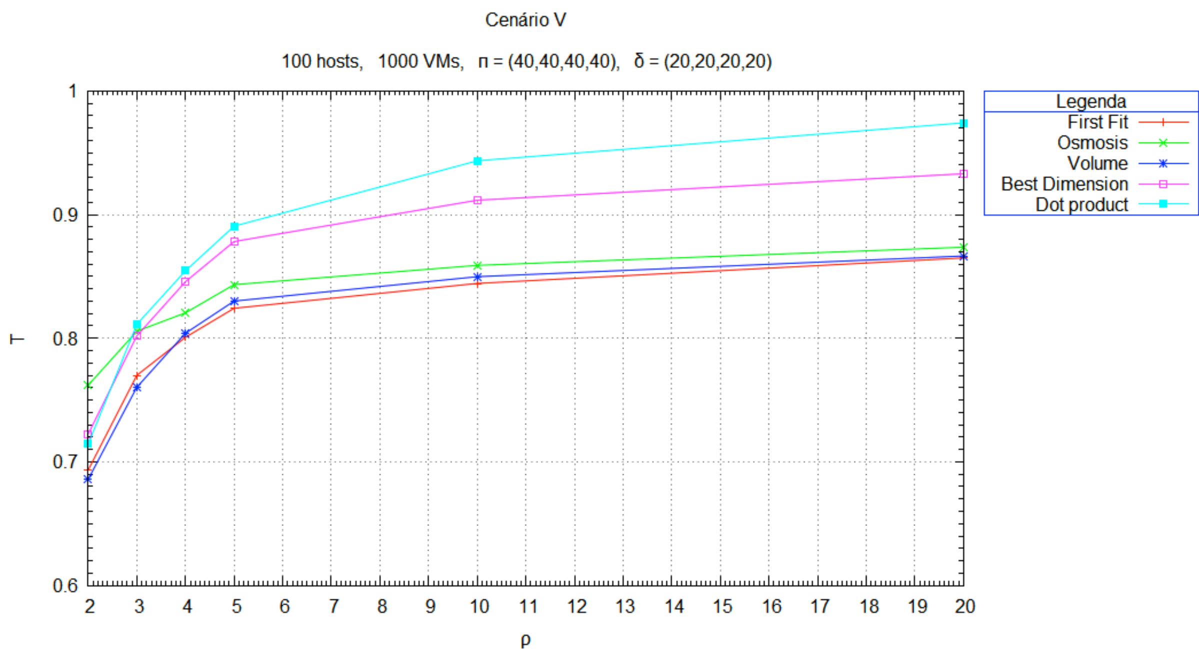



**Figura 16.** Gráfico para a taxa de alocação obtido no Cenário V.

De acordo com a Figura 16, pode-se observar que:

- O aumento da densidade favorece todos os algoritmos.
- Em densidades pequenas ($2 \leq \rho < 3$), o *Osmosis* teve o melhor desempenho, chegando a ser aproximadamente 8% melhor que o *Volume* que, por sua vez, teve o pior desempenho. O *First Fit* teve um desempenho similar ao *Volume*. O *Best Dimension* e o *Dot Product* tiveram desempenhos intermediários similares.
- Em densidades médias e grandes ($\rho \geq 3$), o *Dot Product* teve o melhor desempenho chegando a ser 10% melhor que o *First Fit* que teve o pior desempenho. O *Volume* teve um desempenho similar ao *First Fit*. O desempenho do *Best Dimension* foi similar ao *Dot Product* até $\rho \leq 5$, mas para $\rho > 5$ o *Dot Product* começa a abrir uma boa margem de vantagem, chegando a ser aproximadamente 5% mais eficiente que o *Best Dimension* . O *Osmosis* teve um desempenho intermediário.

### 5.2.6 Cenário VI

O Cenário VI experimenta a influência da variação da **densidade de alocação** na taxa de alocação, isto é, os pontos de observação são valores de $\rho$, supondo:

- $\pi$ = (40,40,40,40)
- $\delta$ = (20,20,20,20)
- A $\rho$ são atribuídos valores compatíveis com sistemas operacionais *microkernel* nas VMs.

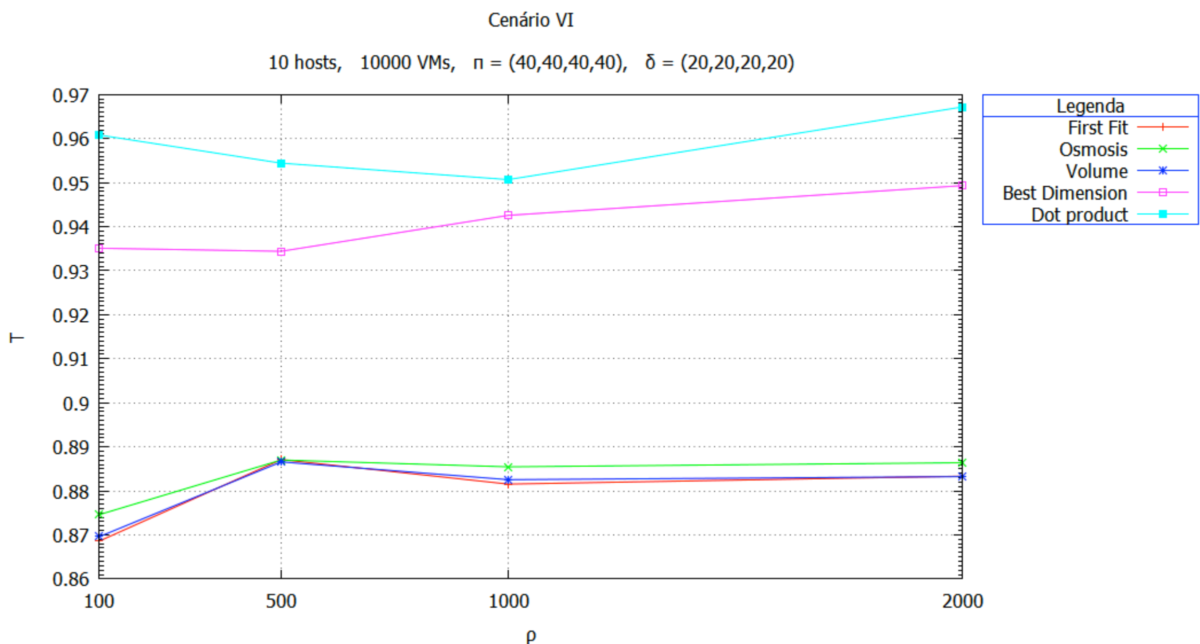



**Figura 17**. Gráfico para a taxa de alocação obtido no Cenário VI.

De acordo com a Figura 17, pode-se observar que:

- O aumento da densidade a partir de $\rho \geq 100$ provoca pequenas oscilações no desempenho de todos algoritmos.
- Os algoritmos *First Fit*, *Volume* e *Osmosis* tiveram o menor desempenho com taxas de alocação similares.
- O *Best Dimension* teve um desempenho intermediário.

- O *Dot Product* teve o melhor desempenho com uma taxa 9% melhor que o *First Fit* e 3% melhor que o *Best Dimension*.

### 5.2.7 Cenário VII

O Cenário VII experimenta a influência da variação das amplitudes das dimensões dos *hosts* e das VMs na taxa de alocação, isto é, os pontos de observação são as tuplas ($\delta$ e $\pi$) de amplitudes das dimensões das máquinas, supondo $\rho$ = 10.

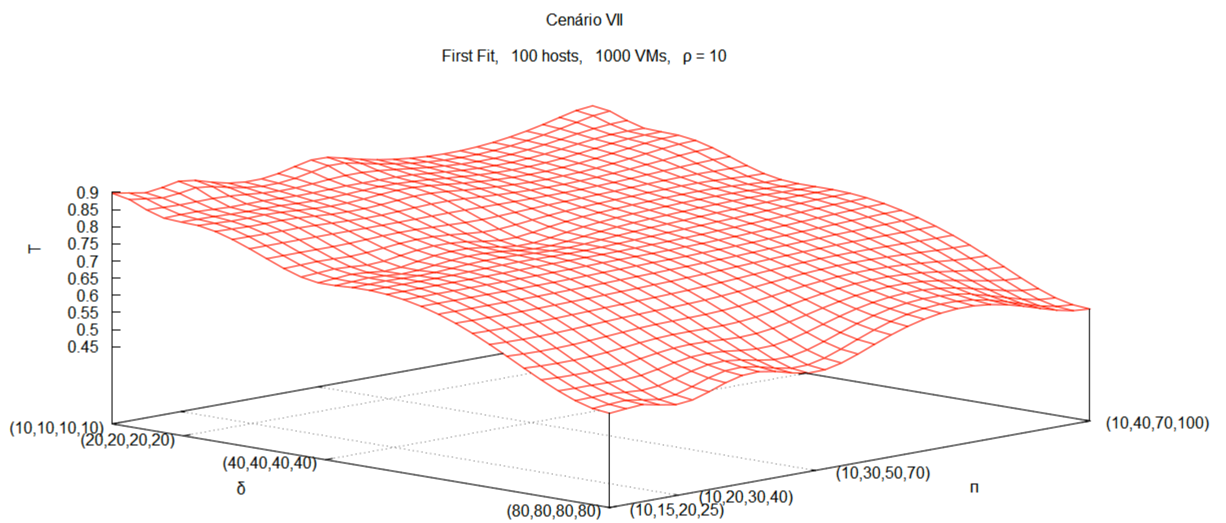



**Figura 18**. Taxa de alocação no Cenário VII para o algoritmo *First Fit*.

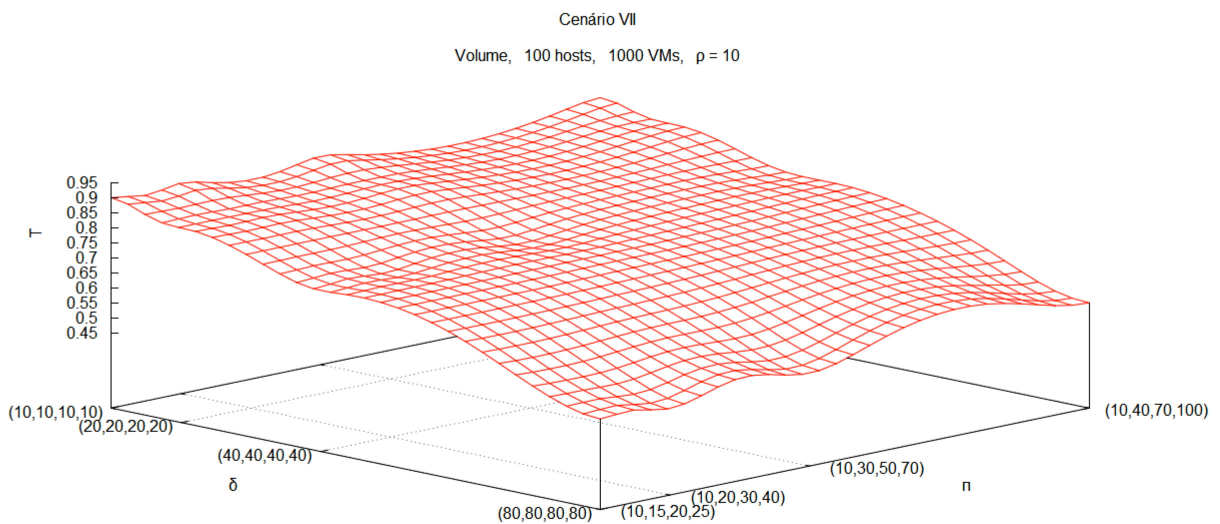



**Figura 19**. Taxa de alocação no Cenário VII para o algoritmo *Volume*.

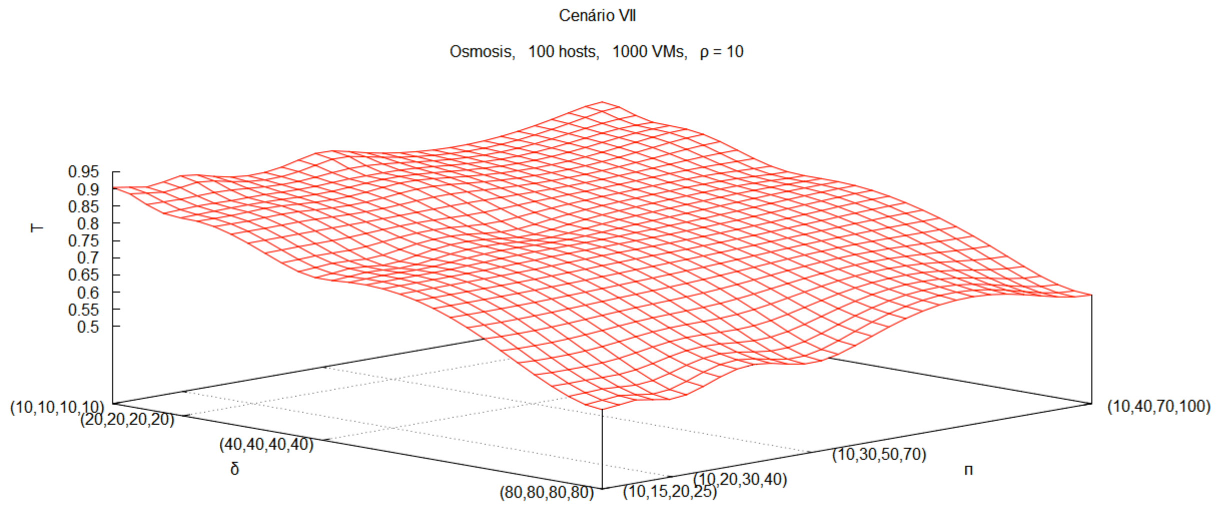



**Figura 20**. Taxa de alocação no cenário VII para o algoritmo *Osmosis*.

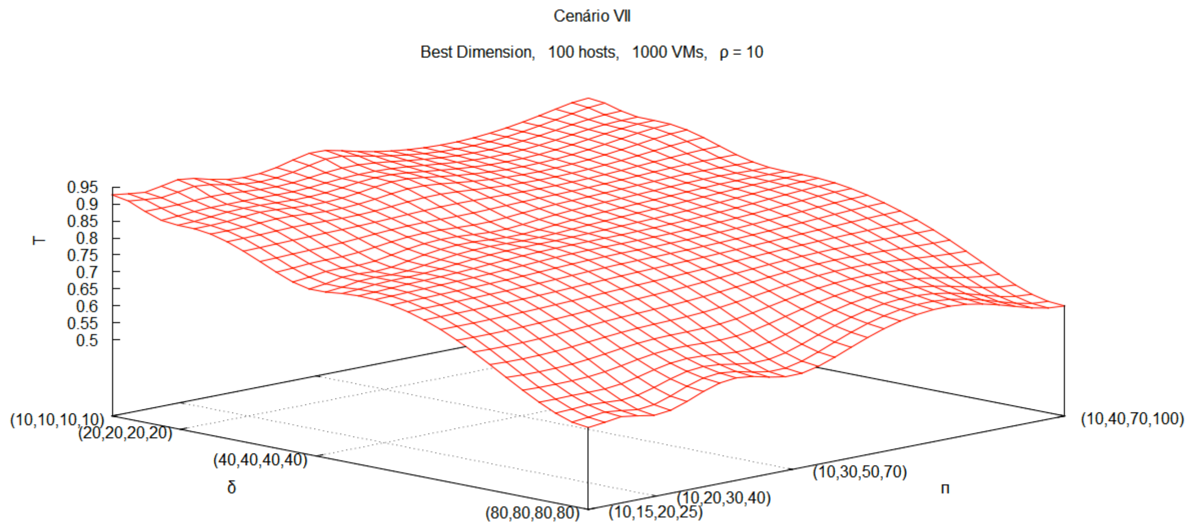



**Figura 21**. Taxa de alocação no Cenário VII para o algoritmo *Best Dimension*.

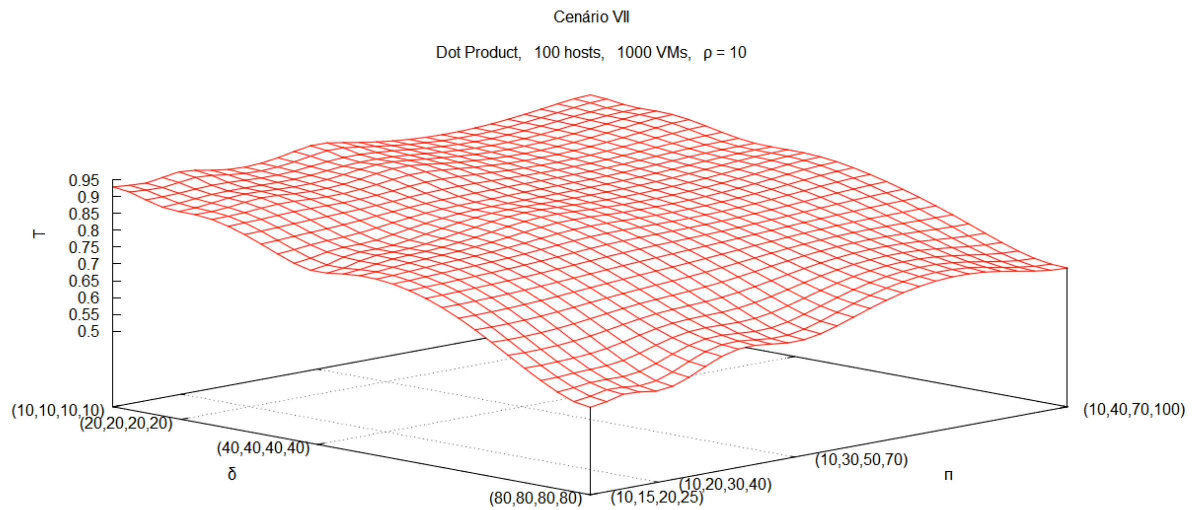



**Figura 22**. Taxa de alocação no Cenário VII para o algoritmo *Dot Product*.

De acordo com os gráficos obtidos neste Cenário, ilustrados nas Figuras 18, 19, 20, 21 e 22, pode-se observar que:

- O aumento de $\delta$ prejudica o desempenho de todos algoritmos.
- O aumento de $\pi$ melhora o desempenho de todos algoritmos.
- Todos gráficos possuem depressões similares, sendo que o *Dot Product* tem depressões menos perceptíveis que os demais algoritmos.

## 5.3 O Simulador *PyCloud*

Quando foi decido usar a simulação para os experimentos, em um primeiro momento foi estudado o simulador *CloudSim* [Calheiros et al., 2010]. Este simulador, mesmo tendo pouco tempo de vida, já é uma referência em muitos estudos de computação em nuvem, sua arquitetura foi construída na linguagem *Java* e em cima do simulador *GridSim* [GridSim, 2013]. Apesar do simulador *CloudSim* ser muito interessante, ele simula muitos recursos que não eram necessários para esta dissertação, tais como:

- diversos *datacenters*.
- nuvens federadas.
- redes.

- usuários.
- tarefas.
- consumo de energia.
- migração de VMs.
- suporte a diferentes políticas para cada etapa da simulação.

A complexidade do simulador *CloudSim* pode ser observada através da ilustração da sua arquitetura na Figura 23.

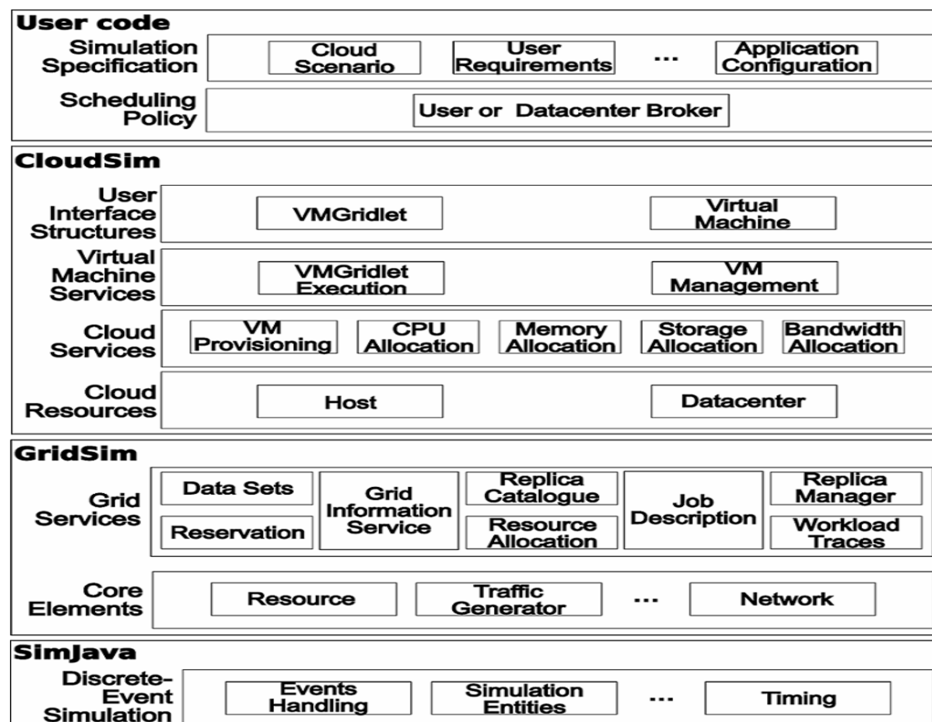



**Figura 23**. A arquitetura do *CloudSim* [Calheiros et al., 2010].

Como esta dissertação não necessitava da maior parte das funcionalidades do *CloudSim*, foi decidido construir outro simulador com a ideia de ser simples, específico e funcional. Os motivos que nos levaram a esta escolha foram os seguintes:

- Seria necessário estudar com detalhes todas as funcionalidades extras do *CloudSim* e como elas poderiam interferir nos resultados deste experimento, este estudo

demandaria mais tempo de pesquisa e poderia inviabilizar o término do projeto em tempo hábil.

- As funcionalidades adicionais implementadas pelo *CloudSim* necessitam de processamento, fazendo com que o tempo de simulação total seja aumentado. Como o tempo de simulação é um fator muito importante para realizar diversas baterias de testes, se o tempo fosse muito grande para cada simulação, as amostras teriam que ser reduzidas e o resultado final seria menos confiável. Apesar dos algoritmos testados realizarem apenas operações matemáticas básicas (subtração, divisão, multiplicação e soma), o processo é repetido inúmeras vezes proporcionalmente ao número de VMs e *hosts*. Sendo a simulação basicamente uma aplicação do tipo *CPU-bound*, queríamos minimizar gastos de tempo, eliminando processamentos desnecessários.
- O novo simulador teria o objetivo único e específico de testar algoritmos de alocação de um conjunto de VMs em um conjunto de *hosts*, esta especificidade ajudaria a depurar o código e minimizaria o seu tempo de construção.

Após decidir fazer um novo simulador, restava escolher em qual tecnologia ele seria desenvolvido; após uma breve análise dos prós e contras de cada linguagem de programação foi escolhido *Python*. Os motivos que nos levaram a esta escolha foram os seguintes :

- Amplo suporte da comunidade de desenvolvedores.
- Ser uma tecnologia de código fonte aberto.
- É linguagem focada no rápido ciclo de desenvolvimento de código.
- Suporte orientação a objetos.
- Possui um grande leque de bibliotecas.
- Produz um código muito legível devido a identação obrigatória exigida pelo interpretador.
- Apesar de *Python* ser fracamente tipada, ela é poderosa justamente devido a sua simplicidade.

Como queríamos que o novo simulador fosse simples, específico e funcional, a combinação com *Python* se encaixou de forma perfeita, permitindo a construção do *PyCloud* em um tempo reduzido.

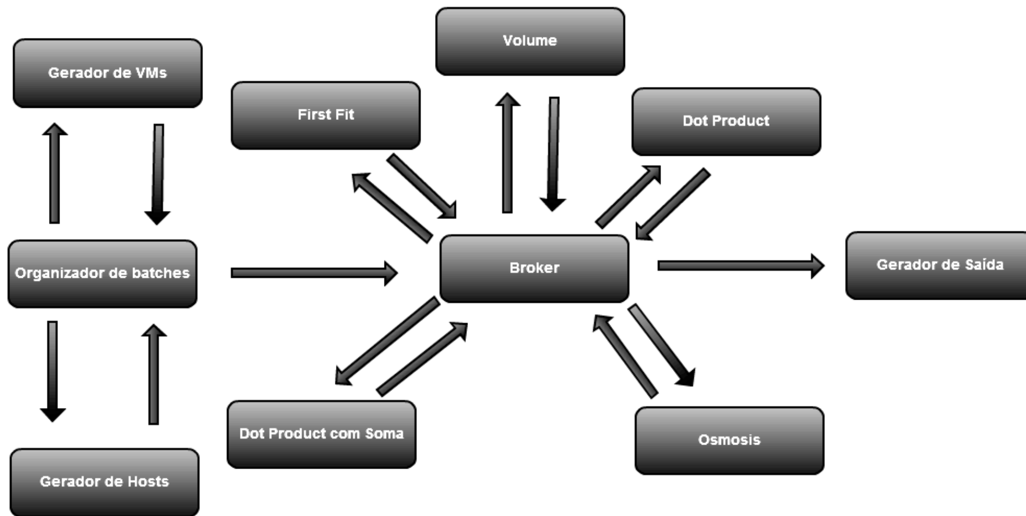



**Figura 24**. A arquitetura do *PyCloud.*

O funcionamento do *PyCloud* é descrito em detalhes abaixo:

- O gerador de VMs e o gerador de *hosts* recebem uma solicitação de geração de arquivos do organizador de *batches*. O gerador de VMs e o gerador de *hosts* recebem os parâmetros |V| (caso gerador de VMs) ou |H| (caso gerador de *hosts*) e seus respectivos $v_{min}$ e $v_{max}$ referentes às dimensões (P, R, D, B). Com os parâmetros carregados na memória, os valores mínimos e máximos de cada dimensão são usados por uma função randômica geradora de números pseudoaleatórios uniformes formando a tupla de dimensões de cada máquina. Assim que uma tupla de dimensões de uma máquina é formada, uma chave única é associada a ela, e estes dados que representam uma máquina são persistidos em arquivo. Cada gerador cria seu próprio arquivo e de acordo com |H| e |V| o valor total de representação de máquinas é concatenado nestes arquivos. Como o número de experimentos escolhidos foi 10, cada gerador produz 10 arquivos diferentes em um cenário.
- O organizador de *batches* especifica uma sequência de cenários (C). A partir de cada cenário é especificado um *loop* de pontos de observação (S) de acordo com as variações de $\delta$, $\pi$ e $\rho$. Para cada ponto de S, uma solicitação de geração de VMs e *hosts* é feita a seus respectivos geradores. Para que os geradores possam atender esta solicitação é necessário que os parâmetros, especificando o total de máquinas (#V e

#H) a serem geradas e os valores $v_{min}$ e $v_{max}$ referentes a cada dimensão (P, R, D, B), sejam passados. Para que o gerador de *batches* encontre os valores de $v_{min}$ e $v_{max}$ referentes à dimensão de cada máquina a ser criada, é usado o valor médio de cada dimensão dos *hosts* e $\delta$ caso as máquinas a serem geradas sejam *hosts,* ou o valor médio de cada dimensão das VMs e $\pi$ caso as máquinas a serem geradas sejam VMs. Após o retorno dos 10 arquivos criados por cada gerador, pares de arquivos são formados, onde cada arquivo de *hosts* corresponde a um arquivo de VMs. Cada par de arquivos é lido pelo organizador de *batches*, um por vez, e carregado em memória, formando uma lista relativa a *hosts* e outra relativa a VMs. Estes pares de listas são passados ao *broker*, um par por vez, até completar os 10 pares de cada ponto em S .

- O *broker* é responsável por tentar alocar uma lista de VMs em uma lista de *hosts*, estas listas são recebidas como parâmetros enviadas pelo organizador de *batches*. A cada par de listas recebido, o *broker* executa os 5 algoritmos (*First Fit*, *Volume*, *Osmosis*, *Best Dimension* e *Dot Product*). Após cada algoritmo retornar ao número de VMs alocadas (#A) é encontrada a taxa de alocação ($\tau$) para cada um deles. As taxas de alocações são passadas como parâmetro ao gerador de saída.

- O gerador de saída armazena a taxa de alocação ($\tau$) de cada algoritmo, acumulando os resultados referentes aos pontos de observação. Estas taxas são somadas e divididas pelo número de experimentos realizados (neste caso 10), encontrando a taxa média de alocação ($T$) de cada algoritmo. Para cada ponto de observação de um algoritmo em um cenário tem-se $T$ persistido em um arquivo.

Nos testes que realizamos, o *PyCloud* mostrou-se estável e eficiente, produzindo resultados consistentes nas diversas simulações realizadas.

## 5.4 Discussão

Os novos algoritmos foram comparados com outros tradicionais através de simulação, empregando o novo método definido. Os experimentos realizados permitiram observar que, de forma geral, a heterogeneidade de recursos entre máquinas físicas prejudica o desempenho dos algoritmos, enquanto que a heterogeneidade de recursos entre máquinas virtuais o beneficia. Também foi possível observar que o aumento na densidade de alocação até certo ponto beneficia o desempenho dos algoritmos.

De um modo geral, podemos afirmar que:

- *First Fit* teve o pior desempenho entre os algoritmos avaliados. Mesmo existindo alternativas muito melhores para o problema de alocação, ele é ainda o mais usado devido a sua simplicidade de implementação.
- *Volume* teve um desempenho muito similar ao *First Fit* e, entre os dois, o First Fit é uma solução mais razoável devido à menor complexidade de implementação.
- *Osmosis* teve o melhor desempenho com baixas densidades, chegando a ser até 8% melhor que o *First Fit*. Em cenários com densidades típicas de computação em nuvem, o seu uso é indicado, e outro algoritmo poderia ser usado caso a densidade aumentasse ($\rho \geq 3$).
- *Best Dimension* teve um desempenho médio superior ao First Fit, ao *Volume* e ao *Osmosis* em todos os cenários.
- *Dot Product* teve o melhor desempenho em todos os cenários, mostrando-se consistente nas mais diversas variações de heterogeneidade e densidade. Teve um ganho de desempenho de até de 13% em relação ao *First Fit*.

O método de avaliação definido e utilizado nos experimentos mostrou-se adequado. Em cada cenário, a avaliação isolada de uma variável exigiu a fixação de valores para as demais variáveis. Os valores escolhidos como fixos foram $\rho$=10, $\pi$=(40, 40, 40, 40) e $\delta$=(20, 20, 20, 20) por serem típicos da computação em nuvem. No entanto, outros valores podiam ser escolhidos de acordo com o ambiente alvo. Por exemplo, para simular ambientes onde as VMs sejam instâncias grandes, o valor de $\rho$ poderia ser fixado em 2 e, dessa forma,

todos os cenários que fixam $\rho$ seriam refeitos, mostrando a flexibilidade do método de avaliação.

# 6 Conclusão

Esta dissertação investigou o problema de alocação de VMs em um *data center*, procurando considerar variáveis realistas, tais como: fluxo contínuo de entrada de VMs, lotes de VM com respeito da ordem de chegada, *N hosts* e as quatro dimensões típicas das máquinas : processador, memória, disco e banda de rede (ou, simplesmente banda).

## 6.1 Contribuições

O trabalho de revisão da literatura e a reflexão sobre o problema a ser resolvido levaram às seguintes contribuições:

a) Definição de uma classificação de abordagens para o problema: a classificação permitiu diferenciar os diversos problemas de alocação de VMs e pode servir como guia no estudo de novas soluções.

b) Definição formal do problema segundo uma abordagem realista: o formalismo definido pode ser usado na verificação de novas propostas de solução para o problema.

c) Proposta de duas novas heurísticas de alocação de máquina virtual: as novas heurísticas podem ser verificadas para o problema de alocação de VM, como também podem servir de inspiração para a solução de outros problemas.

Uma vez definido o problema e identificadas as alternativas de solução a serem verificadas, fez-se necessário definir um método de avaliação de desempenho de algoritmos de alocação de VMs que permitisse comparar essas alternativas. No entanto, observou-se que os métodos de avaliação de desempenho encontrados na literatura possuem limitações. Por isso, definimos um novo método de avaliação baseado nos seguintes conceitos: amplitude de dimensões, densidade de alocação e taxa de alocação. Este método permitiu eliminar as limitações listadas acima. As contribuições consequentes estão no próprio método e também

nos conceitos fundamentais formalizados, pois estes podem ser usados na concepção de novos métodos.

Os novos algoritmos foram comparados com outros tradicionais através de simulação, empregando o novo método definido. Os experimentos realizados permitiram observar que, de forma geral, a heterogeneidade de recursos entre máquinas físicas prejudica o desempenho dos algoritmos, enquanto que a heterogeneidade de recursos entre máquinas virtuais o beneficia. Também foi possível observar que o aumento na densidade de alocação até certo ponto beneficia o desempenho dos algoritmos. De forma geral, o melhor desempenho foi obtido pelo algoritmo *Dot Product,* enquanto o pior desempenho foi do *First Fit*. Apesar dos novos algoritmos propostos (*Osmosis* e *Best Dimension*) apresentarem um desempenho intermediário para o problema de alocação de VMs proposto, eles podem ser testados em outros problemas.

## 6.2 Trabalhos futuros

A classificação dos modelos de delimitação do problema proposta no Capitulo 3.1 permite estudar outras variantes do problema de alocação de VMs. Um exemplo seria considerar outras dimensões além das quatro estudadas, tais como a taxa de vazão do disco e a taxa de vazão da memória.

Os algoritmos utilizados possuem um determinado overhead que não foi calculado. Uma análise de complexidade destes algoritmos poderia ser feita.

O objetivo deste trabalho, foi maximizar a alocação de VMs. No entanto, este objetivo não invalida o propósito de economia de energia, o qual poderia ser contemplado simultaneamente com a maximização de VMs alocadas.

As heurísticas estudadas poderiam ter variantes. Por exemplo, a heurística Volume poderia ter uma simples modificação: antes de calcular o volume de cada *host*, multiplicaria cada dimensão pela dimensão correspondente da VM entrante, para depois ordenar os *hosts* influenciados pelo peso da VM entrante. Da mesma forma, outras variantes poderiam ser criadas para cada heurística.

O método de avaliação proposto pode ser usado para comparar o desempenho de outros algoritmos tais como: genéticos, estocásticos, *annealing*, programação linear inteira, *constraint programming*, colônia de formigas, etc.

O algoritmo *Osmosis* apresentou o melhor desempenho com a densidade de alocação baixa. Por isso, seria interessante refazer todos os experimentos com a densidade de alocação fixada em 2 e 3, ao invés de 10.

Em uma abordagem de problema realista, as VMs podem terminar. Da mesma forma, o ambiente de computação em nuvem tem como sua principal característica a *elasticidade*, isto é, os valores das dimensões de uma VM podem oscilar. Existe também a possibilidade de ocorrer migração de VMs entre *hosts*, especialmente para fins de balanceamento de carga. Nenhum desses aspectos foi contemplado por esta dissertação.

Existe ainda a possibilidade de duas VMs chegarem simultaneamente, este trabalho não tratou esta possibilidade, pois considera a chegada de cada VM em instante diferente. Em um modelo que contemple a concorrência de alocação de VMs, uma heurística adicional para tratar este problema seria necessária.

Finalmente, existe ainda a possibilidade de adaptar os algoritmos citados nesta dissertação, para um ambiente de testes real. O ambiente de testes *Open Cirrus* [Campbell et al. 2010] pode ser usado para validar os resultados. O *Open Cirrus* é explicado em detalhes no Anexo IV.

# Referências bibliográficas

[Barham et al., 2003] Barham, P., Dragovic B., Fraser, K., Hand, S., Harris T., Ho, A., Neugebauer, R., Pratt, I., Warfield A. (2003) Xen and the Art of Virtualization. University of Cambridge Computer Library. Cambridge, UK.

[Barlett, 2003] Barlett, J. (2003) Programming from the ground up. *Edited by Dominick, Jr.* Disponível em: <http://savannah.c3sl.ufpr.br/pgubook/ProgrammingGroundUp-1-0-booksize.pdf, maio, 2011>.

[Barrio, 2001] Barrio, V. M. (2001). *Study of Techniques for Emulation Programming.* Computer Science Engineering – FIB UPC, Chapter 2, p. 14-25.

[Bonde, 2010] Bonde, T. (2010) *Techniques for Virtual Machine Placement in Clouds*. MTP Stage 1 Report. Department of Computer Science and Engineering. Indian Institute of Technology Bombay. Mumbai, p. 01-17.

[Calheiros et al., 2011] Calheiros, R., Ranjan, R., Beloglazov, A., Rose, C., Biyya, R. (2011) CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Journal Software –Practice and Experience,* v. 41, Issue 1, p. 23-50.

[Campbell et al., 2010] Campbell, R., Gupta, I., Heath, M., Ko, S.Y., Kozuch, M., Kunze, M., Kwan, T., Lai, K., Lee, H. Y., Lyons, M., Milojicic, D., O'Halloron, D., Soh, Y.C. (2010). Open Cirrus Cloud Computing Testbed: Federated DataCenters for Open Source Systems and Services Research. Disponível em: <static.usenix.org/event/hotcloud09/tech/full_papers

/Campbell.pdf>.

[Chong, 2011] Chong, R. (2011) Db2 on the Cloud. *Slides da palestra da IBM em abril de 2011 realizada na PUC-PR.* Disponível em: <https://www.ibm.com/developerworks/ mydeveloperworks/groups/service/html/communityview?communityUuid=80857aa6-ecbd-448b-9704-54fcead3a725, abril, 2011>.

[Campbell et al., ]

[Creasy, 1981] Creasy, J. R. (1981). The origin of VM/370 Time-Sharing System. *IBM Journal of Research and Development,* v. 25, n. 5, p. 483-490.

[Endo et al., 2010] Endo, P. T., Gonçalves G. E, Kelner J., Sadok, D. F. H. (2010) A Survey on Open-source Cloud Computing Solutions . VIII Workshop em Clouds, Grids e Aplicações. *Anais do XXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos.* Porto Alegre, p. 3-16.

[Ferrie, 2006] Ferrie, P. (2006). Attacks on Virtual Machine Emulators. *Symantech Advanced Threat Research.*

[Fidanova, 2005]. Fidanova, S. (2005). Heuristics for Multiple Knapsack Problem. *Proceedings of the IADIS International Conference on Applied Computing.* Algarve, Portugal, p. 255-260.

[Firmino et al., 2007] Firmino, R., Cabral G., Covacevice, A., (2007) Design de Kernels: Microkernel, Exokernel e novos Sistemas Operacionais. *Unicamp, Seminario MO806, Tópicos de Sistemas Operacionais.* Disponível em: http://www.ic.unicamp.br/~islene/2s2007-mo806/slides/Microkernel.pdf, fevereiro, 2013.

[Ford *et al.,* 1996] Ford, B., Hibbler, M., Lepreau, J., Tullmann, P., Back, G.,Goel, S., Clawson, S. (1996). Microkernels Meet Recursive Virtual Machines. Technical Report UUCS-96-004, University of Utah.

[Garfinkel and Rosenblum, 2005] Garfinkel, T., Rosenblum, M. (2005). When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. *Proceedings of the 10$^{th}$ Conference on Hot Topics in Operating Systems,* v.10, p. 20-20.

[Garfinkel and Warfield, 2007] Garfinkel, T., Warfield, A. (2007). What Virtualization Can Do for Security. *Login,* v. 32, n. 6, p. 28-34.

[Hines et al., 2009] Hines, M. R., Deshpande, U., Gopalan, K. (2009). Post Copy Live Migration of Virtual Machines. *ACM SIGORPS Operating System Reviews.* Vol. 43, Issue 3, p. 14-26.

[Harrisson, 2005] Harrison, L. (2005) Transmeta Crusoe Processor Presentation series. *University of Illinois*. Disponível em: <http://www.cs.uiuc.edu/homes/luddy/PROCESSORS/TransmetaCrusoe.pdf, maio, 2011>.

[Hyde, 1996] Hyde, R. (1996). *The Art of Assembly Language*. San Francisco: Ni Starch Press, Chapter 7, p. 413-476.

[Jansen et al., 2008] Jansen, B., Ramasamy, H. V., Schunter, M., Tanner, A. (2008). Architecting Dependable and Secure Systems Using Virtualization. *Lecture Notes in Computer Science,* v. 5135, p. 124-149.

[Jones, 2006] Jones, M. T. (2006). Virtual Linux. An Overview of Virtualization Methods, Architetures and Implementations.Disponível em: < http://www-128.ibm.com/

developerworks/library/l-linuxvirt/index.htm>.

[King et al., 2003] King, S., Dunlap, G., Chen, P. (2003). Operating System Support for Virtual Machines. *Proceedings of the Annual USENIX Technical Conference*.

[King et al., 2006] King, S. T., Chen, P. M, Wang, Y., Verbowsky, C., Wang, H. J., Lorch, J. R. (2006). SubVirt: Implementing Malware with Virtual Machines. Disponível em: <web.eecs.umich.edu/~pmchen/papers/king06.pdf>.

[Laureano e Maziero, 2008] Laureano, M., Maziero, C. (2008) Virtualização: Conceitos e Aplicações em Segurança. *Programa de Pós-Graduação em informática da Pontifícia Universidade Católica do Paraná e Centro Universitário Franciscano, Capítulo 4.*

[Lui and Wee, 2009] Liu, H., Wee, S. (2009). Web Server Farm in Cloud: Performance Evaluation and Dynamic Architecture. *Proceedings of the* $1^{st}$ *International Conference on Cloud Computing Technology and Science.* Berlin and Heidelberg: Springer Verlag, p. 369-380.

[LXRun, 2011] Ginzburg, S. (2011) Lxrun Frequently Asked Questions. *Version 3.3.*

Disponível em:<http://www.ugcs.caltech.edu/~steven/lxrun/lxrun-FAQ.html, maio, 2011>.

[Magalhães et al., 2011] Magalhães, D. V., Soares, J. M., Gomes, D. G. (2011). Análise do Impacto de Migração de Máquinas Virtuais em Ambiente Computacional Virtualizado. *Anais do XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos.* Campo Grande, p. 235-248.

[Maziero, 2011] Maziero, C. (2011) Livro de Sistemas Operacionais. *UTFPR, Capítulos 1 e 5.* Disponível em :<http://dainf.ct.utfpr.edu.br/~maziero/doku.php/so:

livro_de_sistemas_operacionais, abril, 2011>.

[Mohammadi et al., 2011] Mohammadi, E., Karimi, M., Heikalabad, S. R. (2011). A Novel Virtual Machine Placement in Cloud Computing. *Australian Journal of Basic and Applied Sciences,* 5 (10), p. 1549-1555.

[Melinda, 1997] Melinda, V. (1997). VM and the VM Community: Past, Present and Future. *Office of Computing and Information Technology.* Princeton University, p. 1-68.

[Mell and Grance, 2010] Mell, P., Grance T. (2010). The NIST Definition of Cloud Computing. *Cloud Computing Forum & Workshop 5/20/2010.*

[Moore, 1965] Moore, G. E. (1965). Cramming More Components onto Integrated Circuit. *Electronics,* v. 38, n. 8.

[Moore, 1997] Moore, J. (1997) SPARC traps under SUNOS. *SunSoft, Sun Microsystems Inc, Version 1.2, Chapter 2.* Disponível em: <http://adam.pra.to/public/docs/tech/sun/SparcTraps/

chap2.html>, março, 2011.

[Morimoto, 2007] Morimoto, C. (2007) As novas versões do Transmeta Crusoe. *Guia do Hardware, artigos.* Disponível em: <http://www.hardware.com.br/artigos/transmeta-crusoe/, abril, 2011>.

[Panigrahy et al., 2011]. Panigrahy, R, Talwar, K., Uyeda, L., Wieder, U. (2011). Heuristics for Vector Bin Packing. *Microsoft Research Sillicon Valley, Microsoft`s VMM product group.*

[PCSX, 2011] (2011) PCSX2 Developer Blog. *Portal do desenvolvimento do emulador PCSX2.* Disponível em: <http://pcsx2.net/blog.php, maio, 2011>.

[Peter et al., 2009] Peter, M., Schild, H., Lackorzynski, A., Warg, A. (2009). Virtual Machines Jailed: Virtualization in Systems with Small Trusted Computing Bases. *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems,* p. 18-23.

[Popek and Goldberg, 1974] Popek, G., Goldberg R. (1974). Formal Requirements for Virtualizable Architetures. *Communication of the ACM,* v. 17, n. 7, p. 412-421.

[Qemu, 2011] (2011) Qemu Open Source Processor Emulator. *Open books for an open world.*

Disponível em: <http://wiki.qemu.org/Main_Page, abril, 2011>.

[Robin and Irvine, 2000] Robin, J., Irvine C. (2000). Analysis of the Intel Pentium`s Ability to Support Secure Virtual Machine Monitor**.** *Proceedings of the 9th USENIX Security Symposium.* Denver, Colorado: USENIX Association.

[Rosenblum, 2004] Rosenblum, M. (2004). The Reincarnation of Virtual Machines. *Virtual Machines,* v. 2, n. 5.

[Rosenblum and Garfinkel, 2005] Rosenblum, M., Garfinkel, T. (2005). Virtual Machine Monitors: Current Technology and Future Trends. *Computer,* 38 (5), p. 39-47.

[Shankar, 2010] Shankar, A. (2010). *Virtual Machine Placement in Computing Clouds.* Technical Report. 23 p.

[Singh et al., 2008] Singh, A., Korupolu, M., Mahapatra, D. (2008). Server-Storage Virtualization: Integration and Load Balancing in Data Centers. *Proceedings of the IEEE/ACM Supercomputing.* 12 p.

[Smith and Nair, 2005] Smith, J. E., Nair, R. (2005). The Architeture of Virtual Machines. *Computer,* 38 (5), p. 32-38.

[Song et al., 2008] Song, Y., Zhang, C., Fang, Y. (2008). Multiple Mutidimensional Knapsack Problem and its Applications in Cognitive Radio Networks. *Military Communications Conference.* Florida, p. 1-7.

[Tanenbaum, 2001] Tanenbaum, S. A. (2001). Sistemas Operacionais Modernos. In: *Redes de Computadores.* 2ª. ed. Trad. Vandenberg D. de Souza, Capítulo 1, p. 18-76.

[Torres e Lima, 2005] Torres G. e Lima C. (2005). Como Funciona a Tecnologia de Virtualização da Intel, *Portal Clube do Hardware.* Disponível em: <http://www.clubedohardware.com.br/printpage/Como-Funciona-a-Tecnologia-de-Virtualizacao-da-Intel/1144, março , 2011>.

[Toscani, 2011] Toscani, S. (2011) Revisão de arquitetura de computadores. *PUCRS, notas de sala de aula.* Disponível em: <http://www.inf.pucrs.br/~ramos/sop_textorevisarq.pdf, abril, 2011>.

[Wine, 2011] (2011) Documentação sobre o Wine, *Site do Wine Wiki. Disponível em:* <http://wiki.winehq.org, abril, 2011>.

[Whitaker et al., 2002] Whitaker, A., Shaw, M., Gribble S. D. (2002). Denali : A Scalable Isolation Kernel. *Proceedings of the* $10^{th}$ *Workshop on ACM SIGOPS European Workshop.* New York, p. 10-15.

[Wu et al., 2012] Wi, Y., Tang, M., Fraser, W.L. (2012) A Simulated Annealing Algorithm for Energy Efficient Virtual Machine Placement. *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics.* Seoul, Korea, p. 14-17.

[Xen, 2011] (2011) Xen Cloud Plataform. *Site oficial.* Disponível em: <http://xen.org/products/cloudxen.html, julho, 2011>.

[Xu and Fortes, 2010]. Xu, J., Fortes, J. A. B. (2010). Multi-Objective Virtual Machine Placement in Virtualized Datacenters Environments. *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing.* Washington, p. 179-188.

[Zhang et al., 2010] Zhang, Q., Cheng, L., Boutaba, R. (2010) Cloud computing: State-of-art and Research Challenges. *Journal of Internet Services and Applications,* volume 1, issue 1, p. 7-18.

# Anexo I Arquitetura de *hardware* para máquinas virtuais

[Popek e Goldberg, 1974] foram os primeiros a descrever formalmente os requisitos mínimos de *hardware* que um computador de terceira geração necessita para suportar máquinas virtuais. Um computador de terceira geração segue a arquitetura Von Newmann com CPU (unidade lógica e aritmética + circuito de controle), memória principal, dispositivos de E/S, barramento de dados, endereços e de controle.

Existe um artifício que é e importante para a construção de máquinas virtuais que se chama interrupção. Sistemas operacionais e hipervisores de máquinas virtuais utilizam o mecanismo de interrupção para várias finalidades, entre elas tratar erros, controlar periféricos, e escalonar processos. A Figura 25 mostra o modelo da arquitetura Von Newmann, juntamente com a chegada de uma interrupção de *hardware* oriunda da controladora de rede. [Maziero, 2011] descreve o mecanismo de interrupção como: o processador está executando um programa (1). Chega um e-mail pela rede (2). Através do barramento de controle, a controladora de rede informa o processador que necessita de uma interrupção (3). O processador interrompe o programa em execução e usa o barramento de endereços para achar a rotina de tratamento daquela interrupção (4). As instruções da rotina de tratamento são lidas e executadas (5). Terminando a rotina de tratamento, o programa volta a executar normalmente no ponto em que parou (6).

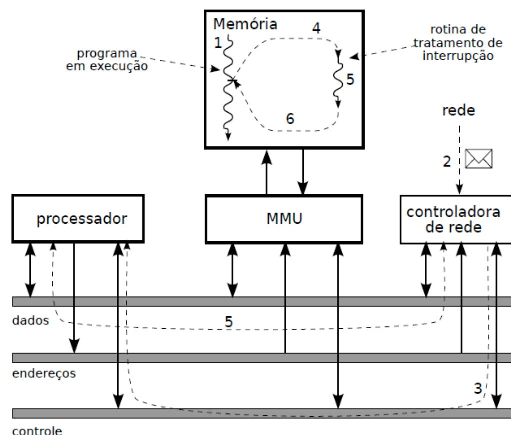



**Figura 25**. Arquitetura Von Newmann e uma interrupção. [Maziero, 2011]

Apesar de que a definição de alguns detalhes mude de um sistema pra outro, podemos falar genericamente que a interrupção é usada para interromper o funcionamento do processador quando necessário. Quando ativada, uma interrupção fará o processador salvar o seu estado atual (gravando este estado em uma região especial da memória chamada de pilha, o valor dos registradores internos da CPU) e irá processar aquela interrupção específica. Terminando o processamento da interrupção, o estado do processador é retornado ao contexto do programa que estava executando anteriormente (carregando os valores dos registradores que anteriormente foram salvos na pilha na CPU) podendo continuar o processamento da tarefa que havia sido interrompida exatamente no ponto em que havia parado. Os fabricantes de computadores descrevem o fenômeno da interrupção de três formas distintas: *exceptions* (exceções), *traps* (armadilhas) e *interrupts* (interrupções). Apesar dos termos serem usados para descrever fenômenos parecidos, existe algumas sutis diferenças entre eles [Hyde, 1996].

A palavra *interrupt*, normalmente é usada para descrever interrupções de *hardware*. O controle de acionar uma interrupção é baseado em algum evento externo ao CPU. Estes tipos de interrupções em uma forma geral não tem relação com as instruções da CPU e são ativadas por eventos de *hardware*, exemplos: pressionar uma tecla no teclado, movimentar o ponteiro do *mouse*, *timeout* de relógio, ou qualquer outro evento gerado pelo *hardware* necessitando notificar a CPU que aquele dispositivo precisa de sua atenção. *Trap* é um mecanismo muito parecido com as interrupções, também referida por alguns autores como interrupções de *software*. Uma *trap* é um desvio de execução de código incondicional. Um exemplo prático

pode ser dado pela arquitetura x86, onde que a instrução "*int"* é o principal veículo para a execução de *traps*. Toda vez que "*int*" é chamada, o controle será transferido para a rotina de código associada à *trap*. É interessante notar que *traps* são sempre ativadas por uma instrução explícita. Uma *exception* é uma *trap* gerada automaticamente que ocorre devido a uma condição excepcional. Não existe uma instrução específica associada a *exception*. Programas podem gerar *exceptions* quando há algum desvio de conduta em seu comportamento tais como: divisão por zero, acesso ilegal de memória, execução de instruções ilegais. Quando a *exception* ocorrer, a CPU suspende imediatamente a execução do programa e transfere a controle para o código do *exception handler* [Hyde, 1996].

Uma "rotina de interrupção" é um procedimento para executar comandos referentes a uma *interruption, trap* ou *exception*. Apesar dos estímulos que causam *interruptions, traps* ou *exceptions* serem classificados de forma diferente, o formato da rotina de tratamento é o mesmo. Para saber o lugar na memória onde se encontram as diversas rotinas de interrupções, o sistema operacional reserva uma área chamada de vetor de interrupção. Em uma arquitetura 80X86 é possível ter 256 interrupções alojadas neste vetor (O vetor de interrupção pode não existir, dependendo da arquitetura utilizada, neste caso o controle é transferido diretamente para uma rotina de tratamento). Este vetor inicia na posição de memória 0:0 para a primeira interrupção, e possui 256 entradas de quatro bytes cada. Os bytes de 0:4 correspondem ao endereço da primeira interrupção, os de 4:8 se referem à segunda interrupção e assim por diante.

[Hyde, 1996] descreve o funcionamento do mecanismo de interrupção:

a) A CPU recebe a solicitação de interrupção (que é referenciada por um valor numérico) e armazena o registrador de *flags* na pilha (para isso conta com um registrador auxiliar chamado *stack pointer* (SP) que indica o topo da pilha).

b) A CPU coloca o endereço de retorno na pilha. Este endereço é apontado pelo *program counter* (P).

c) A CPU determina qual foi a interrupção de acordo com o seu número (em hexadecimal). Como cada entrada no vetor de interrupção tem 4 *bytes* (para a arquitetura X86), o vetor de interrupções é acessado, multiplicando o valor da interrupção por 4, obtendo o valor inicial do endereço da interrupção e a partir deste ponto os quatro próximos *bytes* serão o endereço da rotina de tratamento da

interrupção. O controle de programa passa para o endereço da rotina apontada pelo vetor de interrupções.

d) A rotina de interrupção assume o controle da CPU, executa seus códigos e por fim usa a instrução IRET (retorno de interrupção), retornando os valores armazenados na pilha para os registradores da CPU. O controle é devolvido ao programa que estava sendo executado anteriormente.

[Moore, 1997] complementa, afirmando que interrupções podem ter prioridades: "toda vez que o processador executa uma instrução, ele primeiro verifica se existe alguma interrupção pendente e, se existir, ele seleciona aquela com prioridade mais alta e executa o código de tratamento".

Para poder executar máquinas virtuais (VM de sistemas), além de ter mecanismos de interrupções, o processador deve obrigatoriamente possuir dois modos de operações: o modo usuário e o modo supervisor. O modo supervisor pode executar todo o conjunto de instruções disponíveis, o modo usuário não. Além disso, todo endereçamento de memória deve ser feito de forma relativa, com o uso de um registrador de relocação de endereços. Cumprindo estas exigências, a máquina terá um número finito de estados onde cada estado possui quatro componentes fundamentais [Popek e Goldberg, 1974]:

a) Armazenamento executável (E): É a palavra ou *byte* endereçado na memória. Nesta região de memória ficam as instruções do programa.

b) Modo processador (M): Pode ser modo usuário ou supervisor.

c) Contador de programa (P): que indicará a próxima instrução que será executada.

d) Registrador de relocação de endereços (R): Determina as faixas válidas de memória para o programa em execução, o esquema de um registrador de relocação de endereços dentro da *memory managment unit* (MMU) está representado na figura 25.

Estes estados colocados por [Popek e Goldberg, 1974] trabalham juntamente com o mecanismo de interrupção explicado anteriormente. Quando um programa (gravado no armazenamento executável E) no nível do usuário é interrompido, seu contexto é salvo no topo da pilha indicado pelo registrador SP (*stack pointer)*. Os dados que preservam o contexto se encontram respectivamente nos registradores PSW (*porgram status word*) e P (*program conter*). O modo do processador é chaveado para privilegiado (modo supervisor) e os registradores PSW e P são empilhados. A rotina do tratamento da interrupção é executada,

mas para ela funcionar corretamente, altera-se o estado dos registradores internos de acordo com seu contexto. Por fim a instrução IRET (retorno de interrupção) é executada, e os valores obtidos pelo retorno da interrupção são escritos no espaço de memória do usuário (caso seja necessário). Juntamente com a chamada, a IRET os registradores P e PSW recebem novamente o valor de contexto do programa que sofreu a interrupção, retirando os valores da pilha e fazendo a CPU retornar a execução do programa original (chaveando para modo do processador para usuário novamente), exatamente no ponto em que havia sido interrompido, pois esta informação foi recuperada do contador de programa (P). A figura 26. mostra o contexto de um programa sendo salvo na pilha.

Obs: O registrador PSW contém as *flags* que indicam o estado atual da CPU. [Popek e Goldberg, 1974] afirmam que estas *flags* indicam os estados de M, P e R.

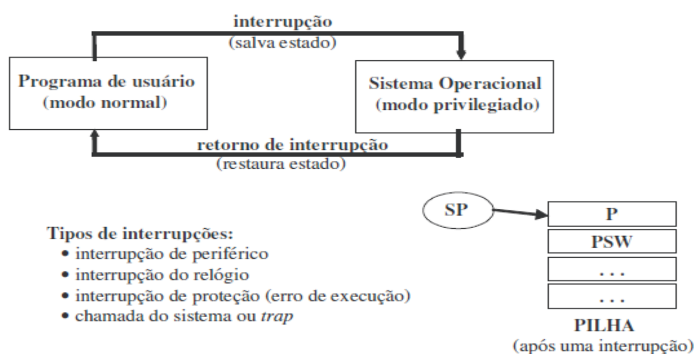



**Figura 26**. Mecanismo de interrupção [Toscani, 2011].

Para ser possível executar hipervisores e sistemas operacionais multiusuários com segurança, o processador precisa ter os dois modos de operação descritos por Popek e Goldberg, mas porque isso? Quando os *bits* do modo do processador (M) são chaveados para usuário, uma parte das instruções da CPU é desabilitada, porque estas instruções são consideradas perigosas e podem trazer desequilíbrio ao sistema, se forem usadas de forma errada. Estas instruções perigosas recebem o nome de “instruções privilegiadas” e devem ser utilizadas apenas pela camada de *software* que controla o *hardware* diretamente, a saber, o núcleo de um hipervisor ou o *kernel* de um sistema operacional. Para monitorar se o programa no nível do usuário tenta acessar alguma instrução proibida são necessárias *traps*, desta forma, se algum programa em modo usuário tentar executar uma instrução não permitida (apontada por P) terá a ação capturada por uma *trap* de “operação ilegal”. Um tipo de interrupção também é usado para monitorar tentativas de acesso à memória, caso o programa provoque uma violação de acesso fora do *range* delimitado pelo registrador de relocação de endereços

(R), a interrupção entra em ação mudando o modo do processador para supervisor e passando o controle para a rotina de tratamento especificada pelo vetor de interrupções do *kernel*.

A Figura 27 mostra o registrador de relocação de endereços dentro da MMU (*memory control unit*) e também uma interrupção de memória gerando uma requisição de interrupção para o processador (IRQ), caso o range de memória acessado seja inválido. O registrador de relocação de endereços é uma das *peças-chave* fundamentais na arquitetura de um processador para tornar possível a virtualização e a proteção de memória.

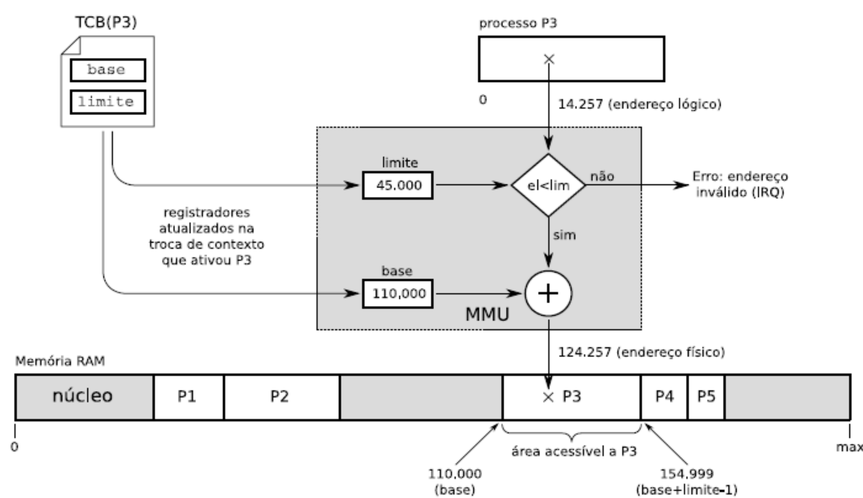



**Figura 27**. Funcionamento da MMU e do registrador de relocação de endereços [Maziero, 2011].

O registrador de relocação de endereços nos primeiros computadores de terceira geração eram muitos parecidos com o modelo exemplificado na figura 27 Com o passar dos anos, a maneira como este registrador é implementado foi mudando, sendo pertinente dar uma revisada no que aconteceu para verificar se o que Popek e Goldberg escreveram na década de 70 ainda é válido para os dias de hoje.

Analisando mais a fundo o primeiro modelo de MMU na figura 27 encontramos a sigla TCB que significa "*task control block*". O TCB é uma estrutura que cada processo deve ter na memória, onde os dados sobre o processo são armazenados permitindo a troca de contexto. A cada troca de contexto é lido do TCB dois inteiros que são carregados nos registradores de relocação de endereços "base" e "limite" (*offset*). A base corresponde ao endereço inicial na memória RAM que o processo poderá ocupar. O limite se traduz pelo deslocamento que contém o último endereço válido a partir da base. Os endereços base e

limite correspondem ao *range* de memória real que o processo pode ocupar. Para determinar se um endereço lógico, que o processo deseja acessar, é valido ou não, o método é bastante simples: O endereço lógico é somado à base real da memória do processo. A soma nunca pode ultrapassar o valor da base + limite. Caso um endereço lógico ultrapasse os limites reais de memória, a MMU irá gerar uma requisição de interrupção (IRQ) e uma rotina de tratamento irá ser acionada. Esta arquitetura de *hardware* é denominada de alocação contígua e tem como principal vantagem a simplicidade, mas não é possível definir políticas diferentes para pedaços de memória [Maziero, 2009].

Para resolver as limitações da técnica de alocação de memória contígua surge um novo modelo de MMU chamado de memória segmentada. O modelo representado pela Figura 28 mostra um diagrama de como a segmentação de memória funciona. Para melhor organizar a memória, o espaço do processo é dividido em pedaços possibilitando políticas específicas para vetores, matrizes, pilhas de *threads*, etc. Na memória segmentada o TCB deve ter uma tabela de segmentação válida para o processo. Nesta tabela os segmentos são associados a suas respectivas posições na memória real. Quando o processo tenta acessar uma posição lógica de memória, a MMU, em primeiro lugar, verifica o número do segmento. Com este número ela poderá checar a tabela referente àquele segmento no TCB, identificando a base e limite na memória real daquele segmento. Se o deslocamento lógico, solicitado pelo processo, para aquele segmento, somando a base do endereço real apontada pelo TCB, referente ao mesmo número de segmento, não extrapolar os limites de segmentação reais da base + deslocamento, o acesso àquela região de memória é legal. Caso o acesso seja ilegal, extrapolando os limites de segmentação, uma interrupção será ativada tratando a violação de acesso à memória, gerando uma IRQ. O modelo de MMU segmentada é mostrado na Figura 28 [Maziero, 2009].

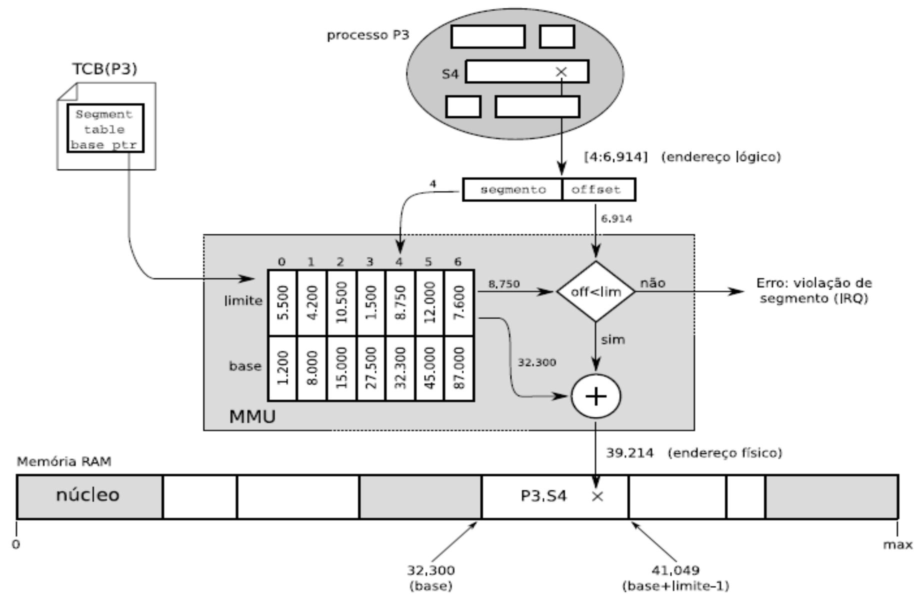



**Figura 28**. Funcionamento da MMU com segmentação de memória [Maziero, 2009].

A principal vantagem do uso de memória segmentada é a possibilidade de implantação de políticas para os diferentes segmentos de um processo, no entanto, perde-se a vantagem da simplicidade. A principal desvantagem comum às técnicas de alocação contígua e segmentada é a fragmentação externa. À medida que os espaços vão sendo alocados e excluídos, surgem buracos na memória, o que é uma consequência da utilização de blocos de tamanho variável.

Para resolver o problema da fragmentação externa, os processadores passaram a usar alocação de memória paginada, em que os blocos físicos e lógicos possuem sempre o mesmo tamanho, acabando totalmente com o problema da fragmentação de memória. Este tamanho de página normalmente é de 4096 bytes, mas pode ser diferente. Uma página na memória real é chamada de quadro (*frame*). Para relacionar as páginas lógicas dos processos com os quadros na memória real é usada uma tabela que fica armazenada no TCB. O endereço usa um número determinado de *bits* para determinar o deslocamento dentro da página, geralmente os menos significativos, enquanto os *bits* mais significativos determinarão qual a página que está sendo utilizada. [Maziero, 2009] dá o exemplo de um endereço lógico de 32 *bits*, sendo que os primeiros 12 *bits* representam o *offset* dentro da página totalizando $2^{12}$ *bits* que representam uma posição entre 0 e 4.095 na página. Os 20 *bits* restantes são o número da página utilizada, com isso podem ser endereçadas até $2^{20}$ páginas, totalizando 1.048.575 endereços, onde cada um deles corresponde a uma página. A MMU, neste caso, precisará separar o endereço lógico em duas partes, verificando o número da página e do *offset*, depois

precisará comparar o número da página com a tabela no TCB, pegando o número do quadro correspondente. Por fim, irá somar o *offset* do endereço lógico ao valor base do início do quadro, verificando se o *range* de memória acessado é válido, verificando se o *offset* lógico somado à base real naquele quadro não é superior à soma da base com os limites reais do quadro. Caso o endereço seja inválido, deverá gerar uma interrupção passando o controle à rotina de tratamento do sistema operacional. A Figura 29 mostra com detalhes como funciona uma MMU, utilizando o recurso da paginação.

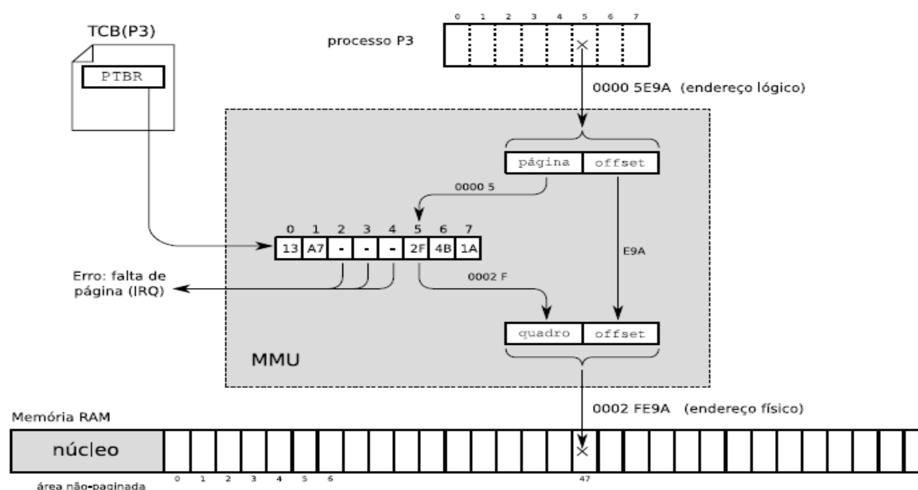



**Figura 29**. MMU com paginação de memória [Maziero, 2009].

É relevante salientar que ainda existem outras técnicas. Citando algumas delas rapidamente, temos a mistura das técnicas de segmentação como a paginação, sendo possível aplicar políticas a segmentos paginados. Nos processadores mais modernos surgiu um poderoso recurso de *hardware* que é o TBL ou *translation lookaside buffer*, que é um *cache* de alta velocidade dentro da MMU, com a função de referenciar os espaços de memória utilizados com maior frequência, evitando novas consultas desnecessárias à tabela de endereços no TCB. As Tabelas multi-níveis são outra técnica de alocação de memória que podem evitar o desperdício de espaço quando temos uma MMU paginada, pois as tabelas de páginas no TCB podem ocupar muito espaço, fazendo o uso do artifício da divisão de tabelas minimizando perdas [Maziero, 2009].

Todas as técnicas de *hardware*, aqui estudadas, servem para conseguir a proteção de memória, umas de uma maneira mais eficiente e outras menos, mas é importante perceber que todas elas usam o mesmo conceito base que é o registrador de relocação de endereços. Após

estudar os detalhes sobre o funcionamento de interrupções, da CPU e da MMU, podemos entender por dedução que a exigência de Popek e Goldberg ainda é satisfatória para CPUs modernas que usam técnicas de segmentação, paginação de memória, entre outras. Mesmo com mudanças na arquitetura da controladora de memória do processador, o principio básico e fundamental para a construção de máquinas virtuais não mudou, pois as técnicas antigas e modernas validam o endereço lógico de um processo, baseando-se no *range* de endereços reais que o processo poderá ocupar.

Sabemos que hoje em dia possuímos computadores modernos com registradores de relocação de endereços utilizando técnicas avançadas, possuindo dois ou mais modos de operações distintos, com suporte a interrupções e uma velocidade de processamento muito maior que qualquer computador da década de 70. Embora tudo isso seja verdade, não garante que seja possível construir uma máquina virtual para este *hardware*. [Popek and Goldberg, 1974] enunciam o seguinte teorema: "Para qualquer computador de terceira geração é possível construir um programa de controle (supervisor) que obedece as três propriedades clássicas das máquinas virtuais (que serão vistas em detalhes mais adiante), desde que o conjunto de instruções sensíveis seja um subconjunto de suas instruções privilegiadas".

Para entender este teorema, podemos relembrar a ideia clássica de virtualização: fazer com que os recursos de *hardware* sejam multiplexados. Multiplexar significa que o tempo do processador e os demais recursos do *hardware* estão sendo divididos entre as diversas instâncias de sistemas virtualizados, permitindo transformar uma máquina real na ilusão de várias máquinas virtuais. As instruções consideradas privilegiadas podem alterar os estados da CPU e modificar a alocação de memória dos recursos compartilhados que fazem parte deste sistema virtual. Se uma máquina virtual começar a utilizar instruções sensíveis e invadir espaços de memória da outra ou de seu supervisor, o sistema tornar-se-á caótico e a virtualização perderá totalmente o seu propósito.

Como já vimos, as instruções privilegiadas necessitam de *traps* para protegê-las da tentativa de execução em modo usuário, garantindo o funcionamento correto das aplicações. Usarei a abreviação $M_1$ para simbolizar o modo supervisor do processador e $M_2$ para o modo usuário. A letra S representará um estado, letra i uma instrução, a letra E o armazenamento executável, a P o contador de programa e a letra R o registrador de relocação de endereços. Uma instrução é privilegiada se, e apenas se, em qualquer par dos estados $S_1=\{E,M_1,P,R\}$ e $S_2=\{E, M_2,P,R\}$ no qual a instrução apontada pelo registrador P (contador de programa) nos

estados $S_1$ e S2 não são capturadas por uma interrupção de memória. Entretanto a instrução $i(S_2)$ é capturada por uma *trap* e a instrução $i(S_1)$ não. Em outras palavras, os estados $S_1$ e $S_2$ apenas se diferenciam pelo modo em que o processador esta atuando. A instrução que ambos estados tentam executar é a mesma. Em modo supervisor é possível a execução da instrução, mas em modo usuário não [Popek and Goldberg, 1974].

Além das instruções privilegiadas, existe outro grupo especial de instruções, chamadas de sensíveis. Uma instrução sensitiva pode ser privilegiada ou não. As instruções sensíveis se subdividem em dois tipos. O primeiro tipo de instrução sensitiva é a "instrução sensitiva ao controle". Em uma definição mais formal, este tipo de instrução inicia com a máquina em um estado $S_1 = \{E_1,M_1,P_1,R_1\}$ é executada e termina com a máquina em um estado $S_2 = \{E_2,M_2,P_2,R_2\}$ sem causar uma *trap* de memória e também, obrigatoriamente, devemos ter $R_1!=R_2$ ou $M_1!=M_2$ ou ambos. Em outras palavras, quando a instrução tenta mudar o tamanho do *range* de memória permitida a um processo em execução, alterando o registrador de relocação de endereços, sem acontecer uma *trap* de memória, ou quando a instrução afeta o modo de funcionamento do processador, dizemos que a instrução é sensitiva ao controle. As propriedades clássicas de máquinas virtuais (estudadas em detalhes mais adiante) afirmam que um monitor de máquina virtual deve ter o controle total dos recursos de *hardware*, logo podemos também afirmar que o hipervisor deve ser o único programa com acesso a instruções sensíveis ao controle [Popek and Goldberg, 1974].

O outro tipo de instrução sensitiva é a instrução sensitiva ao comportamento. A execução desta instrução depende dos valores carregados no registrador de relocação de endereços, ou do modo de operação do processador, ou de ambos. Como vimos anteriormente, no funcionamento da MMU clássica, o valor do registrador de relocação de endereços é uma "tupla". A primeira parte define um endereço, marcando a base inicial de memória real válida para execução do processo que chamaremos de I. A segunda parte define o endereço de memória que será somado ao endereço da base para delimitar o fim do espaço de memória real que o processo poderá ocupar, a qual chamaremos de F. Resumindo, I é fronteira do primeiro endereço válido e I+F é a fronteira do último endereço válido, estas informações são carregadas da tabela do *task control block* de cada processo. Sendo assim, o registrador de relocação de endereços representado por $R=\{I,F\}$ delimita E (o espaço de memória executável), cuja notação será $E|R$ (leia E em função de R). $E|R$ correspondente ao espaço de memória de I até I+F. Vamos adicionar um inteiro X em nossa análise o qual é um inteiro. Agora o operador I pode ter sua base alterada pelo inteiro X, podemos escrever que $R_1$

= {I,F} e $R_2$ = {I+X,F}. A instrução indicada por P onde existe um estado $S_1$={E|R,$M_1$,P,R} não é idêntica à instrução indicada por P no estado $S_2$={E|R+X,$M_2$,P,R+X}, esta instrução depende do valor que é carregado no registrador de relocação de endereços, ou seja, do *range* de endereçamento real de memória. Um exemplo deste tipo de instrução é o caso da instrução LRA (*load real adress*), dependendo do limite especificado por R, ela poderá operar em modo 24 *bits* ou modo 31 *bits* [IBM, 2011]. Este tipo de instrução se encaixa em um subconjunto de instruções comportamentais chamadas de sensíveis à localização. Existe outro tipo de instrução sensível ao comportamento que é classificada como instrução sensível ao modo. Um exemplo de instrução sensível ao modo poderia ser mover a instrução de um espaço anterior MFPI, a qual depende do modo que está em vigor no processador [Popek and Goldberg, 1974].

Apesar do teorema de Popek e Goldberg ser válido, hoje em dia ele não impede mais a virtualização de máquinas com instruções não privilegiadas sensíveis, pois existem técnicas como a tradução dinâmica e a para-virtualização que resolvem este problema, embora tenha perda de *performance* (no caso da tradução dinâmica) e necessidade de modificação do código fonte no sistema convidado, como é o caso da para-virtualização [Laureano e Maziero, 2008].

Para ilustrar a importância do teorema clássico de Popek e Goldberg, temos processadores em que não é possível a virtualização total, como é o caso do Pentium que possui 17 instruções sensíveis não privilegiadas [Robin and Irvine, 2000]. Uma dessas instruções é a POPF (*pop* CPU *flags form stack*) que retira as *flags* da CPU da pilha para controlar e desabilitar as interrupções. Quando esta instrução é executada em modo não privilegiado, ela não é capturada por uma *trap* e pode trazer sérios problemas para outros programas, executando na mesma máquina. Outro problema do Pentium é que instruções não privilegiadas conseguem acessar o registrador de segmentação de código, determinando o nível de privilégio do processador [Garfinkel and Rosenblum, 2005].

Os microprocessadores da arquitetura X86 tem uma particularidade referente ao modo de operação (M). Eles possuem quatro modos de operação conhecidos como *rings* (anéis) que vão do 0 até o 3. O *ring 0* é o mais alto grau de privilégio, este *ring* geralmente é usado pelo *kernel* do sistema operacional. Aplicações usam o *ring 3* que é o menos privilegiado. Os *rings* 1 e 2 não são mais usados por quase nenhum sistema operacional (o últimos foram o OS/2 e o MULTICS). Os *rings* 1 e 2, que não são mais utilizados, podem fornecer a granularidade de

permissões para execução de instruções que as máquinas virtuais necessitam. Um exemplo disto pode ser dado pelo sistema de virtualização Xen, que utiliza o *ring 1* para executar o sistema operacional convidado. Qualquer sistema operacional que use o *ring 0* no modo *kernel* poderá ser portado para executar como uma máquina virtual sob a supervisão do VMM do Xen, desde que mude seu modo de execução para o *Ring 1.* Assim, o Xen consegue limitar melhor o uso de instruções pelo sistema operacional convidado, colocando o sistema convidado em um nível mais privilegiado do que as aplicações convidadas ordinárias, mas menos privilegiado que o hipervisor. A Figura 30 mostra como a virtualização pode ser feita utilizando os anéis de privilégios da arquitetura X86 [Barham et al., 2003].

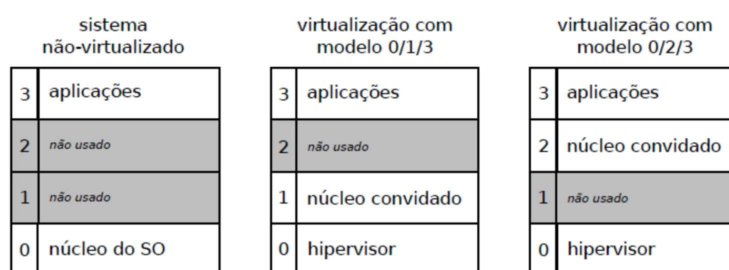



**Figura 30**. Virtualização utilizando os níveis de privilégios da arquitetura X86 [Maziero, 2009].

Processadores mais modernos possuem suporte à virtualização em *hardware* diretamente no processador. Exemplos desta tecnologia são o *intel virtualization tecnology* e o *AMD Virtualization*, ambos são equivalentes no conceito tecnológico [Laureano e Maziero, 2008].

Os processadores da Intel com suporte a virtualização por *hardware* surgiram com o Pentium 4, modelos 672 e 662. Estes processadores fornecem uma interface chamada de VMX (*virtual machine extension*) que é uma extensão da ISA (conjunto de instruções ISA, Capítulo 3) que serve para utilizar o *Ring 0* de uma forma mais inteligente quando existe um hipervisor de máquinas virtuais. No total 10 novas instruções foram adicionadas pela interface VMX, entre elas as mais importantes são: VMLAUCH,VMRESUME,VMXOFF e VMXON. A tecnologia divide o *Ring 0* em dois modos de execução chamados de *root* e de *não-root*. O hipervisor executa em modo *root*, enquanto as máquinas virtuais convidadas trabalham em modo *não-root*. A inicialização da virtualização é feita executando a instrução VMXON, chamando o hipervisor que quando achar necessário executa a instrução VMLAUNCH,

fazendo com que o processador entre no contexto da máquina virtual desejada (modo não *root*). A instrução VMRESUME retorna o controle ao hipervisor (modo *root*). Caso o hipervisor deseje parar todas as máquinas virtuais, pode executar a instrução VMXOFF. A Figura 31 mostra um esquema prático, utilizando a tecnologia da Intel. Estas novas instruções trazem velocidade para a virtualização, pois o controle feito pelo hipervisor se torna mais simples [Torres e Lima, 2005].

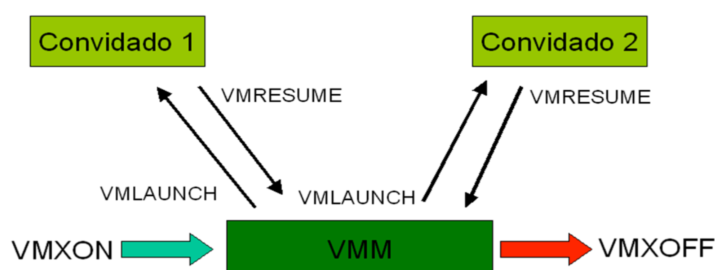



**Figura 31**. Instruções de um processador Intel com suporte a virtualização [Torres e Lima, 2005].

[Maziero, 2009], na Figura 32, mostra o conceito dos *Rings* de privilégio da arquitetura X86 em conjunto com a tecnologia da virtualização por *hardware*. Podemos notar, que com esta tecnologia, uma grande vantagem é que o sistema operacional convidado não necessita ser modificado para executar em outros *rings* de privilégios. O *ring* 0 passa a ser flutuante, ora sob domínio do hipervisor, ora sob domínio do sistema operacional convidado. Ainda na Figura 32, nota-se uma estrutura chamada de VMCS (*virtual machine control structure*), a qual possui 2 áreas distintas. A primeira delas é usada apenas para salvar o contexto do hipervisor, e outra é usada para salvar o contexto dos sistemas virtualizados. O VM*entry* (VMLAUNCH) salva o estado do hipervisor no VMCS e, logo após, faz o processador ler os dados de contexto referentes à máquina virtual que está sendo carregada. O VM*exit* (VMRESUME), salva o estado da máquina virtual em sua devida região de memória dentro do VMCS e faz com que o processador leia os dados de contexto no VMCS, referentes ao último estado salvo do hipervisor, que irá retomar o controle.

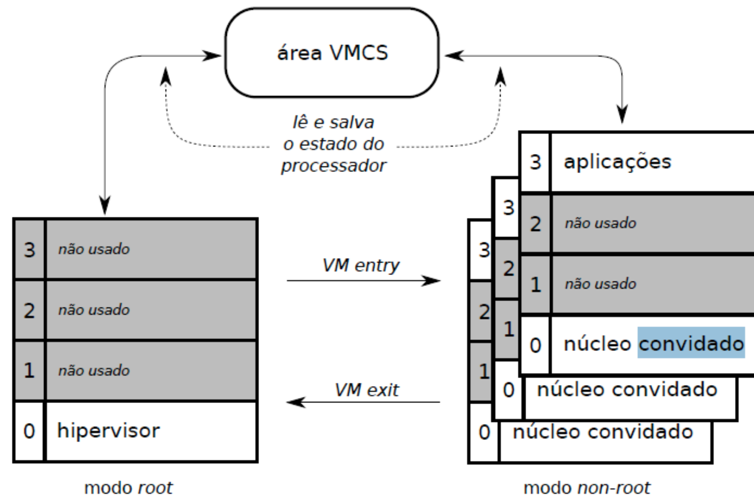



**Figura 32**. Modo *root* e não *root*, virtualização em hardware [Maziero, 2009]

Finalizando este subcapítulo, podemos concluir que para ser possível construir máquinas virtuais, em uma máquina de terceira geração, precisamos, segundo [Robin and Irvine, 2000], no mínimo:

a) Ter um processador com, no mínimo, dois modos de operação (usuário e supervisor) e de preferência com tecnologia de virtualização por *hardware.*

b) Um método para programas não privilegiados chamarem rotinas privilegiadas.

c) Um sistema de relocação e proteção de memória, como a segmentação ou a paginação.

d) Interrupções para tratamento de exceções de *hardware* e comunicação com dispositivos de entrada e saída.

e) As instruções sensíveis da CPU devem ser um subconjunto das instruções privilegiadas [Popek and Goldberg, 1974]. Caso não sejam, um tratamento adequado deve ser feito para que a camada de controle da virtualização intercepte estas instruções, não permitindo que estas instruções sejam acessadas diretamente através de aplicações acima da camada de virtualização na pilha de *software* (tradução dinâmica), ou sistemas convidados precisam ser modificados (para-virtualização).

# Anexo II Classificação de máquinas virtuais

A classificação de máquinas virtuais é derivada do tipo de programa convidado executando nas interfaces virtualizadas, e são divididas em dois grupos: VM de processos e VM de sistema ilustrado pela Figura 33. Máquinas virtuais de processos (também chamadas de máquinas virtuais de aplicação) possuem o único objetivo de executar uma aplicação comum e sempre são executadas em um sistema operacional hospedeiro que pode ser virtualizado ou não. Elas iniciam com a criação do processo e terminam com a sua extinção. Para o sistema operacional *host*, elas são um processo como qualquer outro, e não possuem nenhum privilégio especial. A camada de virtualização pode permitir que a aplicação convidada se comunique com outras aplicações através dos mecanismos de comunicação de processos (IPC – *interprocess communication*). Processos virtualizados podem ainda fazer uso do sistema de arquivos e dispositivos de E/S, desde que não sejam restringidos pelo S.O. *host,* o qual tem o domínio total da máquina virtual. As máquinas virtuais de sistema fornecem uma interface para um sistema operacional completo que podem ainda ter vários processos executando em seu espaço de usuário [Laureano e Maziero, 2008].

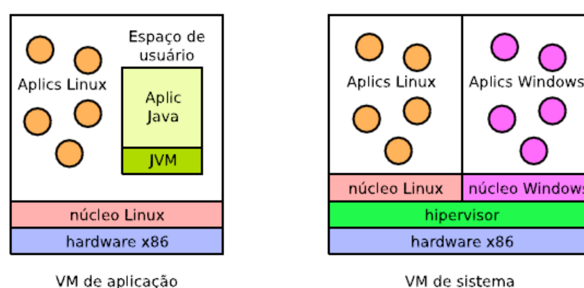



**Figura 33**. Classificação de máquinas virtuais de acordo com a aplicação convidada**.** [Laureano e Maziero, 2008].

Outra classificação possível para máquinas virtuais (Figura 34), é feita de acordo com a interface, que respeitam e exportam nesta classificação, a interface poder ser totalmente diferente, parcialmente diferente ou até mesmo igual a interface que é respeitada (exportar um camada igual a ABI do S. O *host* pode ser considerado um *overhead* desnecessário, mas,

fazendo isso, ganha-se todas vantagens comuns de propriedades de máquinas virtuais) [Smith and Nair, 2005].

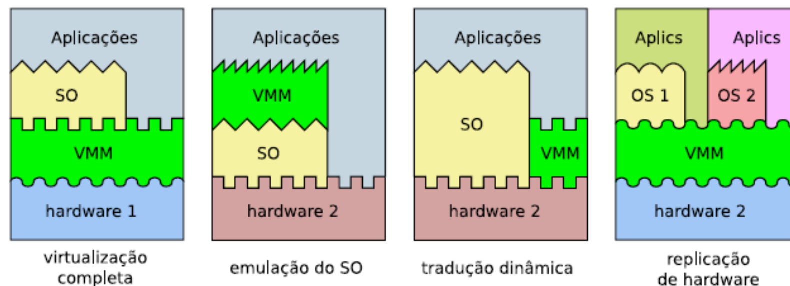



**Figura 34**. Máquinas virtuais classificadas de acordo com a interface respeitam e que exportam [Laureano e Maziero, 2008].

As máquinas virtuais de sistema são, ainda, subdivididas em tipo 1 e tipo 2, de acordo como a interface sob a qual o programa de controle é construído. No tipo 1 (Figura 35. A), a camada de hipervisor fica entre a interface ISA e os sistemas convidados. Já no tipo 2 (Figura 35. B), o hipervisor é construído em cima de um sistema operacional, desta forma é possível aproveitar algumas abstrações do S.O. *host* como, por exemplo, o sistema de arquivos, simplificando muito a implantação e administração das máquinas virtuais. Obviamente, as máquinas virtuais do tipo 2 sofrem um *overhead* maior, devido ao fato de possuírem uma camada a mais na pilha de *software*. Sob o ponto de vista de interfaces, a virtualização do tipo 1 tem um hipervisor, utilizando a interface ISA, o tipo 2 utiliza a ABI.

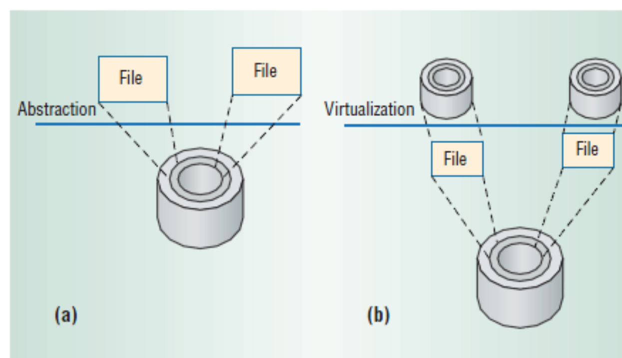



**Figura 35**. Uso da abstração facilitando a virtualização [Smith and Nair, 2005].

[Jones, 2006] classifica alguns tipos especiais de máquinas virtuais: emulação de *hardware*, virtualização total, para-virtualização, virtualização no nível do sistema operacional e virtualização no nível de bibliotecas.

**Emuladores de *Hardware*:**

[Barrio, 2001] faz um estudo detalhado sobre Emuladores de hardware. Este tipo de máquina virtual pode tanto servir para executar sistemas operacionais, como para executar processos. Eles são construídos aproveitando abstrações do sistema operacional *host,* que pode ser virtualizado ou não, ou diretamente sob a ISA do *hardware* real. Eles funcionam instanciando uma ou mais máquinas virtuais, as quais são criadas para interceptar todas as instruções direcionadas ao *hardware* (Figura 36). Uma emulação pode ser de 100 a 1000 vezes mais lenta que um sistema nativo, pois ela simula detalhes da CPU e GPU (processador gráfico) como os *pipelines* e comportamentos de *cache*, além de emular outros periféricos. As instruções que passam pelo emulador são analisadas e modificadas para que a ABI ou ISA subjacente possa executá-las. Uma vantagem oferecida pela emulação é poder executar diferentes sistemas operacionais sem necessidades de modificações, simulando os mais diversos tipos de *hardware.* Outro uso interessante para emuladores é desenvolver e testar *firmwares* para *hardwares* que ainda estão em fase de construção. Emuladores também são muito usados para fazer engenharia reversa de aplicações e *drivers*. O Bosh, QEMU, PCSX2 e o Transmeta Crusoe são exemplos de emuladores.

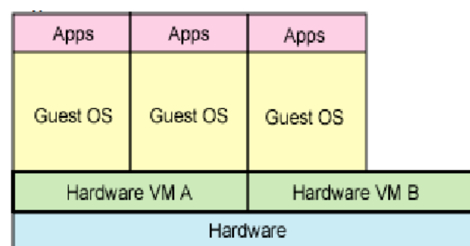



**Figura 36**. Emulação de *hardware* [Jones, 2006].

O Bosh é um simulador *open source* de arquitetura X86 escrito em c++. Ele simula processadores 386, 486, Pentium, Pentium II, Pentium III, Pentium IV, X86-64 com instruções MMX, SSEx e 3DNow. É um emulador altamente portável, que pode ser

executado nas mais diversas combinações de sistemas operacionais e plataformas de *hardware*, tais como: Solaris executando em Sparc. MacOS, BeOS ou AIX executando em Power PC. Linux executando em Power PC, Alpha ou X86. Digital Unix executando em Alpha. IRIX executando em Mips. Windows, FreeBSD, OpenBSD ou BeOS executando em X86. Além de simular a CPU, o Bosh simula todos os periféricos de um computador e possui um excelente *debugger* que pode ser usado para ler conteúdos dos registradores, possibilitando, como exemplo, a engenharia reversa de *drivers*. Apesar de ser muito flexível e prático, o principal defeito do BOSH é a velocidade [Jones, 2006].

O QEMU é outro emulador *open source*. Ele possui velocidade superior a de outros emuladores, porque utiliza a técnica da tradução dinâmica. Suporta dois modos de operações. O primeiro deles simula todo hardware da mesma forma que o Bosh, este modo é chamado de emulador de sistemas e seu o uso é destinado para suportar sistemas operacionais convidados. No segundo modo de operação, o QEMU executa processos compilados para uma CPU em outra CPU diferente, se comportando como um emulador de processos. Diferente do Bosh que apenas emula o X86, o QEMU emula uma grande diversidade de processadores como o X86, PowerPC, ARM, MIPS, Sparc, Alpha, Coldfire, CRIS, MicroBlaze entre outros. Ele tem pacotes pré-compilados para Linux, Windows e MacOS, mas pode também ser recompilado em outras plataformas com possíveis problemas de compatibilidade que devem ser resolvidos no código fonte [QEMU, 2011]. Além de simular a CPU, o QUEMU emula diversos dispositivos periféricos como placas de som, *mouse*, teclados, adaptadores PCI e ISA, *drivers* de disquete, portas seriais, placas gráficas [Jones, 2006].

O PCSX2 emula instruções de um videogame chamado de Playstation 2, possuindo uma arquitetura modular em *plug-ins* e pode ser executado em arquiteturas X86, com suporte a SSE2 (a partir do Pentium IV), desde que possuam no mínimo uma GPU (*graphics processor unit*) com Pixel Sharder 2.0. O emulador é construído em cima da ABI do Windows ou Linux. Ele possui opções para carregamento de diferentes *bios* e *firmwares*, podendo executar jogos comerciais feitos para o Playstation 2. Um *plug-in* interessante para este emulador é o GSdx que permite que o Windows utilize a api do directx para emular e traduzir instruções gráficas do videogame com boa velocidade. Para o Linux é usado o plugin ZZogl que utiliza a api do *opengl*, mas não é tão eficiente igual o GSdx [PCSX, 2011]. Existem emuladores para os mais diversos tipos de videogames, como o Znes (super nitendo), Gens (mega drive), MAME (fliperama) entre outros. Construir emuladores de videogame não configura violação de leis, mas para eles terem utilidade de entretenimento, o *software* (rom)

necessita ser copiado e executado no emulador, violando os direitos de cópia, e sendo assim uma contravenção.

O Transmeta Crusoe (Figura 37) é um processador-emulador [Harrison, 2005]. Ele utiliza uma camada de emulação de *software* embutida, envolvendo a própria ISA da CPU, exportando uma nova interface X86. A CPU faz uso de uma tecnologia chamada de VLIW (*very long instructuion word*). O emulador embutido recebe o nome de *code morphing* e faz a tradução dinâmica de instruções da arquitetura X86 em VLIW para o processador Transmeta Crusoe. O código do emulador é armazenado em uma memória ROM. As instruções traduzidas pelo *code morphing* são armazenadas na *cache* do L1 e L2 do processador e também na RAM [Harrison, 2005].

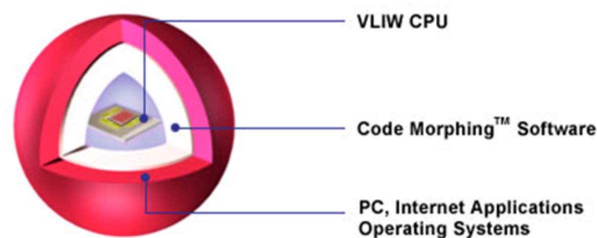



**Figura 37**. O processador e emulador Transmeta Crusoe e suas camadas [Harrison, 2005].

Apesar de não ser um processador complexo e rápido como um X86 de mesma geração, ele dissipa muito menos calor devido ao baixo consumo e menor uso de componentes de *hardware* aglomerados na pastilha de silício. A Figura 38 nos mostra a quantidade de componentes eletrônicos que um processador-emulador pode economizar comparado um processador X86 com o Transmeta Crusoe [Harrisson, 2005].

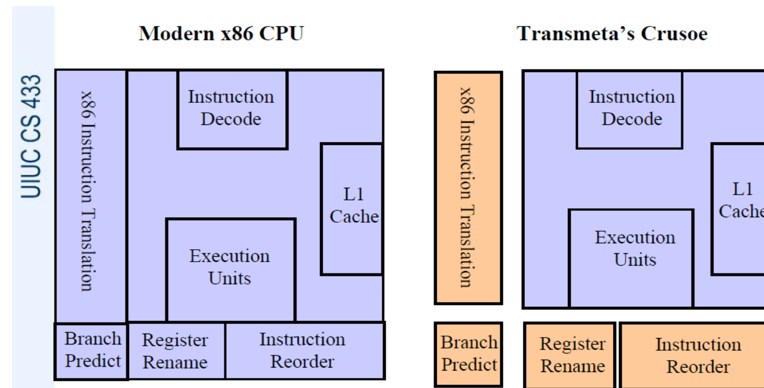



**Figura 38**. Comparação entre componentes do x86 e do Transmeta Crusoe [Harrisson, 2005].

Uma aplicação interessante para este tipo de processador é a dos dispositivos móveis. Ele permite uma maior autonomia destes dispositivos, prolongando o uso de bateria e aumentando a confiabilidade. Como o dispositivo esquenta menos, um trabalho menor será necessário para refrigerar, prolongando a vida útil da pastilha, aumentando estabilidade com menos Vcore (voltagem no núcleo). Usando menos voltagem ganha-se uma grande economia energética e as baterias passam a durar muito mais tempo, além do desgaste de componentes internos do processador ser menor. Diminuindo a complexibilidade do *chip* permite uma maior facilidade de fabricação que se traduz em diminuição de custos.

Para funcionar corretamente, emulador *code morphing* quebra dinamicamente uma instrução X86 e a traduz em 1 a 4 instruções do Transmeta Crusoe. Estas instruções quebradas são chamadas de átomos, e de acordo com certos critérios se unem, formando moléculas de 2 ou 4 átomos. Sendo cada átomo é uma instrução de 32 *bits*, então as moléculas formadas serão de 64 ou 128 *bits*. A Figura 39 representa uma molécula de 128 *bits*. Analisando a molécula temos uma VLIW de 4 átomos, e cada átomo será direcionado a uma das quatro unidades de tratamento distintas da CPU: A *branch unit* é encarrada por fazer decisões como os saltos condicionais/incondicionais e chamadas a procedimentos. A *Load/Store* faz o interfaceamento do processador com o *cache* e a memória. ADD é a unidade lógica e aritmética de números inteiros e a FAAD a de números flutuantes [Trasmeta,2011].

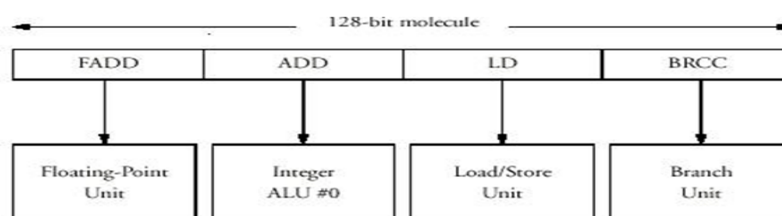


**Figura 39**. A tecnologia do VLIW (molécula do Transmeta Crusoe).

**[Harrison, 2005]**

O processador consegue tratar 1 a 4 instruções por ciclo de *clock* (Existem gerações mais novas do Transmeta Crusoe que utilizam VLIW de 256 *bits* e conseguem tratar até 8 instruções por ciclo de clock). O *code morphing* (emulador) faz o agendamento das instruções (*instruction scheduling*) e renomeia registradores (*register renaming*). Um otimizador de código é usado para analisar e otimizar as instruções mais frequentemente utilizadas [Morimoto, 2007].

**Virtualização no nível de bibliotecas**

A virtualização no nível de bibliotecas é um tipo especial de máquina virtual que geralmente é colocada sob a interface ABI do sistema operacional *host*. Neste tipo de máquina virtual, as *libcalls/syscalls* são modificadas para executar uma aplicação de outro sistema operacional. Uma grande vantagem deste tipo de virtualização é a possibilidade de execução de programas sem ter que comprar licenças de sistemas operacionais (como é o caso do Windows), consumindo muito menos espaço em disco, menos memória e tempo de processamento da CPU. A velocidade de execução dos programas se assemelha muito quando executados em seu sistema operacional nativo. Como exemplo deste tipo de virtualização temos o Wine e o LxRun.

O Wine significa "*Wine is not an emulator",* em português: "Wine não é um emulador". A camada de virtualização do Wine implementa partes do sistema operacional Windows e é usado no Linux e em outros sistemas operacionais compatíveis com o padrão POSIX para executar programas do Windows. Sendo uma implementação de bibliotecas, e não tendo que emular comportamentos complexos da CPU, o Wine tem uma excelente velocidade, sendo comparável à velocidade do programa executando no Windows. A Figura 40 mostra onde é posicionada a camada de virtualização do Wine, podemos notar que as aplicações continuam com livre acesso a *user* ISA e que o Wine serve apenas para interceptar *libcalls/syscalls*, utilizando suas próprias implementações de rotinas de tratamento já otimizadas para o sistema Unix em questão [Wine, 2011].

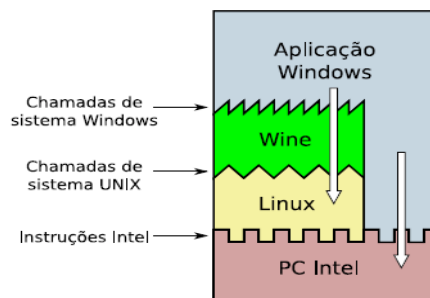



**Figura 40**. Virtualização no nível de bibliotecas : Wine [Laureano e Maziero, 2008].

O LXrun é outra camada de virtualização no espaço do usuário utilizada por sistema Unix como o Solaris, para executar aplicações feitas para Linux. O LXrun apenas funciona em plataforma X86. Seu funcionamento é parecido com o Wine, com a diferença que ele faz o re-mapeamento de *syscalls* do Linux ao invés do Windows [LXrun, 2011]. No Linux, uma *system call* move argumentos para registradores específicos e por fim executa a instrução “int $0x80” que serve para ativar a interrupção de software indicada pelo vetor de interrupções do sistema operacional. O apontamento da rotina de interrupção será de acordo com o valor especificado no registrador EAX, usando os demais argumentos passados nos outros registradores [Bartlett, 2011].

Cada sistema operacional tem um vetor de interrupções de *software* diferente. O Solaris, quando recebe a instrução "int $0x80" gera um sinal “SIGSEGV”, indicando falha de segmentação, além do Solaris, este sinal é compatível com outros sistemas do padrão POSIX. A camada de virtualização do LxRun intercepta este sinal, chamando a *syscall* equivalente no vetor de interrupções do S.O. *host*. Desta forma, é possível executar aplicações do Linux com uma perda quase que irrelevante de *performance*, sem nenhuma modificação no *kernel* do Solaris ou da aplicação do Linux [LXrun, 2011].

Como as *syscalls* não são usadas em cálculos, a *performance* de aplicações CPU *bound* fica quase sem penalidade alguma. Existem alguns casos em que a performance é, inclusive, aumentada pelo LxRun, isso acontece quando o sistema Unix executa uma tarefa de forma mais otimizada que o *kernel* do Linux. O único problema de *performance* acontece quando é feita uma tentativa de emular aplicações gráficas, pois as aplicações do Linux em X-Windows não conseguem utilizar a memória compartilhada para executar em um servidor Unix [LXrun, 2011].

**A Virtualização total**

A virtualização total funciona com um programa de controle chamado de hipervisor, o qual age como um mediador. O *software* convidado neste tipo de virtualização é sempre um sistema operacional, caracterizando uma técnica apenas empregada em máquinas virtuais de sistemas. O hipervisor gera quantas instâncias de máquinas virtuais forem necessárias, uma para cada sistema operacional virtualizado, e para isto exporta uma interface ISA compatível. Os hipervisores podem ser nativos ou convidados, que é sinônimo de virtualização do Tipo I e Tipo II. Um hipervisor nativo é mostrado na Figura 41. Podemos notar nesta figura uma unidade de gerenciamento (Mgmt) que centraliza a administração e monitora todas as máquinas virtuais criadas pelo hipervisor. A velocidade da virtualização total geralmente é superior a de emuladores, porque atalhos entre as camadas da pilha de *software* podem são permitidos. Como exemplo de virtualização total temos o VmWare e Z/VM.

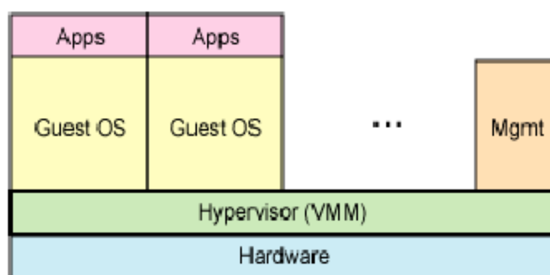



**Figura 41**. Virtualização total com hipervisor nativo [Jones, 2006].

O VmWare é uma solução comercial para a virtualização total. O Hipervisor fica entre o hardware e os sistemas operacionais convidados. Ele fornece uma interface ISA idêntica a que o sistema operacional convidado necessita. Todo sistema operacional convidado tem seus estados salvos em um arquivo, o qual pode ser facilmente migrado para outros servidores. O Z/VM é um hipervisor para o IBM System Z com um programa de controle que permite a virtualização de recursos físicos para sistemas operacionais convidados não modificados como o Linux [Jones, 2006].

**Para-virtualização**:

A para-virtualização é uma técnica muito parecida com a virtualização total, a única diferença é que ela utiliza sistemas operacionais convidados modificados. O *overhead* da para-virtualização é muito menor que o da virtualização total, porque as modificações feitas no sistema operacional convidado facilitam a virtualização, não sendo mais necessário interceptar todo acesso ao *hardware*. A desvantagem é que os sistemas operacionais precisam ser recompilados com o código para-virtualizado. A Figura 42 mostra as diferenças entre uma virtualização total e a para-virtualização. Exemplos de para-virtualização: Denali, Xen, o User mode Linux.

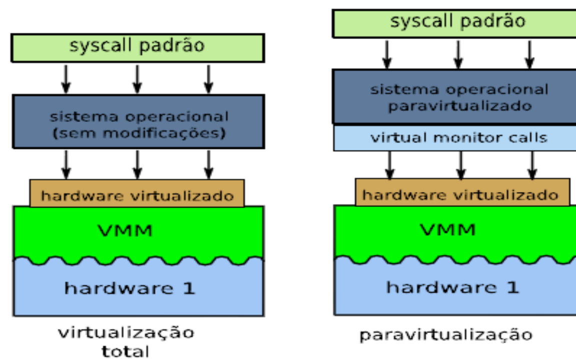



**Figura 42**. Diferenças entre virtualização total e para-virtualização.

O *user mode* do Linux é um tipo de para-virtualização que permite executar sistemas operacionais Linux convidados no espaço do usuário de um Linux *host*, para isso as imagens que serão instanciadas precisam ser recompiladas com modificações. O Xen é um hipervisor *open source* que executa sistemas operacionais modificados que colaboram com o código do hipervisor, conseguindo um desempenho muito superior ao da virtualização total [Jones, 2006].

O Denali é um sistema de para-virtualização com *microkernels*. Como sabemos, serviços de *Internet* não podem confiar um nos outros, visto que *datacenters* e provedores podem utilizar uma mesma máquina para vários serviços diferentes, muitas vezes de organizações diferentes, os quais podem ter vulnerabilidades. A principal ideia por trás do Denali é executar milhares de máquinas virtuais no mesmo *hardware* físico (multiplexação de recursos), onde cada máquina virtual é um serviço de *Internet* totalmente isolado. Se o Denali instanciasse um sistema operacional monolítico para cada serviço de *Internet*, os recursos de

*hardware* facilmente se esgotariam. Para resolver o problema da limitação de recursos, o Denali usa um *microkernel* enxuto chamado de FluxOS kit. Além disso, várias modificações foram feitas no *microkernel,* promovendo simplicidade, escalabilidade, *performance*, segurança e isolamento. O protótipo do Denali foi construído com as seguintes justificativas [Whitaker et al., 2002]:

a) O sistema operacional fornece abstrações de alto nível, e faz a segurança neste mesmo nível o que é um erro, pois os ataques muitas vezes procedem de camadas mais baixas.

b) Sistemas operacionais monolíticos evoluem e possuem uma API com muitas chamadas de sistemas, que são várias portas de entrada para ataques, contrariando o principio de segurança da economia.

c) A perda de performance dos *microkernels* modernos não é mais tão significativa, e é justificada pelo ganho de confiabilidade e facilidade de gerenciamento.

d) Os serviços de *Internet* são projetados para serem operados por usuários independentes, com pouco ou nenhum uso de compartilhamento de recursos entre processos. Desta forma é viável ter a performance sacrificada no compartilhamento de dados entre processos em troca de um forte isolamento.

e) A *Internet* e seus serviços seguem uma distribuição de Zipfian, onde que uma fração considerável de requisições vão para serviços não populares. Isoladamente, os serviços pouco acessados não representam muitas requisições, mas juntos constituem uma grande parcela de acessos.

f) A para-virtualização é justificada, pois a compatibilidade com o *software* legado pode induzir ao erro, quebrando esta compatibilidade é possível criar um novo design para o sistema operacional, fazendo implementações estratégicas.

As modificações introduzidas no *microkernel* convidado no hipervisor do Denali resolveram muitos problemas de *performance* e de segurança enfrentados por serviços de *Internet*; as modificações mais significativas foram [Whitaker et al., 2002]:

a) A introdução de uma nova instrução que pode ser utilizada pelo *microkernel* virtualizado, chamada de IDLE. Esta instrução permite que uma máquina virtual libere a CPU até que chegue uma interrupção externa que necessite de processamento. Para evitar IDLES eternos, um sistema de alarme foi implantado, o qual gera uma interrupção após alguns pré-determinados *ticks* de *clock*.

b) O gerenciamento de *timers* (*ticks*) pode ser um problema para a escalabilidade. Emular este comportamento necessita de um *timer* virtual, resultando em muitas trocas para modo *user/kernel*. No Denali, depois de uma troca de contexto, o *kernel* dispara uma interrupção, indicando que o sistema operacional convidado deve calibrar seu *clock* com o *clock* global. Todas as interrupções pendentes são entregues pelo Denali em um *batch* único, reduzindo o número de trocas de contexto e permitindo, com esta otimização, a execução de muitas máquinas virtuais em conjunto.

c) O Denali não exporta memoria virtual, todas as VM estão no mesmo *adress space*, aumentando a *performance* do sistema e tendo menos perdas de TLB durante as trocas de contexto.

d) Os dispositivos de I/O foram implantados forma muito mais simplificada, pois as interfaces dos *hardwares* são geralmente mais complicadas que o necessário, levando a uma perda de *performance* na virtualização.

e) A inicialização das máquinas virtuais é simplificada, não existe interface de BIOS, e todos os dispositivos virtuais partem de um estado conhecido, não necessitando o sistema operacional inicializar estes dispositivos.

O Denali realmente conseguiu executar centenas de sistemas operacionais virtualizados em uma mesma máquina, mas isso só foi possível devido a técnica da para-virtualização e ao uso de *microkernels*, pois um com um *kernel* monolítico certamente todos recursos da máquina seriam consumidos prematuramente antes da primeira dezena de sistemas operacionais serem instanciados.

# Anexo III Técnicas de minimização do *overhead* para máquinas virtuais

A melhoria de desempenho em ambientes que utilizam a virtualização do Tipo I ou Tipo II é feita com otimizações nas camadas da pilha de *software*. Quanto à melhoria de desempenho em hipervisores nativos, o que pode ser feito é permitir que o sistema convidado acesse diretamente o *hardware* em algumas situações. Para isso são necessárias modificações no hipervisor e no núcleo do sistema convidado (para-virtualização) [Laureano e Maziero 2008].

Sobre a melhoria de desempenho dos hipervisores convidados (tipo II) [Laureano e Maziero, 2008] e [King et al., 2003] argumentam que as seguintes técnicas diminuem o overhead:

a) O sistema operacional convidado acessa diretamente o sistema operacional *host*: As aplicações do sistema operacional convidado são aceleradas pelo hipervisor que fornece parte da API do sistema operacional *host* ao sistema convidado. Desta forma é possível, para o sistema operacional convidado, acessar diretamente alguns recursos específicos do sistema operacional *host*.

b) O sistema operacional convidado acessa diretamente o *hardware*. Este atalho é implementado pelo sistema operacional *host* e pelo hipervisor com um *driver* especial.

c) O hipervisor acessa diretamente o *hardware*. Para isso um *driver* especial é instalado no sistema *host.*

As melhorias descritas nos itens acima são ilustradas pela Figura 43.

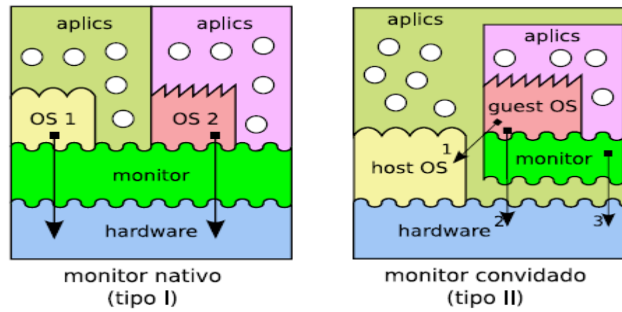



**Figura 43**. Melhorias de desempenho em hipervisores nativos e convidadados. [Laureano e Maziero, 2008]

### Máquinas virtuais recursivas

Máquinas virtuais recursivas é um modelo de máquinas virtuais hierárquico, altamente modular, onde uma máquina virtual é instanciada em cima de outra recursivamente. O principal problema das máquinas virtuais recursivas é justamente o *overhead* que cada camada de virtualização impõe, e para melhorar a *performance* é necessário fazer uma espécie de "curto-circuito" na comunicação entre as camadas.

[Ford et al., 1996] criaram um protótipo de máquina virtual recursiva, utilizando um *microkernel* chamado Fluke, que possui algumas características especiais para suportar a recursividade e o modelo hierárquico. Neste modelo proposto, não existem recursos globais, todos os conceitos são relativos e visíveis apenas se a hierarquia de máquinas virtuais permitir.

# Anexo IV O ambiente de testes *Open Cirrus*

O *Open Cirrus* é um ambiente de testes real, com marca registrada da Yahoo. Este projeto é fruto da associação entre Hp, Intel, Yahoo com colaboração da NSF (*National Science Foundation*), da UIUC (*University of Illinois*), do Instituto de Tecnologia de Karlsruhe e da IDA (*Infocomm Development Authority of Singapore*). O *Open Cirrus* é uma coleção de *datacenters* federados para sistemas *open-source* e pesquisas de serviços. A Figura 44 mostra a localização das federações do ambiente de testes no globo terrestre no início do 2009, cada estrela representa um *cluster* de 1000 núcleos de processamento mais o conjunto de armazenamento associado [Campbell et al., 2010].

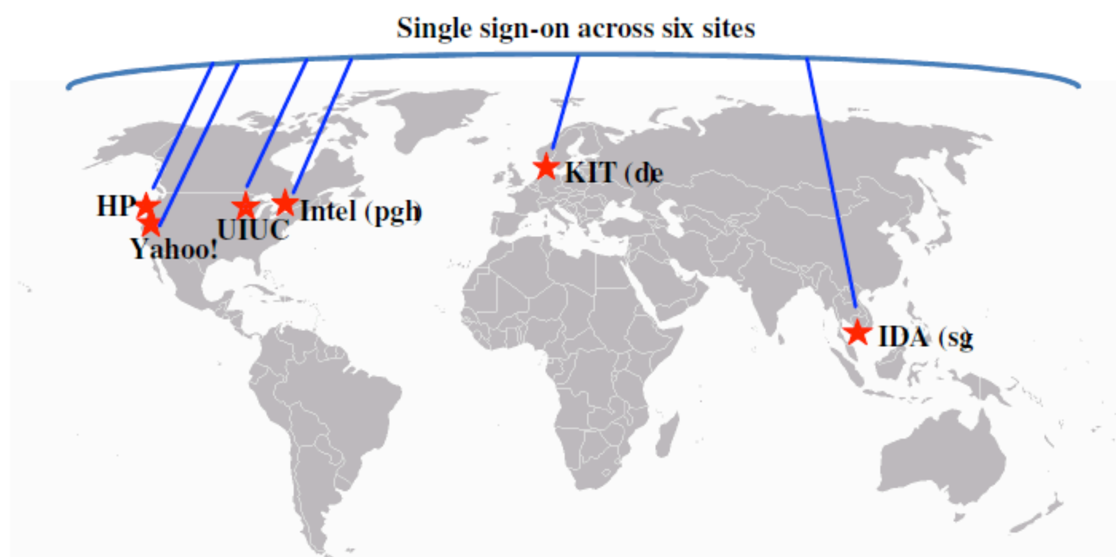



**Figura 44**. Federações de *datacenters* no globo terrestre do projeto *Open Cirrus* 2009 [Campbell et al., 2010].

O *Open Cirrus* fornece um ambiente único que apenas empresas como Amazon, Google e Yahoo possuem e desta forma os pesquisadores não precisam confiar apenas resultados de simulações ou clusters pequenos. O *Open Cirrus* possui duas grandes funcionalidades que tornam ele um ambiente de simulação único: A primeira é que ele permite acesso de baixo nível a nuvem coisa que nenhuma plataforma de *cluster* admite. A segunda é que o ambiente de testes é um ambiente heterogêneo sob domínios administrativos diferentes espalhados pelo mundo, sendo possível estudar o comportamento do *software* desenvolvido em múltiplos *datacenters* [Campbell et al., 2010].

As vantagens fornecidas pelo ambiente *Open Cirrus* [Campbell et al., 2010] são:

*a)* *Datasets* experimentais : Uma grande dificuldade dos desenvolvedores é a falta de conjuntos de dados experimentais como *logs* de carga de *datacenters* e conteúdos obtidos por *web crawlers*. O *Open Cirrus* permite importar dados em grande escala e compartilhar entre outros pesquisadores.

*b)* Desenvolver *software* para a pilha da nuvem: A *cloud computin*g usa a virtualização e cada máquina é uma pilha de *software* em um ambiente distribuído heterogêneo. A *Open Cirrus* é uma plataforma estudo e desenvolvimento de qualquer nível da pilha de *software* da nuvem permitindo o acesso às camadas mais baixas.

*c)* Aumento da importância do trabalho: Um *software* desenvolvido para a nuvem em um ambiente real em grande escala e tem uma importância maior que testes em servidores isolados.

*d)* Validação em ambiente heterogêneo: A qualidade de um sistema distribuído é aumentada se ele se portar de maneira satisfatória em um ambiente com várias máquinas virtuais e sistemas operacionais diferentes.

*e)* Compartilhamento da inovação: Um ambiente de testes em larga escala pode melhorar a eficiência de trabalhos desenvolvidos em paralelo onde um pode aprender com o outro.

*f)* Redução de custos: Compartilhar desenvolvimentos e soluções reduz custos e torna possível um produto final ainda melhor.

*g)* Acesso de *Root*: *Open Cirrus* permite o uso de máquinas físicas ao invés de apenas máquinas virtualizadas. Os servidores podem ser acessados por super usuários (*root*), com isolamento em nível de rede, mas com perda do acesso em caso de comportamento fraudulento ou com alguma conduta não permitida.

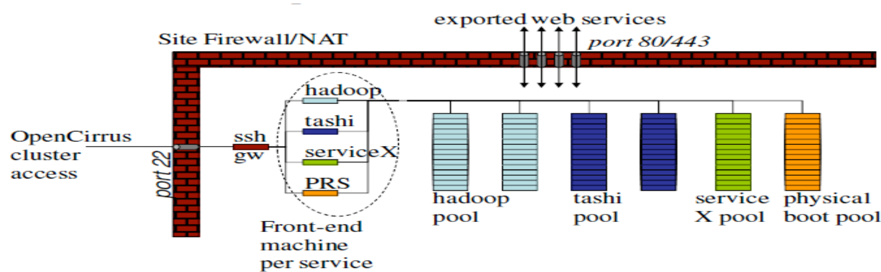
Como os usuários do *open cirrus* tem acesso de *root*, eles podem instalar imagens de sistemas operacionais, acessar diretamente os discos, com exceção dos recursos de rede que são virtualizados através de *switches* (*vlans).* Como os *sites* do *Open Cirrus* são heterogêneos, não é possível ter ambientes idênticos, mas um conjunto de serviços mínimos é obrigatório a todo *site* tais como um sistema de *global sign-on* e um repositório de HDFS (*Hadoop filesystem* – sistema de arquivos distribuídos). Cada *site* do *Open Cirrus* é dividido em alguns serviços [Campbell et al., 2010].:

a) Serviço de PRS (*physical resource set*): É o serviço de mais baixo nível, é um conjunto de servidores, *storages* e *links* isolados por uma *vlan*. Cada *datacenter* é particionado em um ou mais domínios de PRS que são alocados dinamicamente e gerenciados por um serviço de PRS atendendo requisições de clientes PRS. A HP utiliza uma tecnologia chamada *Lights-Out technology* para controlar os servidores remotamente, permitindo instalar sistemas operacionais, fazer *reboot*, *shutdown*. A *vlan* isola os diferentes usuários e permite *firewalls* específicos para cada usuário

b) Serviço de CaaS (*cells as a service*): Um recurso que serve para agrupar de forma segura os recursos virtuais chamados de células de serviço. Nestas células os clientes podem instanciar e operar serviços.

c) Serviço Tashi : É um gerenciador de *cluster* para *cloud computing* e grandes quantidades de dados na *Internet*. O sistema esta sendo desenvolvido na Fundação Apache de *software* pela Intel, Yahoo e universidade de Carnegie Mellon. O Tashi gerencia *clusters* lógicos de máquinas virtuais.

d) Serviços de alto nível: Serviços como Hadoop, Pig e MPI que suportam aplicações no nível de usuário. Os serviços de alto nível são ilustrados na Figura 45.




| Site | Characteristics | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | #Cores | #Servers | Public partition | Memory Size | Storage Size | Spindles | Network | Focus |
| HP | 1,024 | 256 | 178 | 3.3TB | 632TB | 1152 | 10G internal 1Gb/s x-rack | Hadoop, Cells, PRS, scheduling |
| IDA | 2,400 | 300 | 100 | 4.8TB | 43TB+ 16TB SAN | 600 | 1Gb/s | Apps based on Hadoop, Pig |
| Intel | 1060 | 155 | 145 | 1.16TB | 353TB local 60TB attach | 550 | 1Gb/s | Tashi, PRS, MPI, Hadoop |
| KIT | 2048 | 256 | 128 | 10TB | 1PB | 192 | 1Gb/s | Apps with high throughput |
| UIUC | 1024 | 128 | 64 | 2TB | ~500TB | 288 | 1Gb/s | Datasets, cloud infrastructure |
| Yahoo | 3200 | 480 | 400 | 2.4TB | 1.2PB | 1600 | 1Gb/s | Hadoop on demand |

**Figura 45**. Os serviços de alto nível do *Open Cirrus* [Campbell et al., 2010].