PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

CURSO DE SISTEMAS DE INFORMAÇÃO

Juliano Potrich

APRIMORAMENTO DO ESCALONADOR CREDIT

Porto Alegre

2008

Juliano Potrich

APRIMORAMENTO DO ESCALONADOR CREDIT

Trabalho de Conclusão do curso de graduação apresentado ao Departamento de Informática da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Avelino Francisco Zorzo

Porto Alegre

2008

Dedico esse Trabalho aos meus Pais

"Se você obedece todas as regras, acaba perdendo toda diversão."

Juliano Potrich

RESUMO

Este trabalho apresenta o modelo de máquinas virtuais, os tipos existentes de virtualização, e como essa virtualização tem sido utilizado atualmente. Também mostra uma análise do paravirtualizador Xen, descrevendo sua estrutura interna e seus atuais escalonadores, enfatizando o escalonador Credit.

Nesse trabalho de conclusão a idéia principal é realizar uma forma de tornar o Xen em uma ferramenta de virtualização com um maior desempenho de processamento. Para vir a ser real, é realizada uma análise sobre ferramentas de teste (*benchmarking*) e suas respectivas descrições, e detalhamento e análise do escalonador Credit e seus possíveis pontos de otimização. Tal trabalho consiste no aprimoramento no escalonador do Xen, implementando níveis de serviço em seu funcionamento, e dinamicidade em seu tempo de acesso ao CPU, possibilitando um melhor desempenho de suas máquinas virtuais.

Palavras-chave: *Benchmarking*, escalonador Credit, Xen, Máquinas Virtuais

ABSTRACT

This work presents the model of virtual machines, the types of existing virtualization, and how virtualization has been used currently. It also shows an analysis of the Xen system, describing its internal structure and its current schedulers, emphasizing on the Credit scheduler.

The main idea of this work is to change a Xen scheduler in order to improve the performance of the virtual machines that are running over Xen. This work describes a set of benchmark tools and also a detailed description of the Credit scheduler. One of the contributions of this work is to present the places in which it is possible to make some improvements in the Credit Scheduler. An analysis of the new Credit scheduler is presented and shows the improvements for the overall system.

Key-Words: Benchmarking, Credit Scheduler, Xen, SLAs, Virtual Machines

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

VMM – Virtual Machine Monitor

SO – Sistema Operacional

GPL – General Public License

EVT – Efective Virtual Time

AVT – Actual Virtual Time

W – Warp Factor

BVT – Borrowed Virtual Time

EDF – Earliest Deadline First

SEDF – Scan Earliest Deadline First

VCPU – Virtual CPU

SMP – Symetric Multiprocessor

VPS – Virtual Private Server

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, a capacidade de processamento dos computadores tem aumentado consideravelmente. Entretanto, toda a capacidade oferecida não vem sendo utilizada ao máximo. Há situações onde aplicações poderiam ser executadas de uma maneira mais eficiente potencializando o uso de CPU. Uma das soluções é o uso da virtualização, que está ganhando cada vez mais destaque, e sendo utilizada com resultados bastante satisfatórios.

Com a virtualização pode-se desassociar o sistema operacional do *hardware*, trazendo várias novas e úteis ferramentas. A virtualização permite que um operador controle o uso da CPU, memória, armazenamento e de outros recursos do sistema operacional do convidado (máquina virtual), de forma que cada convidado receba apenas os recursos que precise. Este controle elimina o risco de um processo que consuma toda a memória disponível ou a CPU.

Esse tipo de flexibilidade muda o conceito tradicional de aprovisionamento do servidor e de planejamento de capacidade. Nos ambientes virtualizados é possível tratar os recursos computacionais como CPU, memória e armazenamento como um *cache* de recursos e aplicações que podem ser facilmente realocados para receber os recursos necessários, quando necessário.

Essas vantagens da virtualização maximizam o tempo de processamento, controle, e segurança permitindo mais processos, e máquinas virtuais executando simultaneamente, tornando os equipamentos já adquiridos mais produtivos.

O ambiente de virtualização utilizado neste trabalho é chamado Xen [1], que se difere dos outros modos de virtualização [2][4][10], pois não interpreta as instruções passadas ao *hardware*, o que diminuiria o desempenho, apenas se encarrega de repassá-las ao sistema principal, criando uma ilusão de que sistema convidado está executando diretamente sobre o *hardware*.

O Xen possui muitos benefícios como CPU's virtuais (VCPU's); atribuição de peso (*WEIGHT*) que oferece mais ou menos tempo de acesso ao CPU; CAP que estipula uma taxa de processamento para cada máquina virtual; por intermédio do *hypervisor* (camada de virtualização), os operadores conseguem controlar o uso da CPU, da memória, bloco e dos dispositivos de E/S de forma dinâmica; e diversos escalonadores que podem ser alterados na inicialização da máquina, cada um com suas características de gerência de recursos. Embora possua essas melhorias como

paravirtualizador, ainda necessita de maior atenção no escalonamento de processos. O escalonador, disponível no Xen, que faz a melhor distribuição de recursos é o *Credit* [15].

O *Credit*, apesar de ser padrão no monitor de máquinas virtuais do Xen, possui um grande problema a ser resolvido para aperfeiçoar a sua dinamicidade, que consiste em uma fatia de tempo de processamento fixa em trinta milisegundos, onde em máquinas SMP com diversas máquinas virtuais com processamentos diversos, se conseguiria melhorar o desempenho global utilizando fatias de tempo de acordo com seu consumo.

Neste trabalho, é realizado um estudo detalhado dos recursos computacionais no monitor de máquinas virtuais (VMM) Xen, mais especificamente a forma de escalonamento de processos (escalonador Credit), como resultado final é apresentado uma alteração (*patch*) do escalonador Credit para incluir as diferentes fatias de tempo para diferentes máquinas virtuais. Essa alteração cria tempos dinâmicos no escalonamento de processos entre domínios, e camadas de níveis de serviço, proporcionando um melhor aproveitamento das máquinas virtuais.

Neste Trabalho de Conclusão serão apresentados os seguintes capítulos: o Capítulo 2 apresenta a definição de máquinas virtuais, tipos de virtualização, arquiteturas dos tipos de maquinas virtuais; o Capítulo 3 apresenta os trabalhos relacionados, os VMMs que estão sendo utilizados no mercado, e como eles manipulam processos, máquinas virtuais, funcionamento da arquitetura, e gerência de recursos; o Capítulo 4 descreve o funcionamento do VMM Xen, estrutura interna, principalmente um resumo dos escalonadores existentes, e suas respectivas funcionalidades; o Capítulo 5 define e detalha as ferramentas de *benchmarking*; o Capítulo 7 apresenta uma análise do código, solução e análise dos resultados do escalonador *Credit*.

2 MÁQUINAS VIRTUAIS

As máquinas virtuais não são novidades na área da computação, ela data do início dos anos 60, quando a IBM propôs um sistema em seu servidor VM/370, onde cada máquina virtual simulava uma réplica física da máquina real, provendo uma ilusão aos usuários de um acesso exclusivo ao *hardware* [6].

Há muito tempo os microcomputadores são plataformas auto-suficientes e de baixo custo para desenvolvimento. E agora chegamos num ponto que um micro pode simular a si mesmo com velocidade suficiente para uso normal. As ferramentas de desenvolvimento podem subir mais um degrau de qualidade e complexidade.

A grande utilização de máquinas virtuais atualmente se justifica por vários fatores que os parques de máquinas oferecem, tais como:

- Aumento da capacidade de processamento dos computadores pessoais;
- Ociosidade de carga de processamento;
- Agilidade na manutenção;
- Diminuição de custos de energia;
- Suporte a softwares legados.

Todas essas vantagens, trazem facilidades na manutenção de *hardware*, otimização de recursos, maior disponibilidade do parque de máquinas, espaço físico, energia, etc, mas uma das maiores vantagens da utilização de máquinas virtuais é que nesse ambiente existe uma proteção do acesso aos vários recursos do sistema, isolando cada máquina virtual uma da outra, possibilitando segurança.

2.1 Definição de Máquinas Virtuais

Máquinas virtuais são abstrações de *hardware* gerenciadas por um monitor de máquinas virtuais, combinando e dividindo recursos computacionais provendo vários ambientes operacionais de execução.

O monitor de máquinas virtuais (VMM – *Virtual Machine Monitor* ou *hypervisor*) é uma camada de *software* desenvolvida especialmente para não ser intrusivo ao sistema operacional do servidor e que gerencia e repassa chamadas de sistema do sistema convidado em nível privilegiado com o *hardware* [5].

O *hypervisor* introduz um novo nível de privilégio de acesso ao *hardware* e consegue fazer a distribuição de recursos (CPU, memória, I/O, etc.) aos sistemas operacionais convidados.

As implementações de *hypervisor* podem ser um pouco diferentes, mas o conceito como um todo acaba sendo o mesmo: melhorar o desempenho e confiabilidade do mundo virtualizado.

Existem similaridades entre as técnicas de virtualização, porém a diferença entre elas está no nível de abstração e nos métodos usados para a virtualização. Os três modelos de virtualização são: emulação, virtualização e paravirtualização.

2.1.1 Emulação

A emulação vista na Ilustração 1 é uma técnica que simula toda uma arquitetura computacional, de forma que todas as instruções do Sistema Operacional (SO) emulado são traduzidas e executadas no SO original. Conseqüentemente, emuladores são usados para testar o funcionamento de programas escritos para arquiteturas diferentes daquelas em que o executa. A diferença entre emuladores e máquinas virtuais, está basicamente no nível de abstração. Enquanto em um emulador todas as instruções são convencionalmente traduzidas e interpretadas, em máquinas virtuais as instruções são executadas no modo mais nativo possível.

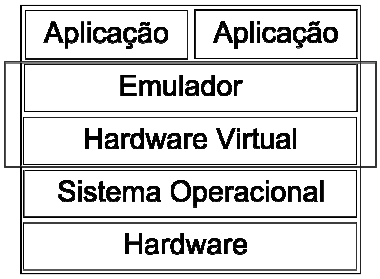

Ilustração 1: Estrutura da Emulação

2.1.2 Virtualização

A virtualização observada na Ilustração 2, é um ambiente de trabalho "falso" dentro de um ambiente "real". Apesar de falso, o ambiente virtual imita bem as características do anfitrião. Esse recurso existe desde os *mainframes*. Os sistemas operacionais de virtualização provêem isolamento das requisições e segurança para rodar múltiplas aplicações ou cópias do mesmo sistema operacional, no mesmo servidor [8]. Portanto tem ganho atenção como ferramenta de segurança, ao confinar serviços sensíveis.

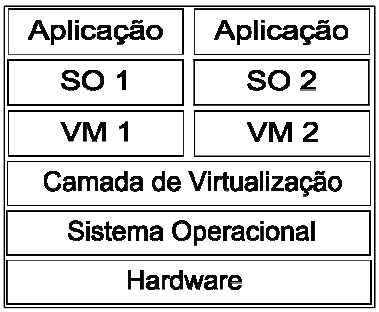

Ilustração 2: Estrutura da Virtualização

2.1.3 Paravirtualização

A paravirtualização ressaltada na Ilustração 3, é um método que consiste em apresentar ao Sistema Operacional que está sendo emulado uma arquitetura virtual que é similar, mas não idêntica à arquitetura física real [13]. Essa solução aumenta o desempenho das máquinas virtuais que a utilizam [14].

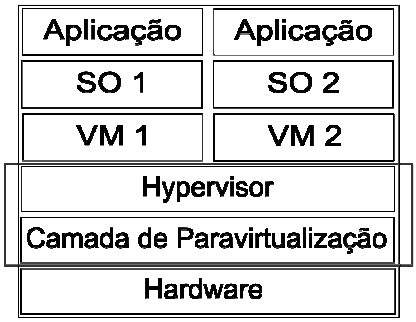

Ilustração 3: Estrutura da Paravirtualização

2.2 Tipos de Máquinas Virtuais

Existem dois tipos distintos de máquinas virtuais, as de Tipo I onde o monitor é implementado entre o *hardware* e os sistemas convidados e as de Tipo II onde o monitor é implementado como um processo de um sistema real.

Em máquinas virtuais de Tipo I, o *hypervisor* sempre tem o controle do *hardware* e cada máquina virtual responde como uma máquina física executando seu próprio sistema operacional.

Já nas máquinas virtuais de Tipo II o *hypervisor* roda como um processo de sistema, sobre o Sistema Operacional “base”, e simula as operações que o sistema base faria, servindo como uma camada de abstração entre as máquinas virtuais e o sistema base.

2.3 Considerações Finais

A virtualização permite que um servidor execute múltiplos Sistemas Operacionais ou aplicações em diferentes partições, que podem ser configurados para atender a necessidades específicas, sem que um interfira no funcionamento no outro. Esta tecnologia pode ser utilizada a partir de ferramentas de software (o uso mais comum) ou de hardware preparado para isto.

Encontramos duas formas de virtualização por software. Na primeira, também chamada de emulação, um utilitário intercepta todas as chamadas ao hardware feitas pelo Sistema Operacional “virtualizado” e traduz para as funções do Sistema Operacional hospedeiro, como OpenVz [17], VMWare [4].

Outra forma de virtualização por software é utilizar um Sistema Operacional “de base”, ou “hospedeiro”, ou paravirtualização, que irá prover, a cada partição virtual, um espaço independente de acesso aos recursos de hardware. Usando estas técnicas temos os utilitários como: User-Mode Linux [10], e o Xen [1].

3 TRABALHOS RELACIONADOS

Neste capítulo apresentamos, resumidamente, alguns conceitos básicos necessários para o entendimento do trabalho, mostrando arquiteturas e formas de virtualização existentes. Seguem-se três seções resumindo outros trabalhos importantes encontrados em utilização no mercado e relacionados ao problema principal tratado neste trabalho que é observar os tipos de virtualização, contudo visando uma forma de melhorar desempenho das máquinas virtuais.

3.1 OpenVZ

O OpenVZ [17] é uma solução de virtualização em nível de sistema operacional. OpenVZ cria ambientes virtuais isolados, que funcionam como servidores *standalone* convencionais, porém utilizando um único *hardware* em comum. Estes ambientes virtuais seguros são conhecidos como VE ou como VPS (*virtual private server*).

As VPS's podem ser reinicializadas independentes umas dos outras. Todas possuem *hostname*, acesso de *root*, endereço IP e tudo mais que um servidor pode ter, sendo assim uma solução extremamente confiável e funcional de virtualização.

Na visão dos usuários virtuais do OpenVZ, sistema virtual é um sistema independente. Esta independência é fornecida por uma camada de virtualização no *kernel* do Sistema Operacional anfitrião e onde somente uma parte insignificante dos recursos do processador central está utilizando esta virtualização (ao redor 1-2%). As características principais da camada de virtualização executada em OpenVZ são as seguintes:

Todo o sistema virtual comporta-se como um sistema Linux. Tem inicialização padrão; vários softwares podem funcionar dentro de um sistema virtual sem modificações específicas do OpenVZ ou ajustes; um usuário pode mudar toda a sua configuração e instalar algum software adicional; os usuários virtuais são isolados completamente (sistema de arquivos, processos, IPC, sysctl, etc.); os processos que pertencem a um VPS são programados para a execução em toda a CPU disponível. Conseqüentemente, o VPS’s não é limitado a somente um processador central e pode usar todo o poder disponível do processador central.

A camada do virtualização da rede do OpenVZ é projetada para isolar os sistemas virtuais da rede física. Cada sistema virtual tem seu próprio endereço de rede; os endereços múltiplos por sistema virtual são permitidos; o tráfego da rede de um sistema virtual é isolado do outro sistema virtual.

A gerência de recurso de OpenVZ controla a quantidade de recursos disponíveis para usuários virtuais. Os recursos controlados incluem parâmetros como o poder do processador central, o espaço de disco, um jogo de parâmetros memória-relacionada, etc. A gerência de recursos permite ao OpenVZ: Compartilhamento eficaz de recursos disponíveis do sistema virtual; Quality-of-Service; fornece proteção dos ataques do negação de serviço; coleta informações do uso para a monitoração do sistema.

A gerência de recursos é muito mais importante para OpenVZ do que para um computador autônomo desde que a utilização dos recursos do computador em um sistema OpenVZ é consideravelmente mais elevada do que aquela em um sistema típico.

3.2 VMWare

O VMWare [4] usa um conceito de virtualização. Ele tenta sempre que possível converter os comandos usados pelo sistema dentro da máquina virtual em comandos que o sistema *host* entenda e execute diretamente. Isso se aplica quando é necessário transmitir dados através da placa de rede, tocar sons na placa de som, ou executar instruções do processador. O VMWare interpreta e converte instruções o mínimo possível, o que faz com que o sistema dentro da máquina virtual execute com um desempenho muito similar ao desempenho real da máquina.

Quando falamos de "sistema *host*", queremos nos referenciar ao micro físico que possui seu próprio Sistema Operacional e que serve de base para a instalação do VMWare.

Do ponto de vista do sistema *host*, cada máquina virtual é um arquivo criado pelo VMWare. Com isso, pode-se levar uma máquina virtual de um micro para outro sem problemas. Outro ponto muito interessante é a possibilidade de ligação entre o sistema *host* e todas as máquinas virtuais como se estivessem em uma rede tradicional sendo cada qual com seu endereço IP. Por exemplo, podemos ter um servidor com o Windows Server 2003 e vários clientes (as máquinas virtuais criadas) todos ligados na mesma rede. É muito interessante para testes de funcionalidade de

recursos de um servidor (DHCP, proxy e outros) ou para testes de rede entre Sistemas Operacionais diferentes (Linux e Windows, por exemplo).

O VMWare é um software que cria máquinas virtuais que simulam um PC completo dentro de uma janela, permitindo instalar praticamente qualquer Sistema Operacional para a plataforma x86. É possível até mesmo abrir várias máquinas virtuais simultaneamente e rodar lado a lado várias versões de Sistemas Operacionais diferentes.

3.3 User-Mode Linux

User-mode *Linux* (UML) [3] é uma arquitetura portável da própria interface do *Linux kernel* para que possa ser chamado a partir do sistema. Um *Kernel Linux* compilado para arquitetura um pode iniciar-se como um processo a partir do próprio sistema operativo *Linux*, e inteiramente gerido no espaço de utilizador (*user space*), sem que isso afete o ambiente do computador que corre o próprio sistema. Inúmeras coisas se tornam possíveis através da utilização do UML. Podem ser lançados serviços de rede a partir do hóspede (guest) UML, e manter-se totalmente isolado do principal sistema que corre *Linux* (*host* no original). Geralmente os administradores de sistemas usam o UML para gerir *honeypots*, que permitem que o resto da segurança de um determinado ponto de rede de um *host* [10].

Após tudo instalado, tem-se uma instância do novo *Linux* rodando em paralelo ao *Linux* original que já estava em operação na máquina. Usando ferramentas de diagnóstico comuns, como os comandos *'ps'* e *'free'*, e exames nos arquivos do */proc* (exemplo: *'cat /proc/interrupts'*), percebe-se que o *kernel* do UML roda sobre uma máquina virtual que não necessariamente corresponde à sua configuração física, incluindo os dispositivos, sistemas de arquivos, quantidade de memória, interfaces de rede e tudo o mais. Aliás, a máquina virtual do UML suporta interface de rede, podendo ter seu próprio IP e comunicar-se com o hospedeiro e com outras máquinas da sua rede local. Pode-se até rodar múltiplas máquinas virtuais no seu computador, cada uma com seu próprio IP comunicando-se com as demais.

O UML permite que se experimentem configurações e force o desempenho de um sistema virtual sem colocar em risco o próprio sistema estável ou manter um sistema extra para testes. Proporciona trabalhar com uma ou mais máquinas virtuais UML e provocar todo o dano que quiser sem se preocupar com seu sistema enquanto testa *drivers* de dispositivo.

Alternativamente, se pode dividir um único sistema em um número de máquinas virtuais independentes. O UML está sendo utilizado por várias companhias de hospedagem de sites para prover *hosts* virtuais *Linux*.

3.4 Considerações Finais

Com a consolidação de servidores há a necessidade de mais processamento então em vez de diversos servidores fisicamente diferentes, no qual aumentaria os custos, basta uma máquina com diversas máquinas virtuais, potencializando os recursos disponíveis.

Neste capítulo vimos diversas maneiras de criar máquinas virtuais, como User-Mode-Linux, que são mais utilizados em servidores totalmente Linux, podendo apenas ser criados máquinas virtuais com distribuição *Linux*. O VMWare que é o mais visado no mercado de trabalho, mostrando maior desempenho na área comercial, consolidando as máquinas virtuais com *Linux* e Windows rodando simultaneamente, e também por fácil migração dessas máquinas virtuais (apenas um arquivo). O OpenVZ que fornece uma camada de virtualização a nível de *kernel*, até então não tinha se visto, aumentando seu desempenho, e transferindo praticamente todos seus recursos para as máquinas virtuais.

Com o estudo detalhado dessas técnicas e formas de virtualização, aprimorou-se o escalonamento de processos do monitor de máquinas virtuais Xen, aumentando seu poder computacional como paravirtualizador.

4 O PARAVIRTUALIZADOR XEN

O Xen [1] foi desenvolvido pelo *Systems Research Group* da Universidade de Cambridge, e é parte de um projeto maior chamado XenoServers, que prove um ambiente de computação global distribuída, e o Xen permite compartilhar uma simples máquina para vários clientes rodando Sistemas Operacionais e seus respectivos programas, desenvolvido com ênfase na proteção de acesso a memória, entre sistemas.

O Xen utiliza um conceito chamado paravirtualização, onde o Sistema Operacional rodando dentro da máquina virtual tem a ilusão de estar sendo executado diretamente sobre o *hardware*.

Xen é um monitor de máquinas virtuais (VMM) para computadores com a arquitetura x86, ia64. Pode executar múltiplas máquinas virtuais, cada uma rodando seu próprio Sistema Operacional. É um software de código aberto, liberado sob a licença GNU General Public License (GPL). Atualmente, o ambiente suporta os sistemas Windows XP, Linux e Unix (baseado no NetBSD).

O Xen está encarregado de organizar as solicitações feitas pelas máquinas virtuais, repassando em seguida para o sistema principal sem interpretá-las. Este processo resulta em uma diminuição de desempenho muito pequena [5]. De acordo com Barham [14], em um *virtual machine monitor* tradicional o *hardware* exposto é funcionalmente idêntico à máquina. Embora a virtualização completa tenha benefícios óbvios de permitir o sistema operacional não modificado de serem hospedados, existem também alguns inconvenientes. Determinadas instruções do *hypervisor* devem ser lidadas pelo monitor de máquinas virtuais para uma correta virtualização, no entanto só podem ser feitas com o nível de privilégio suficiente.

O Sistema Operacional a ser executado na máquina virtual deve ser modificado, ou seja, é necessária uma versão específica de Sistema Operacional para executar dentro do Xen. Para que isso ocorra, é necessária a instalação de um *patch* no núcleo. A proposta do ambiente Xen é suportar aplicações sem a necessidade de alterações, múltiplos sistemas operacionais convidados e a cooperação entre estes sistemas, mas com o máximo de desempenho.

4.1 Estrutura do Xen

A maioria dos sistemas operacionais por padrão funciona em nível 0 (*ring level* 0), onde se pode executar qualquer instrução privilegiada e qualquer acesso de entrada/saída. O Xen modifica o *kernel* de nível 0 para nível 1, passando o próprio Xen a rodar em nível 0. As aplicações não percebem essa mudança por que executam em nível 3, que é menos privilegiado [11].

O Xen é como um *microkernel*, como pode ser observadao na Ilustração 4, onde se podem ter múltiplos Sistemas Operacionais e gerenciando todo o acesso à memória e dispositivos.

O Xen prove uma camada, chamada *hypervisor*, ao *hardware* onde são portados seus Sistemas Operacionais, e em retorno se tem a habilidade para rodar múltiplas instâncias de Sistemas Operacionais.

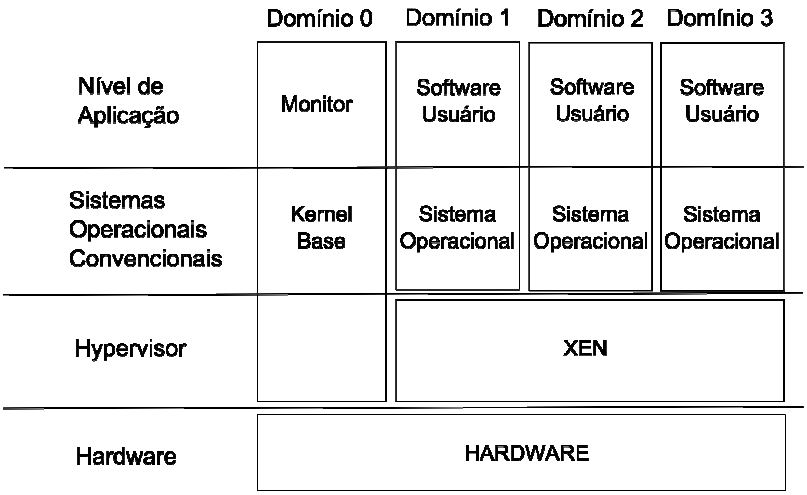

Ilustração 4: Estrutura do Xen

Uma virtualização completa acarretaria perda de desempenho, portanto o Xen faz uma paravirtualização, disponibilizando benefícios significantes em termos de *drivers* de dispositivos e interfaces. Essencialmente, *drivers* de dispositivos podem ser virtualizados utilizando esse modelo paravirtualizado, e assim, garantindo recursos de baixo nível separados por domínios como memória, CPU e outros recursos. Além disso, o próprio *hypervisor* é protegido de eventuais erros e problemas com os *drivers* dos dispositivos, podendo empregar qualquer dispositivo disponível no mercado não precisando de um *hardware* ou *driver* específico.

O Xen se utiliza acesso de somente-leitura às tabelas de páginas da memória e fica a cargo do Sistema Operacional base explicitamente requerer qualquer modificação.

O Xen valida toda requisição e somente aplica modificações seguras ao ambiente. Isto se faz necessário para prevenir que domínios possam adicionar mapeamentos arbitrários em sua tabela de páginas.

O Xen também provê um modo alternativo de operação em que o Sistema Operacional padrão tem a ilusão que suas tabelas de páginas são diretamente graváveis, porem não é assim que ocorre desde que o Xen valida todas as modificações.

4.2 Escalonadores

Em um Sistema Operacional a gerência dos processos é algo vital, pois tem a finalidade de manter o processador o maior tempo ocupado possível, aumentando assim a produtividade do sistema como um todo.

Essa gerência de processos é feita pelo escalonador que tem a função de criar um balanceamento entre os processos, privilegiando a execução de aplicações críticas, apresentando tempos de resposta razoáveis. Na próxima seção serão apresentados os três tipos existentes de escalonamento no Xen.

4.2.1 Borrowed Virtual Time

O BVT (Borrowed Virtual Time) provê de maneira justa o tempo do processador dividindo ele de modo proporcional. É observado que este tipo de escalonamento penaliza sistemas que usam operações de entrada/saída de maneira intensa possibilitando uma melhor distribuição do uso da CPU perante processos *CPU-Bound*, porém ele possui um esquema de compensação que tenta minimizar esse efeito.

Seu escalonamento por tempo virtual consiste em cada *thread* ter seu EVT (*Effective Virtual Time*), AVT (*Actual Virtual Time*), W (*Warp Factor*) e *WarpBack* (se *warp* está ativo ou não).

As *threads* acumulam seu tempo virtual e a próxima thread a ser escalonada é a que tiver o menor EVT.

Alguns detalhes importantes do BVT:

- Permite mudanças de contexto apenas em unidades de tempo para prevenir erros;
- As *threads* podem acumular tempos virtuais com diferentes taxas.

4.2.2 Scan Earliest Deadline First

O *Earliest Deadline First* (EDF) [16] é o mais comum para escalonamento em tempo real de tarefas com *deadline* (prazo final). O *Scan-Earliest Deadline First* (SEDF), combina o algoritmo *Scan* com EDF [16] para reduzir a média do tempo de busca do EDF. É utilizado apenas o EDF, mas quando muitas requisições têm o mesmo *deadline*, os blocos são acessados com o algoritmo *Scan*, se tem diferentes *deadline*, é reduzido para o EDF apenas.

Define um escalonamento baseado em prioridades, onde a escala é produzida em tempo de execução por um escalonado preemptivo dirigido a prioridades. É um esquema de prioridades dinâmicas com um escalonador *on-line* e dinâmico.

As premissas que definem o funcionamento do SEDF são:

- As tarefas são periódicas e independentes;
- O *deadline* de cada tarefa coincide com seu período;
- O tempo de computação de cada tarefa é conhecido e constante (*Worst Case Computation Time*);
- O tempo de chaveamento entre tarefas é assumido como nulo.

A política de escalonamento do SEDF corresponde a uma atribuição dinâmica de prioridades que define a ordenação das tarefas segundo seus *deadlines* absolutos. A tarefa mais prioritária é a que tem o *deadline* absoluto mais próximo do tempo atual.

4.2.3 Credit Scheduler

O *Credit Scheduler* [15] é um escalonador construído para manter um equilíbrio de recursos a todos os processadores em máquinas SMP. A cada domínio convidado são atribuídos dois valores: um peso e um limite. Um domínio com peso 512 terá duas vezes mais direito à execução que um domínio com peso 256. Os valores de peso variam de 1 a 65535, sendo o padrão 256 [15].

O limite geralmente fixa o máximo de CPU que uma máquina virtual pode consumir, mesmo que o sistema anfitrião tenha ciclos de CPU inativos. O limite é expresso em percentagem de uma CPU física como: 100 é uma CPU física, 50 é meia CPU, 400 são 4 CPUs, e assim por diante [16].

O escalonador *Credit* provê automaticamente balanceamento de carga entre os CPUs virtuais (VCPUs) dos domínios convidados, utilizando todas as CPUs físicas de uma máquina SMP.

Uma vantagem desse escalonador, é que o administrador não necessita referenciar cada CPU virtual a cada CPU física, o escalonador já trata dessa maneira, associando 1 CPU virtual a 1 CPU real.

Todos os CPU controlam uma fila local do funcionamento de VCPUs em execução. Esta fila é classificada pela prioridade de cada VCPU. Uma prioridade de VCPUs pode ser dois valores: *over* ou *under*, que representam se a VCPU excedeu ou não a sua parte de acesso ao recurso do processador.

Enquanto um VCPU funciona, consome créditos. Assim, frequentemente é recalculada a contabilidade de quantos créditos cada máquina virtual ativa ganhou. Os créditos negativos implicam em uma prioridade *OVER*. Até que uma VCPU consuma todos seus créditos, sua prioridade está *UNDER*.

Quando um processador não encontra um VCPU da prioridade *UNDER* em sua fila local do funcionamento, olhará em outra CPU por uma prioridade *UNDER*. Assim, garante o balanceamento de carga, onde cada máquina virtual recebe sua parte justa de recursos do processador. Antes que um processador fique inativo, olhará em outro processador para encontrar um VCPU executável. Isto garante que nenhum processador trabalhe lentamente quando existem execuções a cumprir no sistema.

4.2.4 Considerações Finais

Embora o Xen tenha vários escalonadores homologados, para determinadas situações eles não são completamente satisfatórios.

O escalonador chamado *Borrowed Virtual Time* (BVT) provê um compartilhamento proporcional entre as CPUs, porém existe neste escalonador uma grande vulnerabilidade no que diz respeito à precisão de predição dos recursos necessários para o processamento, o que se mostra significativo em aplicações que necessitam de processamento uniforme e também o serviço de interrupção pode

efetivamente roubar ciclos, fazendo o número de ciclos do processador incerto em algum período de tempo.

O escalonador chamado *Scan Earliest Deadline First* usa uma fila de prioridades onde o processo de menor *deadline* é escolhido para a execução. Entretanto, além de problemas de afinidade com os processadores, neste escalonador não existe como controlar qual o percentual de CPU cada máquina virtual vai consumir no domínio 0, embora tenham sido setados pesos de processamento para cada domínio.

O último escalonador é chamado *Credit*, tido como ideal para máquinas SMP (*Symmetric MultiProcessor*) pelos desenvolvedores do Xen, pois tem a capacidade de quando uma CPU real estiver parada, buscar CPUs virtuais de outras CPUs reais para execução.

Embora indicado para máquinas SMP, o *Credit* ainda tem alguns problemas, pois só consegue fazer um bom balanceamento de recursos se houver algum processador livre, o que é difícil de ocorrer em um ambiente de produção utilizando máquinas virtuais. Ainda, quando utiliza processos I/O *Intensive* não existe como fixar limites aos domínios sobre a utilização das CPUs, pois a escrita/leitura é controlada pelo domínio 0 e não pelas máquinas virtuais.

Através do detalhamento e análise do código do escalonador Credit, podemos ver pontos em que o algoritmo de escalonamento foi otimizado, e também estruturas do sistema que incorporou-se camadas de níveis de serviço. Para garantir que a troca desse algoritmo teve relevância e eficiência, utilizamos o monitoramento como forma de avaliação de sistemas, como ferramentas de teste - *benchmarking* de análise de desempenho, descritos no Capítulo 6.

5 FERRAMENTAS DE BENCHMARKING

Avaliar um sistema pode ser definido como toda e qualquer observação feita sobre ele. Tanto situações anteriores como situações atuais podem ser avaliadas para tornar possível a observação da evolução do sistema. Além disso, a observação do comportamento do sistema ajuda a entender seu funcionamento. Podem ser ainda avaliadas situações futuras, com finalidade de modificação para a sua melhoria.

Existem três técnicas de avaliação de sistemas: temos a simulação, métodos analíticos e o monitoramento. Na simulação é criado um modelo com as mesmas características do sistema real. Porém, por se tratar de uma abstração da realidade, a fidelidade das medidas nem é sempre fiel ao sistema real. Métodos analíticos são modelos matemáticos, que simulam o sistema com um nível maior de abstração.

A técnica de monitoramento que é a utilizado nesse trabalho, consiste na observação de sistemas reais, propicia maior fidelidade dos resultados obtidos, pois não é feita nenhuma abstração, ou seja, tem a função de avaliar o desempenho, avaliando a qualidade desse sistema pela capacidade de processamento de certa quantidade de dados, para chegar a um índice qualitativo de desempenho.

Utilizamos *benchmarks* para proporcionar formas padronizadas de avaliação do desempenho de sistemas, retirando estatísticas do seu funcionamento, a medida da carga de processamento atribuída a eles.

Os *benchmarks* são programas destinados a avaliar o desempenho de um sistema perante uma carga bem definida. O *benchmark* é uma ferramenta que permite medir resultados e podem ser reproduzidos facilmente. É a ferramenta de *software* mais utilizada para a avaliação de desempenho por aferição, e é amplamente utilizado para comparar desempenho de máquinas diferentes, reprojetar hardware e software, decisões sobre aquisição de novos sistemas, ajuda na otimização de códigos e previsão de desempenho de aplicações em computadores específicos. Existem quatro tipos de *benchmarks*, os de componente, de sistema, os sintéticos e os de aplicação, classificados quanto à categoria em

proprietários, de subsistemas, de aplicações padrão e de aplicação personalizada e quanto à arquitetura em seqüenciais e paralelos [18].

Utilizamos neste trabalho *benchmark* de *kernel*, e *microbenchmarks*. O *benchmark* de *kernel* é usado para testar as partes essenciais de um determinado tipo de programas de aplicação (funções de sistemas), como:

- Pedaços de código extraídos do código de programas reais que correspondem às zonas onde o programa permanece mais tempo (por exemplo, ciclos) ;
- Desta forma, espera-se que o desempenho seja característico da classe de aplicações;
- Como são poucas linhas de código são fáceis de transportar e manipular;
- Dão indicações importantes (sobretudo relativas) ;
- Muito aplicado em Sistemas Operacionais;
- Não exercita de forma realista a hierarquia de Memória;
- Muitas vezes é aqui que estão os gargalos (*bottlenecks*).

Contudo, os *microbenchmarks* são pequenos programas que testam uma pequena parte do sistema:

- Operações em vírgula flutuante;
- Interação CPU/memória;
- Discos, sistema de arquivos;
- Operações do Sistema Operacional: criação de processos, primitivas de comunicação;
- Em conjuntos podem permitir determinar se os vários componentes de um sistema estão equilibrados;
- Requerem um conhecimento profundo do componente a ser testado.

Portanto, neste trabalho, foi usada uma ferramenta para testar o desempenho das máquinas virtuais no paravirtualizador Xen. A ferramenta escolhida é *open source*, baseadas em aplicações também *open source* e sem restrição com relação à publicação dos resultados obtidos

5.1 UnixBench

O UnixBench [18] é uma série sintética de *benchmarks* distribuída pela BYTE Magazine´s Unix Benchmarks. Assim como o LMBench [19], esta série é melhor indicada para encontrar gargalos em partes específicas de sub-sistemas do *kernel*.

O UnixBench *benchmark* consiste em cinco principais grupos: testes aritméticos, teste de sistema, *dhrystone*, *whetstone*, teste de compilação, e diversos pequenos testes.

- O *Dhrystone* não executa operações de ponto flutuante, mas envolvem matrizes, caracteres de string, endereçamento indireto, e a maioria dos não - ponto flutuante instruções que podem ser encontrados em um aplicativo. Inclui também condicional e outras operações comuns no programa de fluxo controles. O resultado do teste é o número de *dhrystone*, que é *loops* por segundo. É utilizado por causa de sua vasta eleição de operações e porque é um dos mais amplamente usados programas de *benchmark*.
- O *Whetstone* é conceitualmente similar ao *Dhrystone*, mas com uma ênfase em matemática; é uma mistura de ponto flutuante e inteiro aritmético, chamadas de função, operações com matriz, condicionais; e transcendentais chamadas de função.
- Testes Aritméticos têm o mesmo código - fonte com diferentes tipos de dados para substituir as operações: *register*, *short*, *int*, *long*, *float*, *double*, e um laço vazio para calcular o *overhead* exigido pelo programa. O teste envolve atribuição, adição, subtração, multiplicação e divisão.
- Teste de sistema, consiste em *overhead* de chamadas de sistema (*syscall* e *execl*), pipe *throughput*, *pipe context switch*, criação de processos (*spawn*), *throughput* do sistema de arquivos.
- Teste de Compilação, basicamente teste de compilação em C, usando compilador GCC ou G++.
- Diversos testes estão incluídos *Tower of Hanoi* (um teste de recursivas operações) e um teste do UNIX de precisão no cálculo da raiz quadrada de dois a 99 casas decimais.

TEST	BASELINE	RESULT	INDEX
	INDEX VALUES		
Arithmetic Test (type = double)	2541.7	54746.1	21.5
Dhrystone 2 without register variables	22366.3	324439.5	14.5
Execl Throughput Test	16.5	146.2	8.9
File Copy (30 seconds)	179.0	4893.0	27.3
Pipe-based Context Switching Test	1318.5	8114.9	6.2
Shell scripts (8 concurrent)	4.0	18.6	4.7
			=========
SUM of 6 items			83.0
AVERAGE			13.8

Ilustração 5: Exemplo de relatório Unixbench

Após rodar o *benchmark*, é gerado um relatório, como mostrado na Ilustração 6, o resultado de cada um dos testes é comparado com um indicador base. A partir dos resultados individuais é calculado um escore final de pontos (coluna *INDEX*). Este escore final é gerado pela soma dos índices da coluna *index*.

5.2 Considerações Finais

O *benchmark* é o ato de executar um programa de computador, um conjunto de programas ou outras operações, a fim de avaliar o desempenho relativo de um objeto, normalmente executando uma série de testes padrões e ensaios nele.

Benchmark é também comumente usado para os próprios programas (de benchmarking) desenvolvidos para executar o processo. Normalmente, *benchmarking* é associado com avaliação de características de performance de um hardware de computador como, por exemplo, a performance da operação de ponto flutuante de uma CPU, mas há circunstâncias em que a técnica também é aplicável a software. *Benchmarks* de software são feitos, por exemplo, em compiladores ou sistemas de gerenciamento de banco de dados.

Benchmarks provêm um método de comparação da performance de vários subsistemas dentre as diferentes arquiteturas de *chips* e sistemas. Com essas várias condições, as ferramentas de *benchmarking* verificam como se comporta o sistema, simulando de maneira real aplicativos, assim identificamos gargalos que o escalonador *Credit* possui.

6 ESCALONADOR CREDIT

Neste capítulo apresentamos, de forma detalhada, alguns conceitos básicos sobre o escalonador *Credit*, mostrando suas estruturas internas, principais funções, e pontos de otimização para alterar seu funcionamento. Seguem-se três seções, a Análise do Escalonador *Credit* que mostra o funcionamento atual do escalonador, após a descrição da solução para implementação de níveis de serviço, e a dinamicidade de tempo para acesso ao CPU. Depois de realizadas essas alterações há uma análise dos resultados, efetuando uma avaliação de desempenho no *patch* proposto através de *benchmarks*.

6.1 Análise do Escalonador Credit

No escalonador *Credit*, é possível identificar as principais estruturas que o compõe: CPU-física, CPU Virtual (VCPU), estados dos processos, estrutura de domínio, no caso máquinas virtuais ativas no Xen, uma estrutura privada de informações do escalonador.

No escalonador *Credit*, verificamos que também possui variáveis de gerenciamento de estados, contadores, prioridades, definições de créditos do escalonador, constantes básicas, *flags* para controlar o limite de processamento de cada VCPU. Além dessas varáveis globais, apresentam pequenas funções que formam esse contexto do escalonamento de processos, por exemplo, para verificar se a lista de processos da VCPU está vazia, buscar próximo CPU física, adicionar processos a lista de VCPU, associar VCPU a processador, associar uma estrutura privada de escalonamento a um domínio, entre outras.

Como podemos acompanhar na estrutura `csched_pcpu`, observarmos uma lista de processos para o CPU, e um índice para o último elemento da lista.

```
struct csched_pcpu {
    struct list_head runq;
    uint32_t runq_sort_last;
};
```

Na estrutura `csched_vcpu`, encontramos estruturas de processos, lista de VCPUs ativas, estrutura de domínio, a própria estrutura de VCPU, e um contador de créditos (ver seção 4.2.3) para cada VCPU.

```
struct csched_vcpu {
    struct list_head runq_elem;
    struct list_head active_vcpu_elem;
    struct csched_dom *sdom;
    struct vcpu *vcpu;
    atomic_t Credit;
    uint16_t flags;
    int16_t pri;
}
```

Aqui identificamos uma estrutura de dados, *Credit_last*, onde existe controle do estado do domínio, para que o compartilhamento da CPU seja justo entre todas as máquinas virtuais. Há créditos anteriores, contagem de créditos, estado ativo, estado ocioso, e estados para migração de processos.

```
#ifdef CSCHED_STATS
    struct {
        int Credit_last;
        uint32_t Credit_incr;
        uint32_t state_active;
        uint32_t state_idle;
        uint32_t migrate_q;
        uint32_t migrate_r;
    } stats;
#endif
};
```

Na estrutura de domínios, `csched_dom`, mostra uma lista de processo que contem VCPUs ativas, a própria estrutura de domínios, contador de VCPUs, peso e CAP (ver seção 1. Com a devida observação, implementamos CAP limite e CAP base (*cap_limit, cap_base*), que é uma forma de estruturar níveis de serviço, onde esses dois parâmetros não serão ultrapassados.

```
struct csched_dom {
    struct list_head active_vcpu;
    struct list_head active_sdom_elem;
    struct domain *dom;
    uint16_t active_vcpu_count;
    uint16_t weight;
    uint16_t cap;
    #ifdef PATCH_SCHED
    uint16_t cap_limit;
    uint16_t cap_base;
    #endif
};
```

Nessa estrutura, `csched_private`, exibe informações privadas associadas ao domínio, e as VCPUs associadas a ele. Mostra um bloqueio de migração de VCPUs em caso de balanceamento de carga, lista de domínios ativos, números de CPUs, peso e créditos do domínio, créditos para balanceamento de processos, ordenação das filas dos domínios, e a definição do estado do domínio.

```
struct csched_private {
    spinlock_t lock;
    struct list_head active_sdom;
    uint32_t ncpus;
    unsigned int master;
    cpumask_t idlers;
    uint32_t weight;
    uint32_t Credit;
    int Credit_balance;
    uint32_t runq_sort;
    CSCHED_STATS_DEFINE()
};
```

Uma vez descrita as estruturas principais do escalonador *Credit*, explicaremos as funções de inicialização dessas estruturas, padrões, definição das informações iniciais.

Como estrutura de inicialização *default*, temos o `scheduler sched_Credit_def`, como mostra a seguir:

```
struct scheduler sched_Credit_def = {
    .name           = "SMP Credit Scheduler",
    .opt_name       = "Credit",
    .sched_id       = XEN_SCHEDULER_CREDIT,

    .init_domain    = csched_dom_init,
    .destroy_domain = csched_dom_destroy,

    .init_vcpu      = csched_vcpu_init,
    .destroy_vcpu   = csched_vcpu_destroy,

    .sleep          = csched_vcpu_sleep,
    .wake           = csched_vcpu_wake,

    .adjust         = csched_dom_cntl,

    .pick_cpu       = csched_cpu_pick,
    .tick           = csched_tick,
    .do_schedule    = csched_schedule,

    .dump_cpu_state = csched_dump_pcpu,
    .dump_settings  = csched_dump,
    .init           = csched_init,
};
```

Onde temos as seguintes funções discriminadas:

- *csched_vcpu_init*: aloca informações sobre os domínios e inicializa os créditos e pesos;
- *csched_dom_destroy*: retira um domínio de execução;
- *sched_vcpu_sleep*: remove uma virtual CPU da fila de escalonamento;
- *csched_vcpu_wake*: coloca o domínio pronto para execução;
- *csched_dom_cntl*: seta parâmetros como peso e limite;
- *csched_tick*: contabiliza os créditos das CPUs virtuais em execução;
- *csched_schedule*: função que decide qual a próxima CPU virtual será escalonada, faz o balanceamento de carga através das prioridades, verifica em outras CPUs se existem processos com tarefas de maior prioridade;
- *csched_dump_pcpu*: imprime informações sobre as CPUs físicas;
- *csched_dump*: imprime uma lista de informações sobre o escalonamento nas CPUs;
- *csched_init*: função inicial que fixa as CPUs virtuais às CPUs físicas e recebe os valores de pesos e limites.

Após a estrutura de escalonamento inicial, é executada a função `csched_init(void)` para configurar os valores iniciais, como inicialização das listas de processos com domínios ativos, números de CPU, peso, CAP, créditos de balanceamento das máquinas, todas atribuído em zero, ou valores *default*.

Com a inicialização, estruturas iniciais, variáveis e constantes do escalonador *Credit* bem definido, mostraremos as principais funções, propondo formas de otimização, ou alternativas para tornar o sistema mais eficiente, com sugestões de estruturas de níveis de serviço, como dito antes, CAP limite e base.

O SMP *Credit* tem cinco funções principais de escalonamento:

- *__runq_tickle*
- *csched_cpu_pick*
- *csched_vcpu_acct*
- *csched_acct*
- *csched_schedule*

A função *__runq_tickle* é considerada como uma função auxiliar, mas importante, pois recebe um CPU, e uma VCPU nova, no momento que recebe essa VCPU, já a aloca em um processador, mesmo se esse processador já tiver VCPU, e logo após verifica sua prioridade dentre as outras VCPUs ativas. Se a sua prioridade no escalonamento for maior que as outras, então conferi em qual estado se encontra: *IDLE*, *OVER*, *UNDER* (ver seção 4.2.3).

Depois da verificação de estado, e sua associação a um CPU, verificasse, se o CPU tem dois VCPUs executando, se tiver, alerta CPUs *IDLE* que tem um VCPU extra rodando no sistema, e interrompe o designado CPU, rouba o VCPU que sobra para alocar em um CPU *IDLE.*

O *csched_cpu_pick* é uma função de rotina que recebe uma VCPU, e verifica em qual CPU se encontra. Está função prove preferência ao CPU inativo, com taxa de processamento mais baixa em seu agrupamento (VCPUs). Esta distribui trabalho em todos distintos núcleos primeiro, retira VCPUs que tem baixo desempenho ou *IDLE* e aloca em outro processador para obter um melhor aproveitamento de seus recursos. Possui também garantias para não fazer algo que perca tempo como gerir dois VCPUs sobre *co - hyperthreads* enquanto existem inativos núcleos ou *sockts* no sistema.

Já a função csched_vcpu_acct, recebe uma CPU por parâmetro e atribui automaticamente uma virtual CPU a ele. Se virtual CPU estava parada ou ociosa, então associasse uma variável que indica que ela está precisando de créditos para entrar em atividade, e logo após atualiza-se os créditos dela. Coloca-se essa virtual CPU e o domínio em que ela está associada de volta a lista de ativos do sistema, se essa lista de elementos da virtual CPU estiver vazia, reinicia-se a contagem de créditos do VCPU.

Nos Códigos 1,2,3, encontra-se a função *sched_acct*, que elabora a contagem de créditos para balanceamento de processos entre as VCPUs, como pode-se ver, no Código 1, apresentam estruturas de inicialização da função, como listas de iteração, estruturas de virtuais CPUs, domínios, e variáveis que representam crédito e peso para balanceamento.

A linha 18 bloqueia e salve a interrupção da estrutura de escalonamento privada, com a função *spin_lock_irqsave*, logo após atualiza-se com créditos e pesos dos domínios atuais, com as variáveis *weight_total* e *Credit_total*. Logo mais, duas verificações de rotina, uma para garantir que os créditos para balanceamento

não sejam negativos, e outra se consumiu todos os créditos dispostos pára de executar, e reinicia a contagem de créditos.

Código 1: Função de Contagem de créditos para balanceamento de Processamento dos Domínios.

```
01      static void
02      csched_acct(void)
03      {
04          unsigned long flags;
05          struct list_head *iter_vcpu, *next_vcpu;
06          struct list_head *iter_sdom, *next_sdom;
07          struct csched_vcpu *svc;
08          struct csched_dom *sdom;
09          uint32_t Credit_total;
10          uint32_t weight_total;
11          uint32_t weight_left;
12          uint32_t Credit_fair;
13          uint32_t Credit_peak;
14          uint32_t Credit_cap;
15          int Credit_balance;
16          int Credit_xtra;
17          int Credit;
18          spin_lock_irqsave(&csched_priv.lock, flags);
19          weight_total = csched_priv.weight;
20          Credit_total = csched_priv.Credit;
21          if ( csched_priv.Credit_balance < 0 )
22          {
23              Credit_total -= csched_priv.Credit_balance;
24              CSCHED_STAT_CRANK(acct_balance);
25          }
26          if ( unlikely(weight_total == 0) )
27          {
28              csched_priv.Credit_balance = 0;
29              spin_unlock_irqrestore(&csched_priv.lock, flags);
30              CSCHED_STAT_CRANK(acct_no_work);
31              return;
32          }
```

Em seguida, no Código 2, iniciam estados das variáveis, *weight*_left, *cap_total*, *Credit_xtra* e *Credit_balance*, e listam todos os domínios que estão ativados na estrutura *csched_priv*, e faz um teste para saber se o domínio está *IDLE*, e as suas estruturas internas estão devidamente iniciadas.

Assim, de acordo com a rotina imposta pela função, executa-se a contagem de créditos iniciais, nas condições a seguir (linhas 47 a 58), com as atribuições desse domínio, através da variável acumuladoras *Credit_peak* e *Credit_cap*, onde atribui quando *Credit_balance* for menor que zero, um calculo que representa, a multiplicação do peso do domínio (*sdom->weight*) com os créditos de balanceamento da estrutura de escalonamento privada (*csched_priv->Credit_balance*) ,menos o *weight_total* dividido por ele mesmo.

Já se o *sdom->cap* (linha 51) não possuir valor, que é estipulado pelo usuário do sistema, representando fatia de processamento, este domínio pode usar cem por cento do CPU, se não tiver outro domínio concorrente.

Depois de definido esses passos, o escalonador faz um cálculo que considera como *Crédito* justo (*Credit*_fair, linha 58) para cada domínio que é apresentado, da seguinte maneira, *Crédito* total (*Credit_total*) vezes peso do domínio (sdom->weight) mais peso total mais um (weight_total + 1) dividido por ele mesmo. Portanto, o escalonador tenta ajusta conforme os créditos, o tempo de processamento de cada domínio, aproximando as taxas de processamento.

Após a definição do crédito justo, faz a verificação se há necessidade de créditos extra (*Credit_xtra*), assim fazendo uma comparação com o pico de crédito (*Credit_peak*) com o crédito justo (*Credit_fair*), se crédito justo for menor, então aciona a *flag* do *Credit_xtra.* Senão é realizado outro calculo para atribuição do *Crédito* total (*Credit_total),* basicamente igual ao calculo anterior, apenas troca-se as variáveis.

Essa *flag* de créditos extra (linha 69) estiver acionada, reordena a contagem de créditos, apaga-se a lista de domínios ativos, e adiciona-o novamente, retornando a contagem de créditos de forma a priorizar seu escalonamento, assim em não ficando ociosa, ou com muito tempo esperando para ser executada.

Código 2: Função de Contagem de créditos para balanceamento de Processamento dos Domínios (Continuação).

```
33          CSCHED_STAT_CRANK(acct_run);
34          weight_left = weight_total;
35          Credit_balance = 0;
36          Credit_xtra = 0;
37          Credit_cap = 0U;
38          list_for_each_safe( iter_sdom, next_sdom, &csched_priv.active_sdom )
39          {
40              sdom = list_entry(iter_sdom, struct csched_dom, active_sdom_elem);
41              BUG_ON( is_idle_domain(sdom->dom) );
42              BUG_ON( sdom->active_vcpu_count == 0 );
43              BUG_ON( sdom->weight == 0 );
44              BUG_ON( sdom->weight > weight_left );
45              weight_left -= sdom->weight;
46              Credit_peak = sdom->active_vcpu_count * CSCHED_CREDITS_PER_ACCT;
47              if ( csched_priv.Credit_balance < 0 )
48              {
49                  Credit_peak += ( ( -csched_priv.Credit_balance * sdom->weight) +
                                     (weight_total - 1)
                                   ) / weight_total;
50              }
51              if ( sdom->cap != 0U )
52              {
53                  Credit_cap = ((sdom->cap * CSCHED_CREDITS_PER_ACCT) + 99) / 100;
54                  if ( Credit_cap < Credit_peak )
55                      Credit_peak = Credit_cap;
56                  Credit_cap = ( Credit_cap + ( sdom->active_vcpu_count - 1 )
                                 ) / sdom->active_vcpu_count;
57              }
58              Credit_fair = ( ( Credit_total * sdom->weight) + (weight_total - 1)
                              ) / weight_total;
59              if ( Credit_fair < Credit_peak )
60              {
61                  Credit_xtra = 1;
62              }
63              else
64              {
65                  if ( weight_left != 0U )
66                  {
67                   Credit_total += ( ( ( Credit_fair - Credit_peak) * weight_total)
                                       + ( weight_left - 1 )) / weight_left;
68                  }
69                  if ( Credit_xtra )
70                  {
71                     CSCHED_STAT_CRANK(acct_reorder);
72                     list_del(&sdom->active_sdom_elem);
73                     list_add(&sdom->active_sdom_elem, &csched_priv.active_sdom);
74                  }
75                  Credit_fair = Credit_peak;
76              }
77              Credit_fair = ( Credit_fair + ( sdom->active_vcpu_count - 1 )
                              ) / sdom->active_vcpu_count;
```

As linhas 78 a 83 do Código 3 realizam uma varredura de todos virtuais CPUs que estão ativas no sistema, e verificam se eles pertencem ao domínio correspondente, então atribui os créditos justos (*Credit_fair*) ao virtual CPU do domínio, e atualiza a variável *Credit* (linha 83) com a função *atomic_read* que faz a leitura dos créditos da virtual CPU daquele domínio.

Na análise do Código 3, observa-se que entre as linhas 84 e 113, é uma verificação de rotina para declarar em qual estado encontra-se a virtual CPU, por exemplo, se a *Crédito* da VCPU (*virtual* CPU) for menor que zero, então a prioridade é definida como *OVER*, e se ainda for menor que *time-slice* do escalonamento privado (*CSCHED_CREDITS_PER_TSLICE*), atribui-se esse valor ao Crédito (*Credit*, linha 96), pois esse *define* citado acima é o valor padrão de crédito.

Após a verificação desses créditos para balanceamento, atribui-se a variável *Credit* ao acumulador *Credit_balance*, e logo mais a estrutura de escalonamento privada de balanceamento de créditos (*csched_priv->Credit_balance*), assim liberando e restaurando a interrupção dessa estrutura e incrementando a lista de processos dela.

Na função *csched_acct(void)* foi incorporado a implementação de níveis de serviços (serão mais detalhado na seção seguinte). Utilizando CAP base e limite, que seria uma faixa de processamento que não poderia ser ultrapassada, assim estipulando um mínimo e um máximo para cada domínio de acordo com o definido no contrato de SLAs (*Service Level Agreement*).

Código 3: Função de Contagem de créditos para balanceamento de Processamento dos Domínios (Continuação).

```
78              list_for_each_safe( iter_vcpu, next_vcpu, &sdom->active_vcpu )
79              {
80                  svc = list_entry(iter_vcpu, struct csched_vcpu, active_vcpu_elem);
81                  BUG_ON( sdom != svc->sdom );
82                  atomic_add(Credit_fair, &svc->Credit);
83                  Credit = atomic_read(&svc->Credit);
84                  if ( Credit < 0 )
85                  {
86                      svc->pri = CSCHED_PRI_TS_OVER;
87                     if ( sdom->cap != 0U &&  Credit < -Credit_cap &&
               !(svc->flags & CSCHED_FLAG_VCPU_PARKED) )
88                      {
89                          CSCHED_STAT_CRANK(vcpu_park);
90                          vcpu_pause_nosync(svc->vcpu);
91                          svc->flags |= CSCHED_FLAG_VCPU_PARKED;
92                      }
93                      if ( Credit < -CSCHED_CREDITS_PER_TSLICE )
94                      {
95                          CSCHED_STAT_CRANK(acct_min_Credit);
96                          Credit = -CSCHED_CREDITS_PER_TSLICE;
97                          atomic_set(&svc->Credit, Credit);
98                      }
99                  }
100                 else
101                 {
102                     svc->pri = CSCHED_PRI_TS_UNDER;
103                     if ( svc->flags & CSCHED_FLAG_VCPU_PARKED)
104                     {
105                         CSCHED_STAT_CRANK(vcpu_unpark);
106                         vcpu_unpause(svc->vcpu);
107                         svc->flags &= ~CSCHED_FLAG_VCPU_PARKED;
108                     }
109                     if ( Credit > CSCHED_CREDITS_PER_TSLICE )
110                     {
111                         __csched_vcpu_acct_stop_locked(svc);
112                         Credit = 0;
113                         atomic_set(&svc->Credit, Credit);
114                     }
115                 }
116                 CSCHED_VCPU_STAT_SET(svc, Credit_last, Credit);
117                 CSCHED_VCPU_STAT_SET(svc, Credit_incr, Credit_fair);
118                 Credit_balance += Credit;
119             }
120         }
121         csched_priv.Credit_balance = Credit_balance;
122         spin_unlock_irqrestore(&csched_priv.lock, flags);
123         csched_priv.runq_sort++;
124     }
```

No código 4, função *csched_schedule(s_time now) ou do_schedule*, entre as linhas 04 e 08, são definidas as variáveis e estruturas da função, associando um *id processor* a uma variável CPU, declarando uma lista de processos dessa CPU, e determinando qual VCPU que está rodando no sistema irá alocar aquele processador. Após ordena os processos e checa as VCPUs, verificando se a VCPU corrente está no estado *runnable* ou ‘r’, inseri-se na lista de processos do CPU, senão confere-se se está no estado *IDLE* ou se a lista de processos de VCPUs está vazia, se estiver, procura o próximo VCPU, e adiciona a lista de elementos da estrutura VCPU.

Logo mais, averigua-se a prioridade dessa VCPU, se estiver *OVER* remove-se da lista de processos *(__runq_remove(snext)*, linha 17), senão faz o balanceamento de carga (*csched_load_balance(cpu,next)*, linha 19). Detalhando mais o passo acima, se o próximo VCPU de maior prioridade estiver no estado *runnable*, e já tiver usado todos seus créditos, então olha-se para os outros CPUs para ver se tem outra VCPU rodando com créditos. Se isso não acontecer, chama a função *csched_load_balance()* que irá retornar *snext* (próxima VCPU).

O balanceamento de carga consiste em “roubar” uma VCPU e alocar em outro processador, varrendo todos os CPUs existentes e verificando cada VCPU e seu estado corrente. Confere se tem um VCPU *non-IDLE* para colocar em outro CPU. Assim fazendo o balanceamento de VCPUs que estão executando.

As linhas 20 a 28 observam-se uma rotina, que defini em que estado se encontra a VCPU, e aloca-se ou limpa a CPU correspondente. No final da função associa-se o tempo padrão *CSCHED_MSECS_PER_TSLICE* (30 milisegundos) e retorna a VCPU corrente.

Código 4: Função de *Scheduling*, fatia de tempo para os processos dos domínios.

```
01      static struct task_slice
02      csched_schedule(s_time_t now)
03      {
04          const int cpu = smp_processor_id();
05          struct list_head * const runq = RUNQ(cpu);
06          struct csched_vcpu * const scurr = CSCHED_VCPU(current);
07          struct csched_vcpu *snext;
08          struct task_slice ret;
09          CSCHED_STAT_CRANK(schedule);
10          CSCHED_VCPU_CHECK(current);
11          if ( vcpu_runnable(current) )
12              __runq_insert(cpu, scurr);
13          else
14              BUG_ON( is_idle_vcpu(current) || list_empty(runq) );
15          snext = __runq_elem(runq->next);
16          if ( snext->pri > CSCHED_PRI_TS_OVER )
17              __runq_remove(snext);
18          else
19              snext = csched_load_balance(cpu, snext);
20          if ( snext->pri == CSCHED_PRI_IDLE )
21          {
22              if ( !cpu_isset(cpu, csched_priv.idlers) )
23                  cpu_set(cpu, csched_priv.idlers);
24          }
25          else if ( cpu_isset(cpu, csched_priv.idlers) )
26          {
27              cpu_clear(cpu, csched_priv.idlers);
28          }
29          ret.time = MILLISECS(CSCHED_MSECS_PER_TSLICE);
30          ret.task = snext->vcpu;
31          CSCHED_VCPU_CHECK(ret.task);
32          return ret;
33      }
```

Nessa função a idéia de implementação é estipular uma variável global de tempo e analisar o comportamento de cada domínio em relação há esse tempo. Como solução, definimos diversos tempos estáticos, para criar tempos dinâmicos de acesso ao CPU as máquinas virtuais. Assim poderá adquirir um desempenho mais dinâmico, montando uma relação entre domínio, tempo de processamento, e tipo de processo nele executado.

6.2 Soluções para o Escalonador Credit

O principal problema encontrado durante a Análise do Escalonador *Credit*, é o tempo de processamento para as máquinas virtuais, que é estático, ou seja 30 milisegundos.

Para solucionar este problema, precisamos entender como funciona o relacionamento entre domínio, VCPU e CPU, o foco da implementação do trabalho. Na Ilustração 6, representado no diagrama E-R (meramente ilustrativo), nos

mostram a estrutura desse relacionamento antes e depois da solução. Como pode se observar todo o domínio possui uma VCPU, e todo VCPU possui CPU. Sabemos que cada domínio possui créditos que liga com as prioridades dos VCPU, por exemplo, dependendo do número de créditos, se altera a prioridade do VCPU, mas continuamos sem nenhuma conexão com o tempo de escalonamento, que permanece constante em 30 milisegundo. Ainda na Ilustração 6, depois da implementação da solução podemos perceber que há uma relação entre prioridades do VCPU com tempo de escalonamento, dependendo da prioridade imposta com os créditos dos domínios alterará seu tempo de acesso ao CPU.

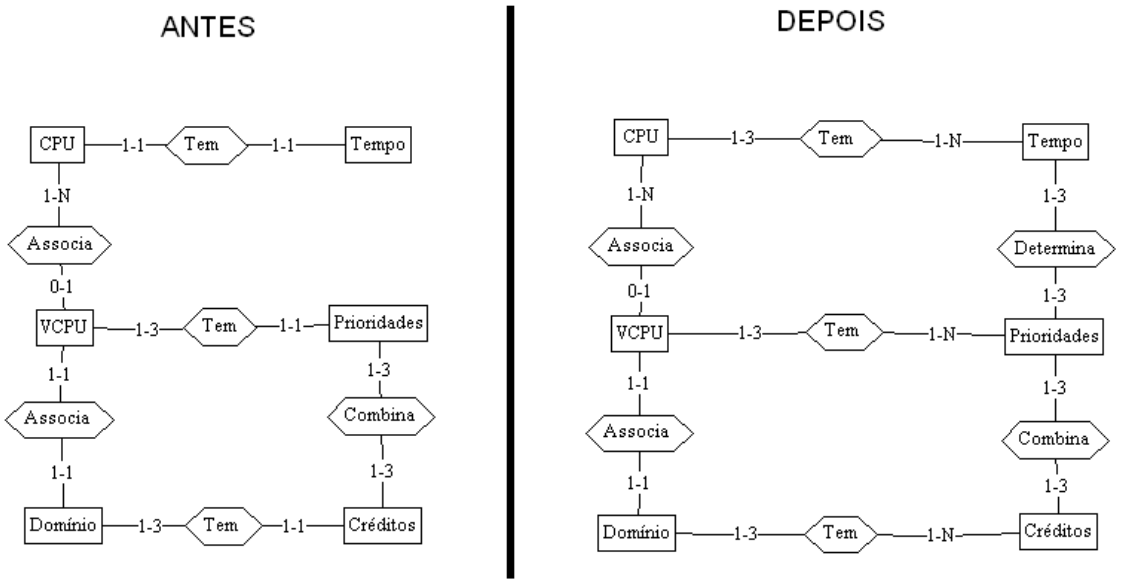

Ilustração 6: Diagrama E-R da VM

Depois de alterado o relacionado, entre domínio, VCPU e CPU, e adicionando uma ligação entre prioridades e tempo, podemos ver como o funciona o escalonador *Credit* após estas modificações.

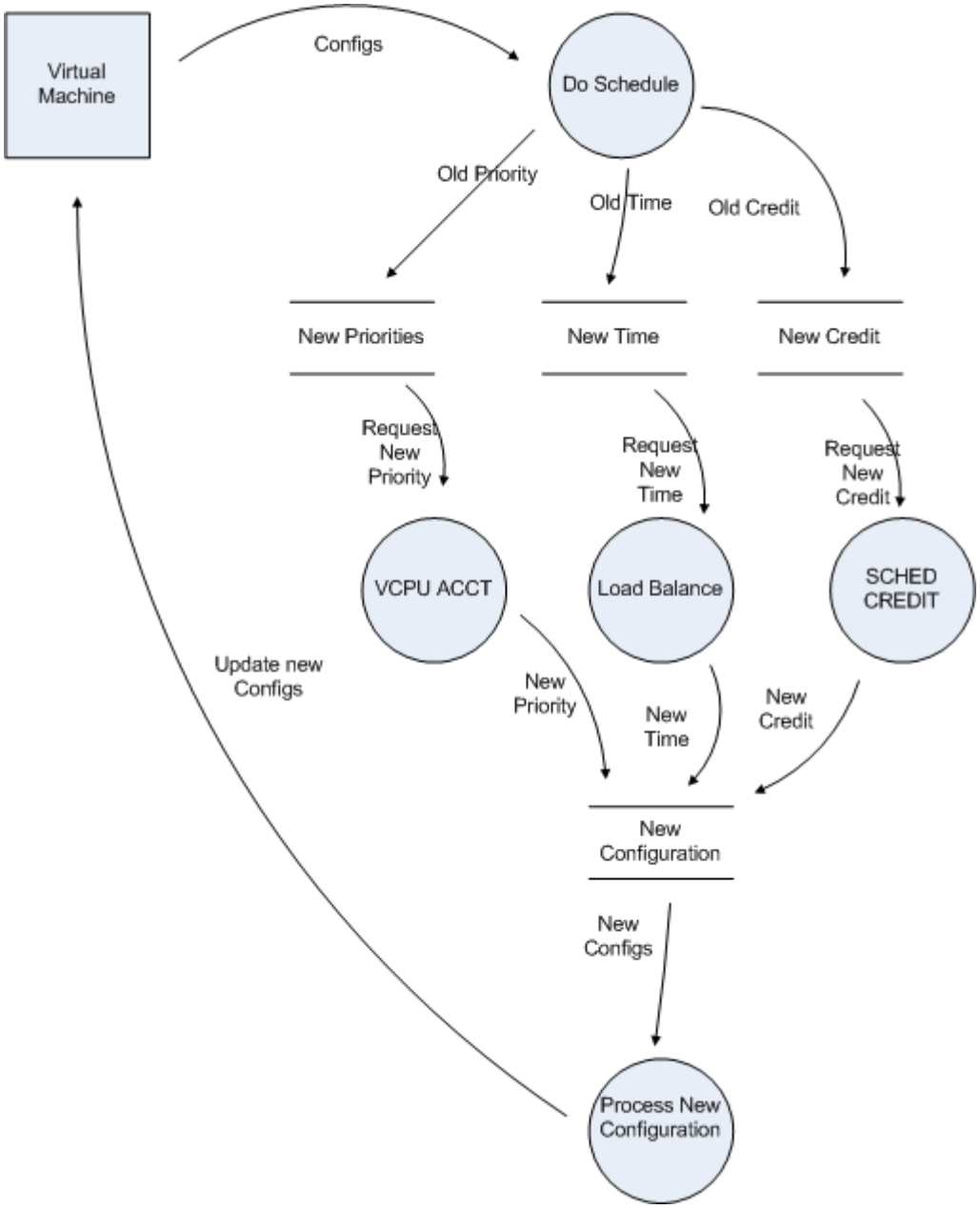

Ilustração 7: Funcionamento do Escalonador Credit

Na Ilustração 7 podemos observar como funciona o escalonador *Credit*, e quais etapas são percorridas para re-alocar suas novas configurações. Primeiramente a máquina virtual (*Virtual Machine*) que será escalonada contém informações como créditos, prioridade do VCPU, e tempo de acesso ao CPU (*configs*), que passa ao *Do Schedule*, que realiza o escalonamento. Estas informações são delegadas para o *New Priorities, New Time, New Credit*, que simplesmente verificam se essas informações estão de acordo com as prioridades, tempo, e créditos do escalonador, e se é necessário mais processamento para a

máquina virtual. Havendo a necessidade de maior desempenho, é feito uma requisição de novos parâmetros (prioridade, tempo e créditos) através dos processos *VCPU ACCT* (aumenta a prioridade do VCPU), *Load Balance* que determina um tempo de acordo com a prioridade, e o *SCHED CREDIT* que realiza a contagem de créditos, assim realizando uma nova configuração (*New Configuration*). Com essas novas configurações, é executado um processo de verificação (*Process new Configuration*), que verifica se esta nova configuração irá aumentar o desempenho, aplicando a mesma para a máquina virtual (*update new configs*).

A próxima seção mostrará como foram implementadas essas alterações, e de que forma alteraram seu funcionamento, incorporamos também níveis de serviço às máquinas virtuais, impondo limites de processamento a elas.

6.2.1 Implementação das Soluções para Escalonador Credit

No Escalonador *Credit*, para a implementação de níveis de serviço e tempos dinâmicos de acesso ao CPU para as máquinas virtuais, foi definido uma nova *flag* de compilação (–DPATCH_TIME), que é atribuída ao *Credit*. Esta implementação criada, deixa como opção ao usuário se deseja incorporar a mudança realizada.

Na implementação de níveis de serviço dentro do escalonador, é utilizado duas definições, o LIMIT_CAP e o BASE_CAP. Essas atribuições foram realizadas no início do código do escalonador *Credit*. As definições LIMIT_CAP e BASE_CAP, são respectivamente o máximo e o mínimo de processamento que máquina virtual atingirá. E essas definições são consideradas como gerais, por exemplo, é definido LIMIT_CAP de 80, e BASE_CAP de 20, então todas as máquinas virtuais poderão no máximo atingir 80%, e um mínimo 20% de processamento do CAP imposto pelo usuário.

```
#define LIMIT_CAP 80
#define BASE_CAP 20
```

Para solucionar o problema de tempo de acesso ao CPU, propomos a redefinição de alguns parâmetros, como o tempo de escalonamento que era estático com 30 milisegundos para cada associação VCPU a domínio. Agora são considerados três tipos de tempos, implementados através de definições. As três definições são:

- O normal, tempo padrão definido pelo *Credit* (30 milisegundos de acesso ao CPU);
- O máximo, que determina 30% a mais tempo de acesso ao CPU que o definido pelo *Credit*;
- O mínimo, que lhe concede 30% a menos de tempo de acesso ao CPU que o definido pelo *Credit*.

Para haver uma ligação entre tempo de escalonamento e a implementação de níveis de serviços, alteramos a contagem de créditos. Por exemplo, se tiver uma prioridade máxima de tempo para alguma máquina virtual, então deve haver uma contagem créditos máxima, assim potencializando ainda mais o desempenho das máquinas virtuais. Para essas definições de contagem de créditos mantemos os mesmo valores que foram imposto para a implementação de tempos dinâmicos para as máquinas virtuais, os 30% a mais para `CSCHED_TICKS_PER_ACCT_MAX` (máxima), e 30% a menos para `CSCHED_TICKS_PER_ACCT_MIN` (mínima).

```
#define CSCHED_TICKS_PER_ACCT_MIN          2.1
#define CSCHED_TICKS_PER_ACCT_MAX          3.9
#define CSCHED_TICKS_PER_TSLICE_MIN        2.1
 #define CSCHED_TICKS_PER_TSLICE_MAX       3.9
```

O Escalonador *Credit* possui uma constante de tempo de 10 milisegundos (`CSCHED_MSECS_PER_TICK`), que é considerado o tempo de funcionamento básico para as máquinas virtuais, por exemplo, para criação de um domínio. Então para não ser sustentado apenas com o processamento mínimo, é adicionado um tempo extra. Esse tempo extra, é calculado utilizando uma constante de *time slice* (3 milisegundos), estipulados pelo *Credit* (`CSCHED_TICKS_PER_TSLICE`).

```
#define CSCHED_MSECS_PER_TSLICE \
(CSCHED_MSECS_PER_TICK * CSCHED_TICKS_PER_TSLICE)
```

Contudo, realizamos uma otimização do tempo padrão do escalonador *Credit*, implementando mais duas constantes de tempo, criando um ambiente dinâmico, atribuindo as definições de tempo mínima e máxima.

```
#define CSCHED_MSECS_PER_TSLICE_MIN \
(CSCHED_MSECS_PER_TICK * CSCHED_TICKS_PER_TSLICE_MIN)
```

```
#define CSCHED_MSECS_PER_TSLICE_MAX \
(CSCHED_MSECS_PER_TICK / CSCHED_TICKS_PER_TSLICE_MAX)
```

Nas prioridades estipuladas pelo *Credit*, para não haver conflitos, é adicionado mais duas definições, uma prioridade máxima, que lhe concede maior prioridade de acesso ao processador, e contagem de créditos. E a prioridade mínima, que diminui o tempo de acesso, e conseqüentemente seus créditos.

```
#define CSCHED_MAX_PRIORY           -3
#define CSCHED_MIN_PRIORY           -4
```

Na função `sched_dom` (descrita na seção 6.1) é adicionado mais dois atributos, o CAP limite e base. Assim incorporando níveis de serviço.

```
uint16_t cap_limit;
uint16_t cap_base;
```

Na implementação da solução, tanto para níveis de serviço quanto para tempos dinâmicos de acesso ao CPU para máquinas virtuais é alterado a função `csched_vcpu_init`. Essa função realiza a associação de um VCPU a um domínio, e um domínio a uma estrutura privada de escalonamento, assim iniciando a lista de processos básicos do VCPU. Portanto, verifica-se o estado da VCPU, se estiver *IDLE* lhe concede a prioridade `CSCHED_PRI_IDLE`, senão `CSCHED_PRI_UNDER` (visto no capitulo do *Credit Scheduler*, seção 4.2.3). Verifica-se também se o domínio está *IDLE*, assim lhe dando a prioridade `CSCHED_PRI_IDLE`, senão a prioridade `CSCHED_PRI_MAX`.

```
svc->pri = is_idle_domain(dom) ? CSCHED_PRI_IDLE : CSCHED_MAX_PRIORY;
```

Na função `csched_vcpu_acct`, que associa uma VCPU a um CPU, e verifica se um domínio está devidamente ligado a essa VCPU. Acrescentamos uma nova condição, no caso de haver a necessidade de trocar a prioridade. Até o momento era alterada sua prioridade somente quando uma VCPU está saindo do estado IDLE. Essa condição atualiza a prioridade do VCPU, analisando o estado atual. Se a VCPU estiver na prioridade mínima e precisar de mais processamento, passamos para prioridade máxima

```
if(svc->pri == CSCHED_MIN_PRIORY)
svc->pri = CSCHED_MAX_PRIORY
```

Na inicialização do domínio, acrescentamos as atribuições de valores ao `cap_limite` e `cap_base` na função `csched_dom_init` (inicialização da estrutura de escalonamento do domínio).

```
sdom->cap_limit = LIMIT_CAP;
sdom->cap_base = BASE_CAP;
```

Na função `csched_dom_cntl`, trabalhamos com controles de domínios, com atribuições costumeiras no código do *credit scheduler*, como as configurações iniciais dos domínios para a estrutura privada de escalonamento (*cap*, *weight*, e suas prioridades).

Para garantir a consistência da implementação de níveis de serviço, é acrescentada uma condição para contagem de créditos. Se tiver algum CAP definido anteriormente pelo usuário, então atribuem os valores definidos pelo LIMIT_CAP e BASE_CAP.

```
sdom->cap = (op->u.credit.cap*CAP_LIMIT)/100;
sdom->cap_limit = CAP_LIMIT;
sdom->cap_base = BASE_CAP;
```

Na função de contagem de créditos, `csched_acct` (visto na seção 6.1), após passar pelas declarações de variáveis, e suas verificações de rotina, como não deixar instanciado créditos negativos, checar as ligações entre VCPUs e os domínios. Adicionamos uma nova condição que verifica se o domínio ou VCPU está *IDLE*, se estiver então seta prioridade mínima, mostrando algo que não existia anteriormente. Diminuindo o tempo de processamento, e a contagem de créditos para a máquina virtual. É uma condição em *real time* que define tempos para VCPUs ter acesso ao CPU.

```
if(is_idle_domain(sdom->dom) || is_idle_vcpu(svc->vcpu))
 {
        svc->pri = CSCHED_MIN_PRIORY;
        credit_peak = sdom->active_vcpu_count *
        CSCHED_CREDITS_PER_ACCT_MIN;
  }
```

Agora, se o domínio estiver associado diretamente há uma CPU (significa sem compartilhamento do CPU com qualquer outra máquina virtual) ou um domínio configurado como privilegiado (possui prioridade na contagem de créditos, ou privilégio de acesso ao CPU), então lhe concede prioridade máxima (maior tempo de acesso ao CPU).

```
if(is_pinned(sdom->dom) || is_privileged(sdom->dom))
  {
        svc->pri = CSCHED_MAX_PRIORY;
         credit_peak = sdom->active_vcpu_count *
        CSCHED_CREDITS_PER_ACCT_MAX;
  }
```

Continuando no `csched_acct`, veremos uma outra rotina que analisa, se está configurado algum CAP para o domínio corrente no escalonamento, então é definido uma contagem de credito conforme estipulado nas definições LIMIT_CAP e BASE_CAP. Sendo recalculado os créditos para este domínio.

```
credit_cap = ((sdom->cap_limit * CSCHED_CREDITS_PER_ACCT) + 99) / 100;
credit_cap_base = ((sdom->cap_base * CSCHED_CREDITS_PER_ACCT) + 99) / 100;
```

A condição a seguir imposta, é elaborada para complementar as implementações de camadas de níveis de serviço. Verificando se *credit cap* (contagem de créditos atuais) está menor que `credit_cap_base` (recalculado na função acima), se verdadeira então impedi de atribuir créditos menores que o devido pelo BASE_CAP , passando `cap_base` para os créditos atuais.

```
if(credit_cap < credit_peak && credit_cap < credit_cap_base)
 {
   credit_peak = sdom->cap_base;
 }
```

Concluindo a implementação da solução, consideramos a função `do_schedule` mais importante do escalonador *Credit*, pois condiciona o tempo para cada máquina virtual. Então como era definida apenas um tempo, de 30 milisegundos, e agora foi alterado para três tempos, de acordo com a condição da máquina virtual. Se estiver IDLE, ou com pouco funcionamento é lhe acrescentado à prioridade mínima. Se estiver com prioridade PRI e for acrescentado mais créditos, então é lhe dado o

acesso máximo, senão o normal ou IDLE (quando uma estiver fora de funcionamento).

```
if ( snext->pri == CSCHED_MAX_PRIORY )
   ret.time = MILLISECS(CSCHED_MSECS_PER_TSLICE_MAX);

else if ( snext->pri == CSCHED_MIN_PRIORY )
       ret.time = MILLISECS(CSCHED_MSECS_PER_TSLICE_MIN);

else {

   ret.time = MILLISECS(CSCHED_MSECS_PER_TSLICE);
}
```

Durante a implementação da solução são condicionadas prioridades e créditos, para distinguir qual máquina virtual terá mais acesso ao CPU. E nesta função (`do_schedule`) é encontrado o relacionamento entre tempo dinâmico proposto e o domínio.

A próxima seção apresentará a Análise dos resultados, mostrando se as alterações realizadas obtiveram o desempenho desejado.

6.3 Análise dos Resultados

Nesse capítulo veremos se implementação proposta (ver seção 6.2.1) resultou em um maior desempenho para as máquinas virtuais.

Na Análise dos Resultados foram realizados testes (execução de *benchmarks*) sobre as máquinas virtuais, proporcionando formas padronizadas de avaliação do desempenho de sistemas, retirando estatísticas do seu funcionamento. Nas próximas seções serão apresentados o ambiente de execução, e como foram realizadas as execuções desse *benchmark*, assim como os resultados gerados.

6.3.1 Ambiente de Execução do Benchmark

Para execução dos benchmark foi escolhido o seguinte ambiente:

- Dell Dimension 5150: 1 processadores Celeron D, 2.8 Ghz, 2 GB de memória RAM
- Benchmark Unixbench: Realização de 45 execuções sem a alteração, e 45 execuções com a alteração proposta.
- Configuração Inicial dos Testes:

CPU: 100% (dividido pelo Credit), configuração *default* do escalonador Credit

Memória: 128 MB para todas as máquinas virtuais

6.3.2 Processo de Execução do Benchmark

No processo de execução de *benchmark*, é necessário preparar o ambiente para que possam ser refeitos os testes, assim criando uma maior confiabilidade dos resultados gerados. Para esta fase de teste, é indispensável que o ambiente possua tais configurações:

- Xen 3.1, instalação e configuração para funcionamento do sistema.
- Domínio 0 (hypervisor) com cento e noventa e seis de memória, que representa a camada de conexão entre as máquinas virtuais.
- Debian como SO em cada máquina virtual.
- De uma até cinco máquinas virtuais ativas no Xen.
- Unixbench (Descrito no Capítulo 5), *benchmark* utilizado para teste de desempenho, na qual é a melhor indicada para encontrar gargalos em partes específicas de subsistemas do *kernel*.
- Execução simultânea do *benchmark* no número de máquinas virtuais ativas.
- Após cada execução reiniciar a máquina.

Todos os itens citados fazem parte de todas as execuções durante este trabalho. Essas características são as configurações fixas, as condições mínimas para que o *benchmark* gere resultados satisfatórios.

6.3.3 Tabela Índices de Desempenho

Nessa seção mostramos os valores que foram escolhidos para avaliar a alteração no sistema, como proposto anteriormente (ver seção 6.2). Através da tabela podemos identificar testes de processamento (CPU), de leitura e escrita, aritméticos, chamada de sistema, *dhrystone*, entre outros.

Para garantir confiabilidade dos resultados, foram executadas três vezes em cada modalidade, para com e sem alteração no escalonador *Credit*. Na tabela 1, está à média dos índices feitos por essas execuções.

DESEMPENHO TOTAL C/ PATCH

	vm01	vm02	vm03	vm04	vm05
Arithmetic Test (type = double)	294,6	153,1	100,5333	74,325	65,54
Dhrystone 2 using register variables	500,1	240,5	152,7333	131,7	90,28
Execl Throughput	412,2	199,85	136	110,9	77,28
File Copy 1024 bufsize 2000 maxblocks	1080	594,9	370,4	278,75	250,78
File Copy 256 bufsize 500 maxblocks	999,5	501	332,3	241,225	183,3
File Copy 4096 bufsize 8000 maxblocks	1600,8	737,4	515,2	361,925	283,54
Pipe Throughput	801,1	402,25	263,2667	191,025	169,72
Pipe-based Context Switching	391,2	182,65	126,6	89,95	71,64
Process Creation	331,2	191,75	120,6	90,3	75,3
Shell Scripts (8 concurrent)	671,8	371,75	251,4	182	159,94
System Call Overhead	682,4	372	251,7333	180,6	173,26
FINAL SCORE	627,6	300,55	207,4333	158,8	122,02

Tabela 1: Índice de Desempenho com *Patch*

DESEMPENHO TOTAL S/ PATCH

	vm01	vm02	vm03	vm04	vm05
Arithmetic Test (type = double)	270,8	133,95	88,86667	68,8	52,58
Dhrystone 2 using register variables	432,9	219,1	143,1	117,2	83,84
Execl Throughput	382,6	190,85	126,7333	94,375	74,84
File Copy 1024 bufsize 2000 maxblocks	1079	542,85	358,5	267,75	213,62
File Copy 256 bufsize 500 maxblocks	900	451,6	299,0667	222,95	178,12
File Copy 4096 bufsize 8000 maxblocks	1398,7	694,85	463,0667	342,85	273,8
Pipe Throughput	722,6	353,25	233,4	178,875	141,04
Pipe-based Context Switching	326,1	159,05	105,1333	79,025	62,86
Process Creation	320,3	163,8	106,7667	79,4	63,26
Shell Scripts (8 concurrent)	668,8	334,3	221,1333	163,65	129,9
System Call Overhead	678,8	330,05	220,0667	164,875	130,6
FINAL SCORE	570,6	283,95	187,7333	141,925	111,36

Tabela 2: Índice de Desempenho sem *Patch*

6.3.4 Gráficos de Desempenho

Os gráficos de desempenho mostram uma comparação entre *patch* inserido no sistema e sistema padrão. Para a criação dos gráficos de desempenho foram utilizados os índices da Tabela 1 (ver seção 6.3.3). Na composição do gráfico nota-se três nomes:

- Nome do teste;
- Número de indexações, que representa índices por segundo (*loop per second,* quanto maior melhor);
- Número de máquinas virtuais (VMs) ativas no sistema.

As Ilustrações 8 e 9 consistem em testes que avaliam mais o desempenho do CPU, através de testes aritméticos, e o *Dhrystone* que envolve *arrays*, endereçamento, em geral controle de fluxo. Como pode se observar, há uma ligeira melhora na alteração proposta, porém com tendência a igualar-se ao tempo estático (30 milisegundos), padrão do escalonador *Credit*. Com uma VM (máquina virtual) ativa no Xen a melhora é mais acentuada e com cinco máquinas virtuais quase não se percebe a melhora.

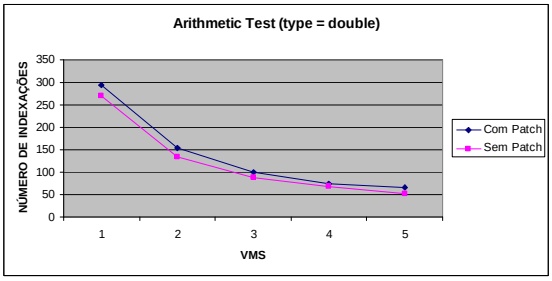

Ilustração 8: Arithmetic Test

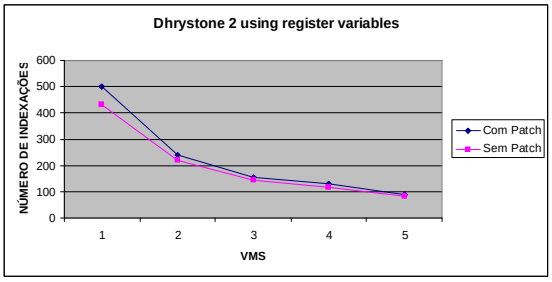

Ilustração 9: Dhrystone Test

O teste *Execl Throughput*, Ilustração 10, consiste em troca o processo corrente por um novo. Podemos ver no gráfico que no sistema alterado há uma melhora em seu desempenho, com uma até quatro VMs, e na quinta praticamente iguala-se ao sistema original.

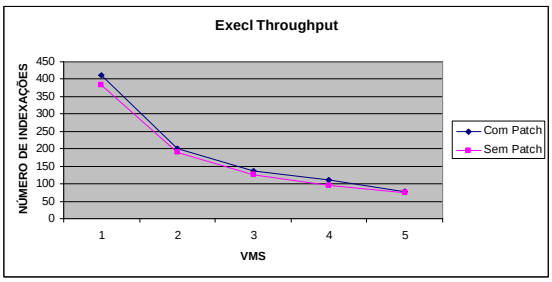

Ilustração 10: Execl Throughtput

O *System Call* Overhead, Ilustração 11, que testa o desempenho das chamadas de sistemas, demonstra um comportamento atípico dos demais testes, mostrando no sistema alterado um desempenho inferior com uma VM, superior com duas e três VMs, igualando-se com quatro VMs, e superior com cinco VMS, em relação ao sistema original. Esse gráfico mostra que provavelmente houve diversas alterações no tempo de escalonamento.

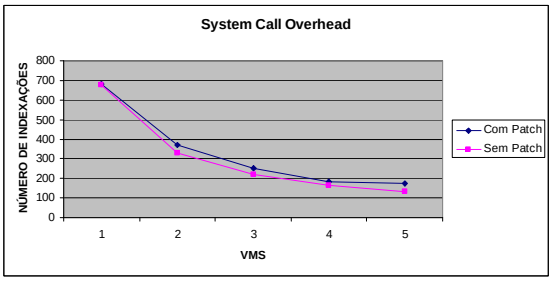

Ilustração 11: System Calls

Os testes *Pipe-base Context Switching*, e *Pipe Throughput* criam canais de conversação entre processos. Respectivamente Ilustrações 12 e 13, começamos a verificar um padrão entre a alteração realizada e o sistema padrão. Um desempenho maior com uma VM, e quando vai aumentando o número de VMs vai diminuindo seu desempenho.

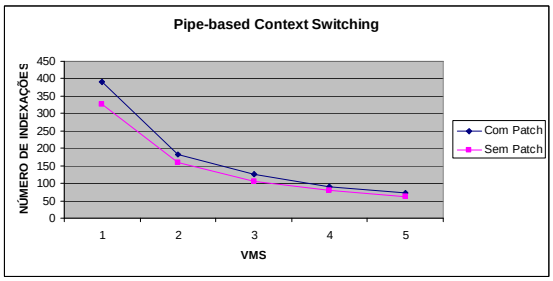

Ilustração 12: Pipe-based

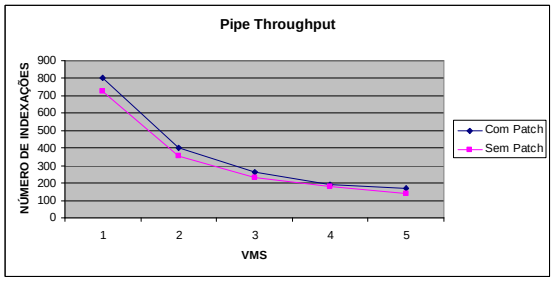

Ilustração 13: Pipe Throughput

Na Ilustração 14, vemos os testes de leitura, escrita e cópia. Podemos observar que houve um maior desempenho, de forma significativa, com uma VM, e quando aumentamos o número de VMs, fica praticamente igual ao desempenho do sistema original.

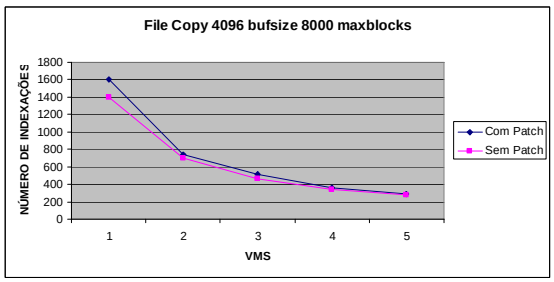

Ilustração 14: File Read, Write & Copy

Nas Ilustrações 15 e 16, mostram respectivamente testes de criação do processo, e execução de um script para executar de um a oito processos concomitantes. O resultado em ambos os testes são similares, havendo apenas uma ligeira melhora quando está com duas e três VMs ativa no sistema modificado.

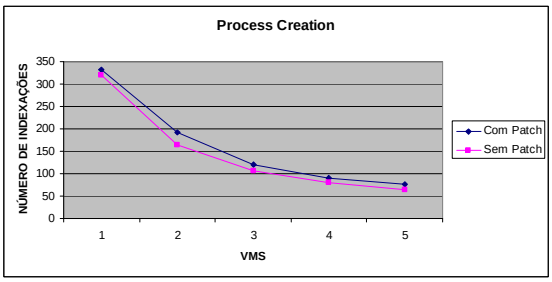

Ilustração 15: Process Creation

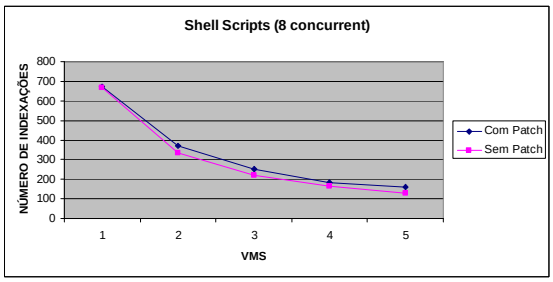

Ilustração 16: Shell Scripts

Como podemos ver, a Ilustração 17 mostra o escore final do *benchmark* Unixbench. O escore final é composto dos testes descritos na seção 5.1, e mostra comportamento das máquinas virtuais diante da modificação realizada em comparação com sistema original.

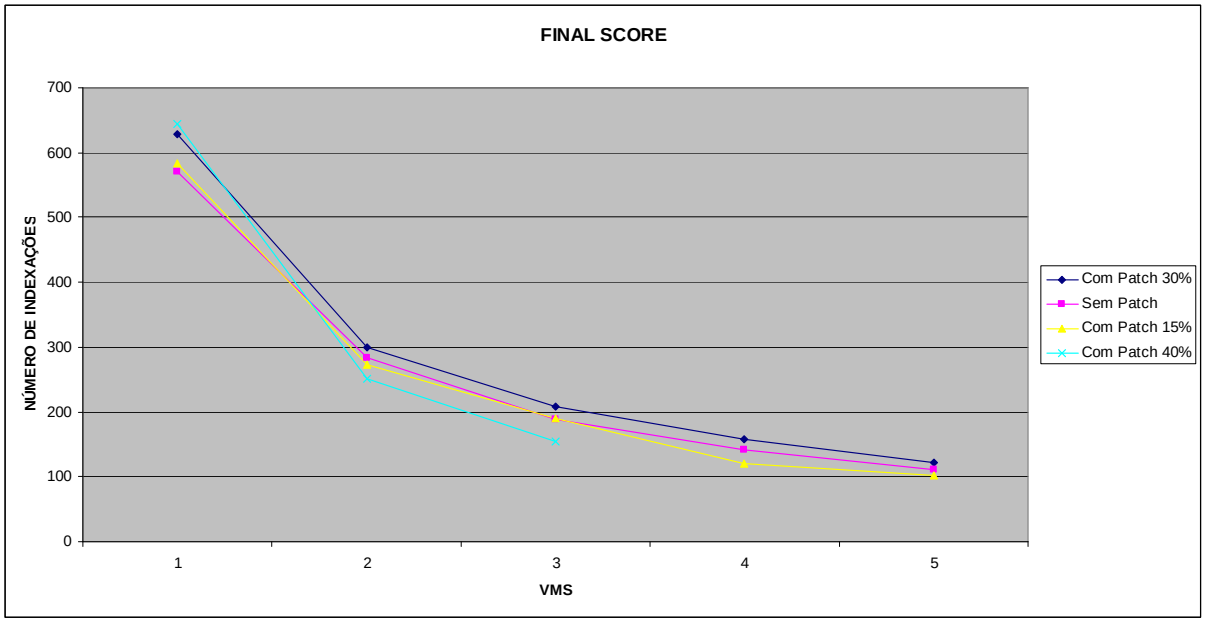

Ilustração 17: Final Score

O gráfico é composto de 4 testes, variando a porcentagem de tempo de escalonamento.. Observando o comportamento dos testes, Com *Patch* de 15% houve um comportamento praticamente igual ao sistema original (sem Patch), já o Com Patch 40% teve o melhor desempenho com uma VM entre os outros testes, mas com duas VMs o pior desempenho, mostrando que essa porcentagem foi

demasiada. Através desses testes que foi retirado a porcentagem que obteve maior rendimento para a implementação do sistema proposto, que é Com *Patch* de 30% ilustrando uma melhora constante no decorrer das máquinas virtuais.

6.4 Considerações Finais

O escalonador *Credit*, como padrão possui apenas uma definição de tempo (30 milisegundos) para cada domínio, tornando as máquinas virtuais, de uma forma geral, igual uma perante a outra em questão de acesso ao CPU. Conforme a Análise do escalonador observou-se isso, e então foi proposta uma solução. Essa solução trabalha a idéia de tornar as máquinas virtuais única para CPU.

Para tornar as máquinas virtuais diferentes em relação a tempo de acesso ao CPU, a implementação propõe novas prioridades ao sistema, adicionando definições máxima e mínima na contagem de créditos e de tempo.

As definições implementadas criam tempos dinâmicos de acesso ao CPU, pois sempre que alguma prioridade for alterada, ou haver necessidade de maior desempenho, seu tempo de acesso modificará.

Conforme a Análise dos resultados, utilizando o *benchmark* Unixbench, observa que houve uma melhora significativa de 8% a 12% a mais de desempenho com as modificações propostas, potencializando o escalonador *Credit* no seu escalonamento de processos, suas contagens de créditos, e suas prioridades.

7 CONCLUSÕES

A Virtualização é um tema de grande pesquisa nos dias de hoje, haja vista o suporte a novas tecnologias de processadores que visam suportar essa tecnologia. A virtualização tende a ajustar uma arquitetura específica as necessidades computacionais de sistemas de software. Na máquina virtual Xen, todo o gerenciamento do processador, memória e dispositivos é realizada pelo *hypervisor*, que oferece uma interface para cada máquina virtual, permitindo de forma individual uma execução independente. Esse gerenciamento é feito através de *system calls* realizadas pelas máquinas virtuais ao domínio principal, e o *hypervisor* vai controlar os acessos dessas chamadas ao *hardware*.

O Xen permite uma separação lógica entre o *hardware* e as máquinas virtuais ou sistemas operacionais convidados, permitindo uma maior flexibilidade, e aproveitando melhor o *hardware*, controlando ainda, o acesso seguro ao *hardware* pelas máquinas virtuais.

Vimos no gerenciamento do processador, como o Xen reconhece o tempo, para que possa manipular escalonamentos. Os três escalonadores suportados atualmente pelo Xen são os padrões, BVT que mostra escalonamento por fatias de tempo virtual, o SEDF que também usa *deadline*, porém complementa com a utilização de múltiplas filas de prioridades, e o SMP Credit que associa uma CPU física a uma CPU virtual, e utilizam prioridades como créditos para o escalonamento.

No capítulo sobre ferramenta de *benchmarking* pudemos notar quais as formas e técnicas utilizadas para observar a evolução do sistema, e qual delas seria mais apropriada a ser aplica no Xen. Portanto, como foi definida essa forma, utilizamos o *benchmark* Unixbench para testar o desempenho do escalonador *Credit* com a alteração proposta (*patch*).

No capítulo sobre análise do escalonador *Credit*, houve um detalhamento das principais funções, e suas respectivas estruturas, de como é feita associação de uma VCPU a um domínio, e este a uma CPU física, o processo de escalonamento de créditos para o tempo de processamento desses domínios, a contagem desses créditos, e como foi elaborado esse calculo.

Na Solução do Escalonador Credit, implementamos tempos dinâmicos de acesso ao CPU. Criando três constantes de tempo, associando as prioridades, definidas como normal, mínima e máxima. Porém, é importante ressaltar a

complexidade do Trabalho, ligando os tempos de acessos ao CPU, com a contagem de créditos de cada domínio, estando de acordo com a implementação de níveis de serviço (se utilizado).

A confirmação do Trabalho vem com a Análise dos Resultados, mostrando uma melhora significativa em seu desempenho, de 8% a 12%. Contudo, foi diagnosticado que quanto maior for o número de máquinas virtuais no ambiente, menor será seu desempenho

8 REFERÊNCIAS BIBLIOGRÁFICAS

[1] Xen Source Download (2007). http://www.cl.cam.ac.uk/research/srg/netos/xen/downloads.html

[2] Bartholomew, D. (2006). Qemu: a multihost, multitarget emulator. Linux Journal 2006, 3.

[3] Dike, J (2000). A user-mode port of the linux kernel. Proceeding of the Fourteen Annual Linux Showcase Conference, Atlanta.

[4] Vmware (2006). Vmware Products. http://www.vmware.com/products. Acessado em 15/08/2007.

[5] Clark, B. et al. Xen and the Art of Repaeated Research. Usenix Freenix Track. 2004.

[6] Creasy, R. J (1981). The origin of the VM/370 time-sharing System. IBM Journal of Research and Development, 25, 483-490.

[7] Salomon, F. A, & Tafuri, D. A (1982). Emulation – A useful tool in the development of computer systems. ANSS'82: Proceeding of the 15th Annual Symposium on the Simulation (pp. 55-71). Tampa, Florida, United States.

[8] Mergen, M. F., Uhlig, V., Krieger, O, & Xenidis, J. (2006). Virtualization for high - perfomance computing. SIGOPS operating system Review, 40, 8-11.

[9] Rosen, R. (2006). Virtualization in Xen 3.0. http://www.linuxjournal.com/article/8540. Acessado em 14/08/2007.

[10] Hoskin, M. E. (2006). User-mode Linux Journal, 2006, 2.

[11] Schroeder, M. D., & Saltzer, J. H. (1972). A Hardware Architecture for implementing protection rings. Communications of ACM, 15, 157-170.

[12] V. A. Cherkasova L., Gupta D. Comparison of the Three CPU Schedulers in Xen. Tecnical Report.

[13] XenEnterprise vs. ESX Benchmark Results: Performance Comparison of Commercial Hypervisors. 2007. Disponível em: http://blogs.xensource.com/simon/wpcontent/ uploads/2007/03/hypervisor_performance_comparison_1_0_3_no_esxdata. pdf, acesso em 11/08/2007.

[14] Barham, P. et al. Xen and the Art of Virtualization. 2003. Disponível em: http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf, acesso

em 15/08/2007.

[15] Xen Source (2007). http://wiki.xensource.com/xenwiki/Creditscheduler. Acesso em 26/08/2007.

[16] Xen User Manual (2007). http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/user/user.html. Acesso em 27/08/2007.

[17] OpenVZ Main Page. http://wiki.openvz.org/Main_Page. Acesso em 27/08/2007.

[18] Unixbench. http://www.nsa.gov/selinux/papers/freenix01/node15.html. Acesso em 18/03/2008.

[19] Lmbench. http://www.nsa.gov/selinux/papers/freenix01/node16.html. Acesso em 23/04/2008.