COPPE/UFRJ

CONTAGEM DE TEMPO EM SISTEMAS DE VIRTUALIZAÇÃO

Diego Leonel Cadette Dutra

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Luis de Amorim

Rio de Janeiro

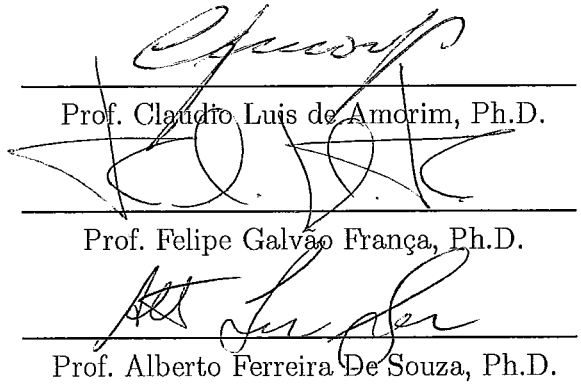
Março de 2009

CONTAGEM DE TEMPO EM SISTEMAS DE VIRTUALIZAÇÃO

Diego Leonel Cadette Dutra

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:



Prof. Claudio Luis de Amorim, Ph.D.

Prof. Felipe Galvão França, Ph.D.

Prof. Alberto Ferreira De Souza, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2009

Dutra, Diego Leonel Cadette

Contagem de Tempo em Sistemas de Virtualização/Diego Leonel Cadette Dutra. – Rio de Janeiro: UFRJ/COPPE, 2009.

XV, 87 p.: il.; 29,7cm.

Orientador: Claudio Luis de Amorim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2009.

Referências Bibliográficas: p. 74 – 79.

1. Técnicas de Virtualização. 2. Máquinas Virtuais. 3. Sistemas Operacionais. 4. *Cluster* de Computadores. 5. Many cores. I. Amorim, Claudio Luis de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha mãe pelo dom da vida,
pelo amparo ao longo desses
anos e paciência para escutar
minhas idéias, mesmo quando
não as entendia.

Agradecimentos

Agradeço ao Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo suporte financeiro.

Agradeço aos meus professores de graduação na UFF que tanto me ensinaram, principalmente os da área de arquitetura de computadores, sistemas operacionais e redes.

Agradeço a minha orientadora de graduação Professora Anna Dolejsis dos Santos pelo apoio incondicional a minha candidatura ao mestrado na COPPE.

Agradeço aos professores do Programa de Engenharia de Sistemas e Computação da COPPE pelas aulas e discussões tão interessantes e divertidas que tive nesses 2 anos.

Agradeço aos Técnicos-Administrativos do Programa de Engenharia de Sistemas, que sempre auxiliaram e estiveram de prontidão para responder quais quer duvidas, por mais bobas que fossem.

Agradeço aos colegas e colaboradores do Laboratório de Computação Paralela (LCP) que me ajudaram ao longo deste periodo de estudos e trabalho.

Um agradecimento especial ao Doutor Lauro Whatley, por tantos ensinamentos e discussões que em muito ajudaram a completar minha formação ao longo do mestrado. Muito obrigado por me deixar participar dos experimentos de sua tese, pois foi durante estes experimentos que tive a idéia para este trabalho.

Um agradecimento especial ao Professor Leonardo Pinho, pelos ensinamentos passados, estando sempre de prontidão a ajudar os outros, mesmo quando estava em vias de defender seu doutorado.

Um agradecimento especial para o Professor Claudio Luis de Amorim que tem me orientado desde meu primeiro dia na COPPE. Graças ao seu convite para integrar a equipe do LCP, pude melhor aproveitar os assuntos discutidos nas disciplinas

melhorando assim minha formação. Muito obrigado pelo suporte e ajuda que nunca faltaram nestes 2 anos e 3 meses.

Um agradecimento a todos meus amigos e as pessoas que de alguma forma acabaram por ficar em segundo plano durante o desenvolvimento deste trabalho.

Um agradecimento espcial a minha mãe que sempre cultivou em seus 2 filhos o prazer de estudar e aprender, mesmo que isso os tenha deixado com trauma de serem jogados pela janela se ficassem reprovados na escola.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

CONTAGEM DE TEMPO EM SISTEMAS DE VIRTUALIZAÇÃO

Diego Leonel Cadette Dutra

Março/2009

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Nos últimos anos, a pesquisa e desenvolvimento na área de sistemas operacionais e arquitetura de computadores tem apreciado um fantástico crescimento no uso de máquinas virtuais em aplicações WEB, computação em grades e escalonamento de recursos, principalmente em *Datacenters*. O estado da arte em máquinas virtuais permite que aplicações executando em domínios virtualizados sejam migradas de forma transparente entre os nós computacionais, mesmo durante a execução, resolvendo problemas como balanceamento de carga entre outros. Nos casos onde as aplicações que são migradas possuem dependências temporais, a migração somente é realizada com sucesso quando os nós de origem e destino encontram-se sincronizados, caso contrário, ao término da migração, a aplicação não irá funcionar corretamente.

Esta dissertação propõe, descreve e avalia uma solução para viabilizar a migração de aplicações entre domínios virtualizados quando as aplicações possuam dependências temporais. O **vTSC** é um mecanismo inteiramente em *software* que resolve o problema de dependência temporal das aplicações dentro de um domínio virtualizado. Em especial, o **vTSC** foi implementado nos sistemas OpenVZ (virtualização do tipo contêineres) e no Linux. Os experimentos realizados demonstram que o **vTSC** não sofre das limitações que afetam as outras soluções para este problema, oferecendo tanto uma medida de tempo mais precisa como um custo computacional inferior.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

TIME COUNTING IN VIRTUALIZATION SYSTEMS

Diego Leonel Cadette Dutra

March/2009

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

In recent years, computer architects and operating system designers have witnessed an astonishing growth in the use of virtual machines for WEB applications, grid computing, and resource scheduling, among other computer applications.

The state-of-art in Virtualization Systems allows for applications running in virtual domains to transparently migrate between computer nodes, without disrupting the services which the applications run. Unfortunately, application's migration can only be successful if no time dependencies are observed, otherwise the computer nodes are required to be synchronized in time with each other so that the application will not crash.

This dissertation introduces the Virtual Time Stamp Counter (**vTSC**), a new time counting solution to support migration of time-dependent applications on virtualized domains. The **vTSC** is a mechanism entirely developed in software that solves the time dependency problem for applications that run above virtualized domains. The **vTSC** has been carried out in OpenVZ as well as the Linux system.

The experimental results showed that **vTSC** not only eliminates the restrictions of current time-counting solutions but also offers better precision for time measurements and lower computational cost.

Sumário

Lista de Figuras xiii

Lista de Tabelas xv

1 Introdução 1

1.1 Contexto 1

1.2 Motivação 3

1.3 Objetivo 4

1.4 Contribuições 4

1.5 Organização da Disssertação 5

2 Conhecimentos Básicos 6

2.1 Aferindo o Tempo em Computadores Digitais 6

2.1.1 Relógio de Tempo-Real 7

2.1.2 Relógio de Intervalo Programável 7

2.1.3 Relógio Local do Processador 8

2.1.4 Temporizador da Interface Avançada de Configuração e Potência 8

2.1.5 Temporizador de Eventos de Alta Precisão 9

2.1.6 Contador de Ciclos do Processador 9

2.2 Processadores com Multiplos Núcleos 10

2.2.1 Processadores AMD 10

2.2.2 Processadores Intel 11

2.3 Máquinas Virtuais e Técnicas de Virtualização 12

2.3.1 Conceitos Gerais 13

2.3.2 Máquinas Virtuais para Linguagem de Alto Nível 13

2.3.3 Sistemas Operacionais Multiprogramáveis 14

2.3.4 Virtualização no Nível do Sistema Operacional 14
2.3.5 Paravirtualização 15
2.3.6 Virtualização Total 15
2.4 Aplicações de Tempo-Real 16
2.4.1 GloVE: Um sistema de Tempo-Real para Transmissão de Mídias Contínuas 17
2.5 Formalização do Problema 20
2.6 Escopo 21

3 Temporizando Aplicações em um *Cluster* de Máquinas Virtuais 22
3.1 Técnicas de Sincronização Global Absoluta 22
3.1.1 Network Time Protocol 23
3.1.2 Sincronização de Relógios Quase Par-a-Par 24
3.2 Técnicas de Sincronização Global Relativa 25
3.2.1 Sincronização de Alta-Precisão com *Time Stamps Counters* . 25
3.2.2 *Precision Time Protocol* 27
3.2.3 Relógio Global Distribuído 28
3.3 Considerações Finais 28

4 vTSC: *virtual Time Stamp Counter* 29
4.1 vTSC 29
4.1.1 Premissas 29
4.1.2 Solução 31
4.2 Implementando o vTSC no OpenVZ 36
4.2.1 Arquitetura Básica 36
4.2.2 Integração vTSC com o OpenVZ 37
4.3 Implementando o vTSC no Linux 41
4.3.1 Arquitetura Básica 41
4.3.2 Integração do vTSC com o Linux 42

5 Validação Experimental 47
5.1 Ambientes de Testes 47
5.1.1 Ambiente Experimental OpenVZ 47
5.1.2 Ambiente Experimental Linux 49

5.2 Avaliação da implementação do vTSC para OpenVZ 49
5.2.1 Avaliando as limitações do NTP no *cluster* local 49
5.2.2 Validação o funcionamento da Instrução RDTSC para os experimentos OpenVZ 51
5.2.3 Experimentos Quantitativos 51
5.3 Avaliação da implementação do vTSC para Linux 62
5.3.1 Validação o funcionamento da Instrução RDTSC para os experimentos Linux 63
5.3.2 Experimentos Quantitativos 63
5.4 Considerações Finais 65

6 Trabalhos Relacionados **67**
6.1 Máquinas Virtuais 67
6.2 Virtualização de Relógios em Sistemas Computacionais 68
6.2.1 Virtualizando a instrução RDTSC 68
6.2.2 Patentes 69

7 Conclusões e Trabalhos Futuros **71**
7.1 Resumo 71
7.2 Conclusões 72
7.3 Trabalhos Futuros 72

Referências Bibliográficas **74**

A Implementação do vTSC - Código Fonte **80**
A.1 OpenVZ 80
A.1.1 *Kernel* OpenVZ 80
A.1.2 vzctl 82
A.2 Linux 82
A.2.1 *Kernel* Linux 82

B Diário de uma Dissertação **84**
B.1 Início 84
B.2 Busca por um tema 85
B.3 Contagem do tempo e virtualização 85

C Memperf 87
C.1 Uso 87

Lista de Figuras

2.1 Arquitetura do processador AMD Athlon X2 11
2.2 Arquitetura do processador AMD Phenom X4 11
2.3 Arquitetura do processador Intel Core 2 12
2.4 Arquitetura do processador Intel Core i7 12
2.5 Arquitetura do sistema de vídeo sob demanda GloVE baseado na técnica CVC-C 18

3.1 Arquitetura do protocolo NTP 23
3.2 Topologia de uma rede de Classe B 25
3.3 Arquitetura Física do Protocolo de Sincronização de Alta-Precisão Utilizando *Time Stamps Counters* 26

4.1 Estrutura de dados utilizada na leitura do registrador TSC 31
4.2 Descrição da estrutura de dados que suporta o vTSC 32
4.3 Inicialização da instância do vTSC vinculada a um processo que esta sendo criado dentro do sistema operacional 32
4.4 Chamada de sistema utilizada pela aplicação para consultar vTSC . . 33
4.5 Algoritmo de Migração do vTSC 33
4.6 Driver para mudança de freqüência do processador utilizando vTSC . 34
4.7 Driver otimizado para mudança de freqüência do processador utilizando vTSC 35
4.8 Tipo de Dado Abstrato T_vTSC 37
4.9 Chamada de sistema para recuperar o tempo em execução 38
4.10 Chamada de sistema para recuperar o tempo em execução + suspenso 38
4.11 Migração do OpenVZ com as modificações feitas para o suporte ao vTSC 39

4.12 Trecho de código executado ao final do processo de *Suspend* do OpenVZ para computar os dados necessários para vTSC 40
4.13 Trecho final do código executado pelo cliente vTSC no Destino 41
4.14 Tipo de Dado Abstrato do vTSC para CPU implementado no Linux *T_vTSC_Base_Per_CPU* 43
4.15 Tipo de Dado Abstrato do vTSC para o processo implementado no Linux *T_vTSC* 43
4.16 Chamada de sistema para recuperar o tempo de execução no Linux . 44
4.17 Migração de processos no Linux com as modificações feitas para o suporte ao vTSC 45

5.1 Ambiente Experimental OpenVZ para avaliar o vTSC 48
5.2 Testa para verificar uso da instrução RDTSC para seu uso na implementação OpenVZ do vTSC 52
5.3 Avaliação da Propriedade de crescimento Monotônico do vTSC 54
5.4 Descrição do *microbenchmark* utilizado nos experimentos 57
5.5 Execução do *microbenchmark* no ambiente experimental Linux 63

Lista de Tabelas

5.1 Nó computacional IBM xSeries 206m 48

5.2 *Notebook* HP Pavilion dv2807nr 49

5.3 Estatísticas dos Δs aplicados pelo protocolo NTP sobre o relógio local 51

5.4 Cenários utilizados na avaliação da chamada de sistema *ve_get_runtime*() 53

5.5 Resultados do tempo médio e desvio padrão da execução do 3º cenário, com intervalo igual a 10 ms 55

5.6 Medidas de disperção para os cinco cenários utilizados na avaliação da chamada de sistema *ve_get_runtime*() 55

5.7 Custo médio da temporização para o *microbenchmark* nos quatro cenários estudados 57

5.8 Interavalo de Confiança para *microbenchmark* nos quatro cenários estudados 59

5.9 Experimento memperf 62

5.10 Experimento FileBench para o Linux 64

5.11 Intervalos de Confiança para o experimento FileBench no Linux . . . 65

Capítulo 1

Introdução

Este capítulo apresenta o assunto desenvolvido nesta dissertação, destacando o problema de interesse que motivou o desenvolvimento de pesquisa sobre o tema e as contribuições resultantes, e delineia a organização dos demais capítulos.

1.1 Contexto

Nos sistemas digitais, a contagem do tempo é importante tanto para o controle de acesso aos barramentos compartilhados em *hardware*, como para os diversos níveis de abstração (*software* e *hardware*) que formam um sistema computacional. Por exemplo, no nível do sistema operacional tanto para o gerenciamento de tarefas como para aplicações de tempo-real. Além disso, dependendo da utilização de um contador de tempo, existem diferentes requisitos em relação à precisão e ao custo da medida de tempo. Neste sentido, há diversos circuitos contadores no computador, os quais diferem geralmente em relação à precisão do contador e ao custo de tempo para acessá-lo.

Entre esses circuitos, o que oferece a maior precisão é o ***Time Stamp Counter*** [1, 2, 3, 4] (TSC), o TSC é um registrador dentro da unidade de processamento que é responsável por contar quantos ciclos já se passaram desde que o processador foi ligado. Dessa forma, o TSC é incrementado na mesma freqüência de trabalho do processador, que se for, por exemplo, de 1 GHz significa que o contador será incrementado a cada 1 ns (nanosegundo).

Entretanto, nos processadores modernos onde, a freqüência de trabalho pode

ser modificada em tempo de execução (escalonamento da freqüência) para reduzir o consumo de energia, por exemplo de 1 GHz para 500 MHz, o contador TSC passará a fornecer uma informação de tempo sob uma nova base quando o sistema fizer a mudança da freqüência. Quando ocorre mudança na freqüência de trabalho, a taxa de atualização do TSC é modificada, fazendo com que o sistema computacional acredite que se passou um intervalo de tempo maior ou menor do que o real, onde a percepção do intervalo depende apenas se a nova freqüência é maior ou menor que a anteriormente utilizada (ex., um processador da AMD que se encontra trabalhando a uma freqüência de 1 GHz e tem sua freqüência de trabalho modificada para 2 GHz). Um problema semelhante ocorre em máquinas com múltiplos processadores de diferentes freqüências de trabalho onde a execução da aplicação não encontra-se presa a um único processador, e portanto ela ficará sujeita as varias freqüências de trabalho. Nesta situação, outro circuito independente deverá ser utilizado para aferir a passagem do tempo de modo a permitir que a aplicação utilize uma única referência ou base de tempo.

Dificuldades em aferir o tempo também ocorrem nos processadores com múltiplos núcleos (*multi-core*), utilizados freqüentemente pelos *datacenters*. Nos *datacenters*, os nós de processamento são formados por um ou mais processadores que são compartilhados pelos aplicativos utilizando técnicas de virtualização. A virtualização garante o isolamento necessário entre as aplicações enquanto permite o compartilhamento do *hardware*, resultando em significativa economia de recursos. Nesse caso, as novas arquiteturas baseadas em *clusters* de processadores com múltiplos núcleos agravaram o problema acima, uma vez que o escalonamento da freqüências pode ser feito no nível do núcleo e não mais do processador, influenciado pela necessidade de se otimizar o consumo de energia elétrica.

Virtualizar consiste em criar abstrações em software, denominadas máquinas virtuais (MMV), responsáveis por multiplexar ou esconder as características do hardware físico onde os aplicativos executam. Desta forma, as técnicas de virtualização abrangem desde o mecanismo de memória virtual nos sistemas operacionais multi-tarefas [5], passando pelos contêineres [6] que criam abstrações de um sistema operacional como o Linux, até a de computadores inteiros como o IBM VM/370 [7] e o Xen [8, 9], que utilizam um Monitor de Máquinas Virtuais (MMV) para controlar

as diversas instâncias virtuais que podem estar executando concorrentemente.

Tanto máquinas virtuais que seguem o modelo de MMV como as que programam contêineres permitem que um domínio virtualizado possa ser migrado entre nós físicos, oferecendo assim suporte para balanceamento de carga, crescimento na demanda por poder computacional e disponibilidade do serviço. Por outro lado, a migração de domínios virtuais introduz um novo problema que é usar os circuitos do hardware para se medir a passagem do tempo entre domínios. Uma solução usual é fazer com que os domínios físicos mantenham alguma forma de sincronização entre os nós de origem e destino. No caso de um *cluster* de computadores, para todo par origem/destino em que ocorreu alguma migração é necessário sincronizar os relógios entre os nós para que uma medida feita na origem seja consistente com uma outra feita no nó destino. Nos casos onde a consistência não possa ser realizada é praticamente inviável realizar uma migração bem sucedida de aplicações que tenham dependências temporais, ou seja possuem em seu processamento limites com relação a duração das tarefas a que são executadas (tempo-real). Observe que em sistemas distribuídos com muitos nós não é possível ter relógios perfeitamente sincronizados [10].

1.2 Motivação

Um caso concreto ilustrará bem o problema de migração de uma aplicação distribuída com dependência temporal onde a inconsistência de contadores de tempo levou à falha de execução da aplicação. A aplicação de tempo-real em questão é um sistema escalável de transmissão de vídeo sob demanda - GloVE [11, 12]. A aplicação GloVE foi executada no OpenVZ [13] que implementa no núcleo do Linux o modelo de contêineres. Uma característica do GloVE é dividir o tempo do vídeo em *slots* de tamanho fixo que são alocados aos clientes do vídeo. A manutenção dos *slots* é crítica para garantir que cada cliente receba um bloco do vídeo num determinado intervalo de tempo de modo que a exibição do vídeo não sofra o efeito de congelamento na tela do cliente. Além disso, o balanceador de carga do sistema migra a aplicação GloVE para outro nó do *cluster* sempre que detectar sobrecarga no nó em que ela estiver executando. Esse balanceamento é realizado para prevenir

que a temporização dos *slots* seja comprometida quando o nó encontra-se sobrecarregado, o que pode causar desde uma perda de qualidade no vídeo assistido até o congelamento da exibição do mesmo.

Ocorreu que a aplicação sempre parava de funcionar após a migração. A causa da falha era a técnica de temporização que utilizava a instrução RDTSC [1, 2, 3, 4] (***ReaD Time Stamp Counter***) da arquitetura x86 [14]. A falha ocorria devido ao programa utilizar os valores retornados pela instrução RDTSC para temporizar o envio dos *slots* de vídeo para os clientes. Especificamente, quando o GloVE passava a ser executado no nó destino os valores retornados pela execução de RDTSC representavam uma temporização inconsistente para a aplicação, que interrompia desta forma sua execução.

A correção adotada foi substituir a RDTSC pela combinação do *Network Time Protocol* [15] (NTP) e a chamada de sistemas gettimeofday [16]. Desta forma, a temporização da aplicação global dentro do *cluster* permitiu que a medida feita no nó de origem mantivesse seu significado quando a aplicação era migrada para outro nó dentro do *cluster*. Apesar de resolver a falha, essa solução apresenta limitações com relação a precisão da medida temporal obtida que é da ordem de dezenas de **milisegundos** em um nó com pouca carga e elevado custo de manutenção da sincronização entre os nós. Partindo deste caso, esta dissertação apresenta resultados de um trabalho de pesquisa buscando melhores formas ou forma para se gerar a medida temporal correta nessas novas arquiteturas de processadores.

1.3 Objetivo

Propor e avaliar um mecanismo inteiramente em software que solucione o problema de inconsistência de temporização local que ocorre quando domínios virtuais (processos, contêineres ou máquinas virtuais) são migrados entre diferentes núcleos ou nós computacionais, buscando minimizar os custos e maximizar a resolução da medida temporal.

1.4 Contribuições

Em resumo, as principais contribuições presentes nesta dissertação são:

1. Redução do problema de temporização em um *cluster* de máquinas virtuais ao problema de temporização em arquiteturas com múltiplos processadores ou processadores com múltiplos núcleos;

2. Discussão das soluções de sincronização existentes e suas limitações, mostrando como estas podem ser utilizadas para resolver o problema de temporização nos domínios virtuais na presença de migração entre nós dentro de um *cluster*;

3. Proposta e descrição de uma solução para resolver o problema de temporização mesmo quando na presença de migrações, denominada ***virtual Time Stamp Counter*** (vTSC);

4. Desenvolvimento e avaliação de uma implementação do vTSC no OpenVZ;

5. Desenvolvimento e avaliação de uma implementação do vTSC no Linux.

1.5 Organização da Disssertação

Esta dissertação apresenta no Capítulo 2 os conceitos básicos necessários para o entendimento deste trabalho, entre eles os circuitos para medição do tempo nos computadores modernos, máquinas virtuais, o sistema de vídeo sob demanda GloVE, o experimento de migração com o GloVE e uma descrição formal do problema. No Capítulo 3 são apresentadas algumas soluções que resolvem, dentro de determinadas condições, o problema descrito no final do Capítulo 2 para um ambiente de *cluster* de computadores. O Capítulo 4 descreve o modelo da solução proposta nesta dissertação, o virtual Time Stamp Counter (vTSC) e as implementações no OpenVZ e Linux da solução proposta neste trabalho. Os experimentos empíricos utilizados para validar a solução implementada encontram-se no Capítulo 5. Os trabalhos relacionados com o conteúdo desta dissertação encontram-se no Capítulo 6. O Capítulo 7 resume os principais resultados obtidos, apresenta as conclusões da dissertação e propõe alguns trabalhos futuros.

Capítulo 2

Conhecimentos Básicos

Este capítulo tem como objetivo explicar os conceitos, definições e questões envolvidas neste trabalho. O sistema de vídeo sob demanda GloVE é apresentado e o experimento que motivou nosso trabalho é melhor detalhado. Após, são feitas algumas considerações finais, encadeando os conteúdos abordados de forma a delimitar o escopo da dissertação.

2.1 Aferindo o Tempo em Computadores Digitais

Nos computadores digitais modernos existem atividades que necessitam ser sincronizadas para seu correto funcionamento, sendo que diferentes circuitos digitais podem ser utilizados para este fim, os relógios do *hardware*. Além de sincronização, esses circuitos também são utilizados para medir a passagem do tempo. Com isso os diversos relógios diferem em como a informação é acessada, a precisão do valor lido e o custo de acesso [17].

É importante ter em mente a diferença entre a precisão da informação que pode ser obtida e o custo para obtê-la. A precisão leva em consideração apenas o número de casas decimais que o circuito do relógio oferece, ou seja, se o relógio envia um sinal a uma freqüência de 100 MHz, a maior precisão que pode ser obtida por este relógio é 100 ns. O custo, é o tempo passado desde que a informação foi requerida até esta encontrar-se disponível para a aplicação.

A arquitetura x86 possui quatro relógios, são eles:

- Relógio de Tempo-Real (*Real-Time Clock*)
- Relógio de Intervalo Programável (*Programmable Interval Timer*)
- Relógio Local do Processador (*Local Advanced Programmable Interrupt Controller*)
- Temporizador da Interface Avançada de Configuração e Potência (*Advanced Configuration and Power Interface Timer*)
- Temporizador de Eventos de Alta Precisão (*High Precision Event Timer*)
- Contador de Ciclos do Processador (*Time Stamp Counter*)

2.1.1 Relógio de Tempo-Real

O Relógio de Tempo-Real (*Real-Time Clock* ou RTC) é o hardware para controle do tempo mais comum nos computadores digitais, todos os computadores que seguem a arquitetura x86 devem possuir este relógio. O circuito que implementa o RTC não possui dependências com o processador ou outros circuitos de controle na placa-mãe, sendo integrado junto com o circuito CMOS RAM (BIOS) e alimentado por uma bateria.

O RTC funciona emitindo periodicamente interrupções de hardware na IRQ8. Sendo que essas interrupções podem ser enviadas com taxas que variam entre 2 Hz a 8.192 Hz (8 KHz), sendo assim, a precisão do intervalo de tempo que pode ser medida utilizando este circuito é de aproximadamente 122 μs [18].

2.1.2 Relógio de Intervalo Programável

Os computadores que seguem a arquitetura x86 contém também o circuito digital do Relógio de Intervalo Programável (*Programmable Interval Timer* ou PIT). Ele é implementado por um chip NMOS Intel 8253 (ou CMOS no caso do Intel 8254) [19].

O 8253 possui três relógios todos implementados com um contador de 16 *bits*. O primeiro é temporizador utilizado pelo sistema operacional (o Linux, por exemplo, configura para emitir interrupções a 1000 Hz na arquitetura x86) [18]), o segundo para controle de atualização das memórias RAM e o terceiro para a saída de áudio padrão (PC *Speaker*).

A freqüência de trabalho máxima (taxa da geração de interrupções) é dada pelo oscilador interno do circuito, que é de 1.193.182 Hz para o 8253 e a freqüência minima de trabalho é de 18.2 Hz [20]. No 8254 a freqüência de trabalho máxima é 10 MHz [21].

2.1.3 Relógio Local do Processador

Presente nos processadores recentes da família x86, o Relógio Local do Processador (*Local Advanced Programmable Interrupt Controller* - APIC) [14, 22], é um circuito de funcionamento similar ao PIT, porém as interrupções geradas são vistas apenas pelo processador que as gerou.

Os contadores deste relógio são registradores de 32 *bits* contra os 16 *bits* do PIT o que lhe permite gerar interrupções com baixíssima freqüência, isso é feito setando-se um valor inicial nos contadores, assim a interrupção somente é emitida para o processador quando o contador chegar a zero. A freqüência máxima de atualização do contador é dada pelo barramento (ex.,100 MHz) e freqüências menores podem ser obtidas esperando-se até 2^n ciclos do barramento (n pode variar de 0 a 7) [18] para emitir uma interrupção.

2.1.4 Temporizador da Interface Avançada de Configuração e Potência

Este relógio é encontrado em placas-mães que suportam o mecanismo de controle de potência (*Advanced Configuration and Power Interface* - ACPI), sendo conhecido como ACPI PMT (*ACPI Power Management Timer*).

Ele é composto por um contador de 32 *bits* que é incrementado a uma freqüência fixa de 3,579545 MHz [23]. A grande vantagem deste relógio é sua independência com relação aos controles de potência implementados nos computadores atuais, o que o torna atraente nas situações onde o processador pode modificar sua freqüência de trabalho [18].

2.1.5 Temporizador de Eventos de Alta Precisão

O Temporizador de Eventos de Alta Precisão [24] (*High Precision Event Timers* - HPET), é um novo relógio desenvolvido pela Intel em parceria com a Microsoft. Como este circuito possui diversos relógios independentes o sistema operacional pode vincular um relógio específico para cada aplicação em nível de usuário.

Seus requisitos de desenvolvimento foram especificados para prover uma granularidade de milissegundos nas medidas e precisão de nanosegundos. O controle do HPET é exclusividade do sistema operacional sendo este o responsável por salvar e restaurar as informações necessárias para o seu correto funcionamento quando o computador entra em um estado de baixo consumo.

O HPET funciona utilizando registradores mapeados na memória pela BIOS, sendo que este mapeamento é estabelecido durante a inicialização do computador sendo posteriormente informado ao sistema operacional [18].

2.1.6 Contador de Ciclos do Processador

O Contador de Ciclos do Processador em inglês, *Time Stamp Counter* (TSC), é um registrador de 64 *bits* na arquitetura x86 que armazena o número de ciclos desde a inicialização do sistema. A lógica de atualização do relógio incrementa em uma unidade o valor do registrador a cada ciclo da CPU, desta forma a freqüência de atualização do contador é dada diretamente pela freqüência de trabalho da CPU.

Devida a sua alta taxa de atualização e largura no número de bits, dentre os circuitos que são encontrados nas arquiteturas x86, o TSC é o que oferece a maior precisão. O TSC ao contrário dos outros relógios da arquitetura x86 não gera interrupções que possam ser utilizadas para indicar a passagem do tempo, esta somente pode ser aferida executando a instrução RDTSC [1, 2, 3, 4], oferecendo custo de acesso inferior aos outros circuitos.

Apesar das vantagens do TSC para temporização das aplicações, como sua freqüência de atualização depedende da freqüência de trabalho atual do processador ele, ele fornece valores inconsistentes quando se esta utilizando o escalonamento de freqüência.

2.2 Processadores com Multiplos Núcleos

A crescente demanda por mais poder computacional, a Lei de Moore [25], os limites para retirada de calor dos processadores, a crescente diferença entre os tempos de acesso a dados em memória com relação a freqüência do processador e incapacidade dos arquitetos em extrairem mais paralelismo a nível de instrução, levou as empresas de microprocessadores a investir maciçamente nos processadores com múltiplos núcleos (*multi-core*), desta forma diversos programas podem estar executando paralelamente em um único computador. Além disso, existe sempre a possibilidade de uma aplicação ser migrada entre núcleos por um procedimento de balanceamento dinâmico de carga.

Os processadores com múltiplos núcleos também implementam controles para escalonamento da freqüência como forma de economizar a energia gasta pelo computador em momentos de baixa demanda, o que torna problemático nestes processadores o uso do TSC como relógio. A forma como estes mecanismos funcionam depende da arquitetura onde se encontram implementados. Por exemplo, pode-se obrigar que o controle de freqüência sempre ocorra no nível do processador ou seja todos os núcleos devem estar executando na mesma freqüência, sendo assim a forma como o mecanismo é implementado impacta diretamente na quantidade de energia que poderá ser poupada.

2.2.1 Processadores AMD

A primeira geração de processadores com múltiplos núcleos da AMD apresentou uma organização interna onde não existia memória compartilhada entre os núcleos e o controlador de memória era interno ao processador; o exemplo desta arquitetura é o processador Athlon X2(Figura 2.1). Uma consequência desta organização é que o acesso a dados armazenados na cache de outro núcleo somente pode ser realizado através da cooperação entre os núcleos. As freqüências dos núcleos do processador Athlon X2 podem ser modificadas individualmente.

Os processadores Phenom X4 (Figura 2.2), por sua vez incorporaram uma memória compartilhada entre os núcleos, desta forma, os processadores passaram a ter uma Cache nível três compartilhada, sendo que, como no caso do Athlon X2, as

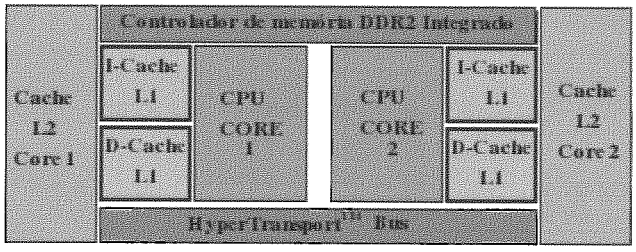


Figura 2.1: Arquitetura do processador AMD Athlon X2

freqüências dos núcleos podem ser ajustadas isoladamente.

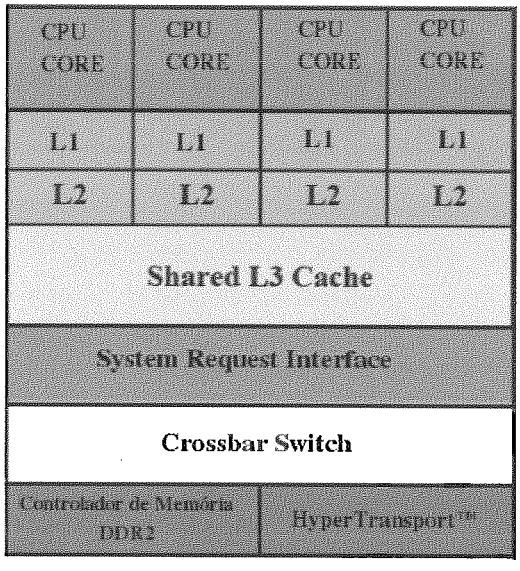


Figura 2.2: Arquitetura do processador AMD Phenom X4

A arquitetura da AMD permite que os registradores TSC em cada núcleo sejam incrementados com taxas diferentes, sendo essas taxas iguais a freqüência do núcleo. Como os núcleos não são obrigados na trabalhar a mesma freqüência isso pode inviabilizar o uso do TSC como contador de tempo em aplicações de tempo-real.

2.2.2 Processadores Intel

A Intel por sua vez, em seus primeiros processadores com múltiplos núcleos construiu uma organização onde dois núcleos possuem memória cache de segundo nível compartilhado e a arquitetura com quatro núcleos incorporava uma memória cache de terceiro nível para interligar esses processadores. Nesta organização, o controle de freqüência foi implementado de forma global no processador, ou seja, todos os

núcleos devem estar trabalhando na mesma freqüência sendo que nas **arquiteturas Core e Core2(Figura 2.3) o TSC é incrementado a uma taxa fixa.**

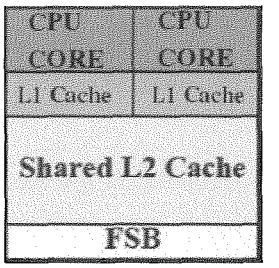


Figura 2.3: Arquitetura do processador Intel Core 2

A nova arquitetura proposta pela Intel o Core i7 (Figura 2.4), por sua vez, passou a ser semelhante à arquitetura da AMD Phenom X4, onde os núcleos possuem memórias cache de primeiro e segundo níveis privadas e uma cache nível três compartilhada, além de passar a integrar a controladora de memória. Outra mudança na nova arquitetura é que os núcleos podem trabalhar em freqüências diferentes.

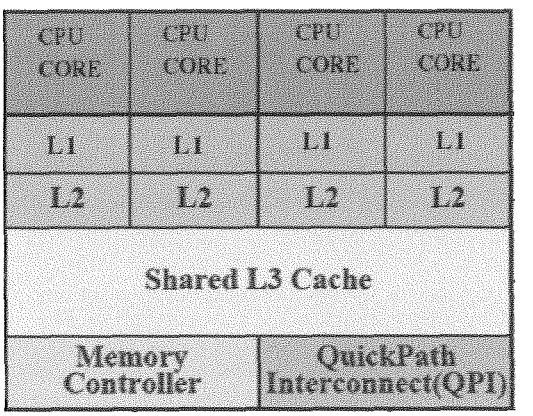


Figura 2.4: Arquitetura do processador Intel Core i7

2.3 Máquinas Virtuais e Técnicas de Virtualização

As tecnologias de virtualização encontram-se difundidas em diversos níveis dos sistemas digitais. Nos últimos anos, o uso de máquinas virtuais ganhou grande impulso

devido à disponibilidade cada vez maior de poder computacional com os processadores com múltiplos núcleos e com o suporte, nestes novos processadores, para tecnologias de virtualização como as arquiteturas *Intel Virtualization Technology* (Intel VT) [26, 27] e *AMD Virtualization* (Pacifica).

Dentro do escopo desta dissertação, máquinas virtuais e técnicas de virtualização aparecem tanto como motivação do problema como solução para ele. Assim sendo, é importante entender esses mecanismos, como eles podem ser úteis na construção de sistemas computacionais e como afetam as aplicações de tempo-real, em particular o sistema de vídeo sob demanda GloVE.

2.3.1 Conceitos Gerais

Um dos conceitos básicos em computação é o uso de abstrações para esconder do desenvolvedor as complexidades do hardware. O uso de múltiplos níveis de abstrações pretende criar, a cada nível, máquinas virtuais que estendem as funcionalidades disponíveis no nível abaixo, construindo desta forma uma hierarquia de abstrações [17].

Além das funcionalidades estendidas, uma máquina virtual deve criar isomorfismos entre as funcionalidades que ela oferece ao nível acima e o que pode ser executado na máquina virtual do nível abaixo. Dentro do escopo desta dissetação, virtualização é definida como a construção de um mapeamento entre o dispositivo virtual (funcionalidade) e o real [28].

2.3.2 Máquinas Virtuais para Linguagem de Alto Nível

Atualmente, Java é o exemplo de virtualização mais conhecido do modelo de Máquinas Virtuais para Linguagem de Alto Nível (*High-Level Language Virtual Machines*)[29]. Essa classe de máquinas virtuais é utilizada para se obter independência de plataforma, assim um código binário (*bytecode*) pode ser executado sem modificações em diversas arquiteturas.

As Máquinas Virtuais desta classe suportam uma abstração que representa tanto o conjunto de instruções como o sistema operacional, encapsuladas essas funcionalidades como um processo de nível usuário.

2.3.3 Sistemas Operacionais Multiprogramáveis

A organização em camadas dos sistemas computacionais levaram a construção de mecanismos de gerenciamento e controle, encontrados nos Sistemas Operacionais (SO). A abstração que o SO oferece às aplicações permite que estas não tenham que lidar com as complexidades das camadas inferiores e fornece mecanismos pelos quais uma aplicação tem um *hardware* virtualmente dedicado [5]. Desta forma, o conjunto de chamadas do sistema Operacional e as instruções não privilegiadas formam um computador virtual [30] que, por meio das técnicas de multiprogramação [31], possibilita que os aplicativos executem como se a máquina física fosse dedicada e não um *hardware* compartilhado [29].

Mémoria Virtual

A técnica de Memória Virtual é um dos mecanismos que dá suporte à multiprogramação nos sistemas operacionais modernos. Ela permite que diversos programas possam utilizar a memória principal do computador de forma concorrente sem que ocorra interferência entre eles. Com a utilização desta técnica, os endereços conhecidos pelo programa não são mais são endereços físicos válidos e sim um deslocamento sobre uma base a ser definida durante a execução.

2.3.4 Virtualização no Nível do Sistema Operacional

A Virtualização no Nível do Sistema Operacional constrói uma nova abstração dentro dos Sistemas Operacionais tradicionais, os contêineres de recursos (resource containers) ou simplesmente contêineres [6, 32]. Um contêiner isola dentro do núcleo do SO todas as estruturas de controle referentes às aplicações que ele encapsula, viabilizando desta forma a utilização de políticas diferentes para o gerenciamento de cada conjunto de aplicações. Uma das vantagens desta classe de virtualização é o baixo custo imposto por suas implementações, sendo esse o motivo de sua escolha para ambientes compartilhados como o PlanetLab [33], onde os nós encotram-se sempre sobre utilizados.

OpenVZ

O OpenVZ [13] é um dos diversos trabalhos [6, 32] na literatura que implementam essa classe de virtualização. Um diferencial com relação a outros trabalhos é que no OpenVZ apenas o núcleo do Linux é compartilhado entre os diversos contêineres. Assim, diversos Sistemas Operacionais (aplicativos de sistemas e bibliotecas) podem executar sobre o mesmo núcleo Linux compartilhado de forma transparente.

2.3.5 Paravirtualização

As máquinas virtuais desta classe executam sobre um Monitor de Máquina Virtual (VMM) que é responsável por gerenciar os acessos feitos pelas máquinas virtuais. O VMM permite a construção de máquinas virtuais que são semelhantes à máquina física que ele virtualiza. Como nesta classe de Máquinas Virtuais o conjunto de instruções original não se encontra todo virtualizado, é necessário que os aplicativos que executem alguma das instruções que não são suportadas substituam essas instruções por chamadas do VMM [29].

Xen

O Xen [8] é um monitor de máquina virtual de código-livre que implementa o modelo de paravirtualização, desta forma o núcleo do sistema operacional deve ser modificado de forma a utilizar as chamadas que substituem os recursos que não são virtualizados. A implementação do Xen consegue fornecer um mecanismo de virtualização de baixo custo computacional para aplicações que sejam dependentes apenas de CPU.

2.3.6 Virtualização Total

A classe de Máquinas Virtuais que implementam a Virtualização Total oferece para as aplicações que executam sobre essa máquina virtual um hardware idêntico ao que é fornecido pela máquina física. Como no modelo Paravirtualizado, existe uma camada de software que é responsável por gerenciar os recursos físicos e exportar para as camadas acima (SO e aplicações) uma interface igual a do hardware [7, 29]. Este modelo de virtualização pode ser implementado com auxilio da arquitetura do

processador ou via tradução binária [29].

VirtualBox

O VirtualBox é um monitor de máquinas virtuais da classe de Virtualização Total que implementa o mecanismo de tradução binária. Quando uma aplicação precisa executar uma instrução privilegiada que não foi virtualizada ele substitui em tempo de execução esta instrução por um conjunto de instruções que realizam a mesma função e que respeitam as regras criadas pelos mecanismos de virtualização.

Xen

Com a chegada no mercado de processadores da arquitetura x86 que possuem suporte em hardware a virtualização [26, 27], as novas versões do Xen [9] passaram a também suportar Virtualização Total. O uso das extensões de virtualização permite que uma instrução que viola as regras impostas pelos mecanismos de virtualização sejam executadas, sendo que a execução destas instruções causa uma interrupção que deve ser tratada pelo VMM.

2.4 Aplicações de Tempo-Real

As aplicações de tempo-real são aquelas onde as tarefas executadas no sistema devem ocorrer dentro de um intervalo de tempo pré-definido. Essas aplicações são classificadas quanto ao rigor das restrições temporais, por exemplo, se uma aplicação consegue continuar executando corretamente mesmo que as restrições temporais sejam esporadicamente desobedecidas é chamada de tempo-real não-críticos. As aplicações que não conseguem executar corretamente após acontecer alguma desobediência com relação às restrições temporais são conhecidas como tempo-real crítico; sistemas de controle militares é um bom exemplo desta categoria de aplicações [34].

A necessidade de controlar o tempo dentro destas aplicações torna-se um problema nas novas arquiteturas de processadores, devido a granularidade do intervalo que pode ser utilizado. Infelizmente, o maior intervalo possível é dado pelo problema atacado e não pelo hardware do computador, por exemplo, se for necessário que a granularidade do intervalo seja da ordem de dezenas de nanosegundos, apenas

o TSC poderá ser utilizado.

2.4.1 GloVE: Um sistema de Tempo-Real para Transmissão de Mídias Contínuas

O sistema de vídeo sob demanda GloVE (*Global Video Environment*) [12, 11] é uma aplicação de tempo-real não-critíco (*soft real-time*), onde, se o intervalo não for cumprido, o usuário sofre apenas uma degradação na qualidade do serviço contratado. O sistema implementa a técnica de Cache de Vídeo Cooperativa Colapsada (CVC-C) [35].

Nesta técnica, os vídeos são divididos em blocos (*slots*) de tamanho fixo para serem enviados para os clientes. Sendo assim, a temporização dos *slots* de vídeo é uma tarefa crítica para o que os clientes consigam receber os blocos no momento correto. Como a taxa de transmissão real não é constante, devido às variações dos retardos de rede (*jitter*), cada cliente possui um *buffer* de vídeo de 10 s, que é totalmente prenchido antes do início da exibição. Desta forma um bloco de vídeo que não se encontra no *buffer* tem até 10 s para chegar ou cliente sofrerá uma degradação na qualidade do serviço.

O GloVE é composto por três programas que executam de forma distribuída em computadores interconectados como pode ser observado na Figura 2.5. As aplicações são:

- Servidor de Vídeo;
- Proxy CVC-C;
- Clientes.

O servidor de vídeo é um nó computacional dedicado responsável por armazenar todos os vídeos que o sistema GloVE provê aos clientes. O servidor de vídeo é uma aplicação cliente/servidor que prevê blocos de vídeos para os proxies CVC-C.

Os proxies CVC-C ou distribuidores são o coração do sistema de distribuição de vídeos, são eles que requisitam blocos de vídeo ao servidor de vídeo e armazenam estes blocos em uma cache local enviando posteriormente os blocos aos clientes a uma taxa constante (*Constant Bit Rate* - CBR). São os distribuidores que implementam

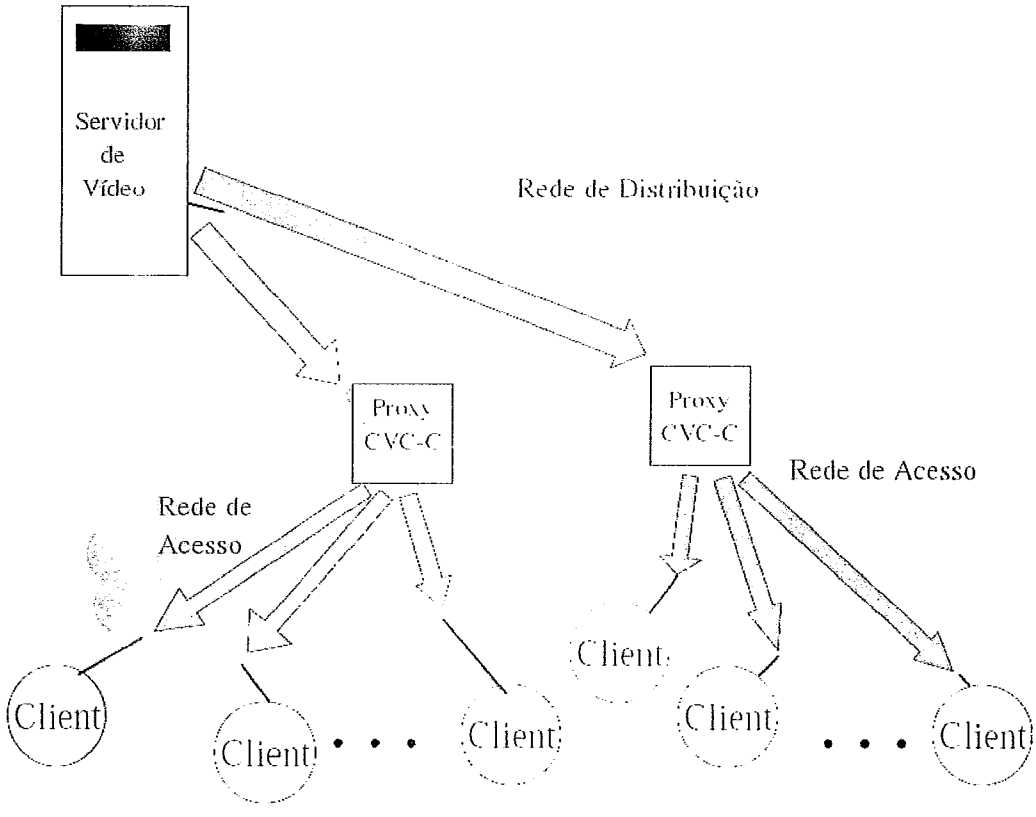


Figura 2.5: Arquitetura do sistema de vídeo sob demanda GloVE baseado na técnica CVC-C

o algoritmo de distribuição de vídeo [35]. O distribuidor é uma aplicação de tempo-real não-crítica composta por quatro linhas de execução (*Threads*):

1. *Thread* de Controle: Responsável por manter a cache de vídeo e requisitar blocos ao servidor de vídeo,
2. *Thread* de Recebimento: Recebe os blocos de vídeo oriundos do servidor,
3. *Thread* de Envio: Fornece os blocos de vídeos aos clientes a uma taxa constante (CBR),
4. *Thread* de Temporização: Provê a temporização necessária no sistema de forma a garantir o envio de blocos a uma taxa constante para aos clientes.

O cliente é uma aplicação dentro do sistema GloVE que executa localmente no computador de um usuário do sistema que deseja assistir um vídeo. O usuário requisita o vídeo para o sistema e, uma vez que esta requisição é aceita, um distribuidor

será responsável por enviar os blocos de vídeos para o cliente. Assim, cada cliente é alocado a um distribuidor, enquanto cada distribuidor serve a um ou mais clientes. O processo de comunicação inicia-se entre o cliente e o servidor de vídeo. Após o sistema ter aceitado o novo cliente (usuário que deseja assistir a um vídeo), o cliente passa a se comunicar apenas com o distribuidor.

SMART: Executando o GloVE sobre o OpenVZ

Em sua concepção original, cada instância do distribuidor executaria em um nó computacional dedicado. Porém, devido às características deste aplicativo, um nó poderia ficar subutilizado, enquanto outro precisaria de mais recursos, sendo que todos os distribuidores deveriam tratar todos os filmes, limitando, desta forma, tanto o número de vídeos como o de clientes por vídeo. O sistema SMART [36], desenvolveu uma solução para este problema executando as instâncias do distribuidor dentro de contêineres do OpenVZ. Desta forma, cada nó físico passou hospedar uma ou mais instâncias da aplicação, sendo que, nesta versão, cada instância do distribuidor ficou responsável por apenas um único vídeo.

A utilização dos contêineres permitiu que o distribuidor fosse migrado entre nós físicos sem a necessidade de se alterar a aplicação, viabilizando desta forma o balanceamento de carga e tolerância a falhas neste sistema. A única restrição imposta pelo sistema foi que a migração tinha que ocorrer em menos de 10 segundos ou o cliente irá experimentar alguma degradação no serviço. Contudo, quando um contêiner migrava, mesmo com o tempo de migração tendo sido muito inferior a 10 segundos, alguns segundos após a migração terminar os clientes paravam de receber os fluxos de vídeo. As investigações realizadas demonstraram que o problema era devido à técnica de temporização utilizada, baseada na instrução RDTSC [3].

A solução encontrada para viabilizar os experimentos foi substituir a instrução RDTSC pela chamada de sistema gettimeofday() e manter o relógio dos nós sincronizados utilizando o NTP. O risco em se utilizar o NTP é que as atualizações periódicas alteram o intervalo de tempo real, sendo que, felizmente, tal problema não ocorreu durante experimentos devido ao intervalo suportado pela aplicação (10 s) ser grande o suficiente para ocultar as mudanças realizadas pelo NTP.

2.5 Formalização do Problema

Observando o ocorrido com o sistema GloVE durante os experimentos iniciais do SMART, é possível descrever formalmente o problema estudado neste trabalho. Considere dois nós computacionais físicos, N_A e N_B, inteconectado entre si e ambos executando a mesma versão do *kernel* do sistema operacional, por exemplo o OpenVZ. Seja V_1 uma máquina virtual hospedada em N_A, V_1 será migrada para o nó N_B, o qual possivelmente já possuí outras máquinas virtuais executando. Sendo C_{V_n} a informação de tempo vinculada a máquina virtual V_n, onde a informação contida em C_{V_n} é similar à do registrador TSC na arquitetura x86. Logo, utilizando dois valores consecutivos de C_{V_n}, é possível saber quanto tempo se passou entre esses valores. O problema é que, se C_{V_n} armazenar o mesmo valor que TSC, esta informação não é mais útil se V_n sofrer uma migração, já que o valor do TSC lido em N_A não tem significado em N_B.

Por exemplo assuma que existe uma máquina virtual V_1 executando em N_A e que será migrada para N_B: por hipótese temos que os processadores de N_A e N_B trabalham na mesma freqüência, onde N_B foi ligada depois de N_A. Sendo que, existe ao menos uma aplicação executando em V_1 onde realiza leituras à C_{V_1}, ou seja executa a instrução RDTSC, armazenando esse resultado em uma variável local. Tendo que V_1 foi migrada para N_B a leitura seguinte à C_{V_1} retornará o valor do TSC de N_B, que é menor que o valor retornado pela leitura feita em N_A. Desta forma, o cálculo do intervalo de tempo resultará em um valor negativo, ou seja, absurdo.

Desta forma, o problema é como manter C_{V_1} neste cénario, tal que a informação nele armazenada não perca seu significado na presença de migrações. Sendo que este problema pode ser dividido em dois subproblemas:

1. Como manter o valor de C_{V_1} monotonicamente crescente, quando V_1 foi migrada para outro nó físico;

2. Como, utilizando o valor armazenado por C_{V_1}, pode se inferir com precisão, uma temporização relativa para a aplicação, sem com isso alterar os valores de C_V de outras máquinas virtuais.

É importante salientar que a principal propriedade que qualquer solução para este problema deve garantir é o comportamento monotonicamente crescente de C_V.

Uma vez que esta condição for satisfeita, por exemplo, em um *cluster* homogênio de computadores, a noção de tempo relativa pode ser facilmente derivada, porém, em *clusters* heterogêneos existrá a necessidade do subproblema dois ser mais elaborado para que a mesma informação seja produzida, pois com a existência de nós com processadores trabalhando a diferentes freqüências o uso do número de ciclos gastos no nó de origem para o cálculo do total de ciclos faz com que o resultado obtido pela aplicação seja incorreto.

2.6 Escopo

Neste trabalho são considerados dois tipos de ambientes onde as aplicações de tempo-real executam, sendo que, em ambos apenas o TSC utilizado como contador de tempo. O primeiro ambiente é um cluster de computadores onde a aplicação pode ser migrada entre os nós deste cluster. O segundo ambiente é uma arquitetura com multiplos núcleos em um mesmo processador onde os núcleos podem realizar escalonamento de freqüência e não são obrigados a executar todos na mesma freqüência.

Dado que a motivação inicial deste trabalho surgiu no primeiro cenário onde o problema é mais acentuado, o capítulo de temporização (capítulo 3) trata apenas de soluções para clusters. Os trabalhos relacionados para o problema de múltiplos núcleos com escalonamento de freqüência foram apresentados anteriormente na Seção 2.1 deste capítulo.

Capítulo 3

Temporizando Aplicações em um *Cluster* de Máquinas Virtuais

Neste capítulo, são descritas como algumas das técnicas de temporização para *clusters* de computadores podem ser utilizadas para resolver, sob condições específicas, o problema apresentado no Capítulo 2. Essas soluções são baseadas na sincronização global entre os nós do *cluster* e a chamada de sistema gettimeofday(). Com relação ao escopo deste trabalho estas soluções encontram-se classificadas em duas categorias:

- Técnicas de Sincronização Global Absoluta
- Técnicas de Sincronização Global Relativa

3.1 Técnicas de Sincronização Global Absoluta

As técnicas de sincronização global absoluta (TSGA) são baseadas no conceito de relógio universal ou relógio de parede (*wall-clock*), com o qual todos os nós devem sincronizar; a técnica mais conhecida de sincronização global para redes de computadores, o NTP, pertence a esta categoria.

3.1.1 Network Time Protocol

O Protocolo de Tempo para Redes (*Network Time Protocol* - NTP) é um padrão do IETF [37, 15] para sincronização de nós na Internet. O NTP [15] versão 3 foi proposto em 1992, ele é uma aplicação cliente/servidor hierárquica que distribuir o tempo universal (UTC), realizando sincronizações sobre o relógio local. O protocolo NTP executa como um serviço nos clientes, que periodicamente contacta um servidor NTP para atualizar seu relógio local baseando-se na informações de tempo passadas pelo servidor. A troca de mensagens entre o servidor e o cliente NTP gera um valor Δ que é aplicado sobre o relógio local.

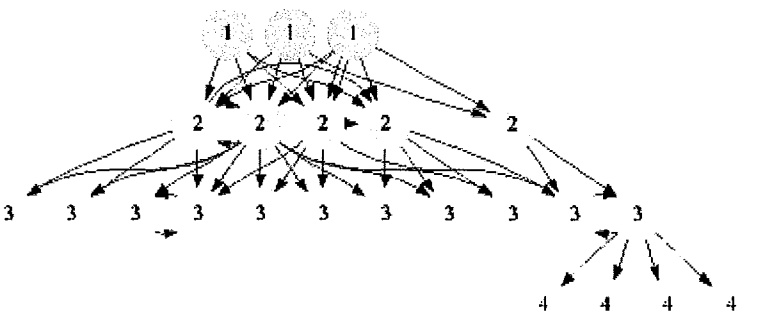


Figura 3.1: Arquitetura do protocolo NTP

Os servidores NTPs são organizados como uma árvore (Figura 3.1), onde a raiz de uma árvore NTP é conhecida como servidor de *stratum* 0 e representa o relógio universal. O maior *straum* possível é o 16 que representa um servidor que não esta respondendo. Os níveis de 1 a 15 representam os nós onde os clientes podem se conectar para sincronizar seus relógios. Em teoria, baseando-se na hierarquia apresentada, nós com *stratum* menores oferecem uma informação mais acurada.

Um cliente NTP pode se conectar a qualquer nó da árvore com exceção da raiz, sendo que os parâmetros, em geral, mais relevantes no processo de seleção são os retardos de rede e carga do servidor. Uma vez que o servidor aceita a conexão do cliente algumas mensagens serão trocadas e, se necessário, o cliente atualizará seu relógio local.

Em uma primeira análise, o NTP consegue solucionar o problema de temporização descrito na Seção 2.5 do capítulo 2, uma vez que todos os computadores têm seus relógios sincronizados com a mesma fonte. Contudo, todos os relógios desviam

inevitavelmente de UTC e com isso será possível que duas chamadas consecutivas em um mesmo nó da função gettimeofday() [16], onde a primeira chamada retorna $T1$, e segunda $T2$ tal que $T1 > T2$. Esta situação deve-se ao fato do NTP poder atualizar o relógio local para um tempo passado. Uma situação semelhante pode ocorrer em uma migração entre nós onde o destino possuí uma referência de tempo mais antiga devido ao NTP.

O NTP usualmente viola a propriedade monotonicamente crescente quando a granularidade da medida é inferior a dezenas de milisegundos, de acordo com medidas feitas na rede do LCP. O problema ocorre quando os clientes encontram-se carregados e não conseguem executar o serviço do NTP com a frequência necessária, nestes casos os desvios podem chegar a dezenas de segundos [33] e não mais milisegundos.

3.1.2 Sincronização de Relógios Quase Par-a-Par

O protocolo de Sincronização de Relógios Quase Par-a-Par [38] (*Almost Peer-to-Peer Clock Synchronization* - AP2P), realiza periodicamente a troca de pacotes (mensagens de sincronização) entre os nós vizinhos. O protocolo funciona elegendo um líder que funciona como se fosse um nó *stratum* 1 no NTP, sendo que a maior diferença entre o procedimento de eleição de líder AP2P e outros algoritmos de eleição de líder em sistemas de sincronização é que no AP2P existe a possibilidade de existir estados transientes no sistema onde existe mais de um líder.

Considere que o AP2P executa sobre em uma rede da classe B como a da Figura 3.2 [38], composta por 8 subredes (nomeadas de A a H) e que esta rede encontra-se totalmente conectada. Nestas condições, apenas um líder será eleito e ele irá ser a raiz de uma árvore geradora que representa a rede. Uma vez que o líder foi eleito todos os nós sincronizam seus relógios realizando trocas de mensagens com seus vizinhos, levando em consideração a distância que estes vizinhos encontram-se do líder. A vantagem desta abordagem é que, se ocorrer alguma desconexão, os nós em ambos os conjuntos conseguem se reoganizar e eleger um novo líder, se necessário, o que mantém a sincronização entre os nós do conjunto. Uma vez que a conexão seja restabelecida um dos líderes passa a ser o novo líder do conjunto de nós.

O artigo que apresenta AP2P descreve apenas os resultados simulados e não

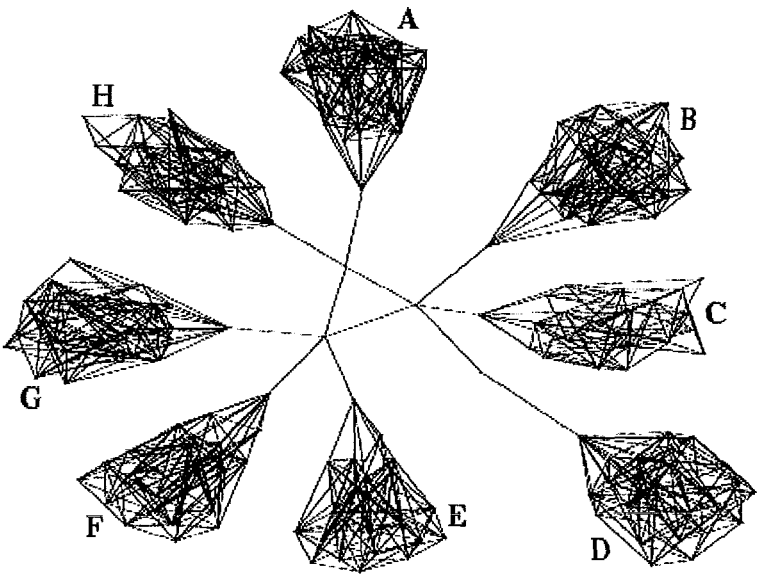


Figura 3.2: Topologia de uma rede de Classe B

uma implementação real. Porém, como este protocolo realiza trocas periódicas de mensagens entre seus pares, sua implementação sofre obrigatoriamente da mesma deficiência do NTP, descrita em [33]. Sendo que estes protocolos ainda podem ser utilizados se a granularidade do intervalo de tempo for grande o suficiente para esconder as violações no crescimento monotônico do relógio.

3.2 Técnicas de Sincronização Global Relativa

As soluções baseadas em Técnicas de Sincronização Global Relativa (TSGR), não garantem que os relógios locais encontram-se sincronizados com o UTC, mas garantem que o tempo encontra-se globalmente sincronizado dentro de um *cluster* de computadores. Essa abordagem permite que o relógio local do nó computacional seja, por exemplo o TSC [39].

3.2.1 Sincronização de Alta-Precisão com *Time Stamps Counters*

O protocolo de Sincronização de Alta-Precisão com *Time Stamps Counters* [40] (*High-Precision Relative Clock Synchronization Using Time Stamps Counters*), foi

construido utilizando o registrador TSC como fonte de tempo. Seus autores fizeram esta escolha devido ao fato de os osciladores de cristal utilizados em outros circuitos serem instáveis e pelo TSC oferecer uma medida de tempo muito mais precisa, dado que ele é incrementado a cada ciclo do núcleo.

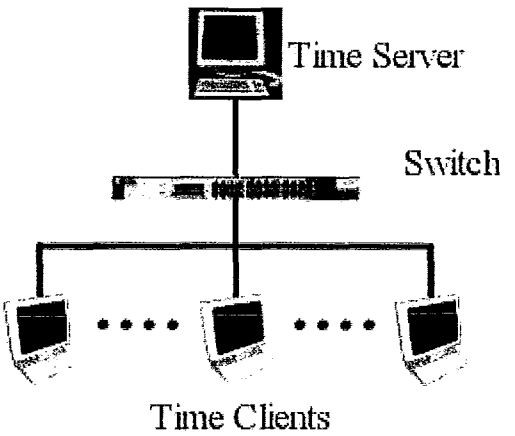


Figura 3.3: Arquitetura Física do Protocolo de Sincronização de Alta-Precisão Utilizando *Time Stamps Counters*

O protocolo utiliza uma estrutura mestre/escravo (Figura 3.3 [40]), onde o mestre é o Servidor de Tempo (*Time Server*) e o algoritmo de sincronização dos relógios é derivado do NTP. A diferença marcante entre este protocolo de sincronização de outras abordagens puramente em software é que ele não utiliza uma fonte de tempo global de alta precisão.

O protocolo Sincronização de Alta-Precisão Utilizando *Time Stamps Counters*, como o NTP pode violar a propriedade de crescimento monotônico quando o intervalo entre as medidas for pequeno. A propriedade é violada devido ao fato de que o nó ter que ajustar seu relógio global baseado na troca de mensagens com o servidor, o ajuste é efeito aplicando-se um Δ ao valor do TSC. Por motivos de precisão, o protocolo foi projetado para executar dentro do *kernel*, contudo, isso não impossibilita que em nós muito carregados o fluxo de execução (*kernel thread*) possa não conseguir executar com a frequência necessária para manter os nós globalmente sincronizados.

3.2.2 *Precision Time Protocol*

O *Precision Time Procol* (PTP) [41] é um padrão IEEE (IEEE 1588 e IEEE1588-2008 [42]) para sincronização de sistemas de tempo-real críticos. O funcionamento básico deste algoritmo é que o relógio mais preciso dentro do *cluster* sincroniza todos os outros. O protocolo funciona utilizando dois tipos de relógios, mestre e escravo, onde inicialmente qualquer relógio pode exercer tanto a função de mestre quanto a de escravo. Neste protocolo, os relógios são classificados com relação a sua precisão em classes (*stratum* no NTP), onde a maior classe é o relógio atômico que possui *stratum* 1. Uma vez que o relógio mestre tenha sido definido, todos os outros relógios tornam-se escravos e devem determinar o deslocamento ($d(t)$) existente entre seu relógio local e o relógio mestre (Equação 3.1), onde $s(t)$ representa o tempo medido no relógio do escravo no instante t e $m(t)$ representa o tempo medido no relógio do mestre no mesmo instante t.

$$d(t) = s(t) - m(t) \tag{3.1}$$

O protocolo funciona através do envio periódico de mensagens a partir do mestre para os escravos. Desta forma, os dispositivos escravos podem recomputar o deslocamento com relação ao relógio mestre que tende a desviar com o o tempo (*drift*). O PTP assume duas premissas para seu correto funcionamento. A primeira é de que as trocas de mensagens ocorrem em intervalos pequenos de forma que o deslocamente aplicado sobre o relógio escravo pode ser considerado constante e a segunda é de que a rede onde as mensagens são trocadas é simétrica. Desta forma, a precisão do IEEE 1588 depende do grau com o qual essas premissas são garantidas.

A precisão da sincronização é altamente dependente da rede e dos componentes utilizados na construção desta. O PTP é baseado em comunicação IP *multicast*, não sendo restrito apenas a *Ethernet*, mas podendo ser utilizado com qualquer padrão de rede que suporte *multicast*. Sendo que os serviços que implementam o protocolo executam como processos de baixa prioridade, o protocolo consegue precisões da ordem de $100\mu s$ para implementações totalmente em software, podendo melhorar esta precisão até $10\mu s$. Como o IEEE 1588 executa como um serviço nos computadores do *cluster*, se o nó encontra-se sobrecarregado o PTP não conseguirá executar com a frequência necessária para manter o sistema sincronizado, similar ao que ocorre

no NTP.

3.2.3 Relógio Global Distribuído

O Relógio Global Distribuído para *Cluster* de Computadores [43] é uma técnica de sincronização por *hardware* desenvolvida no Laboratório de Computação Paralela (LCP). O *cluster* que utiliza o relógio Global possuí uma rede secundária especial para sincronização, onde o *hardware* desta rede de sincronização recebe os tiques de um oscilador remoto (relógio). A rede de sincronização é organizada como uma árvore onde a raiz é o oscilador remoto, onde os cabos desta rede devem ter o mesmo comprimento. Desta forma, os nós conseguem saber o tempo global dentro do *cluster* lendo o registrador local que armazena os tiques do oscilador.

O *hardware* proposto para suportar a rede de sincronização não necessita de interferência das camadas de *software* para realizar a atualização da informação do relógio local, sendo que as únicas funcionalidades que envolvem *software* são as operações de reset e leitura da instância local do relógio global (*time stamp*).

3.3 Considerações Finais

As soluções que necessitam de interação com *software* para manter os nós sincronizados não conseguem garantir a desigualdade descrita na Equação 3.2, para a condição descrita pela Equação 3.3 abaixo, quando sincronização global é utilizada. Isso ocorre principalmente pelo fato de os serviços em *software* não conseguirem executar com a frequência mínima necessária.

$$T1 < T2 \tag{3.2}$$

$$\lim_{(\forall T2, T1 | T2 > T1)} (T2 - T1) \longrightarrow 0 \tag{3.3}$$

Contudo, soluções puramente em *hardware* como o Relógio Global conseguem garantir a propriedade de crescimento monotônico, porém, num ambiente muito mais restrito, onde a diferença entre os comprimento de qualquer enlace físico deve ser inferior a um valor pré definido, onde esse valor é dado pelo comprimento de onda utilizado.

Capítulo 4

vTSC: *virtual Time Stamp Counter*

A solução proposta e descrita nesta dissertação recebeu o nome de ***virtual Time Stamp Counter***, ou **vTSC**, devido à motivação inicial de substituir a instrução RDTSC (***ReaD Time Stamp Counter***) da arquitetura x86 por um mecanismo virtualizado de alta precisão e baixo custo computacional. Neste capítulo o vTSC é descrito, e as implementações para o OpenVZ e Linux são apresentadas.

4.1 vTSC

4.1.1 Premissas

Os algoritmos e estrutura de dados que compõem o virtual Time Stamp Counter (vTSC) são intimamente ligados aos recursos disponíveis nas arquiteturas de computadores modernos. Para fundamentar o estudo realizado, é necessário definir algumas premissas sobre as quais a solução apresentanda nesta dissertação foi elaborada.

Seja C um circuito utilizado para aferir o tempo em computadores digitais; o **vTSC** poderá utilizar este circuito se a Equação 4.1 for verdadeira, onde $U(t)$

representa o tempo universal e $C(t)$ a medida de tempo retornado, pelo circuito.

$$U(t) - C(t) = 0, \forall t \in \Re \tag{4.1}$$

$$\lim_{t \to \infty} \parallel \Sigma[U(t) - C(t)] \parallel \longrightarrow 0 \tag{4.2}$$

$$\exists t | U(t) - C(t) \neq 0 \tag{4.3}$$

Quando a Equação 4.3 é verdadeira, o circuito somente pode ser utilizado se Equação 4.2 também for verdadeira, ou seja, se a diferença acumulada entre o tempo real e o tempo medido tender a zero. Esta condição (Equação 4.2) é necessária e suficiente para utilizar um circuito digital como base para vTSC. As outras premissas assumidas na descrição do vTSC são listadas a seguir:

1. Não existe temporização global entre os nós;
2. Todos os nós que irão sincronizar encontram-se interconectados por enlaces sem perdas e simétricos;
3. O tempo de migração da aplicação entre dois nós quaisquer é insignificante;
4. O Circuito responsável por aferir o tempo não sofre *overflow*.

A Premissa 1 foi um compromisso assumido com o objetivo de garantir que o custo computacional ligado ao número de mensagens trocadas é mínimo. Desta forma vTSC deve ser construido de forma a funcionar corretamente quando os nós não possuem mecanismo de temporização global. O uso de enlaces sem perdas e simétricos tem como objetivo limitar o protocolo do vTSC apenas ao processo de sincronização.

As premissas descritas anteriormente são consideradas suficientes para o vTSC e devem ser respeitadas pelas implementações do modelo proposto, porém, para tornar mais clara a descrição da solução duas outras premissas foram assumidas. A Premissa 3 tem como objetivo assegurar a eficiência da técnica de migração utilizada, sendo que, na descrição do vTSC, o circuitos utilizado para aferir o tempo não deve sofrer *overflow* (Premissa 4).

4.1.2 Solução

O vTSC soluciona o problema descrito anteriormente (Seção 2.5) construindo uma camada de software entre a aplicação que deseja aferir o tempo e os valores retornados pela instrução RDTSC, sendo que ele foi concebido com base nos mecanismos de memória virtual encontrados nos sistemas operacionais modernos e no uso do ***Time Stamp Counter*** (TSC) como circuito para aferir o tempo. O TSC foi escolhido devido as vantagens que ele oferece com relação a resolução (precisão) da medida e o custo computacional para obtê-la.

Utilizando o *Time Stamp Counter* para aferir o tempo em Computadores Digitais

Uma aplicação que deseje utilizar a instrução RDTSC para aferir o tempo necessita armazenar um valor de 64 *bits*, o que em arquitetura de 32 *bits* exige a construção de um tipo de dados abstrato inteiro sem sinal de 64 *bits* para armazenar a informação que for retornada. A Figura 4.1 é um exemplo de como esse tipo de dado pode ser criado, onde os campos da estrutura são inteiros sem sinal de 32 *bits*. Ao ser executada, a instrução RDTSC armazena os 32 *bits* de ordem alta no registrador EDX e os de ordem mais baixa no registrador EAX, sendo que estes valores são armazenados respectivamente nos campos **hi** e **lo** da estrutura. O cálculo da informação temporal é feito multiplicando-se esses valores pela duração do ciclo do processador.

```
typedef struct _T_TSC{
  uint32_t lo;
  uint32_t hi;
}T_TSC;
```

Figura 4.1: Estrutura de dados utilizada na leitura do registrador TSC

Como a freqüência do processador deve ser conhecida pela aplicação que realiza a medida, ela deve ser constante durando toda a execução da aplicação, o que elimina a possibilidade de economizar energia reduzindo a freqüência de trabalho do processador.

Virtualizando o TSC

O *virtual TSC* (**vTSC**) é uma camada de *software* entre o processador e a aplicação que deseja aferir o tempo utilizando o TSC. Esta camada é responsável por criar uma bijeção entre os valores retornados pela execução da instrução RDTSC e os valores apresentados para a aplicação. Uma diferença marcante no uso do vTSC foi a decisão de retornar a informação de tempo para a aplicação em nanosegundos e não em ciclos, permitindo que o vTSC contabilize para a aplicação as mudanças na freqüência do processador, garantindo para aplicação uma medição correta do tempo decorrido.

```
typedef struct _T_VTSC{
  T_TSC v_counter_base;
  unsigned long long run_ns;
}T_VTSC;
```

Figura 4.2: Descrição da estrutura de dados que suporta o vTSC

A estrutura de dados que suporta vTSC pode ser observada na Figura 4.2.Ela é composta por dois campos: o *v_counter_base* e *run_ns*. O *v_counter_base* é do tipo *T_TSC*, apresentado na Figura 4.1 e armazena os valores retornados pela execução da instrução RDTSC. O segundo campo da estrutura *T_VTSC*, o *run_ns*, é um inteiro sem sinal de 64 bits utilizado para medir o tempo em que aplicação encontra-se ativa (ou a quanto tempo vTSC foi criado). O tipo de dado abstrato *T_VTSC* é acrescentado nas informações que o sistema operacional armazena para cada processo, desta forma, quando um novo processo é criado, o trecho de código da Figura 4.3 deve ser executado para inicializar a instância do vTSC. Sendo que a função *atual_ns*() representa a execução da instrução RDTSC.

```
void vtsc_init(t_vtsc *vTSC){
    vTSC.v_counter_base_ns  = atual_ns();
    vTSC.run_ns = 0;
}
```

Figura 4.3: Inicialização da instância do vTSC vinculada a um processo que esta sendo criado dentro do sistema operacional

Como vTSC é inicializado no instante em que a aplicação é criada, ele representa a idade desta aplicação em nanosegundos. Desta forma, a instrução RDTSC pode ser substituída pela chamada de sistema *vtsc_gettime_ns*() (Figura 4.4), onde a função *get_cpu_freq*() é o fator multiplicativo referente a freqüência atual do processador. A chamada *vtsc_gettime_ns*() recebe como parâmetro de entrada o endereço onde a informação deve ser salva.

```
unsigned long long vtsc_gettime_ns(t_vtsc *vTSC){
    unsigned long long rtn;
    rtn = (atual_ns() - vTSC.v_counter_base_ns) * get_cpu_freq();
    return vTSC.run_ns + rtn;
}
```

Figura 4.4: Chamada de sistema utilizada pela aplicação para consultar vTSC

Migração utilizando o vTSC

A utilização da chamada *vtsc_gettime_ns*() permite que a aplicação seja migrada entre núcleos de processamento (nós computacionais) sem causar problemas às medidas realizadas pela aplicação. A Figura 4.5 apresenta o algoritmo utilizado para migrar uma aplicação entre núcleos, onde a estrutura do processo também é migrada.

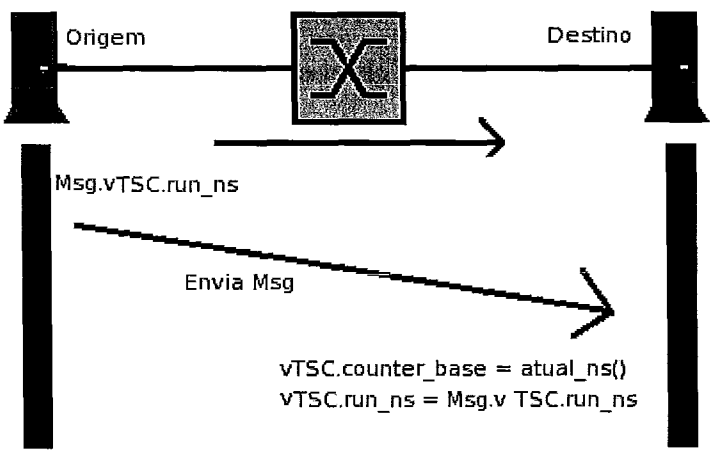


Figura 4.5: Algoritmo de Migração do vTSC

O procedimento de migração é dividido em duas etapas, a primeira é no nó de origem, ou simplimente Origem, onde o sistema operacional suspende a execução

do processo que esta sendo migrado e posteriormente descobre a quanto tempo ele encontra-se executando neste nó utilizando as Equações 4.4 e 4.5, abaixo. Com essas informações geradas a Origem envia para o nó de destinio, ou simplemente Destino, uma mensagem contendo o novo $vTSC.run\_ns$ além das informações do processo. Ao receber esta mensagem, o Destino atualiza $vTSC.counter\_base$ e $vTSC.run\_ns$, com o retorno da função $atual\_ns()$ e $Msg.vTSC.run\_ns$ respectivamente, antes de reiniciar a execução do processo.

$$run\_aux = (atual\_ns() - vTSC.counter\_base) * get\_cpu\_freq(); \quad (4.4)$$

$$vTSC.run\_ns = vTSC.run\_ns + run\_aux; \quad (4.5)$$

O algoritmo descrito anteriormente pode ser executado diversas vezes, uma para cada migração sofrida pela aplicação. Assumindo que as variáveis onde as informações de tempo são lidas e armazenadas possuem precisão infinita, ou seja, não ocorre *overflow* é trivial perceber que o algoritmo proposto consegue resolver o problema da Seção 2.5.

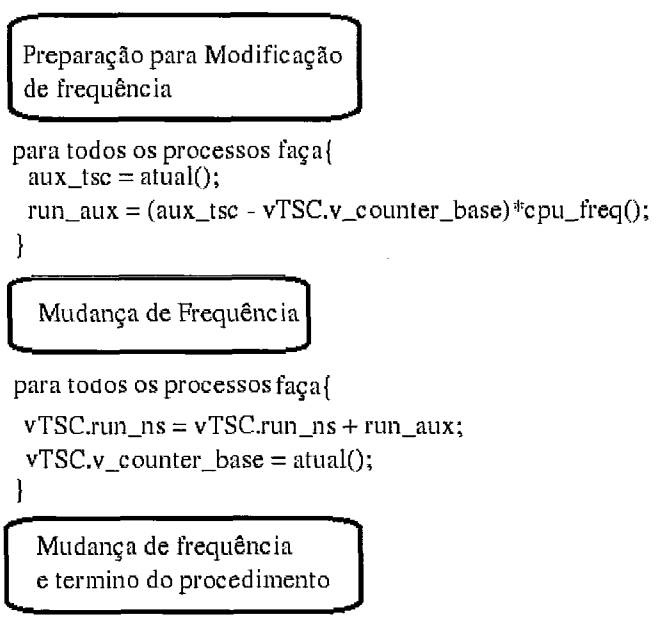


Figura 4.6: Driver para mudança de freqüência do processador utilizando vTSC

A Figura 4.6 representa o *driver* utilizado para mudar a freqüência do processador e como vTSC consegue tornar essa mudança transparente para a aplicação

que deseja medir o tempo utilizando a chamada *vtsc_gettime_ns*(). Na Figura 4.6, as caixas amarelas são as funções que o *driver* executa normalmente e as linhas de pseudo-código, as entradas necessárias para vTSC funcionar. Sendo facil perceber, que o custo deste algoritmo é $\Theta(n)$, onde n é número de processos que encontram-se vinculados ao núcleo que vai mudar de freqüência, enquanto simples esse algoritmo demonstra com facilidade que vTSC consegue resolver o problema causado pelas mudanças de freqüências.

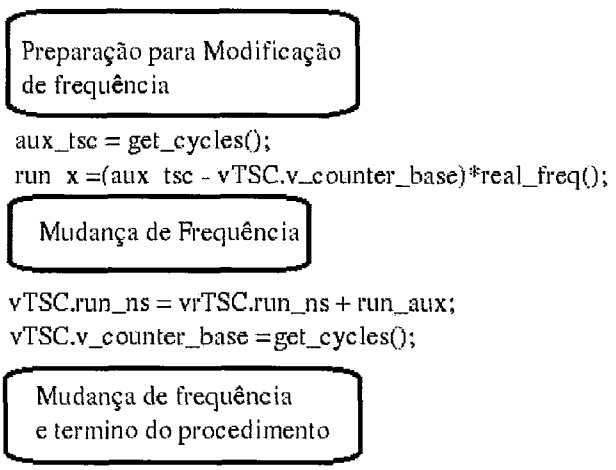


Figura 4.7: Driver otimizado para mudança de freqüência do processador utilizando vTSC

Existe ainda uma otimização que reduz a complexidade desta solução para $\Theta(1)$ (Figura 4.7). Ela é feita construindo uma abstração sobre o processador de forma que a função *atual_ns*() passa a devolver a informação em nanosegundos e não mais em ciclos, e *get_cpu_freq*() retorna sempre o valor 1. Esta solução utiliza uma instância da estrutra *T_VTSC* sobre o processador, onde a intância é que passa a servir de base para a função *atual_ns*(), e apenas a instância do vTSC do processador precisa ser atualizada durante a mudança de freqüência e não mais as intâncias de vTSC ligadas aos processos. É necessário também substituir a função *atual_ns*() por *get_cycles*() que retorna-se o valor em ciclos e *get_cpu_freq* por *real_freq*() que retorna o fator multiplicativo relacionado à freqüência atual do processador. Com essas mudanças o código-fonte da função *atual_ns*() passa a ser semelhante ao da chamada *vtsc_gettime_ns*.

O grande diferencial de vTSC para as soluções apresentandas no Capítulo 3 deve-

se ao fato de vTSC não forçar globalmente uma sincronização entre os nós. Pois como nele as informações necessárias para aferir o tempo encontram-se vinculadas aos processos, somente é preciso que durante o procedimento de migração essas informações sejam atualizadas, o que exige apenas a cooperação já existente entre a Origem e o Destino. Como foi visto vTSC pode ser estendido para tratar também o problema causado pela mudança na freqüência do processador.

4.2 Implementando o vTSC no OpenVZ

A primeira implementação da proposta do vTSC foi realizada para o modelo de virtualização no nível do Sistema Operacional (contêineres), em especial para o OpenVZ. Essa escolha foi feita por motivos históricos, pois foi neste modelo que o problema foi primeiramente identificado, além de fornecer um ambiente controlado onde facilmente as avaliações do algoritmo de migração do vTSC podem ser realizadas. Esta seção explica como o modelo descrito na Seção 4.1 foi implementado e quais foram os compromissos assumidos no decorrer de sua realização.

4.2.1 Arquitetura Básica

A implementação do vTSC para o OpenVZ foi projetada para um ambiente de computação em *cluster* compartilhado onde os nós encontram-se sempre com carga máxima (PlanetLab [33]).Sendo assim, não existe possibilidade de se utilizar escalonamento da freqüência. A exclusão das técnicas de escalonamento de freqüência limita os ambientes computacionais onde o vTSC pode ser utilizado, porém esta premissa pode ser facilmente retirada com um aumento no custo de implementação do vTSC como apresentado na Seção 4.1. Assim como, o tempo de migração anteriormente desconsiderado passa agora a ser levado em consideração.

Arquitetura do Processador

A arquitetura de processador que suporta esta versão do vTSC é a da família x86, mais especificamente a de 32 bits. Essa escolha foi feita tanto por motivos de compatibilidade, quanto por representar um pior caso, pois a leitura do registrador

TSC (64 bits) é um processo mais complexo. Esta escolha representou a criação de um tipo de dado abstrato (TAD) que armazena o valor retornado pela instrução RDTSC, como pode ser observado na Figura 4.1.

4.2.2 Integração vTSC com o OpenVZ

A implementação no OpenVZ foi dividida em duas fases, a primeira consistiu em introduzir dentro do *kernel* do OpenVZ as estruturas de dados e controles necessários para o funcionamento do vTSC, a segunda parte foi modificar os aplicativos de controle do OpenVZ que executam a nível de usuário para passar a utilizar as novas funcionalidades introduzidas.

vTSC no Kernel OpenVZ

A implementação do vTSC foi feita sobre o kernel do Linux versão 2.6.18 com o patch OpenVZ 028stab057.2, desta forma a estrutura de dados que suporta o vTSC nesta plataforma (Figura 4.8) estende a descrição abstrata apresentada no Seção 4.1 passando a contar também o tempo que o contêiner (aplicação no modelo abstrato) esteve parado. Foi escolhido vincular a instância do vTSC aos contêiners do OpenVZ, reduzindo desta forma tanto espaço em memória ocupado pela solução, como o custo de processamento para torna-la funcional.

```
typedef struct _T_VTSC{
  unsigned long id;
  T_TSC v_counter_base;
  unsigned long long runnable_time_ns;
  unsigned long long suspended_time_ns;
}T_vTSC;
```

Figura 4.8: Tipo de Dado Abstrato *T_vTSC*

Um contêiner OpenVZ assim como um processo em um sistema operacional possui uma estrutura de dados que armazena as informações privadas do contêiner a **struct** *ve_struct*. Foi esta estrutura de dados que recebeu a instância do vTSC, sendo que está instância é inicializada no instante em que o contêiner é criado. Desta forma o campo *v_counter_base* recebe o resultado da execução da instrução RDTSC no instante da criação do processo e os campos *runnable_time_ns* e *suspended_time_ns* são inicializados com zero. A chamada de

sistema *vtsc_gettime_ns*() descrita no Seção 4.1 passa a se chamar *ve_get_runtime*() (Figura 4.9).

```
long sys_ve_get_runtime(unsigned long long __user *p_runt)
```

Figura 4.9: Chamada de sistema para recuperar o tempo em execução

Como o tempo de migração não insignificante neste ambiente, foi necessário construir um mecanismo que permita a aplicação descobrir o tempo que efetivamente passou e não somente o tempo de execução. A chamada de sistemas que exporta para fora do *kernel* esta informação pode ser vista na Figura 4.10.

```
long sys_ve_get_suspendtime(unsigned long long __user *p_suspendt)
```

Figura 4.10: Chamada de sistema para recuperar o tempo em execução + suspenso

Migração do OpenVZ com suporte ao vTSC

Neste ambiente, a migração ocorre entre nós computacionais diferentes, desta forma o algoritmo de migração descrito na Seção 4.1.2 teve que ser integrado com o processo de migração de contêiner realizado pelo OpenVZ. A principal diferença decorre da forma como campo *vTSC.suspended_time_ns* é mantido dentro do sistema. A Figura 4.11 apresenta um diagrama de alto-nível da implementação do procedimento de migração do vTSC dentro do OpenVZ.

O algoritmo de migração integrado mostrado na Figura 4.11 utilizado nesta implementação é descrito a seguir:

1. Seja $C0$ o contêiner que se deseja migrar, localizado na Origem;
2. O OpenVZ inicia o processo de Suspend de $C0$. Ao final desta etapa o tempo de execução já encontra-se consolidado em vTSC e o valor retornado por RDTSC utilizado para consolidar o tempo de execução (Figura 4.12) é armazenado em *vTSC.v_couter_base*;
3. O OpenVZ executa seu procedimento padrão de *Dump*, onde a imagem do contêiner (informações privadas e memória usada pelas aplicações) é serializada e escrita em um arquivo que será enviado para o destino, ao final desta etapa um servidor é iniciado na Origem para realizar a migração do vTSC;

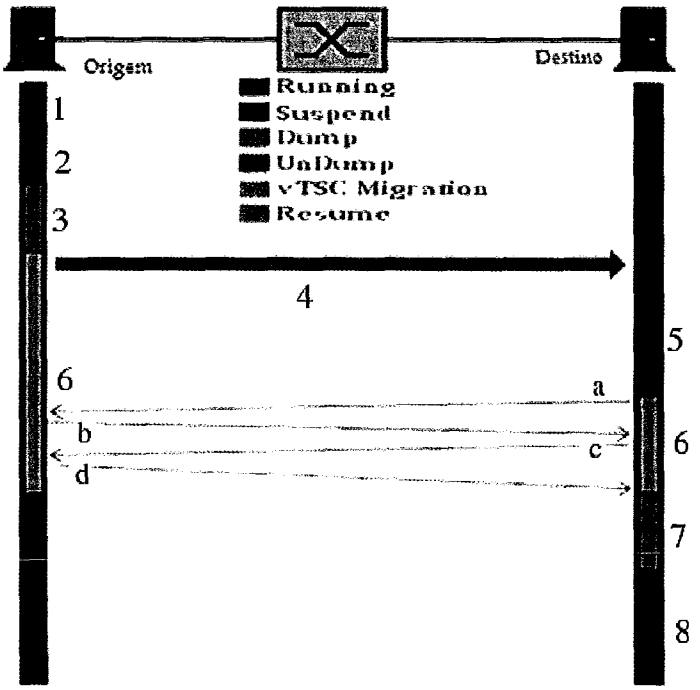


Figura 4.11: Migração do OpenVZ com as modificações feitas para o suporte ao vTSC

4. Cópia dos arquivos necessários para recriar o contêiner no Destino;

5. O nó Destino executa o procedimento de *UnDump* sobre os arquivos gerados na etapa 3. Ao final desta etapa, as estruturas de controle encontram-se já criadas e preenchidas dentro do OpenVZ;

6. Após a etapa 5 ter sido concluída entra em execução o processo de migração do vTSC utilizando o servidor que foi criado na etapa 3. O algoritmo começa com o Destino iniciando um cliente e a Origem o servidor;

   (a) O Destino envia um pedido para origem contendo o $VEID$ do contêiner que foi migrado;

   (b) A Origem responde enviando o vTSC, consolidando o tempo em que o contêiner encontrava-se suspenso (até o envio desta mensagem);

   (c) O Destino envia uma mensagem extra de controle à Origem para poder aproximar o tempo de vôo (tempo que uma mensagem leva para trafegar na rede);

   (d) A Origem responde com a sua medida do tempo de vôo;

(e) O Destino é responsável por somar o tempo de vôo com o valor atual de *vTSC.suspended_time_ns*, armazenando o resultado em *vTSC.suspended_time_ns* e *vTSC.v_counter_base* recebe o valor retornado por RDTSC neste instante (Figura 4.13);

7. O OpenVZ inicia o procedimento de Resume do contêiner *C0*;

8. O contêiner *C0* agora encontra-se executando no Destino.

```
/**
DIEGO INIT: COLETA
*/
.       rdtsc(lo, hi);
.       tempPrev = (uint64_t)env->vTSC.v_counter_base.hi;
.       tempPrev = tempPrev*10000000000ULL +  (uint64_t)env->vTSC.v_counter_base.lo;
.       result = (uint64_t)hi;
.       result = result*10000000000ULL + (uint64_t)lo;
.       result = result - tempPrev;
.       tempPrev = do_div(result, tsc_khz);
.       result = result * 1000000;
.       tempPrev = tempPrev * 1000000;
.       env->vTSC.v_counter_base.lo = lo;
.       env->vTSC.v_counter_base.hi = hi;
.       lo = do_div(tempPrev, tsc_khz);
.       result = result + tempPrev;
.
.       env->vTSC.runnable_time_ns = env->vTSC.runnable_time_ns + result;
/**
DIEGO END: COLETA
*/
```

Figura 4.12: Trecho de código executado ao final do processo de *Suspend* do OpenVZ para computar os dados necessários para vTSC

vTSC em Nível de Usuário no OpenVZ

O sistema OpenVZ possui algumas aplicações de controle que são executadas pelo usuário root para gerenciar os contêineres do sistema, sendo que o processo de criação e migração dos ambientes virtuais utilizam essas aplicações. Entre as aplicações de gerência, as únicas que precisaram ser modificadas para utilizar as novas funcionalidades foram o vzctl e o vzmigrate,

O vzctl é a aplicação que realiza a interface entre o úsuario e o sistema de gerencia, tendo recebido desta forma as modificações necessárias para criar o servidor e o cliente de migração. O vzmigrate foi alterado para passar a utilizar as novas funcionalidades do vzctl.

```
    rdtsc(lo, hi);
    env->vTSC.v_counter_base.hi = hi;
    env->vTSC.v_counter_base.lo = lo;

    result_2 = (uint64_t)hi;
    result_2 = result_2 [illegible] + (uint64_t)lo;
    tempPrev = do_div(result_2, tsc_khz);
    result_2 = result_2 * [illegible];
    tempPrev = tempPrev * [illegible];
    lo = do_div(tempPrev, tsc_khz);
    result_2 = result_2 + tempPrev;

    result = result + (result_2 - result_1);
    tempPrev = do_div(result, [illegible]);

    if (err < 0)
    {
        printk("client_vtsc ERROR - sock_recvmsg(%d).  ", err );
        return -1;
    }
#ifdef DEBUG_DIEGO
    printk("Msg recebida com sucesso!(%d)  ", erro );
#endif


    env->vTSC.runnable_time_ns = mensagem.runnable_time_ns;
    env->vTSC.suspended_time_ns = mensagem.suspended_time_ns + result;
    env->vTSC.id = mensagem.id;

    return 0;
```

Figura 4.13: Trecho final do código executado pelo cliente vTSC no Destino

4.3 Implementando o vTSC no Linux

A implementação no *kernel* do Linux tradicional busca avaliar como vTSC pode ser utilizado para tratar os problemas causados pelos mecanismos de escalonamento de freqüência e migração de núcleo nas aplicações que desejam aferir o tempo. Diferente do caso do OpenVZ (Seção 4.2) na implementação para Linux a rede de comunicação é confiável e não existe migração entre nós computacionais, apenas entre núcleos de processamento em um mesmo nó.

4.3.1 Arquitetura Básica

Esta seção busca descrever os componentes computacionais sobre os quais o vTSC foi implementado, explicando quando necessário as simplificações adotadas. No processo de implementação, parte das premissas que axiliaram a descrição do vTSC não são encontradas, sendo necessário criá-las quando for o caso.

Arquitetura do Processador

A arquitetura de processador que dá suporte a esta versão do vTSC é a da família x86, mais especificamente a de 32 bits, com suporte no nível do sistema operacional para o escalonamento de freqüência e múltiplos núcleos de processamento. O suporte em software para execução da instrução RDTSC foi implementando utilizando o mesmo tipo de dado abstrato (Figura 4.1) da implementação OpenVZ.

Devido à nova organização interna do *kernel* do Linux, a implementação para processadores com um ou mais núcleos pode ser feita de forma quase idêntica. A única diferença é que para o suporte a multiplos núcleos, o protocolo de migração do vTSC é implementado.

4.3.2 Integração do vTSC com o Linux

As funcionalidades oferecidas pelo uso do vTSC no *kernel* do Linux, fizeram a implementação ser dividida em duas fases. A primeira fase consistiu em implementar dentro do *kernel* propriamente dito as estruturas e controles do vTSC, sendo a segunda fase composta pela implementação nos *drivers* do processador dos serviços necessários para o correto suporte ao procedimento de escalonamento de freqüência para suportar o vTSC.

vTSC no *Kernel* do Linux

A implementação do vTSC foi feita sobre a versão 2.6.27.8 do *kernel* do Linux sendo que como discutido na eção 4.1.2, uma implementação ingênua faria com que a solução tive-se um custo computacional alto para o tratamento do procedimento de escalonamento de freqüência. Desta forma, foi implementada uma versão da estrutura de dados do vTSC para cada núcleo de processamento (Figura 4.14), criada durante a inicialização do escalonador de tarefas e com o controle feito pelo *driver* do processador responsável pelo escalonamento da freqüência. O controle é realizado de forma que sempre que a freqüência for alterada o tempo que o processador encontrava-se trabalhando na freqüência anterior é consolidado e armazenado no campo *T_vTSC_Base_Per_CPU.onLine_counter_ns* e o instante aproximado em que o processador mudou de freqüência em *T_vTSC_Base_Per_CPU.v_counter_base*.

```
typedef struct _T_VTSC_Base_Per_CPU{
  unsigned long id;
  T_TSC v_counter_base;
  unsigned long long onLine_counter_ns;
}T_vTSC_Base_Per_CPU;
```

Figura 4.14: Tipo de Dado Abstrato do vTSC para CPU implementado no Linux *T_vTSC_Base_Per_CPU*

A implementação do vTSC foi feita sobre o kernel do Linux versão 2.6.27.8, desta forma, a estrutura de dados que suporta o vTSC nesta plataforma (Figura 4.15) estende a descrição abstrata apresentada no eção 4.1 passando a contar também o tempo em que a aplicação esteve parada. O TAD do vTSC foi incorporado a estrutura **struct** *task_struct* que armazena todas as informações do processo necessárias pelo escalonador de tarefas.

```
typedef struct _T_VTSC{
  unsigned long id;
  unsigned long long counter_base;
  unsigned long long runnable_time_ns;
  unsigned long long suspended_time_ns;
}T_vTSC;
```

Figura 4.15: Tipo de Dado Abstrato do vTSC para o processo implementado no Linux *T_vTSC*

A estrutura de dados da Figura 4.15 é inicializada no instante em que o processo é criado, com os campos *runnable_time_ns*, *suspended_time_ns* inicializados com zero e o campo *counter_base* com o valor corrente do campo *onLine_counter_ns* vinculado ao processador onde o processo foi criado. A chamada de sistema *vtsc_gettime_ns*() descrita no Seção 4.1 passa a se chamar *tsk_get_runtime*() (Figura 4.16).

Mudança no *driver* de escalonamento de freqüência para dar suporte ao vTSC

O suporte ao escalonamento de freqüência dos processadores é implementado pelo vTSC utilizando a estrutara de dados *T_vTSC_Base_Per_CPU* (Figura 4.14) A atualização dos valores armazenados nesta estrutura é realizada pelo *driver* do processador, que deve implementar os conceitos descritos na Seção 4.1.2 em especial os da Figura 4.7.

```
asmlinkage long sys_tsk_get_runtime(unsigned long long __user *p_runt){
    int  err =  ;
    struct task_struct *tsk;
    unsigned long long result;
    unsigned long long tempPrev;
    unsigned long flags;
    struct rq *rq;

    tsk = current;
    rq = task_rq_lock(tsk, &flags);

    if(p_runt){
        tempPrev = get_rq_vtsc_time(rq);
        result = tempPrev - (uint64_t)tsk->vTSC.counter_base;
        result = tsk->vTSC.runnable_time_ns + result;
        err = copy_to_user(p_runt, &result, sizeof(unsigned long long));
    }
    task_rq_unlock(rq, &flags);

    return err ? -EFAULT :  ;
}
```

Figura 4.16: Chamada de sistema para recuperar o tempo de execução no Linux

A forma como o escalonamento de freqüências afeta a instrução RDTSC depende da arquitetura do processador, por exemplo, nas linhas Core e Core2 da Intel a mudança de freqüência do processador não afeta a taxa de incremento do registrador, enquanto que para os processadores da linha Athlon X2 da AMD esta taxa é modificada.

O controle implementado no drivers dos processadores AMD atualizam a estrutura sempre que um pedido de mudança de freqüência é recebido pelo sistema, desta forma, imediatamente antes de alterar a freqüência do núcleo o valor do TSC é lido e armazenado, sendo que o tempo é consolidado imediatamente após a mudança de freqüência for realizada.

Migração de processos no Linux com suporte ao vTSC

O procedimento de migração do vTSC no Linux é utilizado sempre que uma aplicação é migrada entre os núcleos de processamento do computador. Esse procedimento é realizado pelo escalonador de tarefas, com a inclusão do vTSC. Sempre que uma aplicação for migrada entre núcleos o escalonador deve executar o algoritmo descrito na Seção 4.1.2. A Figura 4.17 apresenta em alto-nível como foi implementado o procedimento de migração do vTSC para o Linux.

O algoritmo de migração integrado (Figura 4.17) utilizado neste implementação é descrito a seguir:

1. Seja $P0$ um processo que encontra-se em uma das fila de processos do *Core A*

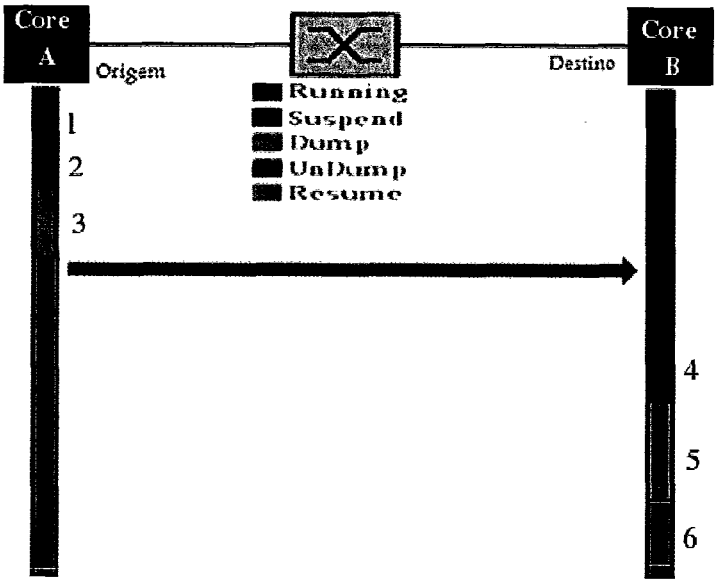


Figura 4.17: Migração de processos no Linux com as modificações feitas para o suporte ao vTSC

e que será migrado para o *Core B*.

2. Durante a etapa de *Suspend* o escalonador do Linux verifica se $P0$ era o processo corrente no núcleo e em caso afirmativo, suspende sua execução. No caso de $P0$ não estar executando e não ser o processo ativo ele é retirado da fila de execução para que não possa ser mais executado no *Core A*. A etapa de *Suspend* é responsável por consolidar o tempo de execução.

3. O procedimento de *Dump* realizado no escalonador do Linux consiste em enviar o processo migrado para uma nova CPU.

4. O código do escalonador ligado ao Core B durante o procedimento de *UnDump* atualiza de $vTSC.counter\_base$ para o processo $P0$ e atualiza o tempo que ele este suspenso.

5. A etapa de *Resume* consiste em colocar em ativar o processo $P0$ caso ele estivesse na fila de execução do núcleo de origem ele é então colocado na fila de execução do *Core B*.

6. O $P0$ encontra-se na fila de processos na fila de processos experando para serem executados do *Core B*.

vTSC em Nível de Usuário no Linux

A implementação do Linux, ao contrario da realizada no OpenVZ, não necessita de modificações nas aplicações de gerenciamento que executam a nível de usuário, sendo necessário somente a criação de novas chamadas de sistemas para as aplicações terem acesso aos valores do vTSC.

Capítulo 5

Validação Experimental

No capítulo anterior, o vTSC foi proposto e suas implementações nos sistemas OpenVZ e Linux descritas. Neste capítulo, são apresentados os resultados experimentais quantitativos gerados durante o desenvolvimento e avaliação do vTSC. Os experimentos realizados servem para avaliar comparativamente o vTSC com o RDTSC e isoladamente onde a instrução RDTSC não pode ser utilizada.

5.1 Ambientes de Testes

Para avaliação empírica do uso do vTSC, foram construídos dois ambientes experimentais, um *cluster* para validar as mudanças necessárias no OpenVZ e um nó isolado para avaliar os mecanismos implementados no Linux.

5.1.1 Ambiente Experimental OpenVZ

O ambiente experimental utilizado na validação do vTSC para OpenVZ é composto por cinco nós computacionais IBM xSeries 206m, interconectados por um *switch Gigabit Ethernet*, cujas características dos nós encontram-se descritas na Tabela 5.1 (com o tipo do compontente na primeira coluna e o modelo utilizado na segunda). Os nós executam o sistema operacional descrito a seguir:

- Sistema Operacional: CentOS 5.1;
- Versão do *kernel* do Linux : 2.6.18;

- Versão do *patch* OpenVZ: 028*stable*057.2.

Tabela 5.1: Nó computacional IBM xSeries 206m

Componente	Modelo
Processador	Intel Pentium D 930 3,0 GHz
Duração Teórica do Ciclo	0,33 ns
Cache L2	2 * 2 MB
FSB	800 MHz
Memória DRAM	2 * 1 GB DDR2
Interface de Rede	10/100/1000 Mbps
Unidade de Disco	80 GB SATA 5400 RPM

A Figura 5.1 apresenta a topologia de rede adotada neste ambiente, onde os nós do *cluster* foram divididos em quatro nós computacionais e um *front-end*. O *front-end* ficou responsável por centralizar o acesso à rede externa utilizando uma segunda placa de rede *gigabit*, além de executar serviços como NTP e NFS. Sendo que os nos *vzware-1* e *vzware-2* eram os que funcionavam como origem e destino para migrações. Os nós *vzware-3* e *vzware-4* apenas foram utilizados nos experimentos com o GloVE.

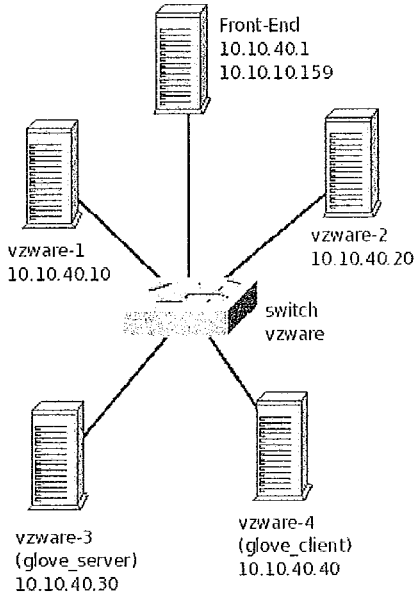


Figura 5.1: Ambiente Experimental OpenVZ para avaliar o vTSC

5.1.2 Ambiente Experimental Linux

O ambiente experimental utilizado na validação do vTSC para Linux é composto por um nó computacional IBM xSeries 206m (Tabela 5.1) em sua etapa de desenvolvimento e um *notebook* HP Pavilion dv2807nr (Tabela 5.2, onde o tipo de componete encontra-se na primeira coluna e o modelo na segunda) para realizar os experimentos de mudança de freqüência do processador. Sendo que em ambas as configurações foi utilizado o sistema operacional descrito a seguir:

- Sistema Operacional: Ubuntu 8.10;
- Linux *kernel*: 2.6.27.2.

Tabela 5.2: *Notebook* HP Pavilion dv2807nr

Componente	Modelo
Processador	AMD Turion 64 `bits` X2 *Dual-Core* TL-60 2,00 GHz
Duração Teórica do Ciclo	0,5 ns
Cache L2	$2 * 512$ KB
Memória DRAM	3 GB DDR2
Unidade de Disco	160 GB SATA 5400 RPM

5.2 Avaliação da implementação do vTSC para OpenVZ

Nesta seção, são avaliados os resultados obtidos para implementação do vTSC no OpenVZ, sendo que o ambiente experimental utilizado encontra-se descrito na subseção 5.1.1. Para simplicidade da leitura, os termos intervalo de tempo e intervalo utilizados equivalentemente.

5.2.1 Avaliando as limitações do NTP no *cluster* local

Dentre as motivações deste trabalho encontra-se as limitações do uso do NTP como mecanismo de sincronização para aplicações de tempo-real em um *cluster* de computadores compartilhados, descritas por Muir [33]. Apesar da informação qualitativa apresentada no referido artigo, optou-se por avaliar como o NTP funciona no *cluster* utilizado na avaliação da implementação para OpenVZ do vTSC, ou seja qual o menor intervalo de tempo que uma aplicação de tempo-real pode medir utilizando o NTP como mecanismo de sincronização. O experimento foi projetado para avaliar quanto apenas o atraso na execução do serviço do NTP pode degradar a medida temporal para uma aplicação, dado que em um sistema real a degradação esperada é ainda maior devido ao tempo gasto pelas interrupções, e de contenção na rede e nos barramentos internos do nó.

Para o experimento foram construídos três cenários distintos, onde variou-se o intervalo de tempo entre as execuções do cliente NTP e cada cenário foi repetido 200 vezes, onde cada repetição realizando 20 execuções do serviço NTP. Em todos os cenários é importante ressaltar que, ao contrario da situação descrita no referido artigo, os nós computacionais e a rede encontravam-se inteiramente disponíveis para o NTP, não ocorrendo contenções devido a sobrecarga do nó ou da rede.

Inicialmente, o experimento foi realizado utilizando um intervalo de 0 s entre as execuções do serviço do NTP, esse cenário representa a situação ideal, onde o cliente do NTP é executado consecutivamente sem executar a chamada de sistema nanosleep() entre as execuções. O segundo cenário utilizou um intervalo de 10 s, ou seja a aplicação esperava 10 s entre as execução do cliente NTP, este cenário representa o funcionamento do protocolo NTP em nós computacionais que não encontram-se sobrecarregados. O terceiro e último cenário utilizou um intervalo de tempo entre as execuções de 600 s, sendo que o intervalo foi escolhido por representar o perfil de execução do NTP em nós computacionais que encontram-se sobrecarregados [33].

A Tabela 5.3 sumariza os resultados obtidos para o experimento onde na primeira coluna tem-se o cenário testado, na segundo coluna o intervalo entre execuções do *deamon* do NTP, na terceira a média dos valores dos Δ's aplicados sobre o relógio local e na quarta coluna o desvio padrão associado a média.

Uma aplicação de tempo-real, cujas tarefas devam ocorrer em intervalos de tempo iguais ou menores que 1 ms não executará corretamente sobre esse *cluster*, pois

Tabela 5.3: Estatísticas dos Δs aplicados pelo protocolo NTP sobre o relógio local

Cenário	Intervalo entre execuções (s)	Média (μs)	Desvio Padrão
1°	0 s	503,17	111,97
2°	10 s	660,11	235,27
3°	600 s	10.169,35	651,26

duas atualizações sucessivas de 503,17 μs fazem com que o tempo medido pela aplicação seja superior a 1 ms. Durante os experimentos o maior Δ aplicado sobre o valor corrente do relógio local realizado pelo NTP foi de 1,879 s. Com base nestes resultados é possível afirmar que os problemas descritos por Muir continuam a ocorrer em sistemas computacionais que utilizam o NTP como mecanismo de suporte as aplicações sensíveis ao tempo.

Os resultados obtidos (Tabela 5.3) demonstraram que mesmo sem a degradação provocada pela perda de interrupções em nós sobrecarregados as atualizações feitas pelo protocolo NTP o tornam uma solução arriscada para sistemas de tempo-real.

5.2.2 Validação o funcionamento da Instrução RDTSC para os experimentos OpenVZ

Antes de se iniciar a validação do vTSC é necessário verificar se o ambiente experimental provê o suporte necessário. Mo caso do OpenVZ é necessário garantir que a freqüência dos núcleos seja constante isso pode ser feito executando o comando com aparece na Figura 5.2. Se a resposta for similar apenas variando o número de núcleos o TSC é estável e o nó pode ser utilizado nos experimentos do vTSC.

5.2.3 Experimentos Quantitativos

Nesta subseção, encontram-se detalhados os experimentos quantitativos realizados de forma verificar o comportamento do vTSC no ambiente OpenVZ. Eles foram criados para verificar a corretude das novas chamadas de sistemas, do funcionamento do mecanismo de migração modificados do OpenVZ para suportar o vTSC e o custo da utilização das chamadas do vTSC relativo ao tempo de execução do programa.

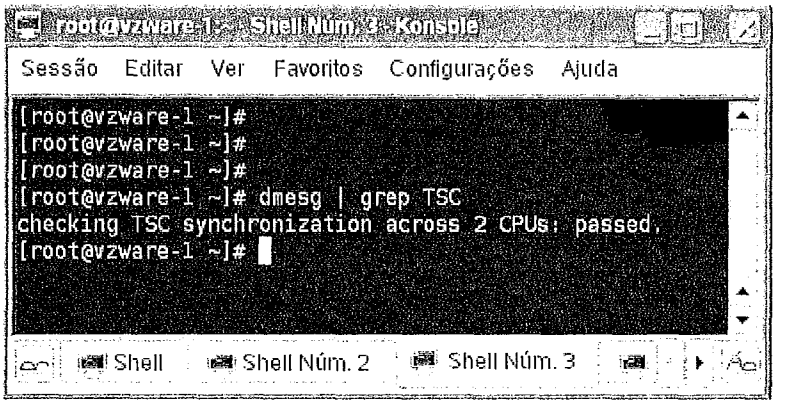


Figura 5.2: Testa para verificar uso da instrução RDTSC para seu uso na implementação OpenVZ do vTSC

Avaliação do Funcionamento da Chamada *ve_get_runtime*()

Este experimento foi realizado para verificar se sucessivas chamadas a função *ve_get_runtime*() por uma aplicação poderia violar a propriedade de crescimento monotônico devido a algum erro de implementação. Uma repetição do experimento era composta pela criação do contêiner e execução da aplicação de teste que realiza 100 chamadas a *ve_get_runtime*().

A aplicação era abortada caso alguma das execuções das chamadas a *ve_get_runtime*() retornasse um valor igual ou menor a chamada anterior. Foram criados cinco cenários, onde em cada um foi a chamada *ve_get_runtime*() executada por 100 vezes e onde foi variado o intervalo de tempo entre as chamadas *ve_get_runtime*(), de acordo com os valores na Tabela 5.4 (na primeira coluna encontra-se o número do cenário e na segunda o intervalo de tempo utilizado entre as chamadas a *ve_get_runtime*()). A avaliação consistiu em executar cada um dos cenários 10 vezes, onde a aplicação abortava se em alguma interação a propriedade de crescimento monotônico fosse violada.

O 1° cenário possui um intervalo de tempo nulo entre as chamadas de *ve_get_runtime*() funcionou como um teste de *stress*, buscando averiguar se sob uma condição extrema o vTSC funcionaria corretamente, enquanto os outros valores representam, aplicações de tempo-real tradicionais. As figuras a seguir apresentam no eixo Y o tempo de execução médio e no eixo X o número da interação em que ela

Tabela 5.4: Cenários utilizados na avaliação da chamada de sistema *ve_get_runtime*()

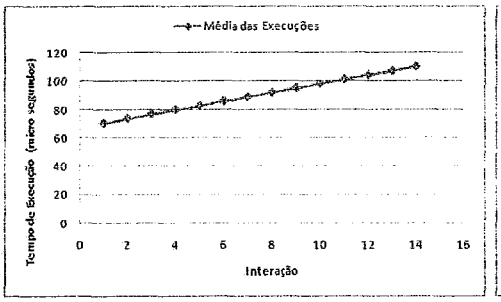
Cenário	Duração do Intervalo
1°	0 ms
2°	1 ms
3°	10 ms
4°	100 ms
5°	1000 ms

foi medido.

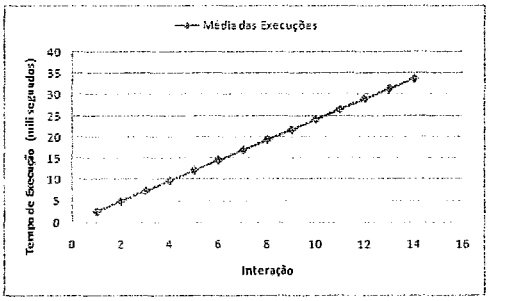
A avaliação obteve sucesso para todos os cenários, como mostrado na Figura 5.3 através dos resultados obtidos para as quinze primeiras interações do experimento. Como se observa pelo gráfico os valores obtidos utilizando o vTSC para um mesmo cenário são sempre crescentes. Note que a Figura 5.3(d) apresenta alterações no valor da derivada, porém mantém-se sempre positiva. Essa alteração ocorre devido ao tamanho do intervalo utilizado na função *nanosleep*(), revelando um comportamento cíclico, possivelmente devido RTC utilizado pelo sistema operacional para chavear os processos.

A Tabela 5.5 apresenta em sua primeira coluna o número da interação em que o valor foi medido, na segunda coluna a média para do tempo de execução, na terceira coluna o desvio padrão e na quarta o intervalo de confiança associado. Esses são os utilizados para construir o gráfico (Figura 5.3(c)), com os desvio padrão e coeficiente de variação para cada uma das 14 primeiras interações. A variação observada na coluna do coeficiente de variação deve-se ao uso da chamada de sistema *nanosleep*() para controlar os intervalos, de modo que o intervalo de suspensão da aplicação seja sempre maior que o requerido. Esta variação depende de fatores como a carga da máquina e as interrupções que chegam ao núcleo do processador.

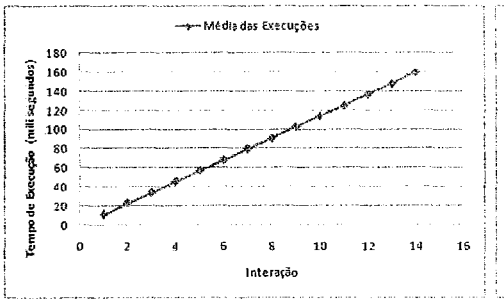
Além do efeito causado nos experimentos devido ao uso da chamada de sistema *nanosleep*(), procedimento de chaveamento entre processos realizado pelo *kernel* do Linux também influenciou alguns cenários deste experimento. A Tabela 5.6, apresenta os valores da média (segunda coluna), desvio padrão (terceira coluna) e intervalo de confiança (quarta coluna) para a duração do intervalo calculado utilizando-se



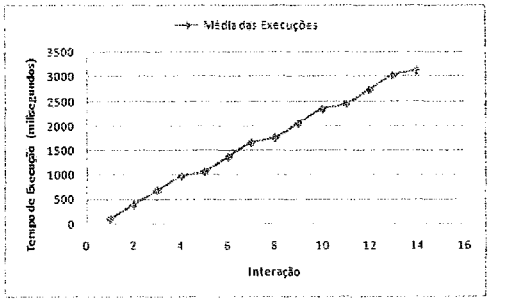
(a) Intervalo de $0ms$



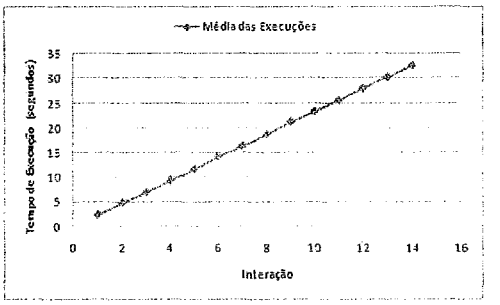
(b) Intervalo de $1ms$



(c) Intervalo de $10ms$



(d) Intervalo de $100ms$



(e) Intervalo de $1000ms$

Figura 5.3: Avaliação da Propriedade de crescimento Monotônico do vTSC

Tabela 5.5: Resultados do tempo médio e desvio padrão da execução do 3° cenário, com intervalo igual a 10 ms

Interação	Média (ms)	Desvio Padrão (ms)	Coeficiente de Variação
1	11,50	0,53	0,046
2	22,89	0,99	0,043
3	34,28	1,49	0,043
4	45,69	2,00	0,043
5	57,08	2,51	0,044
6	68,48	3,03	0,044
7	79,88	3,54	0,044
8	91,28	4,06	0,044
9	102,67	4,06	0,045
10	114,07	5,09	0,045
11	125,47	5,60	0,045
12	136,87	6,12	0,045
13	148,26	6,63	0,045
14	159,76	7,07	0,045

Tabela 5.6: Medidas de disperção para os cinco cenários utilizados na avaliação da chamada de sistema *ve_get_runtime*()

Cenário	Média	Desvio padrão	Intervalo de Confiança de 95%
1°	3,06 μs	1,19 μs	0,02
2°	1,99 ms	0,49 ms	0,01
3°	11,99 ms	148,36 ms	2,91
4°	101,97 ms	482,24 ms	9,47
5°	2,90 s	0,87 s	0,02

os mecanismos oferecidos pelo vTSC. Como é possível observar o 1° cenário não sofre grandes variações em sua execução, principalmente foi devido a não ter sido feita a chamada o que acabou por evitar as instruções vinculadas a entrada dentro do *kernel* do Linux e do escalonador de tarefas. O 2° cenário (Figura 5.3(b)), apresentou uma variação absoluta duas ordens de grandeza acima da encontrada no 1° cenário (Figura 5.3(a)) esse efeito deu-se principalmente devido ao uso da chamada

nanosleep(), sendo importante notar que o desvio padrão e o intervalo de confiança foram menores se comparados com os do 1° cenário.

O 3° e 4° cenários (Figura 5.3(c) e Figura 5.3(d) respectivamente), como pode ser observado na Tabela 5.6, tiveram um aumento significativo com relação ao desvio padrão e desta forma obtiveram um intervalo de confiança muito superior ao cenários anteriores (Figura 5.3(d) apresenta graficamente esse efeito periódico), esse resultado foi causado principalmente pela duração do tempo de processamento ou *quantum* que o Linux oferece para cada processo no sistema, ou seja devido a duração da chamada *nanosleep*() ter sido grande o suficiente o processo que encontrava-se executando estourava seu quantum de processamento, diversas vezes antes que o tempo fosse contabilizado fazendo com que a duração efetiva da *nanosleep*() fosse muito superior a requisitada. Esse efeito não foi percebido no 5° cenário devido a duração da chamada *nanosleep*() ter sido muito superior ao tamanho do quantum de processamento.

Custo do uso de vTSC para as Aplicações no Nível do Usuário

Como vTSC foi proposto como um substituto para a instrução RDTSC e para a chamada *gettimeofday*() para sistemas onde ocorram migrações, é necessário avaliar o custo relativo da nova solução. Com esse objetivo, foi construído um *microbenchmark* (Figura 5.4) para verificar quanto o uso do vTSC para aferir o tempo acrescenta à duração total da aplicação. Todos os experimentos utilizando foram executados por 8000 vezes, onde o tempo que cada execução levou é calculado utilizando duas instruções RDTSC. O *microbenchmark* foi construído para aproximar os custos das diferentes técnicas de temporização testadas. Durante os experimentos, a precisão oferecida pela instrução RDTSC foi de 1 ns. As métricas utilizadas foram a média, intervalo de confiança de 95% e o desvio padrão.

A etapa de Inicialização é executada a cada interação do *microbenchmark*, sendo esta a etapa onde a freqüência do processador é inferida, para ser posteriormente utilizada nas medidas. Os blocos (Bloco1 e Bloco2) limitados pela instrução RDTSC executam um laço vazio de 32.768 interações cada um. Finalmente, a etapa de consolidação calcula o tempo médio que a execução levou. Essa estrutura foi cons-

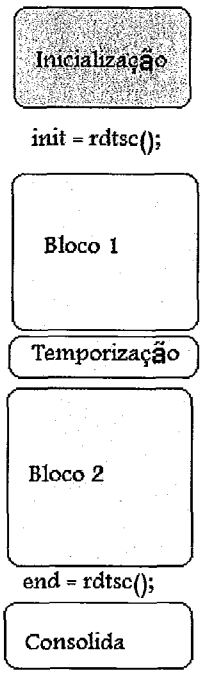


Figura 5.4: Descrição do *microbenchmark* utilizado nos experimentos

truída para medir o custo adicional quando bloco de temporização é incluído entre as instruções de um programa.

A Tabela 5.7 apresenta em sua primeira coluna o cenário utilizado no experimento, na segunda coluna o tempo médio de execução para cada um dos cenários e na terceira coluna o desvio padrão das 8000 execuções para cada um dos cenários.

Tabela 5.7: Custo médio da temporização para o *microbenchmark* nos quatro cenários estudados

Cenário	Tempo Médio	Desvio Padrão
1° Cenário (Sem Temporização)	176,27 ns	51,63 ns
2° Cenário (RDTSC)	338,38 ns	10,82 ns
3° Cenário (*gettimeofday*())	6.690,54 ns	201,72 ns
4° Cenário (vTSC)	2.134,44 ns	74,69 ns

O primeiro cenário aferire o tempo de execução médio e desvio padrão do *microbenchmark* sem executar temporização. A linha 1° Cenário da Tabela 5.7 mostra que o tempo de execução médio foi de 176,273 ns com um desvio padrão de 51,628 ns.

O segundo cenário utiliza uma instrução RDTSC e uma multiplicação (64 *bits*) na

etapa de Temporização. Observando-se a linha 2° Cenário (RDTSC) na Tabela 5.7 verificar-se que o tempo médio de processamento foi de 333,84 ns. O acrescimo no tempo de processamento no 2° Cenário em comparação com o primeiro, deu-se devido ao temporização realizada, desta forma os proximos cenários são comparados apenas com o 2° Cenário.

Entretanto, como visto anteriormente, o RDTSC não pode ser utilizado em ambientes onde ocorram migrações, desta forma, se uma aplicação precisar aferir o tempo ela o deve fazer utilizando a chamada de sistema *gettimeofday*(). Assim sendo é importante avaliar os custos que o uso desta chamada tem sobre o tempo de execução total das aplicações. Para isso, substitui-se a etapa de Temporização por esta chamada de sistema. Como pode ser observado na linha 3° Cenário(*gettimeofday*()) da Tabela 5.7, tanto o tempo médio de execução de 6690,54 ns como o desvio padrão 201,72 ns, foram quase 20 vezes maiores que os resultados obtidos pela instrução RDTSC. Isto ocorre devido ao fato de que a utilização de uma chamada de sistema implica na execução de um número maior de instruções, além do chaveamento do processador para o modo núcleo e retardos causados por possíveis contenções dentro do *kernel*.

O experimento que avaliam o custo do vTSC foi realizado com o uso de uma chamada de sistema construída para o vTSC no OpenVZ na etapa de Temporização. O cenário utilizado foi o mesmo dos experimentos anteriores. A linha 4° Cenário (vTSC) na Tabela 5.7 mostra que o tempo médio de execução foi de 2134,44 ns, 6,3 vezes maior que o obtido utilizando-se o RDTSC e 3,14 vezes mais rápido do que o *gettimeofday*().

A Tabela 5.8, apresenta os Intervalos de Confiança de 95% (segunda coluna), obtidos para o experimento utilizando o *microbenchmark* nos quatro cenários estudados (primeira coluna). Sendo possível observar que as instruções extras executadas pela chamada *gettimeofday*() fizeram com que seu intervalo de confiança fosse o maior dos quatro cenários.

Em resumo, o que é importante observar é que o vTSC não tem o problema de temporização do RDTSC e seu custo de execução é relativamente menor do que o obtido quando se utiliza a chamada de sistema *gettimeofday*().

Tabela 5.8: Interavalo de Confiança para *microbenchmark* nos quatro cenários estudados

Cenário	Intervalo de Confiança de 95%
1° Cenário (Sem Temporização)	1,13 ns
2° Cenário (RDTSC)	0,23 ns
3° Cenário (*gettimeofday*())	4,42 ns
4° Cenário (vTSC)	1,63 ns

Migração do vTSC

O objetivo deste experimento foi avaliar o funcionamento do mecanismo de migração do vTSC que foi incluído dentro da migração realizada pelo OpenVZ. Desta forma, o experimento consistiu em construir uma aplicação que realiza as chamadas de sistemas *ve_get_runtime*() e *ve_get_suspendtime*(), nesta ordem dentro de um laço durante 100 interações. No primeiro cenário, o intervalo entre as interações foi de dois segundos e os nós computacionais não eram reiniciados entre as migrações. O segundo cenário utilizou um intervalo entre interações de 5 segundos e o nó de origem era sempre reiniciado após uma migração. Em ambos os cenários o contêiner onde a aplicação encontrava-se executando era migrado entre os nós. Sendo que a aplicação terminava com erro se em alguma interação o valor retornado por *ve_get_runtime*() fosse maior que o retornado por *ve_get_suspendtime*() ou se uma dessas chamadas retornasse um valor inferior ao retornado na interação anterior pela mesma chamada.

O experimento foi repetido 200 vezes para cada um dos cenários, sendo que em cada execução foram realizadas de três a cinco migrações. Ambos os cenários terminaram com sucesso. Como o nó de origem eram sempre reiniciados após uma migração o seu valor do TSC quando ele funcionar como nó de destino na proxima migração será obrigatoriamente menor que o do nó para onde o contêiner foi migrado.

Em resumo, esse experimento foi realizado para validar o funcionamento das alterações realizadas no protocolo de migração do OpenVZ para o suporte ao vTSC. Desta forma, pode-se concluir com base neste experimento que o vTSC mantém a aplicação sincronizada após a migração entre nós físicos, seja para apenas uma ou para múltiplas migrações.

Migração do Distribuidor GloVE

Este foi o experimento que motivou o desenvolvimento do mecanismo de virtualização vTSC, sendo desta forma utilizado para validar essa solução, o vTSC em um sistema construído sobre a instrução RDTSC. O GloVE teve de ser modificado para passar a realizar a chamada de sistema *ve_get_suspendtime*() em vez da instrução RDTSC. Foi utilizada a *ve_get_suspendtime*() e não *ve_get_runtime*() devido à necessidade da aplicação conhecer o tempo de parede (*wall-time*) e não apenas o tempo em execução do contêiner.

Os experimentos utilizando GloVE consistiram em iniciar o distribuidor dentro de um contêiner do OpenVZ, e uma vez que um cliente estivesse assistindo o vídeo, o contêiner seria migrado para outro nó físico. Sendo que, se o tempo total de migração fosse menor que a quantidade (duração) de vídeo no *buffer* do cliente, este não deveria sofrer degradações no serviço devido à migração. As migrações foram realizadas sempre que o buffer do cliente encontrava-se quase cheio e após a migração o nó de origem se reiniciava para poder zerar seu TSC e com isso garantir que o mecanismo de migração do vTSC funcionasse corretamente.

O vídeo utilizado foi a Máscara do Zorro com uma taxa de exibição de 411,7 Kbps, onde para cada exibição do filme foram realizadas 10 migrações entre os nós vzware-1 e vzware-2. Desta forma, o experimento é composto por 20 exibições do vídeo, onde em cada exibição foram realizadas 10 migrações. Sendo que uma exibição era concluída com sucesso, se e apenas se, para as 10 migrações realizadas um cliente estivesse assistindo o vídeo não sofresse perdas de qualidade, ou seja não parasse de receber o fluxo de vídeo.

Foram testados dois cenários neste experimento. No primeiro, o contêiner era apenas migrado entre os nós físicos, e no segundo cenário após uma migração o nó de origem era reiniciado. Com o reinício do nó físico o valor do TSC (global no Pentium *Dual Core*) do nó que recebeu o contêiner era obrigatoriamente sempre menor que o do nó de origem. Em ambos os cenários descritos o experimento começava aproximadamente dez minutos após o cliente ter começado assistir o filme e o intervalo entre migrações foi de aproximadamente um minuto para o primeiro cenário e cinco minutos no segundo cenário. Em nenhuma das 20 exibições realizadas um cliente precisou realizar outra requisição para assistir o filme. Isto confirmou

que o tempo de migração foi suficientemente curto e que o vTSC teve sucesso em mantar a sincronização da aplicação. O tempo médio de migração foi em média 1,9 s e o desvio padrão 0,014 s, sendo que o tamanho da memória transferiada pelo procedimento de migração foi de aproximadamente 70 MB.

Migração do contêiner com o FileBench

O FileBench [44] é uma aplicação criada para avaliar o desempenho de sistemas de arquivos. O experimento foi composto por três cenários no primeiro, a aplicação não foi modificada, ou seja, funcionava utilizando a instrução RDTSC e nenhuma migração foi realizada. O segundo cenário era similar ao primeiro, porém o contêiner onde o *benchmark* executava era migrado entre os nós. No terceiro cenário a instrução foi substituída pela chamada de sistemas *ve_get_runtime*() do vTSC. Em todos os cenários foram realizadas 100 execuções utilizando o teste de criação de arquivos do *benchmark*, sendo que o tamanho dos arquivos criados era de 778 MB.

O primeiro cenário consistiu na execução da aplicação normal, sendo seus resultados utilizados como base para a comparação entre os dois outros cenários do experimento. A média amostral do tempo de execução obtida para primeiro cenário foi de 7,02 s, com um desvio padrão de 0,65 s. Como esperado, o segundo cenário não concluiu com sucesso em nenhuma das execuções realizadas, com a aplicação abortando durante a execução devido às inconsistências temporais encontradas nas medidas. Como não foi possível obter uma saída útil para as execuções não foi possível realizar um estudo com relação à perda de precisão provocada pela migração.

No terceiro cenário a aplicação passou a usar o vTSC, com o contêiner era migrando durante as execuções. A aplicação sempre terminou com sucesso e a duração média para o experimento foi de 8,26 s, com um desvio padrão de 0,71 s.

Migração do contêiner com o Memperf

A aplicação Memperf [45] avalia o comportamento da hierarquia de memórias, onde para a arquitetura x86 os cálculos temporais são realizados utilizando-se a instruções RDTSC. Note que o Memperf exlui valores medidos que sejam 75% menores que o valor médio. O primeiro cenário consistiu em realizar as execuções sem que o contêiner

fosse migrado. O segundo cenário, por sua vez, consistiu em migrar contêiner onde a aplicação executava entre os nós computacionais sem que esta fosse modificada (ainda utilizando a instrução RDTSC), No terceiro cenário, a aplicação foi modificada para usar a chamada de sistema *ve_get_runtime*(). O experimento compreendeu em 20 execuções com 100 interações para cada um dos cenários, onde o contêiner era migrado duas vezes durante a execução.

Devido ao efeito de memória cache, cada execução realizada, foi precedida por 1000 instruções NOP. O experimento teve início com a execução do primeiro cenário, onde não houve migrações, permitindo avaliar como o processo de migração afeta a aplicação. Nesta pré-execução, o número de valores excluidos que eram menores que 75% da média experimental foi de 5, 1, com um desvio padrão de 0, 79. A Tabela 5.9 apresenta o cenário utilizado no experimento (primeira coluna) e número de valores exluidos (segunda coluna). O segundo cenário quando comparado com o primeiro cenário tem um crescimento de mais de dez (10, 418) vezes os valores excluidos. Isso deve-se ao fato de que os valores correntes dos registradores TSC nos computadores não estarem sincronizados. Enquanto que para o terceiro cenário a sincronização.

Tabela 5.9: Experimento memperf

Cenário	Média de Valores Exlcuídos	Desvio Padrão
1° Cenário (RDTSC)	5, 10	0, 79
2° Cenário (RDTSC + Migrações)	53, 15	9, 16
3° Cenário (vTSC + Migrações)	5, 15	0, 99

5.3 Avaliação da implementação do vTSC para Linux

Nesta seção, são avaliados os resultados obtidos para implementação do vTSC no *kernel* do Linux, sendo que o ambiente experimental utilizado encontra-se descrito anteriormente na seção 5.1.2.

5.3.1 Validação o funcionamento da Instrução RDTSC para os experimentos Linux

Antes de se iniciar a validação do vTSC é necessário verificar se o ambiente experimental provê o suporte necessário. Para os experimentos utilizando a implementação Linux é necessário garantir que a freqüência dos núcleos possa ser ajustada individualmente. A Figura 5.5 apresenta o resultado da execução, onde é possível perceber que a mudança na freqüência de trabalho dos núcleos efetivamente altera a taxa de atualização do registrador TSC apresentada no campo *Frequency*.

```
root@dutranote:/tmp/vzwl_vtsc# cpufreq-selector -c 0 -f 1000000
root@dutranote:/tmp/vzwl_vtsc# cpufreq-selector -c 1 -f 1000000
root@dutranote:/tmp/vzwl_vtsc# ./exp_base
Frequency: 1002124993,000000
Duration: 5,340192E-05
```

(a) Intervalo de $0ms$

```
root@dutranote:/tmp/vzwl_vtsc# cpufreq-selector -c 0 -f 2000000
root@dutranote:/tmp/vzwl_vtsc# cpufreq-selector -c 1 -f 2000000
root@dutranote:/tmp/vzwl_vtsc# ./exp_base
Frequency: 2004213758,000000
Duration: 2,717716E-05
```

(b) Intervalo de $1ms$

Figura 5.5: Execução do *microbenchmark* no ambiente experimental Linux

5.3.2 Experimentos Quantitativos

Nesta seção, encontram-se detalhados os experimentos quantitativos realizados de forma a verificar o comportamento do vTSC no ambiente Linux. Eles foram projetados para verificar o suporte oferecido pelo vTSC ao procedimento de escalonamento de freqüências.

FileBench

O experimento utilizando o FileBench foi composto por três cenários; onde no primeiro, a aplicação não foi modificada, ou seja, funcionou utilizando a instrução RDTSC e sem o escalonamento da freqüências. No segundo cenário, foi realizado o escalonamento de freqüência uma vez durante a execução, especificamente na metade do tempo médio obtido no primeiro cenário. O terceiro cenário foi ao segundo,

porém o FileBench foi modificado para utilizar a chamada de *ve_get_runtime*(). O escalonamento de freqüências realizado neste experimento teve como objetivo mostrar como as arquiteturas modernas podem sofrer de problemas de sincronização do tempo, ou seja ao usar a instrução RDTSC a aplicação que deseja medir o tempo não mais consegue obter valores realistas.

O primeiro cenário consistiu na execução normal da aplicação, e os resultados serviram de base de comparação para os dois outros cenários do experimento. A média amostral do tempo de execução obtida para o primeiro cenário foi 10,95 s, com desvio padrão de 0,98 s. O escalonamento de freqüências no segundo cenário foi realizado cinco segundos após o início da execução, com a freqüência reduzida para a metade de seu valor máximo. A média do tempo de execuçãofoi 11,67 s, com desvio padrão de 1,07 s. Durante as execuções do segundo cenário, o aplicativo *date* foi executado imediatamente antes do início e ao término da execução. Isso foi feito para comparar a duração medida pelo FileBench e a duração real do experimento, e como esperado, a duração real foi quase duas vezes (1,53) maior que a reportada pelo *benchmark*. A média do tempo de execução real foi 17,9 s com desvio padrão de 1,96 s.

No terceiro cenário, a aplicação usou o vTSC e cenário o contêiner era migrado durante as execuções, e o tempo de execução médio foi 16,35 s, com desvio padrão de 1,49 s. A Tabela 5.10 apresenta os resultados obtidos pelo experimento, com a primeira coluna representando o cenário executado, a segunda coluna o tempo médio de execução do benchmark e a terceira o desvio padrão associado a média. Sendo que, as duas linhas do segundo cenário representam o tempo medido pelo FileBench e o medido externamente pelo aplicativo *date*.

Tabela 5.10: Experimento FileBench para o Linux

Cenário	Tempo Médio	Desvio Padrão
1° Cenário Cenário RDTSC (FileBench)	10,95 s	0,98 s
2° Cenário Cenário RDTSC (FileBench)	11,67 s	1,07 s
2° Cenário Cenário RDTSC (date)	17,9 s	1,96 s
3° Cenário Cenário vTSC (FileBench)	16,35 s	1,49 s

O intervalo de confiança obtido para esse experimento é apresentando na Ta-

bela 5.11, novamente a primeira coluna contém o cenário que foi executado e a segunda o intervalo de confiança calculado para o experimento. Sendo que, novamente as duas linhas do segundo cenário representam o tempo medido pelo FileBench e o medido externamente pelo aplicativo *date*.

Tabela 5.11: Intervalos de Confiança para o experimento FileBench no Linux

Cenário	Intervalo de Confiança de 95%
1° Cenário Cenário RDTSC (FileBench)	0,017 s
2° Cenário Cenário RDTSC (FileBench)	0,018 s
2° Cenário Cenário RDTSC (date)	0,021 s
3° Cenário Cenário vTSC (FileBench)	0,024 s

5.4 Considerações Finais

Os resultados apresentados neste capítulo demonstra que o uso do vTSC não causa impactos significativos no tempo de execução das aplicações, o que o viabiliza como substituto para o uso da instrução ***ReaD Time Stamp Counter***. Como foi demonstrado, que a utilização do NTP como mecanismo de temporização apresenta fortes limitações com relação ao intervalo de tempo que pode ser medido, corroborando as afirmações de Muir [33].

Os experimentos realizados na implementação para OpenVZ do vTSC demonstram que ele pode ser utilizado como mecanismo para suportar as aplicações que realizam medidas temporais, em especial foram feitos experimentos com aplicativos reais como o sistema de vídeo sob demanda GloVE e os benchmarks; FileBench e memperf.

A implementação para o Linux tradicional do vTSC, teve como objetivo resolver os problemas causados pelo escalonamento de freqüências. Os resultados experimentais para o aplicativo FileBench apresentaram como esses problemas podem ocorrer e demonstraram que o vTSC resolve efetivamente esse tipo de problema.

Em resumo, as contribuições deste capítulo são:

1. Demonstração prática dos problemas discutidos por Muir no artigo *The Seven Deadly Sins of Distributed Systems* [33].

2. Um estudo dos diversos mecanismos de temporização disponíveis na arquitetura x86.

3. Validação da implementação do vTSC para OpenVZ e Linux.

4. Investigação do comportamento das chamadas de sistemas criadas para o vTSC.

5. Demonstração de como o vTSC pode ser utilizado em sistemas de tempo-real.

Capítulo 6

Trabalhos Relacionados

Este capítulo visa relacionar esta dissertação com outros trabalhos relevantes encontrados na literatura, divididos em duas áreas: uma geral - Máquinas Virtuais e outra específica - Virtualização de Relógios em Sistemas Computacionais.

6.1 Máquinas Virtuais

A necessidade de se multiplexar o uso dos recursos computacionais deu origem as máquinas virtuais, hoje utilizadas pelos *datacenters* de grandes empresas e também objeto de investigação no meio acadêmico, principalmente nas áreas de gerência de *datacenters* compartilhados e Grades Computacionais [46, 47], tendo sua origem no final da década de 60 [48].

O grande foco tem sido às técnicas de virtualização total e virtualização parcial (paravirtulização), principalmente nos trabalhos de Grades Computacionais, sendo um exemplo disso o fato de o *middleware* de grades *OurGrid* [49, 50] fornecem uma integração como Xen em uma de suas opções de instalação padrão. Por outro lado, os mecanismos de virtualização no nível do sistema operacional [6] (contêineres) oferecem um custo consideravelmente inferior aos obtidos utilizando os Monitores de Máquinas Virtuais. Um exemplo marcante é o uso do VServer na infra-estrutura do PlanetLab [51] para multiplexar o *hardware* disponível.

O vTSC tem um objetivo ortogonal a esses trabalhos, buscando funcionar como um mecanismo que suporte de forma transparente os procedimentos de migração de máquinas virtuais e escalonamento de freqüência que ocorrrem nestes ambientes

compartilhados. Com o uso do vTSC, um *middleware* de Grades, poderá realizar migrações nas máquinas virtuais, mesmo que as aplicações que nelas executam dependam de medidas temporais precisas.

O uso dos mecanismos de escalonamento de freqüência, possibilita a redução no consumo de energia por parte de grandes *datacenters*, sendo este assunto é tratado em trabalhos como o de Fan [52] e por Lim [53]. Infelizmente aplicações temporizadas não funcionará corretamente devido ao escalonamento de freqüências. Com o uso do vTSC a freqüência de trabalho pode ser modificada sem que isso afete as medidas realizadas pelas aplicações de tempo-real que executam nesses ambientes.

6.2 Virtualização de Relógios em Sistemas Computacionais

O uso de máquinas virtuais levou ao desenvolvimento de soluções para virtualizar os circuitos do hardware responsáveis por aferir o tempo, esta seção trata de alguma dessas técnicas.

6.2.1 Virtualizando a instrução RDTSC

Esta subseção apresenta os mecanismos tradicionalmente utilizados para virtualizar a instrução RDTSC nas arquiteturas x86.

Processadores sem suporte a virtualização

No modelo paravirtualizado para arquitetura x86 a instrução RDTSC não pode ser virtualizada, sendo assim é criada uma chamada ao monitor de máquinas virtuais que substitui a instrução. Essa chamada funciona aplicando um valor delta sobre o resultado da instrução RDTSC, para somente assim disponibilizá-lo.

Processadores com suporte à virtualização

O suporte à virtualização permite que a instrução RDTSC possa ser virtualizada, isso é feito em duas etapas no nível do monitor e no nível do hardware. No nível do hardware o processador implementa uma base que é subtraída do valor corrente

do TSC, sendo posteriormente retornado para o fluxo de execução que realizou a instrução. A execução da instrução é iniciada com a geração de uma interrupção que deve ser tratada pelo monitor de máquina virtual, que é responsável por armazenar o valor base correto para a execução corrente para então dar continuidade a execução da instrução.

6.2.2 Patentes

Esta subseção descreve algumas patentes publicadas que solucionam o problema de virtualizar relógios em computadores digitais.

Scalable Virtual Timer Architecture for Efficiently Implementing Multiple Hardware Timers With Minimal Silicon Overhead

A AMD depositou a patente [54], onde apresenta um suporte em *hardware* para implementação de relógios tipo RTC virtuais. A diferença básica deste trabalho com relação ao vTSC é a construção de um mecanismo de envio de interrupções, que fornece os meios para que o *software* realize a contagem de tempo como se estivesse utilizando o RTC, por sua vez o vTSC não necessita deste tipo de mecanismo pois a aplicação pode a qualquer instante realizar um consulta e descobrir quanto tempo se passou desde a última consulta realizada.

Method and Apparatus for Emulating Multiple Virtual Timers in a Virtual Computer System When The Virtual Timers Fall Behind The Real Time of a Physical Computer System

A VMware depositou a patente [55], onde apresenta a construção um relógio virtual tipo RTC, sendo que este relógio virtual consegue detectar que o tempo que ele tem como atual encontra-se errado, devido, possivelmente a perda de interrupções. Se o relógio virtual encontra-se muito distante do relógio real do *hardware*, entra em execução um mecanismo para que o relógio virtual seja corrigido. Como na patente [54] existe um relógio real que gera interrupções e o próprio relógio virtual gera interrupções para as máquinas virtuais. O vTSC por outro lado, não necessita que sejam geradas interrupções para seu correto funcionamento, o que o torna viável mesmo em ambientes onde não existem interrupções de relógio (*tickless*).

Controlling Virtual Time

A patente [56], depositada pela Microsoft, apresenta um mecanismo para virtualizar os relógios que geram interrupções como o HPET e RTC. O relógio virtual também funciona gerando interrupções de software e como na patente [55] existe um mecanismo para corrigir o valor do relógio virtual quando a diferença entre ele e o real for maior que um limite pré estabelecido.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste capítulo são apresentadas as principais conclusões decorrentes desta dissertação, bem como trabalhos futuros identificados como relevantes.

7.1 Resumo

Esta dissertação propôs, descreveu e implementou o *virtual Time Stamp Counter* um mecanismo em software que mantém consistente a visão da aplicação sobre à passagem do tempo durante sua execução. O vTSC foi concebido como uma camada de *software* que virtualiza o *Time Stamp Counter* (TSC) encontrado nas arquiteturas x86 e que é o mecanismo em *hardware* mais preciso para aferir o tempo nessas arquiteturas.

Foram realizadas duas implementações do vTSC, uma para o OpenVZ e a outra para o Linux. Na implementação OpenVZ todo contêiner possui uma instância do vTSC. Desta forma, quando um contêiner é migrado entre nós computacionais, basta que a instância ligada ao contêiner seja atualizada para que a aplicação que usa o vTSC para aferir o tempo não receba valores inconsistentes. O foco da implementação para OpenVZ foi seu uso em *cluster* de computadores, desta forma foi necessário modificar as ferramentas de gerencia do OpenVZ. A implementação Linux utilizou duas instâncias, uma ligada à fila dos núcleos de processamento e outra para cada processo no sistema. Esta implementação do vTSC teve como objetivo suportar o escalonamento de freqüências disponível nos processadores atuais.

7.2 Conclusões

Com base nos experimentos realizados nesta dissertação pode-se concluir que o vTSC consegue suportar as consultas realizadas para aferir o tempo em computadores digitais sem que com isso incorrer em um acréscimo significativo no tempo de execução da aplicação. A implementação do vTSC para OpenVZ, em especial o experimento utilizando o sistema de vídeo sob demanda GloVE, demonstrou a viabilidade de se utilizá-lo como mecanismo de suporte para aplicações de tempo-real que podem ser migradas entre os nós de um *cluster*. Os experimentos realizados com a implementação para Linux demonstraram como o procedimento de escalonamento de freqüência pode afetar as medidas realizadas pela aplicação. O uso do vTSC consegue resolver este problema sem incorrer em um *overhead* significativo para aplicação.

A implementação do vTSC foi comparada com o uso da instrução RDTSC e o da chamada de sistemas *gettimeofday*(), como forma de se avaliar comparativamente o custo causado pelo uso do vTSC. Como esperado, o vTSC tem um custo maior que RDTSC, aproximadamente 6 vezes, porém, quando comparado com o *gettimeofday*(), vTSC é 3,14 vezes mais rápido. Sendo que o uso de RDTSC é inviável em ambientes onde aconteçam migrações, enquanto o *gettimeofday*() nesses ambientes necessita de auxilio externo (NTP), por sua vez o vTSC não possui estas limitações.

As características do vTSC o tornam um mecanismo ideal para ser implementado em sistemas operacionais *tickless*, pois ele não necessita que interrupções sejam geradas para seu funcionamento, o que oferece ao *kernel* maior flexibilidade com relação aos procedimentos para economia de energia.

7.3 Trabalhos Futuros

Entre os trabalhos futuros para vTSC encontra-se testar mas profundamente o mecanismo de migração para implementação no kernel do Linux, além disso, fica em aberto um estudo sobre quais funcionalidades simples podem ser incluidas em um processador de forma a melhorar o desempenho de vTSC quando comparado a instrução RDTSC.

Outra frente de trabalho importante é migrar outras aplicações que realizam

medições temporais e ver como elas se comportam quando utilizam vTSC, sendo interessante averiguar como vTSC pode auxiliar na redução do consumo em redes de sensores sem que estas percam sua temporização.

Referências Bibliográficas

[1] AMD, *AMD64 Technology AMD64 Architecture Programmers Manual Volume 2*, September 2007.

[2] AMD, *AMD Technical Bulletin - TSC Dual-Core Issue and Utility Fix*, June 2007.

[3] INTEL, *Intel 64 and IA-32 Architectures Software Developers Manual. Volume 2B: Instruction Set Reference, N-Z*, Nov. 2007.

[4] INTEL, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Nov. 2007.

[5] TANENBAUM, A. S., WOODHULL, A. S., *Sistemas Operacionais: Projeto e Implementação*. 3ª ed. Artmed, 2008.

[6] OSMAN, S., SUBHRAVETI, D., SU, G., et al., "The design and implementation of Zap: a system for migrating computing environments", *SIGOPS Oper. Syst. Rev.*, v. 36, n. SI, pp. 361–376, 2002.

[7] CREASY, R. J., "The Origin of the VM/370 Time-Sharing System", *IBM Journal of Research & Development*, v. 25, n. 5, pp. 483–490, Sept. 1981.

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., et al., "Xen and the art of virtualization". In: *ACM Symposium on Operating Systems Principles*, pp. 164–177, 2003.

[9] PRATT, I., FRASER, K., HAND, S., et al., "Xen 3.0 and the Art of Virtualization". In: *Proceedings of Linux Symposium 2005*, July 2005.

[10] COULOURIS, G., DOLLIMORE, J., KINDBERG, T., *Sistemas Distribuídos - Conteitos e Projeto*. Quarta ed. BookMan, 2006.

[11] BRAGATO, L. L. S., *Implementação e Avaliação de um Sistema de Vídeo Sob Demanda Baseado em Cache Cooperativa Colapsada de Vídeo*, Dissertação de Mestrado, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, Setembro 2006.

[12] PINHO, L. B., AMORIM, C. L., "Assessing the efficiency of stream reuse techniques in P2P video-on-demand systems", *Journal of Network and Computer Applications (JNCA)*, v. 29, n. 1, pp. 25–45, January 2006, ISSN: 1084-8045, Academic Press - Elsevier.

[13] SWSOFT, *OpenVZ Users Guide*, Nov. 2005.

[14] INTEL, *Intel 64 and IA-32 Architectures Software Developers Manual. Volume 1: Basic Architecuture*, Nov. 2007.

[15] MILLS, D. L., *Network Time Protocol (Version 3) Specification, Implementation and Analysis*, Tech. rep., 1992, http://www.faqs.org/rfcs/rfc1305.html.

[16] THOMPSON, K., RITCHIE, D. M., *gettimeofday*, Nov. 1971.

[17] PATTERSON, D. A., HENNESSY, J. L., *Computer Organization and Design: The Hardware/Software Interface*. 3ª ed. Morgan Kaufmann, August 2004).

[18] BOVET, D. P., CESATI, M., *Understanding the LINUX KERNEL*. 3ª ed. O'Reilly Media, 2006.

[19] INTEL, *8254/82C54: Introduction to Programmable Interval Timer*, Nov. 2008, http://www.intel.com/design/archives/periphrl/docs/7203.HTM.

[20] OSDEV.ORG, "Intel 8253", Wiki, Nov. 2008, http://wiki.osdev.org/*Programmable_Interval_Timer*.

[21] INTEL, *8254/82C54 Programmable Interval Timer*, Nov. 2008, http://www.intel.com/design/archives/periphrl/docs/7178.htm.

[22] INTEL, *Intel 64 and IA-32 Architectures Software Developers Manual. Volume 3A: System Programming Guide, Part 1*, Feb. 2008.

[23] HP, INTEL, MICROSOFT, et al., *Advanced Configuration and Power Interface Specification*, December 2005, Revision 3.0a.

[24] INTEL, *IA-PC HPET (High Precision Event Timers) Specification*, Oct.. 2004.

[25] MOORE, G. E., "Cramming More Components onto Integrated Circuits", *Electronics*, v. 38, n. 8, pp. 114–117, April 1965.

[26] UHLIG, R., NEIGER, G., RODGERS, D., et al., "Intel virtualization technology", *Computer*, v. 38, n. 5, pp. 48–56, May 2005.

[27] NEIGER, G., SANTONI, A., LEUNG, F., et al., "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization", *Intel Technology Journal*, v. 10, n. 3, Aug. 2006.

[28] POPEK, G. J., GOLDBERG, R. P., "Formal requirements for virtualizable third generation architectures", *Commun. ACM*, v. 17, n. 7, pp. 412–421, 1974.

[29] SMITH, J. E., NAIR, R., *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[30] TANENBAUM, A. S., *Organização Estruturada de Computadores*. 5ª ed. Pearson, 2006.

[31] VAHALIA, U., *UNIX Internals: The New Frontiers*. Prentice Hall Press: Upper Saddle River, NJ, USA, 1996.

[32] SWSOFT, "Virtuozzo Containers 4", May 2008, http://www.parallels.com/en/products/virtuozzo/.

[33] MUIR, S., "The Seven Deadly Sins of Distributed Systems". In: *First Workshop on Real, Large Distributed Systems, WORLDS'04*, Dec. 2004, http://www.usenix.org/events/worlds04/tech/muir.html.

[34] WILLIAMS, R., *Real-Time Systems Development*. Butterworth-Heinemann©: Upper Saddle River, NJ, USA, 2005, Books24x7 accessed December 11, 2008.

[35] ISHIKAWA, E., AMORIM, C. L., "Collapsed Cooperative Video Cache for Content Distribution Networks". In: *Proceedings of the Brazilian Simposium on Computer Networks (SBRC)*, pp. 249–264, Natal, RN, Brazil, May 2003.

[36] WHATELY, L. L. A., *Uso de Virtualização na construção de serviços de streaming escaláveis*, Tese de Doutorado, COPPE/Sistemas, UFRJ, July 2008, (in Portuguese).

[37] MILLS, D. L., *Network Time Protocol (NTP)*, Tech. rep., Nov. 1985, http://www.faqs.org/rfcs/rfc958.html.

[38] SOBEIH, A., HACK, M., LIU, Z., et al., "Almost Peer-to-Peer Clock Synchronization", *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–10, March 2007.

[39] INTEL, *Intel 64 and IA-32 Architectures Software Developers Manual. Volume 3B: System Programming Guide, Part 2*, Feb. 2008.

[40] TIAN, G.-S., TIAN, Y.-C., FIDGE, C., "High-Precision Relative Clock Synchronization Using Time Stamp Counters". In: *ICECCS '08: Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pp. 69–78, IEEE Computer Society: Washington, DC, USA, 2008.

[41] SOCIETY, I. I., "Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", *IEC 61588 First edition 2004-09; IEEE 1588*, 2004, http://ieee1588.nist.gov/.

[42] EIDSON, J. C., *Measurement, Control, and Communication Using IEEE 1588*. 1ª ed. Springer, 2006.

[43] AMORIM, C. L., DE SOUZA, A. F., "Distributed global clock for clusters of computers", United States Patent No. 20060212738, September 2006.

[44] "FileBench", Dec 2008, http://www.solarisinternals.com/wiki/index.php/FileBench.

[45] STRICKER, T., GROSS, T., "Optimizing memory system performance for communication in parallel computers", *SIGARCH Comput. Archit. News*, v. 23, n. 2, pp. 308–319, 1995.

[46] FIGUEIREDO, R., DINDA, P., FORTES, J., "A case for grid computing on virtual machines". In: *In Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, pp. 550–559, May 2003.

[47] SANTHANAM, S., ELANGO, P., ARPACI-DUSSEAU, A., et al., "Deploying virtual machines as sandboxes for the grid". In: *WORLDS'05: Proceedings of the 2nd conference on Real, Large Distributed Systems*, pp. 7–12, USENIX Association: Berkeley, CA, USA, 2005.

[48] GOLDBERG, R. J., "Survey of virtual machine research", *Computer*, v. 7, n. 6, pp. 34–35, June 1974.

[49] ANDRADE, N., CIRNE, W., BRASILEIRO, F., et al., "OurGrid: An approach to easily assemble grids with equitable resource sharing". In: *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[50] CAVALCANTI, E., ASSIS, L., GAUDENCIO, M., et al., "Sandboxing for a free-to-join grid with support for secure site-wide storage area". In: *First International Workshop on Virtualization Technology in Distributed Computing, 2006. VTDC 2006.*, Nov. 2006.

[51] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., et al., "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors", *SIGOPS Oper. Syst. Rev.*, v. 41, n. 3, pp. 275–287, 2007.

[52] FAN, X., WEBER, W.-D., BARROSO, L. A., "Power provisioning for a warehouse-sized computer". In: *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pp. 13–23, ACM: New York, NY, USA, 2007.

[53] LIM, K., RANGANATHAN, P., CHANG, J., et al., "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing En-

vironments". In: *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pp. 315–326, IEEE Computer Society: Washington, DC, USA, 2008.

[54] CRAYCRAFT, D. G., RUSSELL, R. G., GODFREY, G. M., et al., "Scalable Virtual Timer Architecture for Efficiently Implementing Multiple Hardware Timers With Minimal Silicon Overhead", United States Patent No. 6,550,015 B1, Feb 1999.

[55] MANN, T. P., "Method and Apparatus for Emulating Multiple Virtual Timers in a Virtual Computer System When The Virtual Timers Fall Behind The Real Time Of a Physucal Computer System", United States Patent No. 7,475,002 B1, Feb 2004.

[56] NICHOLAS, A. E., VEGA, R. A., "Controlling virtual time", United States Patent No. 20070033589, Aug. 2004.

Apêndice A

Implementação do vTSC - Código Fonte

Esse apêndice tem como objetivo apresentar as principais mudanças realizadas pela implementação do vTSC nos sistemas OpenVZ e Linux.

A.1 OpenVZ

O suporte ao vTSC no OpenVZ fui realizado em duas etapas distintas, a primeira foi a modificação do *kernel* OpenVZ e a segunda a modificação da aplicação de controle. A seguir são apresentados os arquivos que sofreram modificações ou foram incluídos durante o processo de desenvolvimento.

A.1.1 *Kernel* OpenVZ

As modificações realizadas dentro do kernel OpenVZ tiveram como objetivo incluir as novas chamadas de sistemas e modificar as estruturas de controle associadas aos contêineres de forma a viabilizar o correto funcionamento do vTSC. Os arquivos apresentados nesta seção são relativos ao *kernel* Linux 2.6.18 com o *patch* OpenVZ 028stab057.2, podendo ser necessário a modificação de outros arquivos para implementação do vTSC em outra versão do *kernel* ou do *patch*.

A introdução das estruturas de dados e controles associados a criação de uma instância do vTSC no sistema OpenVZ foi realizada alterando-se os arquivos a seguir:

- $< openVZ\_src > /kernel/ve/vecalls.c$

- $< openVZ\_src > /include/ve.h$

Uma vez que o vTSC foi implementado dentro do OpenVZ foram construídas duas chamadas de sistemas que são a única forma de a aplicação acessar as informações oferecidas pelo novo mecanismo, essas chamadas foram criadas alterando-se os seguintes arquivos:

- $< openVZ\_src > /kernel/sched.c$
- $< openVZ\_src > /arch/i386/kernel/syscall\_table.S$
- $< openVZ\_src > /include/linux/syscalls.h$
- $< openVZ\_src > /include/asm - i386/unistd.h$
- $< openVZ\_src > /include/asm/unistd.h$

As alterações anteriormente garantem o funcionamento do vTSC em um ambiente onde não existe migrações, sendo assim o suporte a migração do vTSC foi incluído dentro do procedimento de migração realizado pelo OpenVZ, para isso foram alterados os arquivos a seguir:

- $< openVZ\_src > /kernel/cpt/cpt\_context.h$
- $< openVZ\_src > /kernel/cpt/cpt\_proc.c$
- $< openVZ\_src > /kernel/cpt/rst\_proc.c$
- $< openVZ\_src > /kernel/cpt/cpt\_dump.c$

O procedimento de migração do vTSC para OpenVZ contou também com a criação de novos arquivos responsáveis pelo procedimento cliente/servidor realizado para enviar as informações do vTSC entre os nós de origem e destino. Os arquivos criados foram:

- $< openVZ\_src > /kernel/cpt/cpt\_vtsc.h$
- $< openVZ\_src > /kernel/cpt/cpt\_vtsc\_srv.c$
- $< openVZ\_src > /kernel/cpt/rst\_proc\_clt.c$

A.1.2 vzctl

O vzctl é o principal mecanismo de interação entre o administrador e o OpenVZ, sendo ele o responsável por operacionalizar no nivél do usuário o procedimento de migração. Assim o procedimento de migração apenas realiza também a migração do vTSC se o administrador desejar, isso foi feito para que o mesmo *kernel* pudesse ser utilizado nos testes que com ou sem o vTSC.

O vzctl passou a contar com quatro novos paramentos que podem ser utilizados para o procedimento de migração são eles; set_vtsc_ip, set_vtsc_port, set_vtsc_srv e set_vtsc_clt. Os parâmetros set_vtsc_srv e set_vtsc_clt servem para identificar informar ao *kernel* do OpenVZ modificado que o servidor e o cliente de migração do vTSC devem ser executados nos nós de origem e destino respectivamente.

A.2 Linux

Para construir o suporte ao vTSC no Linux foi necessário apenas modicar os arquivos ligados ao *kernel*. A seguir são apresentados os arquivos que sofreram modificações durante o processo de desenvolvimento.

A.2.1 *Kernel* Linux

As modificações no *kernel* do Linux tiveram como objetivo incluir as novas chamadas de sistemas e modificar as estruturas de controle associadas aos processos e as fila de processos vinculadas aos núcleos. Os arquivos apresentados nesta seção são relativos ao *kernel* Linux 2.6.27.8 com o, podendo ser necessário a modificação de outros arquivos para implementação do vTSC em outra versão do *kernel*.

A introdução das estruturas de dados, os controles associados a criação de uma instância do vTSC e a criação de duas chamadas de sistemas para aceso as informações no Linux foi realizada alterando-se os arquivos a seguir:

- $< linux\_src > /include/linux/sched.h$
- $< linux\_src > /kernel/sched.c$
- $< linux\_src > /kernel/fork.c$

- $< linux\_src > /arch/i386/kernel/syscall\_table.S$
- $< linux\_src > /include/linux/syscalls.h$
- $< linux\_src > /include/asm - x86/unistd.h$
- $< linux\_src > /include/asm/unistd.h$

Uma vez que o vTSC foi implementado dentro do Linux foi necessário incluir os controles ligados ao escalonamento de freqüencias, isso foi realizado alterando-se os arquivos específicos dos processadores utilizados nos experimentos, neste caso os processadores da AMD, para isso foram alterados os seguintes arquivos:

- $< openVZ\_src > /arch/x86/kernel/cpu/cpufreq/powernow - k6.c$
- $< openVZ\_src > /arch/x86/kernel/cpu/cpufreq/powernow - k7.c$
- $< openVZ\_src > /arch/x86/kernel/cpu/cpufreq/powernow - k8.c$

As mudanças realizadas nestes arquivos tiveram como objetivo informar ao controle do vTSC que um dos núcleos do processador estava tendo sua freqüência alterada.

Apêndice B

Diário de uma Dissertação

Esse apêndice resume os trabalhos realizados dentro do Laboratório de Computação Paralela que deram origem a esta dissertação de mestrado.

B.1 Início

O caminho desta dissertação teve início em Janeiro de 2007 quando a convite do professor Claudio Amorim, passei a integrar a equipe do LCP, sob a supervisão direta do Lauro Whately, na época doutorando. Como primeira tarefa o Lauro sugeriu que eu realiza-se a integração prevista entre um procedimento de migração de discos virtuais defendido por outro aluno em seu mestrado e o OurGrid. Essa integração levou a alterações na forma como a solução para transferência de discos foi implementada, de forma a reduzir a complexidade do algoritmo proposto e com isso utilizando de forma efetiva a largura de banda disponível.

O processo de integração entre essa nova implementação da solução e o OurGrid, teve fim em meados de Julho de 2007, sendo que os experimentos realizados demonstravam que a nova implementação consegui melhorar o desempenho da solução quando a banda efetiva era de 80 Mpbs, sendo que a implementação original so era vantajosa quando banda efetiva era de 20 Mpbs. Sendo que se a largura de banda efetiva for 800 Mpbs ou maior, a melhor solução encontrada era o uso de compactação.

B.2 Busca por um tema

Com base nos resultados obtidos em meu primeiro projeto dentro do LCP e em conversas com o Lauro, comecei a estudar o problema de como fazer o envio de discos virtuais não para um mas para múltiplos nós computacionais dentro de um sistema de grades. Sendo que no escopo desta ideia foram estudadas as soluções como *bitorrent* e outras soluções existentes para *clusters*.

Enquanto realizava esse estudo foi convidado pelo Lauro a realizar uns experimentos com o OpenVZ, inicialmente buscamos formas de melhorar (reduzir) o tempo em que uma aplicação ficava suspensa durante o procedimento de migração deste sistema de virtualização, apesar de não ter sido possível na época modificar o procedimento de migração esse estudo me permitiu entender como foi implementado o procedimento de migração do OpenVZ.

Em paralelo ao trabalho com o Lauro, comecei a imaginar como a solução de envio de discos virtuais poderia se beneficiar do modelo de implementação das grades computacionais, ou seja o fato de uma grade computacional ser um *cluster* de *clusters*. A implementação inicial desta ideia ficou suspensa enquanto participava de alguns experimentos que o Lauro estava realizado para sua tese de doutorado, sendo que um problema que encontrava-se ocorrendo em um dos experimentos de migração deu origem a essa dissertação.

B.3 Contagem do tempo e virtualização

A pesquisa reportada nesta dissertação de mestrado teve início em Maio de 2008 durante a realização de alguns experimentos com o sistema de vídeo sob demanda GloVE, a pesquisa bibliográfica realizada não deu indícios de que existia uma solução fechada para o problema de temporização em domínios virtualizados, sendo que a prática utilizada era o uso do NTP como mecanismo de sincronização entre os nós.

Devido as limitações impostas pelo NTP para esse tipo de solução, pois o mesmo não foi criado para sincronizar pares. A solução que escolhi seguir foi a criação de um novo contador, que passaria a ser responsável por manter a temporização das aplicações mesmo que ocorressem migrações, desta forma as limitações impostas pela sincronização global entre os nós deixariam de existir.

A nova solução (vTSC) foi implementado inicialmente para o sistema OpenVZ e uma vez que essa implementação foi concluída, a implementação da versão para Linux teve início (sugestão do professor Claudio Amorim).

Apêndice C

Memperf

Esse apêndice tem como objetivo apresentar de forma sucinta o uso do memperf.

C.1 Uso

O memperf é um aplicativo utilizada para medir a banda de memoria em duas dimensões, variando o tamanho de bloco o que permite descobrir informações com relação a hierarquia de memória e variando a forma como os dados são acessados (contíguos ou não).

Para os experimentos realizados nesta dissertação foi utilizado o comando apresentado abaixo a seguir para executar o memperf:

- $memperf - m0 - c10$

Onde as opção $-m$ representa o tipo de teste a ser realizado e $-c$ representa quantas vezes o teste deve ser repetido.

Uma característica importante do memperf é que ele descarta os valores medidos durante os experimentos que são menores que 75% do valor médio experimental, sendo reportado para o usuário o total de valores descartados por essa regra.