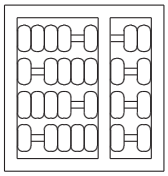

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Uma Análise Comparativa Entre o Desempenho de Máquinas Virtuais Interpretadas de Sistema e de Processo

R. Barboza Jr *D. C. S. Lucas*
A. S. Ferreira

Technical Report - IC-12-??? - Relatório Técnico

September - 2012 - Setembro

Uma Análise Comparativa Entre o Desempenho de Máquinas Virtuais Interpretadas de Sistema e de Processo

Roberto Barboza Jr * Divino C. S. Lucas † Anderson Soares Ferreira ‡

Resumo

Máquinas Virtuais são ferramentas amplamente utilizadas para resolução de diversos problemas computacionais e podem ser categorizadas por diversos atributos, entre eles escopo de emulação (sistema ou processo) e técnica de emulação (interpretação ou tradução). Como forma de ampliar a compreensão do *overhead* decorrente da emulação do sistema operacional e dispositivos de *hardware* em uma máquina virtual de sistema, este trabalho apresenta uma comparação de desempenho entre máquinas virtuais interpretadas de sistema e de processo. Para tanto, conduzimos uma investigação utilizando duas máquinas virtuais distintas. A primeira delas, chamada *Bochs*, é uma máquina virtual de sistema que emprega interpretação como técnica de emulação, a segunda é uma máquina virtual de processo chamada *Box*, desenvolvido a partir da ferramenta anterior especificamente para esse estudo.

1 Introdução

Uma Máquina Virtual (MV) é uma plataforma versátil que pode ser empregada para resolver diversos problemas na área de computação. Uma MV pode ser vista como uma camada utilizada para prover a integração entre duas interfaces (possivelmente distintas). Esta camada pode ser implementada com emulação em software, virtualização em hardware ou uma composição das duas abordagens. As interfaces podem ser tanto a *Application Binary Interface* (ABI) em MVs de processo, quanto a *Instruction Set Architecture* (ISA) em MVs de sistema. Dessa forma, tais ferramentas ocupam uma posição estratégica que pode ser explorada de diversas formas: *cross-platform emulation* - permitindo que aplicações escritas para uma plataforma sejam executadas em outra; *simulação* - simulação do comportamento do hardware durante a execução do programa; *análise* - análise do perfil de execução das aplicações; *otimização* - aplicar otimizações no código da aplicação utilizando informações obtidas durante a execução do código.

Duas abordagens são frequentemente empregadas para implementar a emulação de código em uma máquina virtual: interpretação e tradução. Uma MV que emprega interpretação utiliza funções para simular o comportamento de cada instrução da aplicação

*RA: 035712, rbarboza@gmail.com
†RA: 115121, divcesar@gmail.com
‡RA: 974530, asferreira.ferreira@gmail.com

sendo emulada. O processo de tradução porém, emprega técnicas da área de compiladores para produzir um código binário nativo (possivelmente otimizado) equivalente àquele da aplicação sendo emulada. É comum encontrarmos MVs que, visando maximizar o desempenho, empregam estas abordagens em conjunto.

Note que uma MV de sistema emula não apenas as instruções da arquitetura alvo, mas toda uma infraestrutura de hardware necessária para executar um sistema operacional. Nisto estão inclusos dispositivos como placa de rede, hierarquia de memória, placa gráfica e etc. Para tarefas como *cross-platform emulation* e perfilamento da execução do código de uma aplicação em específico, o uso de uma MV de sistema também agrega ao *overhead* de emulação da aplicação o *overhead* de emulação destes dispositivos e do sistema operacional. Portanto frequentemente nestes cenários uma máquina virtual de processo é preferível em relação a uma MV de sistema.

Nesse texto apresentamos uma proposta de avaliação do *overhead* de emulação do sistema operacional e dispositivos de hardware em uma máquina virtual de sistema interpretada em relação a uma máquina virtual de processo interpretada.

Para tanto, propomos a transformação de uma máquina virtual de sistema em uma máquina virtual de processo. Além de uma melhor compreensão do *overhead* de emulação em uma máquina virtual de sistema, este trabalho tem como resultado uma infraestrutura para a realização de experimentos voltados para a área de máquinas virtuais.

Este texto está organizado da seguinte forma. Na Seção **??** apresentamos com certo nível de detalhamento nossos objetivos. Na Seção 2 apresentamos a fundamentação teórica para a realização do projeto. Na Seção 3 apresentamos um levantamento bibliográfico da área de máquinas virtuais a nível de binários. A Seção 4 e a Seção **??** descrevem a infraestrutura e a metodologia proposta para a realização do projeto, respectivamente. Por fim a Seção 6 apresenta a conclusão do trabalho.

2 Fundamentação Teórica

2.1 Máquinas Virtuais

Uma máquina virtual tem como finalidade implementar as interfaces de um sistema a partir de outro sistema. Seguindo a taxonomia apresentada por Jim Smith e Nair em [11], podemos classificar uma máquina virtual em de sistema ou de processo dependendo de qual o escopo de emulação é provido:

- **Máquina Virtual de Sistema:** A interface emulada é a nível da *Instruction Set Architecture*. A ISA de um computador é composta de duas partes: as instruções de computação de propósito geral (*user-ISA*) e as instruções de controle dos periféricos do sistema (*system-ISA*). Uma máquina virtual de sistema deve, portanto, emular tanto os dispositivos de hardware como placas de rede e vídeo quanto as instruções que controlam tais dispositivos. Dessa forma, uma máquina virtual de sistema permite a execução de um sistema operacional completo de forma transparente para a aplicação sendo emulada.

- **Máquina Virtual de Processo:** A interface emulada é a nível da *Application Binary Interface*. A ABI define diversos padrões, onde os principais são: um subconjunto da ISA que é visível aos programas de usuário (*user-ISA*) e uma interface para que programas possam se comunicar com o sistema operacional para utilizar os recursos disponíveis no mesmo. Em sistemas operacionais *Unix-like*, esta interface com o sistema operacional é feita através de *system calls* (syscalls).

  A ABI também define convenções para tipos de dados, *endianness*, alinhamento, chamadas de funções (como devem ser passados os argumentos e o retorno), e o formato utilizado para representar arquivos binários executáveis e bibliotecas dinâmicas. No Linux o formato utilizado para representar tais arquivos é o *Executable and Linking Format* (ELF).

2.2 Técnicas de Emulação

Segundo [11], a emulação é um processo de implementação das funcionalidades e interfaces de um sistema em outro sistema com características diferentes. Por exemplo, para criar uma máquina virtual que executa um sistema compilado para o x86 em uma arquitetura PowerPC será necessário emular tanto a interface do x86 (instruções) quanto funcionalidades (ex: *calling convention*) utilizando instruções disponíveis na arquitetura PowerPC. O processo de emulação é central em uma máquina virtual e a técnica utilizada para implementá-lo é fundamental para o desempenho do sistema. Entre as técnicas frequentemente utilizadas para implementar emulação em uma máquina virtual estão:

- **Interpretação:** Processo no qual cada instrução do programa original (*guest*) possui uma rotina na plataforma destino (*host*) que implementa a semântica da instrução na arquitetura *guest*. A interpretação envolve a recuperação da instrução original (*fetch*), sua decodificação (*decode*) e finalmente a execução da operação equivalente.

- **Tradução de binários:** Neste processo um código nativo na arquitetura destino é gerado dinamicamente (*Just-in-time*) a partir do código original da aplicação e tem como função reproduzir o comportamento do código da aplicação original [10]. Esta forma de implementação assemelha-se ao processo de compilação estático onde um código fonte é decodificado e traduzido para uma linguagem destino.

- **Tradução em duas etapas:** Nesta forma de implementação, o sistema utiliza uma combinação de interpretação e tradução com o intuito de reduzir o *overhead* de emulação. Regiões de código que são raramente executadas são apenas interpretadas (uma vez que o custo de tradução seria maior que o de sempre interpretar a região), enquanto porções de código que são frequentemente executadas são traduzidas, uma vez que a execução do código de forma otimizada amortiza o custo da tradução.

As implicações do uso de interpretação ou tradução influenciam diretamente o desempenho e a aplicabilidade da máquina virtual. Em sistemas interpretados, o desempenho é relativamente baixo, uma vez que cada instrução é emulada individualmente por uma função, o que adiciona todo o *overhead* de chamada e retorno de funções. Apesar disso,

tais sistemas são extremamente portáveis, o que permite sua execução em diversas plataformas com poucas ou até mesmo nenhuma alteração em seu código original. Nos sistemas de traduções de binários, existe um custo (*overhead*) inicial elevado, visto que o código original precisa ser traduzido antes de sua execução. No entanto, uma vez traduzido, o código resultante é executado nativamente, o que garante melhor desempenho. Sistemas que empregam tradução em duas (ou mais) etapas são ainda mais sofisticados e utilizam técnicas para prever regiões de código que serão frequentemente executadas a fim de traduzir e otimizar tais regiões.

2.3 Técnicas de Otimização para Interpretação

Na forma mais básica de interpretação há um *loop* principal que busca a instrução original, decodifica-a e chama a função responsável pela execução da instrução, retornando ao *loop* principal logo em seguida. No entanto, existem diversas técnicas que tornam a interpretação um processo mais eficiente. Algumas delas estão listadas a seguir:

- **Threaded Interpretation:** São adicionadas ao final das funções que emulam cada uma das instruções a busca pela próxima instrução, sua decodificação e a chamada para a função responsável pela execução da instrução. Note que isto reduz drasticamente o número de iterações do *loop* principal do interpretador.

- **Pre Decoding:** As instruções são decodificadas, por exemplo, sempre que uma nova página de código é carregada do disco, e os campos como *opcode* e operandos são salvos em estruturas de dados padronizadas, o que torna possível uma implementação mais simples e eficiente dos mecanismos de despacho e interpretação.

- **Direct Threaded Interpretation:** Utilizado em conjunto ao *Pre Decoding*. Anota-se junto com as informações de decodificação o endereço da rotina que deve fazer a interpretação de cada instrução, de forma que ao final de cada rotina de interpretação é feito um salto indireto "diretamente" para a rotina que interpreta a próxima instrução - evitando a consulta por uma rotina de instrumentação.

- **DICache:** É uma proposta de utilização de uma cache de decodificação para as instruções interpretadas frequentemente. A DICache [3] funciona como uma cache convencional e pode ser implementada tanto em software como em hardware. Cada linha da cache contém uma rótulo para identificar a instrução mapeada para aquela linha, os operandos da instrução e o endereço da rotina que faz a interpretação dessa instrução.

2.4 Executable and Linking Format

O *Executable and Linking Format* (ELF) [9] é o formato de arquivos binários executáveis, bibliotecas compartilhadas, códigos objetos, etc. utilizados em ambientes *Unix-like*. Um arquivo ELF encaixa-se em um dos três tipos:

- **Relocatable File:** Representa um arquivo objeto que contém dados e código que serão "linkados"a outro binário para criar um executável ou objeto compartilhado. Em geral são arquivos utilizados durante o processo de compilação estática.
- **Executable File:** Representa um programa compilado. O programa pode possuir dependências que devem ser resolvidas em tempo de execução ou pode ter "linkagem"estática, o que significa que todas as bibliotecas necessárias para sua execução já foram resolvidas durante a compilação.
- **Shared Object:** Representa um biblioteca dinâmica que contém código e dados que podem ser "linkados"a outro objeto compartilhado, criando um terceiro objeto compartilhado, ou pode ser combinado a um executável pelo *linker* dinâmico e a outros objetos compartilhados para a criação de uma imagem de processo.

A organização de um arquivo ELF é composta por um cabeçalho principal (*ELF Header*), seguido por cabeçalhos de seção - descrevendo seções de informações (instruções, dados, símbolos) utilizados na "linkagem"estática - ou cabeçalhos de programa - que contém os segmentos da aplicação e as informações necessárias para a criação do processo.

3 Levantamento Bibliográfico

O Bochs [8] é uma máquina virtual de sistema com suporte a emulação de diversos processadores da família x86 32 e 64 bits. Ele preza por portabilidade, e nesse sentido utiliza interpretação como técnica de emulação. Para reduzir o *overhead* de interpretação, o Bochs utiliza técnicas como *pre-decoding* [7], *threaded-interpretation* [5], e *lazy evaluation* [4]. O sistema que estamos propondo se diferencia do Bochs por ser voltado à emulação de processos e não de sistemas.

O Pin [6] é uma máquina virtual de processo que emula os ISAs IA-32 e x86-64. Ele é uma ferramenta de código fechado e possui versões tanto para Windows como para Linux, sendo reconhecido por prover uma rica API para criação de ferramentas (Pintools) para análise do comportamento dinâmico de aplicações. O Box difere do PIN por ser uma máquina virtual de código aberto e utilizar interpretação como técnica de emulação. Em contrapartida, estes dois sistemas tem em comum o aspecto de proverem uma interface para instrumentação de binários e serem emuladores *same-ISA* [1].

O HDTrans [12] é uma máquina virtual de processo para a arquitetura IA-32. O HDTrans difere do Bochs e do PIN por empregar tradução de binários como técnica de emulação. No entanto, o HDTrans abre mão de empregar otimizações no código traduzido (o que geralmente é feito para amortizar o *overhead* de tradução) em favor da implementação de uma tradução simples e rápida. O Box diferencia-se do HDTrans por empregar interpretação de binários.

O Dynamo [1] é uma máquina virtual de processo para a arquitetura do HP PA-8000. Diferentemente do PIN, Bochs e HDTrans, o Dynamo é uma máquina virtual cujo objetivo

[1] Em uma máquina virtual same-ISA as duas interfaces da máquina virtual são iguais, possivelmente diferindo apenas em relação a ABI.

é otimizar a execução do programa sendo emulado. Para tanto, o Dynamo emprega uma combinação das técnicas de interpretação e emulação. O sistema inicialmente interpreta o código da aplicação sendo emulada. Uma vez que uma região é declarada como quente [2], o Dynamo aplica otimizações nesta região e salva o código otimizado em uma cache de traduções. Subsequentes invocações do trecho de código original serão emuladas utilizando o código traduzido. O Box diferencia-se do Dynamo por não ter o foco na otimização de binários mas sim em ser um sistema que disponibilize uma interface simples para instrumentação de binários.

O IA-32 EL [2] é uma máquina virtual de processos com o objetivo de suportar a execução de aplicações IA-32 em processadores da família IA-64. De forma similar ao Dynamo, o IA-32 EL emprega uma abordagem de emulação em duas etapas, porém as duas etapas realizam tradução de código. Inicialmente o IA-32 EL efetua a tradução de código em uma granularidade de bloco básico. Durante essa tradução, código de instrumentação é inserido para detectar blocos básicos quentes. Quando um número mínimo de blocos básicos é identificado, o sistema forma uma região de código envolvendo esses blocos básicos, os otimiza e posteriormente salva em uma cache de traduções. Execuções subsequentes desses blocos básicos usam a tradução otimizada. O Box diferencia-se do IA-32 EL por não fazer tradução de binários.

O StarDBT [13] é uma máquina virtual de pesquisa capaz de fazer traduções de aplicações compiladas para x86 32/64 bits para execução em hardware x86 32 bits. De forma similar ao IA-32 EL e ao Dynamo, o StarDBT é um tradutor em duas fases. Regiões de código que são infrequentemente executadas são traduzidas utilizando um tradutor simples e rápido, enquanto regiões de código que são frequentemente executadas são traduzidas empregando otimizações de código e posteriormente persistidas em uma cache de traduções. O Box diferencia-se do StarDBT por ser capaz de traduzir apenas binários de 32 bits, compilados para Linux e não empregar tradução de binários.

[2] Uma região de código é quente quando ela é frequentemente executada.

4 Infraestrutura de Pesquisa

4.1 Bochs

4.2 Box

4.2.1 Componentes Removidos

4.2.2 Carregador

4.2.3 Emulação de Syscalls

4.2.4 Emulação da Memória

4.3 Mibench

4.4 μClibC

5 Resultados

5.1 Metodologia

5.2 Overhead de Emulação de Sistema

Comparação com o Bochs.

5.3 Otimizando a Interpretação

Gráficos mostrando o tempo de execução do Box com DICache, Threaded Interpretation e ambas as técnicas habilitadas.

5.4 Caracterizando o Overhead do Sistema

Gráfico, por exemplo na forma de pizza, mostrando quais são os pontos onde o Box gasta a maior parte do tempo durante a emulação.

6 Conclusão

Referências

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on itanium-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, 2003. IEEE Computer Society.

[3] W. Chen, D. Chen, and Z. Wang. An approach to minimizing the interpretation overhead in dynamic binary translation. *The Journal of Supercomputing*, 61:804–825, 2012. 10.1007/s11227-011-0636-y.

[4] R. J. Hookway and M. A. Herdeg. DIGITAL FX!32: combining emulation and binary translation. *Journal of Digital Technology*, 9(1):3–12, Jan. 1997.

[5] P. Klint. Interpretation techniques. *Software: Practice and Experience*, 11(9):963–973, 1981.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[7] P. Magnusson and D. Samuelsson. A compact intermediate format for simics. Technical Report R94:17, Swedish Institute of Computer Science, September 1994.

[8] D. Mihocka and S. Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infraestructure. In *1h Workshop on Architectural and Microarchitectural Support for Binary Translation, AMAS-BT'01, 2001.*, 2001.

[9] SCO. System V application binary interface, March 1997. `http://www.sco.com/developers/devspecs/gabi41.pdf`. (Visitado em 25 de setembro de 2012.).

[10] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, Feb. 1993.

[11] J. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[12] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. Hdtrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 175–185, New York, NY, USA, 2006. ACM.

[13] C. Wang, S. Hu, H.-s. Kim, S. Nair, M. Breternitz, Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In L. Choi, Y. Paek, and S. Cho, editors, *Advances in Computer Systems Architecture*, volume 4697 of *Lecture Notes in Computer Science*, pages 4–15. Springer Berlin / Heidelberg, 2007.