Fábio Mendonça Maciel
Nelson Antônio de Oliveira

Geração de grafo de execução para códigos executando em plataformas baseadas na família x86

São José do Rio Preto

Novembro de 2005

Fábio Mendonça Maciel
Nelson Antônio de Oliveira

Geração de grafo de execução para códigos executando em plataformas baseadas na família x86

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista Júlio de Mesquita Filho, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientador:
Prof. Dr. Aleardo Manacero Jr.

São José do Rio Preto

Novembro de 2005

Fábio Mendonça Maciel
Nelson Antônio de Oliveira

Geração de grafo de execução para códigos executando em plataformas baseadas na família x86

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista Júlio de Mesquita Filho, como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Fábio Mendonça Maciel
Orientado

Nelson Antônio de Oliveira
Orientado

Prof. Dr. Aleardo Manacero Jr.
Orientador

Banca examinadora:
Prof. Dr. José Márcio Machado
Prof. Dr. Mário Luiz Tronco

São José do Rio Preto

Novembro de 2005

À nossa família e amigos.

Agradecimentos

Primeiramente agradecemos a Deus pela vida que nos deu e por estar conosco nos momentos mais difíceis.

Aos nossos pais, pelo amor, paciência, apoio, motivação, confiança, sabedoria e pelo esforço realizado para que chegássemos até aqui.

Ao nosso orientador, Prof. Dr. Aleardo, pela confiança, ajuda e tempo dedicado à nós, durante a orientação do projeto e também pelo apoio ao longo do curso.

Ao Almir, Diego, Fábio "Ponêis" e seu Escortinho, Guilherme, Leandro e Thiago "Arregão", com os quais passamos grandes momentos e alegrias durante esses quatro anos.

Às nossas namoradas, por sempre serem tão compreensívas e por nos incentivarem nas situações de dificuldade.

Ao Prof. Dr. Walter, pelos ensinamentos sobre ciência, a vida e principalmente pela amizade.

Aos meus tios, José Antônio e Cleunice, por não terem medido esforços durante o período em que me acolheram e por todo o apoio durante o período da faculdade.

A todos os demais amigos e pessoas que conviveram conosco durante esse período.

"Se não posso realizar grandes coisas,
posso pelo menos fazer pequenas
coisas com grandeza."
Clarck

Resumo

Cada vez mais o desempenho tem sido um fator altamente considerado no desenvolvimento de sistemas, sejam paralelos ou não. Em ambientes de programação paralela exige-se uma atenção especial quanto ao desempenho obtido pelo conjunto programa-máquina, em função de seu custo.

O alto custo de equipamentos paralelos faz com que o desempenho desejado seja o máximo possível. Assim, o problema passa a ser determinar medidas para verificar se o sistema está atendendo os requisitos desejados.

Existem diversas ferramentas e métodos que auxiliam na determinação de medidas de desempenho sobre programas, mas infelizmente a maioria delas trabalha com grandes aproximações no modelo do ambiente paralelo, fazendo com que os resultados obtidos não sejam totalmente precisos.

Este projeto apresenta a implementação de um componente de uma nova metodologia que dispensa a inserção de código adicional ao programa e dispensa o uso da máquina simulada para realizar as medidas de desempenho. Tal metodologia consiste na reescrita do código executável em um grafo de execução. Este é um grafo dirigido que armazena informações sobre o tempo de processamento de cada instrução de máquina incluída no programa.

Neste trabalho desenvolveu-se o módulo de geração do grafo de execução a partir da leitura do código executável de um programa, compilado para a família de processadores x86. Esse módulo é necessário pois a geração do grafo depende do código *assembly* utilizado e os processadores dessa família são amplamente utilizados em equipamentos de alto desempenho.

Abstract

Performance has been a highly considered factor in the development of software systems, parallel or not. In parallel programming environments, due to its high cost, a special attention is demanded in how much performance is achieved from the program-machine pair.

The high costs involved with parallel equipment implies that the desired performance must be the maximum possible. Thus, the actual problem becomes the performance measurement to verify if the system is achieving the desired requirements.

There are several tools and methods to assist in the determination of performance measurements on programs. However, most of them make large approximations in the parallel environment model, turning their results somewhat inaccurate.

This work presents the implementation of a component of a new methodology that avoids the insertion of additional code to the program and the use of the simulated machine to execute the performance measurements. Such methodology consists on the rewriting of the executable code in an execution graph. This is a directed graph that stores information about the processing time of each machine instruction enclosed in the program.

In this work, it was developed the module that generates the execution graph from the executable code for the x86 processors family. This module is required since the graph generation process depends on the assembly code used and that processors of the x86 family are largely used on high performance equipments.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A eficiência dos sistemas computacionais tem sido alvo de grandes estudos na computação atualmente. Como os custos com equipamentos de grande porte são elevados, é necessário que se tenha sistemas altamente eficientes e que utilizem todo o pontencial que o *hardware* possa oferecer. A determinação da eficiência de um programa é feita através de sua análise de desempenho. Para isso é preciso criar ferramentas que possam verificar se determinado programa está obtendo um grau de eficiência elevado para evitar ao máximo o desperdício de recursos, tanto financeiros quanto computacionais.

1.1 Problemas na análise/predição de desempenho

Atualmente existem diversas ferramentas que auxiliam a determinação da eficiência de um determinado programa, fornecendo vários dados sobre o mesmo. Entretanto, a maioria delas não produz resultados precisos pela necessidade da inserção de código adicional para se obter as medidas, comprometendo assim o desempenho do próprio programa. Essa inserção de código caracteriza uma técnica invasiva, já que programa analisado e o analisador irão disputar o acesso ao processador.

Um grande problema é obter mecanismos que independam do *hardware* em que se faz a medida. Isso é complicado, pois a maior parte das técnicas conhecidas é invasiva, ou seja, o programa analisado e o analisador disputam o acesso ao processador.

1.2 Uma nova metodologia para análise/predição de desempenho

Para o desenvolvimento de ferramentas de análise de desempenho existem diversas metodologias distintas, cada uma com suas particularidades e limitações. Em sua tese de doutorado Manacero [1] propôs uma nova metodologia para fazer a análise de desempenho

de programas em sistemas distribuídos. Essa técnica baseia-se na predição do desempenho através da simulação do grafo de execução do programa. Este grafo é gerado a partir do código executável e posteriormente é simulado e os resultados obtidos são utilizados para a análise de desempenho do programa. Por ser uma metodologia não invasiva, os resultados obtidos apresentam-se mais confiáveis do que os obtidos por programas invasivos.

1.3 Objetivos do trabalho

Pelo que foi citado, a análise/predição de desempenho é uma tarefa que não apresenta soluções simples, precisas ou fáceis de serem obtidas. Também foi visto que bons resultados de desempenho podem trazer ganhos (financeiros e de tempo), tanto para a pessoa que utiliza o sistema quando para o sistema em si.

Originalmente concebida para a arquitetura MIPS, a metodologia proposta por Manacero [1] foi portada por Moraes [2] para a implementação de um gerador de grafo de execução para a arquitetura SPARC. De acordo com os resultados obtidos nas duas arquiteturas, este projeto utilizou a arquitetura x86 devido a sua crescente expansão e utilização no mercado e em *clusters* Linux.

O objetivo desse projeto é o desenvolvimento de uma ferramenta que leia e interprete código executável de programas compilados para arquitetura x86 e gere um grafo de execução do programa, para que posteriormente seja possível fazer as medidas de desempenho necessárias para determinar se um sistema é eficiente ou não.

1.4 Descrição do texto

Para que se compreenda o trabalho realizado, são tratados no capítulo 2 vários conceitos fundamentais de sistemas paralelos e medidas de desempenho. Também é feita uma descrição do método para a análise de desempenho, essencial para a compreensão da metodologia utilizada.

O capítulo 3 trata da implementação do gerador de grafo, explicando suas particularidades. O capítulo 4 apresenta testes e resultados obtidos pela ferramenta desenvolvida. No capítulo 5 são apresentadas as conclusões obtidas com o desenvolvimento do projeto e também possíveis trabalhos futuros. Finalmente, o apêndice A trata da arquitetura IA-32 e faz uma breve introdução às instruções dessa arquitetura.

2 Fundamentação teórica

Assim que um programa foi escrito e os erros foram eliminados, os programadores geralmente focam a sua atenção para seu desempenho. As ferramentas de medição de desempenho existem para providenciar aos programadores uma forma de entender porque os seus programas não são executados rápido o suficiente.

Neste capítulo são apresentados conceitos fundamentais para uma boa compreensão do texto e do projeto. Na seção 2.1 são apresentadas medidas de desempenho bem como metodologias para selecionar as que serão mais importantes para a análise de desempenho de um sistema específico.

Na seção 2.2 são apresentadas medidas para avaliar sistemas paralelos em particular. É apresentado também o conceito de ganho de velocidade (*speedup*), que é uma das medidas mais importantes para avaliação de sistemas paralelos.

Nas seções 2.3 e 2.4 são apresentadas abordagens de avaliação de desempenho e métodos para predição e/ou análise de desempenho respectivamente.

Na seção 2.5 são apresentadas técnicas de modelagem e avalição de desempenho, as situações em que são utilizadas e exemplos de ferramentas. Na seção 2.6 é apresentada a metodologia proposta por Manacero [1] em sua tese de doutorado e que será utilizada para o desenvolvimento desse projeto.

2.1 Medidas de desempenho

A grande tarefa dos programadores é desenvolver programas que sejam livres de erros e que satisfaçam os requisitos de desempenho. Entretanto, o desempenho de programas paralelos é algo muito complexo e difícil de se definir [3]. Deve-se considerar além do tempo de execução e escalabilidade, os mecanismos que geram, armazenam, transmitem os dados sobre a rede, entre outras. A importância dessas medidas varia de acordo com a natureza do problema, tornando-se assim necessário considerar várias técnicas utilizadas

para tais medições. De um modo geral essas considerações são apresentadas nas próximas duas seções [4, 5, 6, 7].

2.1.1 Selecionando medidas de desempenho

Para cada sistema a ser avaliado um conjunto de medidas de desempenho deve ser escolhido. Uma maneira de preparar esse conjunto é listar os serviços oferecidos pelo sistema. Para cada requisição de serviço feita pelo sistema, há um número possível de saídas. Essas saídas podem ser classificadas de acordo com a figura 2.1.

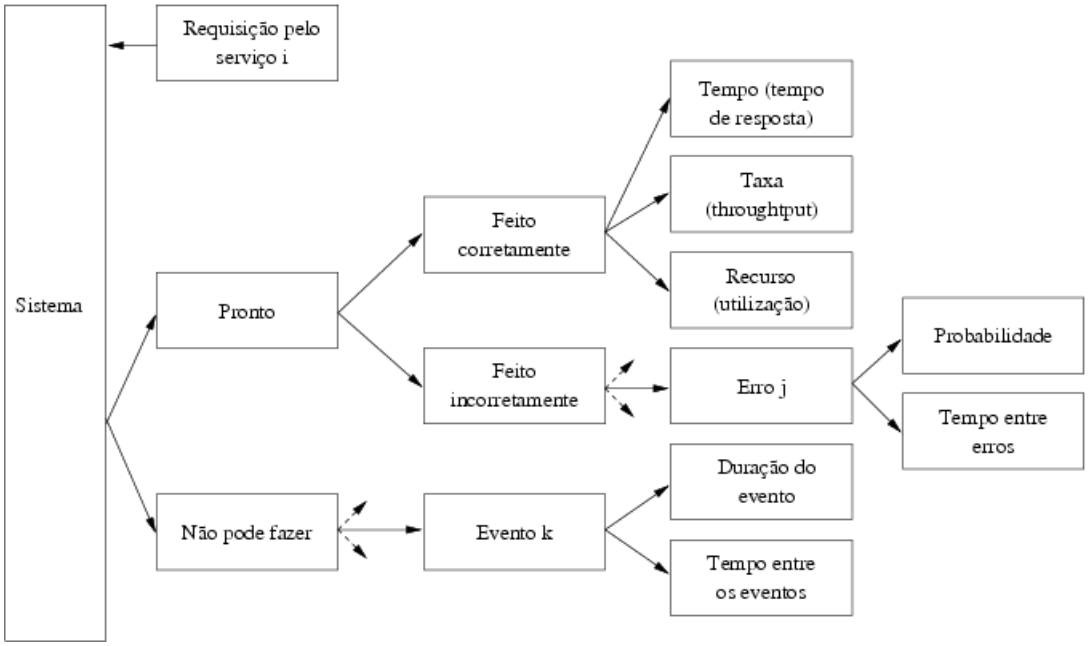

Figura 2.1: Saídas das requisições do sistema.

Como pode-se ver pela figura 2.1 o sistema pode realizar a requisição do serviço corretamente, incorretamente ou recusar a realizar o serviço.

As medidas associadas com esses três tipos de saída, com sucesso, com erro e indisponível são também chamadas de medidas de velocidade, confiabilidade e disponibilidade. Muitos sistemas oferecem mais de um tipo de serviço e, com isso, o número de medidas cresce proporcionalmente.

Em sistemas compartilhados por muitos usuários dois tipos de medidas de desempenho precisam ser consideradas: individual e global. A utilização dos recursos, confiabilidade e disponibilidade são medidas globais enquanto que o tempo de resposta pode ser medido para cada usuário individualmente.

É muito importante que se escolha as medidas corretas para avaliar o desempenho do sistema. Depois de selecionado um número de medidas que forem importantes, deve-se utilizar algumas considerações para selecionar um subconjunto dessas medidas a fim de se obter: variabilidade baixa, ausência de redundância e integridade. Por exemplo, se duas medidas fornecem essencialmente a mesma informação, é melhor se considerar apenas uma para o estudo do desempenho do sistema.

A partir dessas técnicas deve-se obter um conjunto completo de medidas a serem analisadas.

2.1.2 Medidas de desempenho mais utilizadas

São muitas as medidas de desempenho de um programa. Nesta seção serão apresentadas algumas das medidas de desempenho mais utilizadas, deixando-se claro que estas medidas não são as únicas.

Tempo de resposta

É definido como o tempo em que o sistema demora a responder a partir do momento que o usuário requisitou determinado serviço. Esta é uma versão bastante simplista do que pode ser tempo de resposta. Em outro caso, pode-se calcular o tempo de resposta considerando o que o usuário gasta para digitar a requisição pelo serviço e o intervalo de tempo que o sistema gasta para enviar a resposta ao usuário.

Throughput

É definido como a taxa de requisições que o sistema pode atender em uma determinada unidade de tempo. Para UCP's o *throughput* é medido em milhões de instruções por segundo (MIPS) ou milhões de operações de ponto flutuante por segundo (MFLOPS). Para redes de computadores o *throughput* é medido em pacotes por segundo (pps) ou em bits por segundo (bps).

A taxa do maior *throughput* alcançável em um sistema é denominado eficiência. Por exemplo, se em uma rede de 10 Mbps (megabits por segundo) o maior *throughput* alcançável é de 5 Mpbs, então a eficiência dessa rede é de 50%.

Utilização

É medida como sendo a fração do tempo que os recursos ficam ocupados atendendo as requisições dos usuários.

Confiabilidade

É medida como sendo a probabilidade de que ocorram erros no sistema ou pelo tempo entre a ocorrência de um erro e outro. Nesse último caso, é denominado como segundos livres de erro (*error-free seconds*).

Disponibilidade

É definido como o intervalo de tempo em que o sistema está disponível para atender as requisições dos usuários. O tempo no qual o sistema se encontra disponível é chamado *uptime* e o tempo no qual o sistema encontra-se indisponível é chamado *downtime*.

2.2 Medidas de desempenho de sistemas paralelos

Existem várias medidas que avaliam o desempenho de sistemas paralelos. No entanto, o mais importante deles é o ganho de velocidade (*speedup*). Na próxima seção é feita uma análise mais detalhada dessa medida [8].

2.2.1 *Speedup*

Do ponto de vista do *hardware*, é fácil construir um sistema paralelo com enormes taxas de velocidade. O que o usuário deseja é uma máquina que faça o trabalho ser executado rapidamente. Ao assumir que se pode decompor um trabalho em N partes, então o *speedup* é o quanto mais rápido o trabalho decomposto é executado nas N partes.

Segundo Jain [4], com o aumento de N, o desempenho começa a ser dominado por gargalos no sistema e na comunicação. Assim, o melhor *speedup* obtido seria $O(log_2 N)$. Este é um resultado desapontador, se verdadeiro, já que ele diz que não existe muito benefício no paralelismo após uma certa quantidade de processadores.

Em contrapartida, existe a noção de *speedup* ideal. Para que o speedup ideal seja realizado, o problema deve ser perfeitamente decomposto em N partes e não devem existir comunicação ou gargalos no sistema. Assim, o *speedup* é linear, resultando em $S = N$.

Entre estes dois extremos existem duas medidas do que ocorre quando acontecem gargalos no sistema, *overhead*, e decomposição paralela imperfeita. A lei de Amdahl [4] trata cada programa como uma composição de um componente seqüencial s e um componente paralelo $p = 1 - s$. A observação crucial é que o *speedup* do programa será limitado severamente pela quantidade de código não-paralelizável. De forma geral, se existem N processadores, então o *speedup* S (eq. 2.1) é dado por:

$$S \leq \frac{s+p}{s+\frac{p}{N}} \tag{2.1}$$

Por exemplo, se $N = 512$ e $s = 0$ (eq. 2.2), então:

$$S \leq \frac{1}{0+\frac{1}{512}} \tag{2.2}$$

ou $S \leq 512$, que é, essencialmente, o caso ideal de *speedup*. No entanto, se mesmo um pequeno componente seqüencial está presente, tal que $N = 512$ e $s = 0.02$ (eq. 2.3), então:

$$S \leq \frac{1}{0.02+\frac{0.98}{512}} \tag{2.3}$$

ou $S \leq 45.63$. Pelo modelo de Amdahl [4], as limitações de paralelização tornam-se aparentes. Finalmente, quando apenas 50% de um código é reconhecido como paralelo ($p = 0.5$), então 50% não é paralelizável ($s = 1 - p = 0.50$). Com isso, o *speedup* máximo (eq. 2.4) é

$$S \leq \frac{1}{0.50+\frac{0.50}{N}} \tag{2.4}$$

ou $S \leq 2$, não importando a quantidade de processadores utilizados.

Esses resultados parecem desapontadores. No entanto, Gustafson [9], observou que o tempo seqüencial, que é o tempo para carregar um programa, coletar os resultados e fazer cálculos de *overhead*, permanece relativamente constante através de problemas computacionais de variados tamanhos. Este modelo assume que, em contraste à lei de Amdahl, p não é independente de N. Assim, pode-se calcular o *speedup* através da equação 2.5.

$$S_s = \frac{s + p \times N}{s + p} \tag{2.5}$$

2.2.2 Utilização de processador

A próxima medida de desempenho é a utilização do processador. Essa medida é aplicada a todos os processadores ativos para executar uma dada aplicação, e mostra o quanto cada processador contribui para o trabalho. Assumindo que N processadores estão sendo utilizados em um programa paralelo, a utilização do processador P_i (eq. 2.6) é definida por:

$$Utilização(P_i) = \frac{TempoComputaçãoP_i}{TempoOcioso(P_i) + TempoComputação(P_i)} \tag{2.6}$$

em que o TempoComputação(P_i) representa uma quantidade de tempo gasta pelo processador para fazer o trabalho atual, e TempoOcioso(P_i) representa a quantidade de tempo no qual o processador ficou sem fazer nada.

A utilização do processador é uma medida muito intuitiva e a única particularidade sobre ela é como são computados os tempos de computação e ociosidade.

2.2.3 Eficiência do processador

A eficiência é definida como a média da contribuição de cada processador para todo o sistema. É dado como o *speedup* dividido pelo número de processadores (eq. 2.7):

$$Eficiência = \frac{Speedup(N)}{N} \tag{2.7}$$

em que Speedup(N) se refere ao *speedup* medido nos N processadores.

2.2.4 Eficácia

Uma outra medida similar à eficiência é a eficácia (eq. 2.8), sendo útil para determinar o melhor número de processadores (relação custo/benefício) para uma determinada aplicação.

$$Eficácia = n(N) = \frac{(Speedup(N))^2}{N} = Eficiência(N) \times Speedup(N) \qquad (2.8)$$

Uma propriedade importante da curva da eficácia, quando desenhada como uma função do número de processadores N, é que o primeiro máximo corresponde a um ponto de operação ótimo do sistema. Quando um processador pode ser adicionado ao sistema com um resultante acréscimo na eficácia, então o ganho do processador extra mede o custo de adicioná-lo, em que o custo é definido pelo inverso da eficiência. Quando o contrário ocorre (diminuição da eficácia), então o custo de adição supera o ganho de potencial obtido no desempenho.

2.2.5 *Throughput* de pacotes

A próxima medida de desempenho é o *throughput* de pacotes (eq. 2.9), que é a medida que caracteriza a eficiência da estrutura de comunicação, responsável por rotear os pacotes de informação de suas origens aos seus destinos. Isso não deve ser confundido com o *throughput* definido como o número de processos por segundo que a máquina paralela é capaz de completar. Esta segunda medida é associada com sistemas de compartilhamento de tempo. Aqui o interesse recai sobre o número médio de pacotes de informação gerados por unidade de tempo, e pode ser expresso como bits/seg, bytes/seg, pacotes/seg. É dado por

$$Throughput = \frac{\sum_i tamanho(pacote_i)}{T} \qquad (2.9)$$

em que T é o tempo durante o qual os pacotes são gerados.

2.2.6 Outras métricas de desempenho

Como dito anteriormente, a medição do desempenho de uma aplicação sendo executada em uma máquina paralela pode ser feita utilizando diferentes métricas, cada uma focada em um determinado aspecto da máquina ou da aplicação. Quando a atenção está no roteamento de mensagens, por exemplo, então o atraso médio sofrido pelo pacote enquanto ele trafega pela rede pode ser mais significativo do que o *speedup* ou a utilização do processador. Como resultado, uma coleção rica de métricas existe, sendo que a maioria delas é especializada em descrever um comportamento da combinação aplicação/máquina.

2.3 Formas de medição

Existem três abordagens para avaliação de desempenho: analítica, simulação e mensuração (*benchmarking*). Existe um grande número de considerações que ajudam o programador a decidir qual a melhor técnica a ser utilizada. Essas considerações estão apresentadas na tabela 2.1.

Critério	**Modelo Analítico**	**Simulação**	**Mensuração**
1. Estágio	Qualquer	Qualquer	Pós-protótipo
2. Tempo requerido	Pequeno	Médio	Varia
3. Ferramentas	Analistas	Linguagens de computador	Instrumentação
4. Precisão[a]	Baixa	Moderada	Varia
5. Avaliação de *trade-off*	Fácil	Moderada	Difícil
6. Custo	Pequeno	Médio	Alto
7. Escalabilidade	Baixa	Média	Alta

[a] Em todos os casos, o resultado pode não ser confiável ou estar errado.

Tabela 2.1: Comparação entre as abordagens para avaliação.

A grande consideração a ser feita para decidir qual técnica deverá ser utilizada é o estágio do ciclo de vida em que se encontra o sistema. Mensuração é possível somente se algo similar ao sistema proposto já existe, como por exemplo, no desenvolvimento de uma versão mais atualizada do sistema. Portanto, se o sistema é algo novo, a ser implementado, então as técnicas que podem ser utilizadas são a simulação e os modelos analíticos.

A próxima consideração é o tempo disponível para a avaliação. Na maioria das vezes, os resultados são requeridos em pouco tempo e nesse caso modelos analíticos se apresentam mais rápidos de serem utilizados do que as técnicas de simulação.

O nível de exatidão das medidas é outro fator importante a se considerar. Em geral, modelos analíticos requerem muitas simplificações e suposições surpreendendo os analistas quando os resultados se tornam precisos. Simulações podem incorporar um número maior de detalhes e menos suposições que os modelos analíticos e, portanto, estão mais pertos da realidade. Mensurações podem não retornar um resultado preciso porque muitos dos parâmetros, como configuração do sistema, tipo de carga do processador e tempo da mensuração podem ser únicos para aquele experimento.

Em alguns casos pode ser útil utilizar mais de uma técnica simultaneamente. Por exemplo, pode-se utilizar modelos analíticos e simulações para verificar e validar os resultados obtidos por cada técnica.

2.4 Métodos para predição e/ou análise de desempenho

Para que se compreenda claramente os fatores que influenciam o desempenho dos programas existem duas abordagens distintas [10]. A primeira abordagem constitui um conjunto de técnicas que permitem estudar a aplicação. Inicialmente executa-se o programa, verificando-se as medidas de desempenho desejadas. Após os testes concluídos faz-se as alterações necessárias no código para que seu desempenho seja melhorado. Repete-se a primeira e a segunda etapa até que os resultados sejam satisfatórios.

Na segunda abordagem elaboram-se modelos matemáticos para representar o comportamento do programa e as características são abstraídas através de um conjunto de funções e parâmetros.

Segundo Meira [10], pode-se seguir duas estratégias distintas durante a elaboração dos modelos dos programas. A escolha por uma delas depende da necessidade do avaliador que desenvolverá o estudo sobre o desempenho do programa. As estratégias citadas em [10] são:

- *Bottom-up*: o sistema é representado através de um conjunto de funções e parâmetros. Esta técnica é utilizada para analisar fatores que são inerentes ao ambiente da aplicação.
- *Top-down*: considera os aspectos relacionados ao código do programa. Modelos baseados em descrições e análise estatística são exemplos de técnicas que fazem parte dessa estratégia.

2.5 Técnicas de modelagem e avaliação de desempenho

A avaliação de desempenho é utilizada em duas situações distintas. Na primeira procura-se comparar dois sistemas distintos, assim são feitas medidas relevantes pertencentes a cada sistema e posteriormente essas medidas são comparadas para se descobrir qual deles é mais eficiente. Na segunda se esta interessado em estudar alguns parâmetros do sistema e descobrir qual o melhor valor para cada um deles. As técnicas mais utilizadas atualmente [11, 12] são: as mensurações, as simulações, as modelagens analíticas, estrutural e híbrida.

2.5.1 Mensurações

O problema da mensuração de desempenho de programas pode ser dividido em duas partes: análise de desempenho e instrumentação. A análise de desempenho tenta filtrar a grande quantidade de estatísticas disponível sobre a execução de um programa e providenciar informações úteis para o programador. A instrumentação de desempenho focaliza-se em coletar eficientemente informações suficientes sobre a computação. Assim, o programa é executado por diversas vezes e os resultados são comparados. Entre um teste e outro o código do programa é modificado a fim de se obter um melhor desempenho do programa. Um dos maiores objetivos desse processo é encontrar informações relacionadas com a aplicação que puderam ser ajustadas para se obter um desempenho melhor.

Esta técnica é utilizada somente quando se possui o sistema completamente disponível para que se possa executá-lo e assim obter os parâmetros de desempenho desejados.

Para colher essas informações pode-se utilizar monitores, tanto em nível de hardware como em nível de software. Geralmente esses monitores utilizam comandos de alto nível para coletar estatísticas sobre o desempenho do programa.

Dentre as várias ferramentas de *benchmark* tem-se o LINPACK [13], utilizado como uma medida de desempenho para o *ranking* de supercomputadores da lista TOP500[1], o LAPACK [14], versão mais eficiente do LINPACK, SLALOM [9], MEDEA [15] e VISPAT [16], ambos para obtenção de traços de eventos e o ALPES [17], um *benchmark* de análise.

2.5.2 Simulações

As simulações são utilizadas tanto na avaliação de desempenho dos sistemas como na validação de resultados obtidos em modelos analíticos. Ao contrário das medições, as simulações baseiam-se em modelos abstratos do sistema, isso faz com que não seja necessário ter o sistema implementado totalmente para se obter medidas de desempenho sobre o mesmo. Além disso, as simulações permitem criar modelos mais detalhados do sistema, gerando assim resultados mais precisos sobre o desempenho.

A vantagem dos métodos de simulação em relação aos modelos analíticos é que esses métodos de simulação são de mais fácil adaptação, assim quando se faz alguma alteração no programa é mais fácil realizar alterações no simulador do que no conjunto de equações. Essa modelagem é bastante flexível, permite realizar mudanças no modelo sem alterar sua

[1] A lista dos 500 computadores mais rápidos do mundo pode ser acessada através do endereço http://www.top500.org/

estrutura e pode ser refinado após a realização de *benchmarks*.

O principal problema dessa abordagem é encontrar um modelo que represente fielmente o sistema a ser analisado.

Como exemplos de simuladores, existem o PDL [18], AXE [19], PAWS [20], Rsim [21] e o Simics [22].

2.5.3 Modelagem analítica

Estas técnicas são geralmente aplicáveis apenas para uma pequena fatia de arquitetura de sistemas e para estruturas específicas de programas. Este método emprega tanto o modelo de redes quanto o modelo de grafos, onde é traçado cada estado do sistema durante a execução do programa. Um exemplo dessa categoria são as redes de Petri.

Basicamente a modelagem analítica utiliza um conjunto de equações e funções que determinam o comportamento do sistema. Desse modo, todos os fatores que influenciam no comportamento do sistema são modelados como parâmetros do modelo para que assim, possa se obter uma avaliação de desempenho mais precisa.

A modelagem correta desses valores permite uma análise bem ampla e aprofundada dessas características sobre o sistema e ainda permite criar relacionamentos entre os diversos parâmetros.

A modelagem analítica possui um menor custo de execução quando comparada com as outras técnicas de avaliação de desempenho e os resultados obtidos através do modelo analítico podem ser validados através de comparações com testes reais executados a partir do próprio programa. Porém criar um modelo analítico para um problema pode ser uma tarefa custosa, pois geralmente os programas são grandes e complexos, principalmente em sistemas paralelos. Apesar desse problema essa é a única solução quando não se possui o programa e o sistema disponíveis.

Algumas das técnicas de modelagem analítica são baseadas no modelo de rede de Petri estocástica generalizada (GSPN) [23, 24], modelo gerado em tempo de compilação, métodos baseado em teoria de filas [25], métodos baseados em cadeia de Markov [26, 27] e modelos determinísticos [28].

2.5.4 Modelagem baseada em descrição

Esta modelagem é baseada em informações fornecidas pelos usuários sobre o comportamento do sistema, para isso o modelo utiliza uma representação gráfica auxiliar, baseada em grafos e denominada *task graph*, que indica os principais pontos onde se pode paralelizar o programa.

2.5.5 Modelagem com análise estática

Nessa modelagem deve-se ter a aplicação disponível, pois as informações são obtidas pelo compilador em tempo de compilação, por isso essa modelagem é denominada estática. Existem algumas ferramentas que fazem esse tipo de coleta de informações, mas o alto custo associado com essa tarefa não permitiu que essa técnica fosse muito difundida.

2.6 Predição do desempenho através da simulação do código executável

Esta seção irá apresentar a técnica proposta por Manacero em sua tese de doutorado [1], que consiste na análise de desempenho de programas através de simulação. O detalhamento deste modelo pode ser visto na referida tese, ficando aqui apenas uma breve descrição. Esta técnica consiste na geração de um grafo de execução a partir de um código executável, tornando assim possível obter medidas de desempenhos. Isto permite uma obtenção de dados de uma forma não invasiva.

2.6.1 Descrição da metodologia

Este método é baseado na metodologia de três passos de Herzog [29], que busca separar em três diferentes partes o modelo de análise, sendo uma delas dedicada ao **programa** a ser analisado. Outra parte dedica-se aos detalhes da **máquina** onde será feita a execução do programa e finalmente, a última parte interage com as outras duas primeiras. Um grafo é obtido através da reescrita do código executável, pelo modelo proposto por Manacero.

De forma geral, este modelo exige um programa executável, a ser analisado, e que é reescrito na forma de um grafo de execução, sendo que os caminhos de execução do programa são descritos de forma exata pelo grafo. Este grafo, em um próximo passo, é simulado com o objetivo de se obter os dados desejados para a análise.

2.6.2 Módulos do método

O modelo consiste de três módulos [2]: um responsável pela criação do grafo de execução, obtido através de um código executável; outro é encarregado de obter um grafo mínimo a partir do grafo obtido pelo primeiro estágio; por último, o módulo responsável pela simulação e obtenção de medidas de desempenho. Cada um desses módulos é descrito a seguir.

2.6.3 Geração do grafo de execução

Um grafo de execução é um grafo orientado, onde estão representados todos os caminhos que o programa pode seguir. Cada um dos vértices do grafo pode ter uma das seguintes classificações:

- Inicial: existe apenas um único vértice inicial em cada grafo de execução. Não apresenta vértices ascendentes, pois representa o ponto inicial do programa.
- Passagem: possuem apenas uma aresta de entrada e uma aresta de saída, representando assim partes do programa que não possuem desvios condicionais.
- Decisão: possuem um vértice ascendente e dois ou mais descendentes, sendo que cada vértice descendente representa um caminho possível a ser seguido. Este tipo de vértice representa os desvios condicionais.
- Agrupamento: possuem duas ou mais arestas ascendentes, e apenas uma descendente. Representa o final de um desvio condicional, onde são agrupados os ramos criados em seu início.
- Final: não possuem vértices descendentes, pois representam o final da execução do programa.

2.6.4 Descompilação

A descompilação é um processo feito de forma diferente para cada arquitetura de computador, pois necessita ser levado em conta o conjunto de instruções de máquina para o qual foi compilado, estrutura utilizada para sua geração e a quantidade esperada de ciclos de máquina consumida com cada instrução.

Com isso, obtém-se o grafo da seguinte forma:

1. Leitura do código executável;
2. Interpretação das instruções;
3. Agrupamento das instruções em fluxo contínuo.

Na primeira etapa lê-se o código executável e a partir dele é gerado o código em linguagem *assembly* desse programa. Nessa etapa as sub-rotinas são separadas, com um mapeamento dos endereços lógicos ao longo do programa.

Terminada essa etapa, passa-se para a etapa de geração do grafo, feita em conjunto com a terceira e última etapa do processo, até que todas as instruções e rotinas sejam analisadas. A segunda etapa pode ser feita com um mapeamento entre cada instrução e a ação tomada quando se encontra tal instrução.

As instruções podem ser agrupadas em três grupos distintos: saltos condicionais, saltos incondicionais e instruções que não são de salto.

Nesse método a interpretação das instruções deve ser feita de uma maneira seqüencial, assim quando o modelo encontra uma sub-rotina deve ser feita a interpretação de todo o corpo da sub-rotina antes que a próxima instrução seja analisada. Tal metodologia é adotada pois é preciso que se tenha um grafo conexo, uma vez que os programas são conexos.

A terceira e última etapa é agrupar em um único vértice seqüências de instruções em que não ocorrem desvios condicionais ou incondicionais. Assim, definem-se quais instruções devem ser agrupadas dentro de um único vértice e como esses vértices são conectados.

Portanto, para a geração do grafo deve-se levar em conta fatores como tratamento de desvios, tratamento de ciclos de repetição e chamadas de sub-rotina.

Tratamento de desvios

No momento da geração do grafo de execução deve-se levar em conta alguns aspectos com relação ao tratamento de desvios: estabelecimento da quantidade de arestas que saem dos vértices de decisão, determinação dos vértices de destino dessas arestas, determinação do endereço de retorno das chamadas de sub-rotinas, determinação do tempo de execução de cada ramo, estabelecimento de uma política de decisão para o caminho a ser seguido.

Sempre que possível os vértices de decisão devem ser agrupados, para que se possa determinar quantas arestas partem de cada vértice e para onde se dirigem.

Tratamento de ciclos de repetição

Um problema no tratamento de ciclos de repetição é determinar quantas vezes o ciclo será executado. Nesse caso o simulador utiliza uma função de distribuição de probabilidade (*fdp*) previamente definida para determinar quantas vezes o ciclo será executado.

Porém, o grande problema nesse caso é diferenciar um ciclo de repetição de uma estrutura de decisão. Para se diferenciar uma estrutura da outra é necessário observar o contador de programa, que em algum momento será decrementado para que se possa voltar ao início do ciclo. A identificação da instrução em que o ciclo começa e termina é feita com a identificação da instrução em que ocorre o decremento, seu endereço e o endereço para a qual o contador de programa foi desviado.

Chamadas de sub-rotinas

O tratamento das chamadas de sub-rotinas é feito de forma a manter o grafo conexo. Assim, quando é encontrada uma chamada para uma sub-rotina uma aresta sai do grafo corrente com destino ao grafo onde se encontra a sub-rotina e quando é encontrada a função de retorno cria-se uma aresta de saída na função de retorno para a próxima instrução do grafo corrente.

2.6.5 Otimização do grafo

Alguns critérios devem ser levados em consideração no momento em que será feita a otimização do grafo, como o tempo gasto em cada vértice e a precisão pretendida com a simulação. Isso ocorre porque a redução do número de vértices do grafo implica em resultados mais rápidos, porém, menos precisos.

Assim, é necessário estabelecer níveis de otimização, pois uma redução máxima implica em resultados menos precisos e uma redução mínima implica em uma precisão máxima. Pode-se ainda estabelecer níveis intermediários que serão escolhidos pelo usuário de acordo com seus interesses.

A otimização do grafo é baseada em técnicas utilizadas para a otimização de código em compiladores. Entretanto, tais técnicas não podem ser aplicadas a todos os conjuntos

de vértices do grafo. O objetivo é reduzir o número de vértices do grafo sem prejudicar o tempo consumido pelo grafo e conseqüentemente a avaliação do desempenho.

2.6.6 Simulação

Existe um módulo responsável pela simulação do grafo de execução, o qual é baseado em uma estrutura de controla sobre a ocorrência de eventos. Cada passagem de um vértice ao outro do grafo caracteriza um evento, ou seja, esse é gerado quando se percorre uma aresta.

Seu funcionamento se dá da seguinte maneira: um grafo de execução do programa é lido. Após a leitura do grafo, os dados de configuração da máquina são lidos pelo simulador, como número de processadores, carga, quantidade de processos, velocidade de processamento, entre outros.

O próximo passo consiste na criação de uma estrutura de controle para cada processo a ser executado. Informações sobre o estado de cada processo são armazenadas nessa estrutura, podendo assim um processo estar em execução, em comunicação ou em espera por sincronismo.

A etapa seguinte é responsável pelo atendimento de eventos, atualizando o tempo de execução simulado somando o tempo já decorrido durante a simulação. Logo depois, percorre-se o grafo até alcançar o vértice indicado pela aresta e também se atualiza a estrutura de controle de processos. Isso se repete até que todos os processos, com exceção dos encarregados pela comunicação, cheguem ao estado "encerrado". Até que todos os processos não estejam nesse estado, é escolhido o próximo evento a ser atendido. Tal escolha é dada pelo evento que possui o menor tempo de ocorrência entre todos os processos que estejam no estado "em execução".

Concluída a simulação, os dados sobre ela são coletados e analisados, como por exemplo, tempo de execução, *speedup*, tempo gasto no processamento, entre outros.

2.7 Considerações finais

Durante o decorrer desse capítulo foram apresentados conceitos relevantes sobre avaliação de desempenho. Foram discutidas medidas e considerações a serem feitas na escolha de cada medida e também foi mostrado que a escolha varia de acordo com a natureza do problema. Também foram abordadas algumas técnicas de modelagem e avaliação de

desempenho, além da técnica a ser utilizada para desenvolver esse projeto.

Um protótipo utilizando a técnica descrita em 2.6, implementado com a linguagem C, apresentou bons resultados mas falta a implementação de um gerador de grafo para diversas arquiteturas, inclusive para a família de processadores x86 que é a base desse projeto. Assim, será discutido no capítulo 3 a implementação da ferramenta para essa arquitetura.

3 Detalhamento e desenvolvimento do projeto

Neste capítulo será apresentada a metodologia e implementação de um pacote de classes Java capaz de gerar grafos de execução a partir de programas executáveis, na arquitetura x86. Na seção 3.1 serão abordados tópicos necessários para a compreensão da implementação. Na subseção 3.1.1 será feita uma breve introdução à arquitetura x86 e também mostrado porque ela é amplamente utilizada, justificando assim a criação de uma ferramenta para tal arquitetura. Na subseção 3.1.2 serão mostradas características da linguagem adotada para a implementação da ferramente proposta. Uma descrição sobre *disassembler* e seu funcionamento será feita na subseção 3.1.3, tendo destaque para a ferramenta que permite ler arquivos binários *objdump*. A seção 3.2 discutirá toda a implementação, envolvendo as ferramentas apresentadas nas seções já apresentadas.

3.1 Conceitos fundamentais

Nesta seção serão apontados alguns conceitos fundamentais que justificam a escolha da arquitetura e da linguagem de programação. Tais conceitos são importantes para que se compreenda a implementação do projeto de uma maneira melhor.

3.1.1 Porque o x86

IA-32 [30, 31, 32], algumas vezes genericamente chamado de x86-32, é a arquitetura de microprocessadores de maior sucesso da Intel [33]. Entre várias diretivas de linguagens de programação ela também é referenciada como i386. O termo pode ser utilizado para referenciar tanto as extensões de 32 bits como a arquitetura original x86, ou para a arquitetura como um todo. Essa arquitetura define o conjunto de instruções para a família de microprocessadores instalada na vasta maioria de computadores no mundo.

O termo significa *Intel Architecture 32-bit*, que se distingue das versões de 16 bits da arquitetura que precedeu, e da arquitetura de 64 bits IA-64 (que é muito diferente, apesar de ter um modo de compatibilidade com IA-32). O nome mais genérico para todas as versões de 16 e 32 bits dessa arquitetura é **x86**.

Esse conjunto de instruções foi introduzido no microprocessador 80386 em 1985, sendo ainda a base da maioria dos microprocessadores de PC, vinte anos mais tarde, em 2005. Ainda que o conjunto de instruções tenha permanecido intacto, as sucessivas gerações de microprocessadores tornaram-se mais rápidas na execução delas.

A arquitetura x86 tem se destacado durante as últimas décadas, mesmo que uma grande variedade de outras arquiteturas tenham desaparecido ou acabando sendo relegadas para nichos menores de mercados. A arquitetura continua evoluindo através de novas tecnologias, mais recentemente passando a utilizar 64 bits e *dual core*, com *multicore* em planejamento pelos fabricantes. Uma forte competição entre a Intel e AMD, junto com um contínuo progresso previsto pela lei de Moore sugere que as inovações continuarão em um ritmo constante. Considerando a enorme (e crescente) pesquisa e capital envolvidos no desenvolvimento e produção de um processador moderno (e compiladores, sistemas operacionais e outras ferramentas necessárias para fazê-lo útil), a arquitetura x86 provavelmente irá continuar a substituir processadores especializados em um grande número de áreas.

3.1.2 Java

A plataforma Java é o nome de um ambiente de computação, ou plataforma, da Sun Microsystems que pode executar aplicações desenvolvidas utilizando a linguagem de programação Java. Neste caso, a plataforma não é um *hardware* específico ou um sistema operacional, mas um *engine* de execução chamado de máquina virtual e um conjunto de bibliotecas padrão que provêem funcionalidades comuns.

Tecnologias Java

A plataforma Java consiste de um grande conjunto de tecnologias, cada uma provendo uma porção distinta do ambiente de desenvolvimento e execução. Por exemplo, usuários finais tipicamente interagem com a máquina virtual Java e com o conjunto padrão de bibliotecas de classes. Em adição, existem numerosas maneiras de uma aplicação Java ser liberada, incluindo a inclusão em páginas *web*. Ultimamente, desenvolvedores que

estejam criando aplicações para a plataforma utilizam um conjunto de ferramentas de desenvolvimento chamado de *Java Development Kit*.

Ambiente de execução Java

Um programa feito para a plataforma Java necessita que dois componentes estejam presentes: uma máquina virtual Java e um conjunto de biblioteca de classes que provê os serviços do qual o programa depende. A distribuição da Sun de sua Máquina Virtual Java (JVM - *Java Virtual Machine*) e de sua implementação de classes padrões é conhecida como Ambiente de Execução Java (JRE - *Java Runtime Environment*).

Máquina virtual Java

O coração da plataforma Java é o conceito de um processador "virtual" comum que execute programas *bytecode* Java. Esse *bytecode* é sempre o mesmo, não importanto o *hardware* ou o sistema operacional onde o programa esteja sendo executado. A plataforma Java provê um interpretador chamado JVM, que traduz em tempo de execução o *bytecode* Java em instruções nativas do processador. Isso permite que a mesma aplicação seja executada em qualquer plataforma que tenha disponível uma máquina virtual.

Java não foi a primeira plataforma baseada em máquina virtual, no entando é de longe a mais bem sucedida e conhecida. Antigas implementações de tecnologia de máquinas virtuais envolviam primariamente emuladores para ajudar no desenvolvimento de *hardwares* e sistemas operacionais ainda não desenvolvidos, mas o JVM foi especificado para ser implementado completamente em *software*, fazendo com que seja fácil de portar eficientemente uma implementação para *hardwares* de todas espécies.

3.1.3 *Disassembler*

Um *disassembler* é um programa de computador que traduz linguagem de máquina em código *assembly*, desempenhando a operação inversa de um montador. Um *disassembler* difere de um descompilador, pois mira uma linguagem de alto nível ao invés de uma linguagem de montagem. A saída de um *disassembler* é geralmente formatada para leitura humana ao invés de uma entrada específica para um *assembler*, fazendo dele uma ferramenta propícia para engenharia reversa.

Códigos fonte em linguagem *assembly* geralmente permitem o uso de constantes simbólicas e comentários do programador. Esses são geralmente removidos do código de

máquina final pelo *assembler*. Sendo assim, um *disassembler* operando em um código de máquina irá produzir um código em que estará faltando essas constantes e comentários; a saída do *disassembler* torna-se mais difícil para um humano interpretar do que o código fonte comentado e documentado.

Na etapa de obtenção do código *assembly* de um arquivo executável é utilizado o aplicativo *objdump*, que faz parte de um conjunto de ferramentas chamado *Binutils* [34]. Sua principal função é a de mostrar informações sobre um ou mais objetos. Através da opção *disassemble*, ele é capaz de exibir os mnemônicos das instruções de máquina, obtidos do objeto. Um exemplo da saída obtida pelo *objdump* pode ser visto na figura 3.1.

```
0804836d <main>:
 804836d:       55                      push   %ebp
 804836e:       89 e5                   mov    %esp,%ebp
 8048370:       83 ec 08                sub    $0x8,%esp
 8048373:       83 e4 f0                and    $0xfffffff0,%esp
 8048376:       b8 00 00 00 00          mov    $0x0,%eax
 804837b:       83 c0 0f                add    $0xf,%eax
 804837e:       83 c0 0f                add    $0xf,%eax
 8048381:       c1 e8 04                shr    $0x4,%eax
 8048384:       c1 e0 04                shl    $0x4,%eax
 8048387:       29 c4                   sub    %eax,%esp
 8048389:       e8 da ff ff ff          call   8048368 <func>
 804838e:       b8 01 00 00 00          mov    $0x1,%eax
 8048393:       c9                      leave
 8048394:       c3                      ret
 8048395:       90                      nop
```

Figura 3.1: Exemplo de saída obtida através do *objdump*.

3.2 Implementação

A obtenção do grafo de execução a partir de um programa executável é feita através das etapas distintas de leitura do código executável, interpretação das instruções de máquina e agrupamento de instruções em blocos sequenciais. Abaixo apresentamos detalhes referentes a implementação de cada etapa do processo.

3.2.1 Leitura do código executável

A etapa de leitura do código executável é a mais simples, comparando com as outras, e consiste em ler o código executável, instrução após instrução, e agrupar códigos em conjunto de rotinas. Em um próximo passo, separa-se cada uma dessas rotinas em arquivos individuais, criando juntamente uma tabela associativa, com a função de "ligar" cada rotina a um determinado arquivo.

Cada arquivo gerado é formatado de tal maneira que contenham:

- Endereço lógico da instrução;
- Código hexadecimal da instrução;
- Mnemônico para a instrução.

Assim como os arquivos contendo as rotinas do programa, a tabela associativa possui uma estrutura pré-definida.

- Nome da rotina;
- Nome do arquivo, que é dado pelo endereço lógico da rotina no programa.

Por trás da tarefa de realizar a desmontagem e formatação do código binário, de uma maneira que o arquivo resultante possa ser lido pelo programa escrito em Java, foi utilizado o *objdump*, presente em praticamente todos os sistemas *Unix-like*[1] como Solaris, FreeBSD e Linux, além de haver versões portadas dele para outros sistemas operacionais como Windows.

Após a saída gerada pelo *objdump*, o arquivo resultante é dividido em arquivos menores, cada um contendo uma rotina. Como descrito anteriormente, através desses arquivos separados e de uma tabela associativa, é feita a leitura de todo o código *assembly*. Os arquivos contendo as rotinas do programa serão lidos de acordo com a ordem em que forem necessários durante a execução do programa.

O arquivo obtido com a utilização do *objdump* é fragmentado através da utilização de uma classe Java implementada para esse fim. Assim, após o arquivo de saída ser gerado, o mesmo é fragmentado para que assim possa ser utilizado pelo decodificador. Essa

[1]Uma boa descrição de sistemas *Unix-like* pode ser obtida através do endereço http://en.wikipedia.org/wiki/Unix-like (disponível em Inglês).

classe utiliza uma estratégia de fragmentação para facilitar que se encontre o arquivo onde determinada rotina está localizada. Ela salva o arquivo contendo a rotina com o número do endereço inicial da rotina para facilitar na momento da leitura. Por exemplo, se determinada rotina tem início no endereço **8048c0f** o arquivo gerado conterá o mesmo nome desse endereço.

3.2.2 Interpretação das instruções

Após a separação das rotinas em vários arquivos, inica-se a análise de cada rotina separadamente, permitindo assim que trechos do grafo de execução já analisados sejam reutilizados. Tal procedimento é útil pois evita que a mesma função seja analisada várias vezes em chamadas diferentes dentro do programa. Assim, quando determinada função for chamada, mais de uma vez dentro do programa, será utilizado o mesmo grafo para efetuar a análise.

O processo inicia-se gerando o grafo para o arquivo contendo a função "start_", que é a função que inicia a execução do programa. Deve-se também gerar o grafo para as outras funções chamadas pela função "start_" inclusive para a função "main" (ou equivalente, dependendo da linguagem utilizada), onde começa o código escrito pelo programador. Esse processo é repetido até que todas as funções pertencentes ao programa tenham sido analisadas.

Para interpretar uma instrução contida no arquivo o programa lê um byte em hexadecimal (D8, por exemplo) e aplica uma função "hash" que mapeia o valor em hexadecimal para um valor inteiro. Tal procedimento é repetido para facilitar no momento da implementação, pois a linguagem Java oferece estruturas de decisão para os tipos primitivos da linguagem (inteiro e caracter). Assim, após o valor, em hexadecimal, ser mapeado para um valor inteiro é efetuada a decodificação para saber qual é a instrução correspondente aquele valor. Esse procedimento é adotado para todas as instruções do programa.

Como Java não trabalha explicitamente com ponteiros, é necessário criar uma classe com os atributos que serão utilizados e incluir referências à própria classe. Por exemplo, no código 3.2.1, a classe *Exemplo* possui dois "ponteiros", um chamado "esquerda" e outro "direita". Através do código 3.2.2, pode-se observar com mais clareza o funcionamento de tal mecanismo. Nas linhas 1 e 2, são declarados dois objetos do tipo *Exemplo*. Logo em seguida, na linha 3, é feito com que o atributo "esquerda" de *objeto1* referencie o *objeto2* e o atributo "direita" referencie *null* (nulo). Pode-se dizer portanto que o lado esquerdo do *objeto1* está apontando para o *objeto2* enquanto o lado direito não aponta para algum

lugar (figura 3.2).

```
1 public class Exemplo {
2         int numero;
3         Exemplo esquerda;
4         Exemplo direita;
5 }
```

Código 3.2.1: Exemplo de classe em Java.

```
1 Exemplo objeto1 = new Exemplo();
2 Exemplo objeto2 = new Exemplo();
3 objeto1.esquerda = objeto2;
4 objeto1.direita = null;
```

Código 3.2.2: Exemplo de referência em Java.

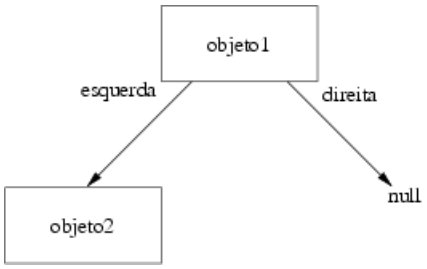

Figura 3.2: Representação de referências entre objetos.

Através da utilização de referência em Java conseguimos montar a estrutura necessária para a geração do grafo de execução. O mais importante é perceber que o grafo de execução do programa está organizado na forma de uma árvore de grafos e que cada nó da árvore contém um grafo de execução. Esse nó é basicamente formado por algumas informações gerais sobre a função, além dos apontadores para o grafo a esquerda e o grafo a direita, ou seja, os nós vizinhos desse grafo na árvore.

Na tabela de vértices são armazenados todos os vértices de um grafo com informações sobre o tipo de vértice (chamada de função, retorno de função, execução), endereço inicial e final do vértice e apontadores para os vértices sucessores.

O processo de geração do grafo passa a ser a interpretação de todas as instruções até que se tenha o fim de um bloco, ou seja, se tenha determinado um novo vértice do grafo. Nesse caso a estrutura do grafo é atualizada com a inserção de um ou mais vértices na tabela de vértices.

3.2.3 Agrupamento em blocos

No momento da interpretação das instruções se pode perceber que o agrupamento das instruções em blocos com execução sequencial pode ser uma tarefa útil, pois se todo o processo fosse realizado sem qualquer tipo de agrupamento seria necessário uma quantidade maior de memória, além de ser necessário o armazenamento exato da sequência de instruções do programa. Assim, agrupar as instruções diminui o consumo de memória pelo programa sem aumentar o tempo total de processamento.

O agrupamento em blocos é controlado pelo tipo de instrução que estiver sendo lida. O resultado da interpretação vai determinar se a instrução é de salto ou não. Quando a instrução for classificada como sendo de salto são determinados os endereços possíveis de desvio.

Assim, para instruções de chamada de função são determinados dois endereços, o da função chamada e o endereço de retorno da função. Para instruções de retorno de função nenhum endereço precisa ser calculado, apenas é recuperado o endereço da lista de caminhos ainda não percorridos.

Se a instrução for um desvio condicional são calculados dois endereços, o primeiro é o endereço de desvio quando o teste obtiver sucesso e o outro o endereço da próxima instrução.

O agrupamento de instruções não prejudica o resultado da simulação. Na figura 3.3 percebemos que se o simulador considerar um tempo *t1* para o primeiro vértice e *t2* para o segundo vértice, o tempo total gasto seria *t1+t2*, que é o tempo atribuído para o vértice *V2* após o agrupamento.

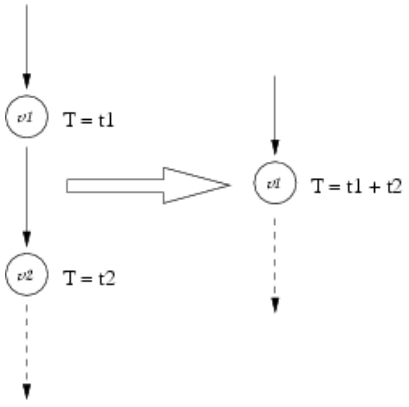

Figura 3.3: Agrupamento de vértices de passagem.

3.2.4 Geração do grafo de execução

Um grafo de execução é um grafo orientado que apresenta todos os possíveis caminhos que o programa pode seguir durante uma instância de execução, como pode ser visto na figura 3.4. As arestas desse grafo indicam os caminhos que podem ser seguidos, enquanto os vértices definem os pontos de processamento.

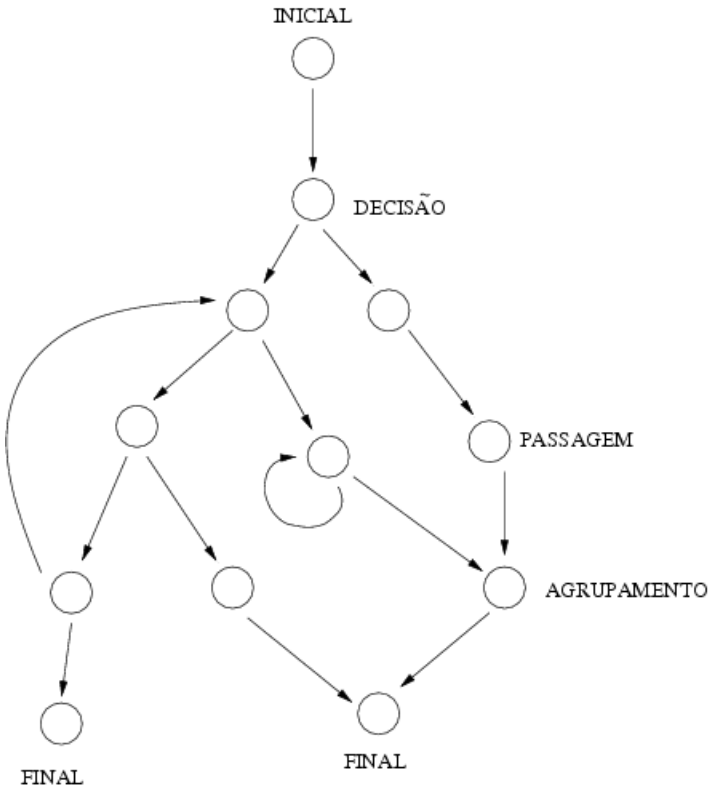

Figura 3.4: Exemplo de um grafo de execução de um programa.

O objetivo dessa técnica é reescrever o código executável a partir de suas instruções de máquina, medindo os tempos consumido por cada instrução.

As funções são armazenadas em um tipo de tabela *hash*, utilizando o endereço da função como índice. Essa estrutura pode ser vista pela figura 3.5.a. Cada nó (figura 3.5.b) aponta para um grafo do programa, sendo que a organização desse grafo é dada num formato de árvore de grafos, com os vértices apontados em uma espécie de tabela de vértices (figura 3.5.c), que contém informações sobre todos os vértices de cada grafo, como endereço inicial, apontadores e dados de sincronismo ou comunicação.

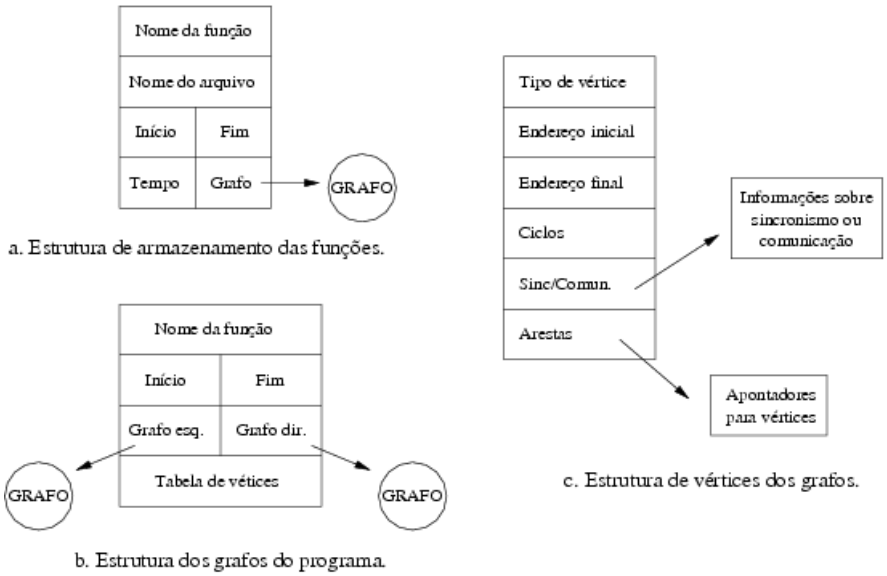

Figura 3.5: Estruturas utilizadas para armazenagem do grafo de execução.

4 Testes e resultados

Neste capítulo serão apresentados testes feitos com o gerador de grafos de execução, tendo utilizado para isso dois programas executáveis. Cabe ressaltar que o único objetivo dos testes foi averigüar se o grafo de execução estava sendo gerado de forma adequada. Como o objetivo desse projeto é a obtenção de um grafo de execução, não foram feitas simulações envolvendo os grafos gerados.

4.1 Exemplos de teste

Para ambos os testes apresentados nesta seção, serão utilizados pequenos programas já compilados. Cada um desses programas foi lido, individualmente, pelo *objdump*, chamado à partir da ferramenta desenvolvida. A saída obtida pelo *objdump* é então redirecionada a um arquivo, que é posteriormente lido pelo gerador de grafo.

Esse arquivo de saída é então separado em vários arquivos pelo programa desenvolvido. Cada arquivo recebe um nome que contém o endereço da função a que corresponde. O arquivo inicial gerado pelo *objdump* termina divido em vários pequenos arquivos portanto.

O próximo passo executado é a leitura do primeiro arquivo da tabela de arquivos. É feita uma leitura de forma que cada instrução seja identificada. Encerrada a análise das funções, o grafo que foi sendo gerado à medida em que se interpretavam as instruções é armazenado em um arquivo contendo a tabela de arquivos e a relação dos grafos produzidos pelo programa.

Como exemplos de teste neste capítulo foram utilizados quatro programas executáveis distintos e já compilados, cujos códigos fonte não serão mostrados pois o gerador de grafo de execução pode ser utilizado em qualquer programa executável compilado para a família x86. Nas figuras 4.1, 4.3, 4.5 e 4.7 são apresentados os arquivos de exemplo gerados pelo gerador de grafo. Nas figuras 4.2, 4.4, 4.6 e 4.8 são apresentadas a relação dos vértices de cada grafo do programa, lembrando que cada função do programa executável gera um

grafo distinto.

Na figura 4.1 são mostradas as funções obtidas do primeiro programa executado no gerador de grafo. A saída está formatada de modo a oferecer algumas informações relevantes ao usuário. Na primeira coluna temos o nome do arquivo em que está localizada a rotina, cujo nome está localizado na segunda e terceira coluna. Na última coluna é fornecido o endereço inicial da função.

```
Funcoes do programa binario
      f_8048278                      _init                      8048278
      f_8048290                       .plt                      8048290
      f_80482c0                     _start                      80482c0
      f_80482e4           call_gmon_start                       80482e4
      f_8048310     __do_global_dtors_aux                       8048310
      f_8048350             frame_dummy                         8048350
      f_8048384                    main                         8048384
      f_804846f                  funcao                         804846f
      f_8048480           __libc_csu_init                       8048480
      f_80484e0           __libc_csu_fini                       80484e0
      f_8048530    __i686.get_pc_thunk.bx                       8048530
      f_8048540     __do_global_ctors_aux                       8048540
      f_8048570                    _fini                        8048570
Numero de arquivos: 13
```

Figura 4.1: Saída obtida pelo gerador de grafo (funções e arquivos).

Na figura 4.2 é mostrado um arquivo de saída do gerador de grafo com os grafos e suas respectivas tabelas de vértices. Nos exemplos utilizados algumas tabelas de vértices foram simplificadas e alguns grafos omitidos devido a grande quantidade de informação produzida pelo gerador de grafo. Como mencionado anteriormente, cada função gera um grafo com sua respectiva tabela de vértice. Assim, é fornecido ao usuário o endereço inicial do grafo e sua tabela de vértice em seguida, como pode ser observado na figura 4.2. Na primeira coluna da tabela de vértices é impresso a *string* "vtx" que serve apenas para indicar o início de um novo vértice. A segunda coluna indica o tipo de vértice gerado. A terceira e a quarta coluna representam o endereço inicial e final do vértice, respectivamente. Logo em seguida temos o total de ciclos de máquina consumidos pela instrução do vértice e as arestas para os vértices sucessores. Os vértices podem possuir 0, 1 ou 2 arestas, dependendo do tipo do vértice em questão. Assim, 0 arestas representa um vértice final, 1 aresta representa vértices inicial, de passagem ou agrupamento e 2 arestas representam vértices de decisão.

Os demais exemplos não serão comentados pois o esquema de entendimento segue o mesmo padrão utilizado no exemplo anterior e ficam somente a carater ilustrativo da ferramenta.

Vale citar que o arquivo gerado contém a tabela de arquivos e a relação de árvores. É apresentado o arquivo separado apenas para melhor visualização.

Cada árvore possui uma lista dos vértices e arestas correspondentes. A identificação dos vértices é dada pelo tipo (salto, retorno de função, entre outros), tempo de execução e endereço inicial dos vértices.

Finalizada essa etapa, o arquivo gerado pelo programa está pronto para ser utilizado em uma simulação (onde poderá se obter estimativas e análises de desempenho do programa).

```
Grafo: 8048278
  vtx  6   8048278   804827d   0   804827e      null
  vtx  5   804827e   8048282   2   80482e4      null
  vtx  5   8048283   8048287   2   8048350      null
  vtx  5   8048288   804828c   2   8048540      null
  vtx  6   804828d   804828d   3   804828e      null
  vtx  7   804828e             3    return
Grafo: 8048290
  vtx  6   8048290   8048295   3   8048296      null
  vtx  4   8048296   804829c   1   804829e      null
  vtx  6   804829e   804829f   2   80482a0      null
  vtx  4   80482a0   80482a6   1   80482ab      null
  vtx  4   80482ab   80482b0   1   8048290      null
  vtx  6   80482b6   80482ba   1      null      null
Grafo: 80482c0
  vtx  6   80482c0   80482db  12   80482dc      null
  vtx  5   80482dc   80482e0   2   80482a0      null
  vtx  6   80482e1   80482e3  12      null      null
...
Grafo: 8048350
  vtx  6   8048350   804835c  15   804835d      null
  vtx  3   804835d   804835e   3   804835f   8048380
  vtx  6   804835f   8048365   6   8048366      null
  vtx  3   8048366   8048367   3   8048368   8048380
  vtx  6   8048368   804836e   3   804836f      null
  vtx  5   804836f   8048373   2   8048374      null
  vtx  6   8048374   804837f   2   8048380      null
  vtx  6   8048380   8048382   2   8048383      null
  vtx  7   8048383             1    return
Grafo: 8048480
  vtx  6   8048480   804848a  21   804848b      null
  vtx  5   804848b   804848f   2   8048530      null
  vtx  6   8048490   8048495   2   8048496      null
  vtx  5   8048496   804849a   2   804849b      null
  vtx  6   804849b   80484ad   6   80484ae      null
  vtx  3   80484ae   80484af   2   80484b0   80484cc
  vtx  6   80484b0   80484bf   4   80484c0      null
  vtx  5   80484c0   80484c6   1   80484c7      null
  vtx  5   80484c7   80484c7   1   80484c8      null
  vtx  6   80484c8   80484c9   2   80484ca      null
  vtx  3   80484ca   80484cb   2   80484cc   80484c0
  vtx  6   80484cc   80484d2  10   80484d3      null
  vtx  7   80484d3             2    return
```

Figura 4.2: Saída obtida pelo gerador de grafo (vértices).

```
Funcoes do programa binario
      f_8048330                      _init                      8048330
      f_8048348                       .plt                      8048348
      f_80483d0                     _start                      80483d0
      f_80483f4           call_gmon_start                      80483f4
      f_8048420     __do_global_dtors_aux                      8048420
      f_8048460             frame_dummy                      8048460
      f_8048494                      main                      8048494
      f_8048625                   invalido                     8048625
      f_80486b1                       cont                     80486b1
      f_80486be                   endereco                     80486be
      f_8048737                     classe                     8048737
      f_80487fc                    formato                     80487fc
      f_804886e                  proposito                     804886e
      f_80489c8                       bits                     80489c8
      f_8048a7d                  intervalo                     8048a7d
      f_8048bd7                     numero                     8048bd7
      f_8048d31                      hosts                     8048d31
      f_8048e80           __libc_csu_init                      8048e80
      f_8048ee0           __libc_csu_fini                      8048ee0
      f_8048f30    __i686.get_pc_thunk.bx                      8048f30
      f_8048f40     __do_global_ctors_aux                      8048f40
      f_8048f70                      _fini                      8048f70
Numero de arquivos: 22
```

Figura 4.3: Outro exemplo de saída obtida pelo gerador de grafo (funções e arquivos).

```
Grafo: 8048330
  vtx  6   8048330   8048335   0   8048336      null
  vtx  5   8048336   804833a   2   80483f4      null
  vtx  5   804833b   804833f   2   8048460      null
  vtx  5   8048340   8048344   2   8048f40      null
  vtx  6   8048345   8048345   3   8048346      null
  vtx  7   8048346             3    return
...
Grafo: 8048460
  vtx  6   8048460   804846c  15   804846d      null
  vtx  3   804846d   804846e   3   804846f   8048490
  vtx  6   804846f   8048475   6   8048476      null
  vtx  3   8048476   8048477   3   8048478   8048490
  vtx  6   8048478   804847e   3   804847f      null
  vtx  5   804847f   8048483   2   8048484      null
  vtx  6   8048484   804848f   2   8048490      null
  vtx  6   8048490   8048492   2   8048493      null
  vtx  7   8048493             1    return
...
Grafo: 8048e80
  vtx  6   8048e80   8048e8a  21   8048e8b      null
  vtx  5   8048e8b   8048e8f   2   8048f30      null
  vtx  6   8048e90   8048e95   2   8048e96      null
  vtx  5   8048e96   8048e9a   2   8048e9b      null
  vtx  6   8048e9b   8048ead   6   8048eae      null
  vtx  3   8048eae   8048eaf   2   8048eb0   8048ecc
  vtx  6   8048eb0   8048ebf   4   8048ec0      null
  vtx  5   8048ec0   8048ec6   1   8048ec7      null
  vtx  5   8048ec7   8048ec7   1   8048ec8      null
  vtx  6   8048ec8   8048ec9   2   8048eca      null
  vtx  3   8048eca   8048ecb   2   8048ecc   8048ec0
  vtx  6   8048ecc   8048ed2  10   8048ed3      null
  vtx  7   8048ed3             2    return
```

Figura 4.4: Outro exemplo de saída obtida pelo gerador de grafo (vértices).

```
Funcoes do programa binario
      f_8048480                    _init                    8048480
      f_8048498                     .plt                    8048498
      f_8048590                   _start                    8048590
      f_80485b4           call_gmon_start                    80485b4
      f_80485e0     __do_global_dtors_aux                    80485e0
      f_8048620              frame_dummy                    8048620
      f_8048654                     main                    8048654
      f_804867a                bcp_start                    804867a
      f_80486e7                rand_time                    80486e7
      f_8048748                     menu                    8048748
      f_8048870                  inserir                    8048870
      f_80489b4                finalizar                    80489b4
      f_8048b06                 bloquear                    8048b06
      f_8048c0a               desbloquear                    8048c0a
      f_8048d0e                visualizar                    8048d0e
      f_8048e0d                  continua                    8048e0d
      f_8048e1f                nao_existe                    8048e1f
      f_8048e51                    inicia                    8048e51
      f_804904a                   faz_es                    804904a
      f_8049200              insere_lista                    8049200
      f_80492b9                   getnode                    80492b9
      f_80492d3                      park                    80492d3
      f_80492e2              remove_lista                    80492e2
      f_804939d              disk_control                    804939d
      f_80493a2                 algorithm                    80493a2
      f_804945c                    vis_es                    804945c
      f_804951c                 entra_sai                    804951c
      f_8049590           __libc_csu_init                    8049590
      f_80495f0           __libc_csu_fini                    80495f0
      f_8049640    __i686.get_pc_thunk.bx                    8049640
      f_8049650     __do_global_ctors_aux                    8049650
      f_8049680                    _fini                    8049680
Numero de arquivos: 32
```

Figura 4.5: Outro exemplo de saída obtida pelo gerador de grafo (funções e arquivos).

```
Grafo: 8048480
  vtx  6   8048480   8048485  10   8048486      null
  vtx  5   8048486   804848a   2   80485b4      null
  vtx  5   804848b   804848f   2   8048620      null
  vtx  5   8048490   8048494   2   8049650      null
  vtx  6   8048495   8048495   3   8048496      null
  vtx  7   8048496             3    return
...
Grafo: 8048620
  vtx  6   8048620   804862c  15   804862d      null
  vtx  3   804862d   804862e   3   804862f   8048650
  vtx  6   804862f   8048635   6   8048636      null
  vtx  3   8048636   8048637   3   8048638   8048650
  vtx  6   8048638   804863e   3   804863f      null
  vtx  5   804863f   8048643   2   8048644      null
  vtx  6   8048644   804864f   2   8048650      null
  vtx  6   8048650   8048652   2   8048653      null
  vtx  7   8048653             1    return
...
Grafo: 8048e0d
  vtx  6   8048e0d   8048e12   9   8048e13      null
  vtx  5   8048e13   8048e17   2   80484a8      null
  vtx  5   8048e18   8048e1c   2   80484a8      null
  vtx  6   8048e1d   8048e1d   3   8048e1e      null
  vtx  7   8048e1e             3    return
...
Grafo: 80492b9
  vtx  6   80492b9   80492c5  12   80492c6      null
  vtx  5   80492c6   80492ca   2   80492cb      null
  vtx  6   80492cb   80492d1   7   80492d2      null
  vtx  7   80492d2             3    return
...
Grafo: 8049680
  vtx  6   8049680   8049683   3   8049684      null
  vtx  5   8049684   8049688   2   8049689      null
  vtx  6   8049689   8049690   6   8049691      null
  vtx  5   8049691   8049695   2   80485e0      null
  vtx  6   8049696   8049699   5   804969a      null
  vtx  7   804969a             3    return
```

Figura 4.6: Outro exemplo de saída obtida pelo gerador de grafo (vértices).

```
Funcoes do programa binario
      f_80483e4                    _init                    80483e4
      f_80483fc                     .plt                    80483fc
      f_80484c0                   _start                    80484c0
      f_80484e4           call_gmon_start                   80484e4
      f_8048510     __do_global_dtors_aux                   8048510
      f_8048550              frame_dummy                    8048550
      f_8048584                     main                    8048584
      f_80485af               zera_disco                    80485af
      f_80485e4                bcp_start                    80485e4
      f_804863e                     menu                    804863e
      f_8048766                   inserir                   8048766
      f_804895d                finalizar                    804895d
      f_8048ab3                 bloquear                    8048ab3
      f_8048bbd              desbloquear                    8048bbd
      f_8048cc7               visualizar                    8048cc7
      f_8048daa                 continua                    8048daa
      f_8048dbc               nao_existe                    8048dbc
      f_8048dee              disk_control                   8048dee
      f_8048df3                 algorithm                   8048df3
      f_8048df8                      park                   8048df8
      f_8048e07              apaga_disco                    8048e07
      f_8048e9f                 vis_disco                   8048e9f
      f_8048f1b              insere_lista                   8048f1b
      f_80490ae              remove_lista                   80490ae
      f_804914b                   getnode                   804914b
      f_8049165                 vis_lista                   8049165
      f_80491d0           __libc_csu_init                   80491d0
      f_8049230           __libc_csu_fini                   8049230
      f_8049280    __i686.get_pc_thunk.bx                   8049280
      f_8049290     __do_global_ctors_aux                   8049290
      f_80492c0                     _fini                   80492c0
Numero de arquivos: 31
```

Figura 4.7: Outro exemplo de saída obtida pelo gerador de grafo (funções e arquivos).

```
...
Grafo: 80484c0
  vtx  6   80484c0   80484db  12   80484dc      null
  vtx  5   80484dc   80484e0   2   80484e1      null
  vtx  6   80484e1   80484e3  12      null      null
Grafo: 80484e4
  vtx  6   80484e4   80484e7  12   80484e8      null
  vtx  5   80484e8   80484ec   2   80484ed      null
  vtx  6   80484ed   80484fc  10   80484fd      null
  vtx  3   80484fd   80484fe   2   80484ff   8048501
  vtx  5   80484ff   8048500   2   8048501      null
  vtx  6   8048501   8048504   5   8048505      null
  vtx  7   8048505             3    return
...
Grafo: 8048550
  vtx  6   8048550   804855c  15   804855d      null
  vtx  3   804855d   804855e   3   804855f   8048580
  vtx  6   804855f   8048565   6   8048566      null
  vtx  3   8048566   8048567   3   8048568   8048580
  vtx  6   8048568   804856e   3   804856f      null
  vtx  5   804856f   8048573   2   8048574      null
  vtx  6   8048574   804857f   2   8048580      null
  vtx  6   8048580   8048582   2   8048583      null
  vtx  7   8048583             1    return
...
Grafo: 8048daa
  vtx  6   8048daa   8048daf   9   8048db0      null
  vtx  5   8048db0   8048db4   2   804840c      null
  vtx  5   8048db5   8048db9   2   804840c      null
  vtx  6   8048dba   8048dba   3   8048dbb      null
  vtx  7   8048dbb             3    return
...
Grafo: 8048dee
  vtx  6   8048dee   8048df1   9   8048df2      null
  vtx  7   8048df2             3    return
Grafo: 8048df3
  vtx  6   8048df3   8048df6   9   8048df7      null
  vtx  7   8048df7             3    return
Grafo: 8048df8
  vtx  6   8048df8   8048e05  13   8048e06      null
  vtx  7   8048e06             2    return
...
```

Figura 4.8: Outro exemplo de saída obtida pelo gerador de grafo (vértices).

5 *Conclusões e perspectivas*

Este capítulo apresenta as conclusões que puderam ser tiradas do desenvolvimento desse trabalho. Assim, serão enunciados os pontos relevantes e suas principais contribuições. Também são apresentados futuros trabalhos que podem ser desenvolvidos em cima da ferramenta aqui implementada.

5.1 Conclusões gerais

O gerador do grafo de execução foi implementado como planejado, testado e verificado que é capaz de ler e identificar corretamente instruções da arquitetura x86. A ferramenta foi capaz de gerar grafos de execução de forma adequada, fazendo com que os objetivos do projeto fossem alcançados e satisfeitos.

Cada vez mais a análise de desempenho vem crescendo tanto entre a comunidade acadêmica quanto entre a comunidade que visa o desenvolvimento de aplicações comerciais. Assim, essa é uma área bastante promissora, com destaque especial para a metodologia aqui proposta, pois essa elimina a necessidade de inserção de código adicional ao programa e dispensa a disponibilidade do uso da máquina simulada.

O desenvolvimento desse trabalho contribuiu para a aprendizagem da linguagem Java, que é altamente poderosa e o seu uso vem crescendo a cada ano. Com certeza essa linguagem se tornará uma das mais utilizadas no mundo inteiro e o aprendizado contribuiu de forma significativa em nossa formação profissional.

Outro proveito tirado desse trabalho foi um conhecimento mais aprofundado da arquitetura x86, que é a arquitetura mais utilizada em todo o mundo e da linguagem *assembly* para a mesma. Isso possibilitou uma visão mais ampla do funcionamento do processador que habitualmente se utiliza, mas do qual não se tem um conhecimento aprofundado.

5.2 Perspectivas de trabalhos futuros

O processo de predição de desempenho por simulação do grafo de execução traz consigo diversas vantagens que foram relacionadas ao longo desse documento. Assim, é interessante que se tenha a ferramenta completa de predição de desempenho.

Como trabalhos futuros se faz necessário a implementação de um *disassembler* para programas binários, pois a utilização dos programas utilizados neste projeto podem não estar disponíveis em determinados sistemas. A implementação de tal ferramenta auxiliaria na portabilidade da ferramenta, uma vez, que a mesma não dependeria de programas instalados no sistema operacional, passando a utilizar somente a máquina virtual Java (JVM) necessária para executar o programa compilado.

O grafo gerado pela ferramenta desenvolvida não é mínimo, assim também se faz necessário a implementação de uma versão em Java para o protótipo do otimizador do grafo de execução, uma vez que a utilização do grafo mínimo proporciona ao simulador um melhor controle sobre o grafo e um menor consumo de memória pela ferramenta.

Outro trabalho que se torna interessante neste momento é a implementação em Java do simulador do grafo de execução, uma vez que a ferramenta implementada gera o grafo de execução mas não o simula e a simulação é uma parte do método proposto por Manacero. Cabe ressaltar que já existe uma implementação em C para as arquiteturas MIPS e SPARC de todo o método apresentado, contendo o gerador de grafo de execução e o simulador.

Assim, com a implementação e a integração das ferramentas descritas acima teria-se o método totalmente implementado e independente de programas previamente disponíveis no sistema operacional.

APÊNDICE A – A arquitetura IA-32 Intel

As codificações das instruções IA-32 são um subconjunto do formato apresentado na figura A.1. Instruções consistem de prefixos opcionais de instrução (em qualquer ordem), bytes primários do *opcode* (até três bytes), um especificador de endereçamento (se requerido) consistindo do byte ModR/M e algumas vezes do byte SIB (*Scale-Index-Base*), um deslocamento (se requerido) e um campo imediato de dados (se requerido).

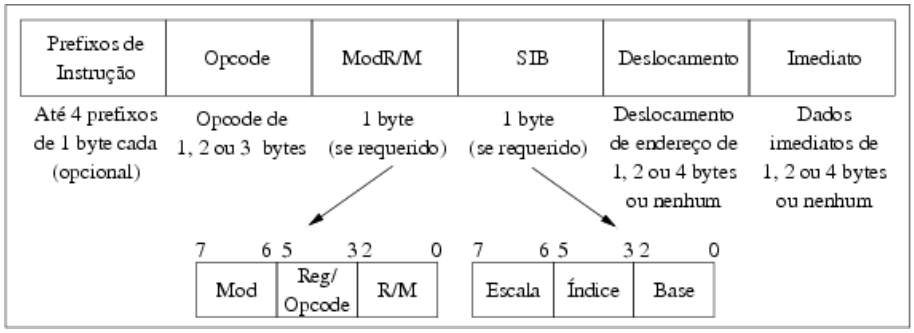

Figura A.1: Formato de instruções IA-32.

A.1 Prefixos de instruções

Prefixos de instruções são divididos em quatro grupos, cada um com códigos de prefixos permitidos. Para cada instrução, um prefixo pode ser utilizado de cada um dos quatro grupos (Grupos 1, 2, 3, 4) e ser disposto em qualquer ordem.

A.1.1 Grupo 1

Prefixos de *lock* e repetição:

- F0H – LOCK;

- F2H – REPNE/REPNZ (usado apenas com instruções de *string*; quando utilizado com o *opcode* 0FH, esse prefixo é tratado como um prefixo mandatório para algumas instruções SIMD);
- F3H – REP ou REPE/REPZ (usado apenas com instruções de *string*; quando utilizado com o *opcode* 0FH, esse prefixo é tratado como um prefixo mandatório para algumas instruções SIMD).

A.1.2 Grupo 2

Prefixos de *override* de segmento:

- 2EH – *override* de segmento CS (uso com qualquer instrução de ramificação é reservado);
- 36H – prefixo de *override* de segmento SS (uso com qualquer instrução de ramificação é reservado);
- 3EH – prefixo de *override* de segmento DS (uso com qualquer instrução de ramificação é reservado);
- 26H – prefixo de *override* de segmento ES (uso com qualquer instrução de ramificação é reservado);
- 64H – prefixo de *override* de segmento FS (uso com qualquer instrução de ramificação é reservado);
- 65H – prefixo de *override* de segmento GS (uso com qualquer instrução de ramificação é reservado).

Ramificação:

- 2EH – *Branch not taken* (usado apenas com instruções Jcc);
- 3EH – *Branch taken* (usado apenas com instruções Jcc).

A.1.3 Grupo 3

- 66H – Prefixo de *override* de tamanho do operando (quando utilizado com o *opcode* 0FH, esse prefixo é tratado como um prefixo mandatório para algumas instruções SIMD).

A.1.4 Grupo 4

- 67H – Prefixo de *override* de tamanho de endereço.

A.2 *Opcodes*

Um *opcode* primário pode possuir tamanho de 1, 2 ou 3 bytes. Um campo adicional de 3 bits é alguma vezes codificado no byte ModR/M. Campos menores podem ser definidos dentro do *opcode* primário. Tais campos definem a direção da operação, tamanho do deslocamento, codificação do registrador, códigos de condição ou extensão de sinal. Campos de codificação utilizados por um *opcode* variam dependendo da classe de operação.

Os formatos de *opcode* de dois bytes para uso geral e instruções SIMD consistem de:

- Um byte de *opcode* 0FH como o *opcode* primário e um segundo byte de *opcode*;
- Um prefixo mandatório (66FH, F2H, F3H), um byte de *opcode* e um segundo byte de *opcode*.

Por exemplo, CVTDQ2PD consiste da seguinte sequência: F3 OF E6. O primeiro byte é um prefixo mandatório para instruções SSE/SSE2/SSE3. Note que todos os três bytes do *opcode* são reservados.

A.3 Bytes ModR/M e SIB

Muitas instruções que se referem a um operando em memória têm um byte especificador de forma de endereçamento (chamado de byte ModR/M), seguindo o *opcode* primário. O byte ModR/M contém três campos de informação:

- O campo *mod* combina com o campo *r/m* para formar 32 possíveis valores: oito registradores e 24 modos de endereçamento.
- O campo *reg/opcode* especifica tanto um númeto de registrador ou mais três bits de informação de *opcode*. A função do campo *reg/opcode* é especificada no *opcode* primário.
- O campo *r/m* pode especificar um registrador como um operando ou pode ser combinado com o campo *mod* para codificar um modo de endereçamento. Algumas vezes,

certas combinações dos campos *mod* e *r/m* são usadas para fornecer informações do *opcode* para algumas instruções.

Certas codificações do byte ModR/M requerem um segundo byte de endereçamento (o byte SIB). Os formatos de endereçamento de 32 bits *base-plus-index* e *scale-plus-index* requerem o byte SIB. O byte SIB inclui os seguintes campos:

- O campo de escala especifica o fator de escala.
- O campo de índice especifica o número do registrador do registrador índice.
- O campo base especifica o número do registrador do registrador base.

A.4 Bytes Deslocamento e Imediato

Alguns métodos de endereçamento incluem um deslocamento imediatamente após o byte ModR/M (ou o byte SIB se estiver presente). Se um deslocamento é requerido, ele pode ser de 1, 2 ou 4 bytes.

Se uma instrução especifica um operador imediato, o operando sempre segue qualquer byte de deslocamento. Um operando imediato pode ser de 1, 2 ou 4 bytes.

Referências

[1] MANACERO, A. J. *Predição do Desempenho de Programas Paralelos por Simulação do Grafo de execução.* Tese (Doutorado) — Unicamp, 1997.

[2] MORAES, M. D. *Geração do Grafo de Execução para Análise de Desempenho de Programas Paralelos.* Monografia (Projeto Final) — Unesp, 1999.

[3] CASAVANT, T. L.; TVRDÍK, P.; PLÁSIL, F. *Parallel Computers: Theory and Practice.* [S.l.]: IEEE, 1996.

[4] JAIN, R. *The Art of Computer Systems Performance Analysis: Techniques for experimental design, measurement, simulation and modeling.* [S.l.]: Wiley, 1991.

[5] MEDIDAS de Desempenho (Análise de programas paralelos). Disponível em: <http://www.inf.puc−rio.br/∼noemi/victal/medese.html>. Acesso em: 01/09/2005.

[6] DESIGNING and Building Parallel Programs. Disponível em: <http://www−unix.mcs.anl.gov/dbpp/text/book.html>. Acesso em: 01/09/2005.

[7] PARALLEL Programming in C. Disponível em: <http://maven.smith.edu/∼thiebaut/transputer/toc.html>. Acesso em: 01/09/2005.

[8] CULLER, D. E.; SINGH, J. P. *Parallel Computer Architecture: A Hardware/Software Approach.* [S.l.]: Morgan Kaufmann Publishers, Inc., 1998.

[9] GUSTAFSON, J. et al. *The design of a scalable, fixed-time computer benchmark.* [S.l.]: Journal of Parallel and Distributed Computing, 1991.

[10] MEIRA, W. J. Modeling performance of parallel programs. Rochester, New York, 1995.

[11] LAINE, J. M. *Desenvolvimento de Modelos para Predição de Desempenho de Programas Paralelos MPI.* Dissertação (Mestrado) — USP, 2003.

[12] MARCARI, E. J. *Classificação e Comparação de Ferramentas para Análise de Desempenho de Sistemas Paralelos.* Dissertação (Mestrado) — Unesp, 2002.

[13] LINPACK. Disponível em: <http://www.netlib.org/linpack/>. Acesso em: 01/09/2005.

[14] LAPACK. Disponível em: <http://www.netlib.org/lapack/>. Acesso em: 01/09/2005.

[15] CALZAROSSA, M. et al. Parallel performance evaluation: The medea tool. *Intl. conf. and Exhibition on High-Performance Computing and Networking*, Brussels, Bélgica, 1996.

[16] HONDROUDAKIS, A.; PROCTER, R.; SHANMUGAM, K. Performance evaluation and visualization with vispat. *Third Intl. Conf. on Parallel Computing Technologies*, St. Petersburg, Russia, 1995.

[17] KITAJIMA, J. P.; PLATEAU, B. *Modelling parallel program behaviour in ALPES.* [S.l.]: Information and Software Techonology, 1994.

[18] VERMURI, R.; MANDAYAM, R.; MEDURI, V. Performance modelling using pdl. *IEEE Computer*, 1996.

[19] SARUKKAI, S. R.; MEHRA, P.; BLOCK, R. J. Automated scalability analysis of message passing parallel progams. *IEEE Parallel and Distributed Technology*, 1995.

[20] PEASE, D. Paws: a performance evaluation tool for parallel computing systems. *IEEE Computer*, 1991.

[21] THE RSIM Project. Disponível em: <http://rsim.cs.uiuc.edu/rsim/>. Acesso em: 01/09/2005.

[22] VIRTUTECH Simics. Disponível em: <http://www.virtutech.com/products/>. Acesso em: 01/09/2005.

[23] MARSAN, M. A.; BALBO, G.; CONTE, G. *A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems.* [S.l.]: ACM Trans. Comput. Sys., 1984.

[24] ZUBEREK, W. M. Performance evaluation using unbounded timed petri nets. *Proc. of the Third Intl. Workshop on Petri nets and Performance models*, Kyoto, Japan, 1989.

[25] THOMASIAN, A.; BAY, P. F. Analytic queueing network models for paraller processing of tasks systems. *IEEE Trans. on Computers*, 1986.

[26] GANDRA, M.; DRAKE, J. M.; GREGORIO, J. A. Performance evaluation of parallel systems by using unbounded generalized stochastic petri nets. *IEEE Trans. on Software Engineering*, 1992.

[27] JKIM, J.; SHIN, K. G. Execution time analysis of communicating tasks in distributed systems. *IEEE Trans. on Computers*, 1996.

[28] ADVE, V. S. *Analyzing the behavior and perfomance of paraller programs.* Tese (Doutorado) — Universidade de Wisconsin, 1993.

[29] HERZOG, U. Formal description, time and performanace analysis. *Entwurf und Betrieb Vertwilter Systeme*, Verlag, Berlin, 1990.

[30] INTEL. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture.* 2005.

[31] INTEL. *IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference.* 2005.

[32] INTEL. *IA-32 Intel Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference.* 2005.

[33] BREY, B. B. *The Intel microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro, and Pentium II processors: architecture, programming, and interfacing.* [S.l.: s.n.], 2003.

[34] GNU Binutils. Disponível em: <http://www.gnu.org/software/binutils/>. Acesso em: 10/10/2005.

Índice Remissivo