UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

OTÁVIO OLIVEIRA NAPOLI

ESTUDO E IMPLEMENTAÇÃO DE TÉCNICAS DE PERFILAMENTO POR AMOSTRAGEM E ANÁLISE DE FLUXO DE CONTROLE EM MÁQUINAS VIRTUAIS DE PROCESSO

TRABALHO DE CONCLUSÃO DE CURSO

CAMPO MOURÃO - PR

2015

OTÁVIO OLIVEIRA NAPOLI

ESTUDO E IMPLEMENTAÇÃO DE TÉCNICAS DE PERFILAMENTO POR AMOSTRAGEM E ANÁLISE DE FLUXO DE CONTROLE EM MÁQUINAS VIRTUAIS DE PROCESSO

Trabalho de Conclusão de Curso do Curso de Bacharelado em Ciência da Computação pela Universidade Tecnológica Federal do Paraná - Campus Campo Mourão.

Orientador: Prof. Me. Juliano Henrique Foleiss

CAMPO MOURÃO - PR

2015

ATA DA DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO

Às **VINTE E UMA HORAS E VINTE MINUTOS** do dia **DOIS DE DEZEMBRO DE 2015** foi realizada no **AUDITÓRIO DA UTFPR-CM** a sessão pública da defesa do Trabalho de Conclusão do Curso Superior de Bacharelado em Ciência da Computação do acadêmico **OTÁVIO OLIVEIA NAPOLI** com o título **ESTUDO E IMPLEMENTAÇÃO DE TÉCNICAS DE PERFILAMENTO POR AMOSTRAGEM E ANÁLISE DE FLUXO DE CONTROLE EM MÁQUINAS VIRTUAIS DE PROCESSO**. Estavam presentes, além do acadêmico, os membros da banca examinadora composta pelo professor **JULIANO HENRIQUE FOLEISS** (Orientador-Presidente), pelo professor **RODRIGO HÜBNER** e pelo professor **MARCO AURÉLIO GRACIOTTO SILVA**. Inicialmente, o aluno fez a apresentação do seu trabalho, sendo, em seguida, arguido pela banca examinadora. Após as arguições, sem a presença do acadêmico, a banca examinadora o considerou **APROVADO** na disciplina de Trabalho de Conclusão de Curso e atribuiu, em consenso, a nota 8,8 (OITENTA E OITO). Este resultado foi comunicado ao acadêmico e aos presentes na sessão pública. A banca examinadora também comunicou ao acadêmico que este resultado fica condicionado à entrega da versão final dentro dos padrões e da documentação exigida pela UTFPR ao professor Responsável do TCC no prazo de onze dias. Em seguida foi encerrada a sessão e, para constar, foi lavrada a presente Ata que segue assinada pelos membros da banca examinadora, após lida e considerada conforme.

Observações:

_____

_____

_____

_____

_____

Campo Mourão, 2/12/2015

Prof. Rodrigo Hübner Membro	Prof. Marco Aurélio Graciotto Silva Membro	Prof. Juliano Henrique Foleiss Orientador

AGRADECIMENTOS

A Agradeço a minha família pelo apoio e em especial aos meus Pais que puderam me proporcionar saúde, educação e condições financeiras para estar aqui e concluir mais esta etapa.

Agradeço aos meus professores e em especial ao meu orientador Juliano Henrique Foleiss que me proporcionou conhecimento na área e pegou no meu pé até tarde para realizar este trabalho.

Agradeço também à todos os meus amigos e colegas que me apoiaram direta ou indiretamente para conclusão deste trabalho.

À todos, muito obrigado!

RESUMO

NAPOLI, Otávio Oliveira. ESTUDO E IMPLEMENTAÇÃO DE TÉCNICAS DE PERFILAMENTO POR AMOSTRAGEM E ANÁLISE DE FLUXO DE CONTROLE EM MÁQUINAS VIRTUAIS DE PROCESSO. 41 f. Trabalho de Conclusão de Curso – Curso de Bacharelado em Ciências da Computação, Universidade Tecnológica Federal do Paraná. Campo Mourão - PR, 2015.

Máquinas virtuais estão se tornando cada vez mais importantes no cotidiano. Portanto, o maior desempenho e a maior segurança destas também estão se tornando cada vez mais requisitadas. Um problema relacionado a máquinas virtuais consiste em emular um conjunto de instruções de uma arquitetura convidada em uma arquitetura hospedeira de maneira eficiente. Para isso, perfis de execução são coletados e um grafo de fluxo de controle que denota o fluxo do programa é construído para determinar o comportamento do programa, servindo como base para a tomada de decisões dos processos de otimização. Todavia, as técnicas utilizadas para realizar a coleta destes dados podem degradar bastante o desempenho da máquina virtual.

O objetivo deste trabalho foi realizar um estudo das técnicas de perfilamento baseadas em amostragem, para coleta de dados, que possuem um custo adicional relativamente baixo sobre a emulação em relação as abordagens tradicionais baseadas em instrumentação. Além disso, uma técnica de amostragem baseada em contagem e tempo foi implementada em um emulador de *Nintendo Entertainment System* (NES). Um algoritmo de baixo custo para construção do grafo de fluxo de controle das regiões recorrentes (regiões de real interesse para otimizações) baseado em amostragem, chamado GFCGaussiano, foi proposto e implementado. Este algoritmo é incremental e dinâmico, utilizando apenas de funções gaussianas para construção do grafo de fluxo de controle. O algoritmo constrói um grafo de fluxo de controle com aproximadamente 1,5% das instruções totais executadas pela máquina virtual que consegue ser até 83% similar ao grafo de fluxo de controle real, obtido por instrumentação.

Palavras-chave: Perfilamento por amostragem, Máquinas virtuais de processo, Medidas gaussianas, Construção de grafos de fluxo de controle

ABSTRACT

NAPOLI, Otávio Oliveira. . 41 f. Trabalho de Conclusão de Curso – Curso de Bacharelado em Ciências da Computação, Universidade Tecnológica Federal do Paraná. Campo Mourão - PR, 2015.

Virtual machines play a big role in modern computing. Therefore it is crucial to provide higher performance implementations. A problem related to high performance virtual machine consists in efficiently emulating the instruction set of a guest architecture in a host machine. In order to achieve this, execution profiles must be collected and a control flow graph must be build in order to determine program execution behavior and serve as basis for decision making in dynamic optimization techniques. The main challenge related to collecting execution profile data is that this process incurs in considerable overhead to the virtual machine.

The aim of this work was to study sampling-based profiling techniques to collect execution profile data, which are less costly then traditional instrumentation-based approaches, albeit having a performance-accuracy trade-off. Furthermore, a sampling-based technique was implemented in a *Nintendo Entertainment System* (NES) emulator. The most significant contribution of this work is a sampling-based profiling hot control flow graph construction algorithm named GFCGaussiano, along with its experimental implementation. This is a low-cost incremental algorithm that is based on simple gaussian measures that builds a control flow graph depicting the hot regions of a program during runtime. The algorithm achieves this by sampling just 1.5% of all instructions executed by the virtual machine and generates graphs with up to 83% similarity to the exact hot control flow graph gathered by instrumentation.

Keywords: Sampling-based profiling, process virtual machines, gaussian measures, control flow graph construction

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Máquinas virtuais é um tópico que possui grande influência na comunidade científica e industrial em geral. O desenvolvimento desta tecnologia proporcionou a possibilidade para solução de uma variedade de problemas relacionados a sistemas operacionais, linguagens de programação e arquitetura de computadores em geral. Mesmo que o conceito de máquinas virtuais não seja novo, este permitiu a padronização e a criação de novos conjuntos de instruções, tão quanto a otimização dinâmica para redução de energia e melhoria do desempenho (SMITH; NAIR, 2005).

No contexto industrial, as máquinas virtuais puderam consolidar e levar ao desenvolvimento de diversas arquiteturas que não são amplamente aceitas (SMITH; NAIR, 2005). O uso delas permite a execução de conjuntos de instruções distintos do conjunto de instruções do hardware real. Uma máquina virtual é basicamente uma máquina real executando um software de virtualização que permite a execução de outro conjunto de instruções em uma máquina com conjunto de instruções diferente ou igual (SMITH; NAIR, 2005). Em diversos contextos, máquinas virtuais podem ser utilizadas devido sua versatilidade e também sua economia, pois permitem a criação de um ambiente virtual para execução de sistemas completos, os quais levaria a um custo de obtenção de hardware potencialmente alto (SMITH; NAIR, 2005).

Contudo, para a construção de uma máquina virtual, o conhecimento da arquitetura do programa que sofrerá emulação é de extrema importância. Conhecendo esta arquitetura, as técnicas para execução de programas sobre a máquina virtual são discorridas. Entre as técnicas de emulação temos a interpretação, que consiste em percorrer o programa alvo, buscando, decodificando e executando as instruções de forma à alterar o estado do conjunto de instruções mantendo o isomorfismo. Outra vertente à emulação é a tradução de binários que converte códigos do programa alvo para arquitetura da máquina real (SMITH; NAIR, 2005). Otimizações dinâmicas comumente utilizam um grafo de fluxo de controle, que é um grafo que denota o fluxo de execução do programa, como base para tomada de suas decisões. A coleta de informações para realização do perfil para geração deste grafo podem ser realizadas de duas formas: com instrumentação e com amostragem.

Técnicas de coleta baseadas em instrumentação consistem na inserção de instruções de

controle em regiões do emulador para realização da coleta de dados, enquanto técnicas baseadas em amostragem realizam interrupções periódicas ou aleatórias ao emulador para realização da coleta. A instrumentação possui uma acurácia maior, porém o custo adicional de execução também é mais alto. A amostrage,m por sua vez, possui uma acurácia menor e o custo adicional de execução também menor (SMITH; NAIR, 2005).

O objetivo principal deste trabalho é propor uma técnica para a construção do grafo de fluxo de controle a partir dos dados coletados por meio de amostragem. Para cumprir este objetivo, os seguintes objetivos específicos foram traçados:

- Realizar uma pesquisa bibliográfica sobre as técnicas de perfilamento por amostragem;
- Implementar algumas técnicas e comparar seus pontos positivos e negativos;
- Explorar o comportamento dinâmico dos programas a partir dos dados coletados na amostragem;
- Nas observações realizadas sobre o comportamento dos programas com base nos dados coletados por amostragem, propor uma técnica para construção dografo de fluxo de controle; e, por fim
- Avaliar os resultados obtidos e compará-los a uma técnica de instrumentação.

Com base nestes objetivos específicos, a principal contribuição deste trabalho é o desenvolvimento de um algoritmo para construção do grafo de fluxo de controle a partir dos dados coletados pela técnica de amostragem, também desenvolvida, juntamente com a análise da similaridade do grafo de fluxo de controle gerado por este algoritmo com o grafo de fluxo de controle real, gerado pela instrumentação.

2 MÁQUINAS VIRTUAIS

O gerenciamento da complexidade nos sistemas de computação modernos é possibilitado pela divisão lógica dos componentes do sistema em níveis de abstração, separados por interfaces bem definidas, de forma hierárquica. Com os níveis de abstração, podemos simplificar o desenvolvimento de componentes de alto nível utilizando interfaces de comunicação bem definidas, ignorando detalhes de implementação de componentes de mais baixo nível à ele (SMITH; NAIR, 2005).

A interface do sistema operacional, por exemplo, define um conjunto de funções e chamadas de sistema padronizadas para o gerenciamento e utilização de forma simples dos recursos de hardware, permitindo desenvolvedores interagirem e implementarem aplicações sem se preocuparem com o gerenciamento de detalhes de baixo nível (SMITH; NAIR, 2005). Todavia, apesar das vantagens da abstração da complexidade através destas interfaces de comunicação, esta também pode limitar o uso e gerar uma dependência desta interface. Desta forma, um programa executável está amarrado a uma combinação específica de um sistema operacional e um conjunto de instruções (que define uma plataforma).

2.1 VIRTUALIZAÇÃO

Máquinas virtuais partem da virtualização, que provê uma forma de relaxamento entre as limitações geradas pelo conceito de abstração com a intenção de ampliar a flexibilidade e interoperabilidade dos sistemas de computação (SMITH; NAIR, 2005). Assim, um conjunto de sistemas reais pode ser transformado para aparentar ser diferente em um ambiente virtual.

Segundo Popek e Goldberg (1974), para que a virtualização seja realizada, é necessária a construção de um isomorfismo que mapeie o estado do sistema convidado para o sistema hospedeiro. Como mostra na Figura 1, existe um conjunto de estados `S` das instruções `I`, tanto no sistema convidado quanto no hospedeiro. Para todo conjunto de sequência de instruções `e` no sistema do convidado, existe uma sequência de instruções `e'` que modifica o estado do sistema hospedeiro, utilizando a função de virtualização `V`, de forma a manter a equivalência de estados entre ambos.

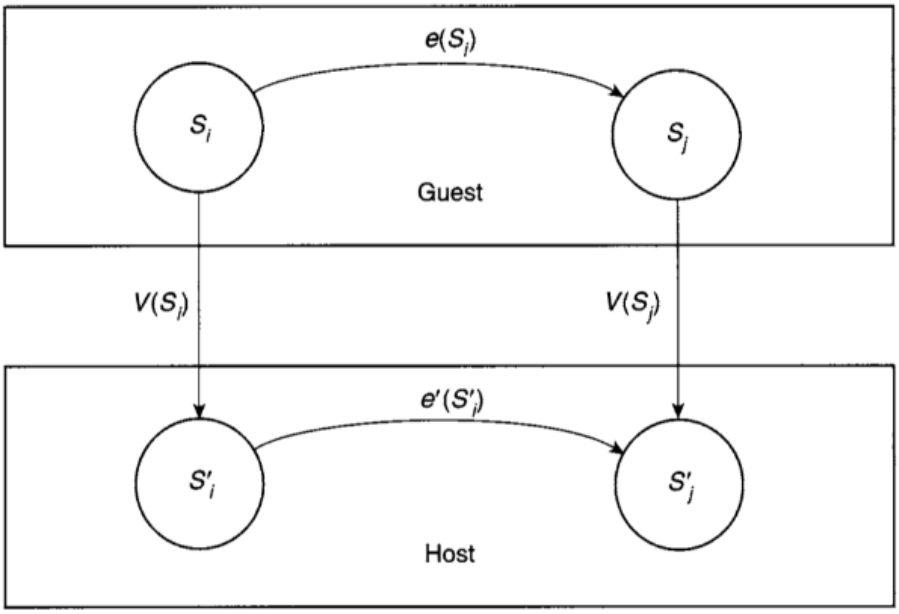

Figura 1: Isomorfismo entre o sistema convidado e o sistema hospedeiro (SMITH; NAIR, 2005)

2.2 ARQUITETURA

O conhecimento da arquitetura do sistema que sofrerá a virtualização é de extrema importância, tanto quanto a arquitetura do sistema que fará a mesma, pois apenas através de tais fundamentações é possível obter o isomorfismo entre ambos. Para Smith e Nair (2005), a arquitetura pode ser descrita como a especificação de uma interface e o comportamento lógico dos recursos manipulados através da interface, assim qualquer arquitetura pode ter diversas implementações, com diversas características diferentes para cada. Pelos conceitos dos níveis de abstração para um sistema computacional, a abstração corresponde a implementação das camadas nos níveis de software e hardware. Algumas interfaces importantes do sistema computacional são mostradas na Figura 2 e descritas a seguir.

O conjunto de instruções da arquitetura (*Instruction Set Architecture - ISA*) denota a fronteira entre o hardware e o software (interfaces 7 e 8, na Figura 2). Esta é a interface que o programador tem com o hardware. Ela expõe todas as funcionalidades do hardware ao software, permitindo o gerenciamento dos recursos da máquina (SMITH; NAIR, 2005). Tal interface deve ser bem definida para permitir a interoperabilidade e garantir a abstração da simplicidade de implementação. No contexto deste trabalho, a máquina virtual utilizada tem o objetivo de emular as interfaces 7 e 8 que são: a ISA de usuário e ISA de sistema, respectivamente e são descritas a seguir.

Três conceitos importantes apresentados por Smith e Nair (2005) são: a ISA de usuário (*User ISA*), que é a ISA visível para a aplicação, apresentado na Figura 2 pela interface 7,

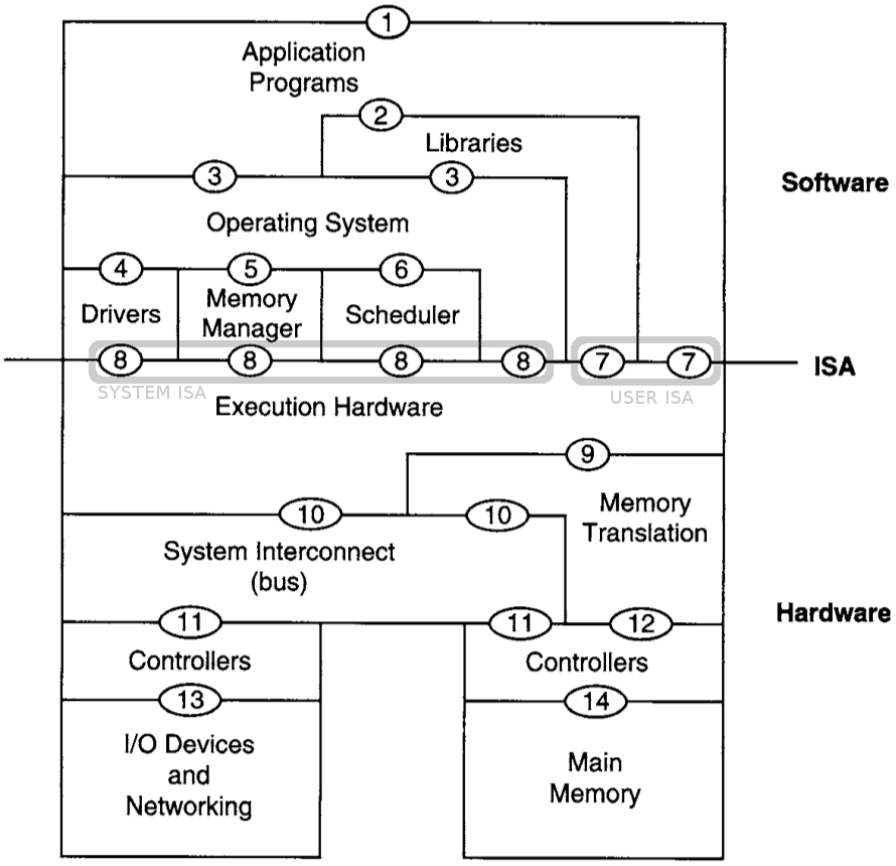

Figura 2: Camadas de comunicação de um sistema computacional (SMITH; NAIR, 2005)

a ISA de sistema (*System ISA*), que é a ISA visível para o sistema, apresentado na Figura 2 pela interface 8 e a ABI (*Aplication Binary interface*), apresentado na Figura 2 pela interface 3 (interface de chamadas de sistema), a qual prove acesso para os recursos de hardware e serviços dispostos pelo sistema operacional para as aplicações e as bibliotecas de usuário. A ABI possui um conjunto de instruções de usuário, sem instruções de sistema, que permite à aplicação interagir com recursos de hardware indiretamente, por meio de chamadas de sistema do sistema operacional.

2.3 MÁQUINAS VIRTUAIS DE PROCESSO

Uma máquina virtual é implementada como sendo a combinação da máquina real com um software de virtualização. Com o isomorfismo, o processo de virtualização consiste no mapeamento dos recursos ou estado virtual da máquina convidada para a real e o uso de instruções da máquina real para realizar a consistência e manutenção dos estados da máquina virtual (SMITH; NAIR, 2005). Desta forma, uma máquina virtual de processo é capaz de suportar a execução de um processo e prover toda a ABI e as chamadas de sistema necessárias

para este processo de forma transparente (interfaces 3 e 7) (SMITH; NAIR, 2005). O software de virtualização de uma máquina virtual de processo pode também ser chamado de *runtime*.

2.4 EMULAÇÃO

Para (SMITH; NAIR, 2005) a emulação é o processo de implementar uma funcionalidade de um sistema que possui uma funcionalidade diferente. Uma aplicação pode, por exemplo, estar compilada para uma plataforma específica, em sua forma binária, e a plataforma em uso é distinta.

Assim sendo, a emulação do conjunto de instruções da arquitetura alvo é a forma mais apropriada para reprodução do comportamento de uma aplicação que possui o conjunto de instruções diferente da máquina real, de forma a se obter o isomorfismo previamente discutido. Em uma máquina virtual, a emulação do conjunto de instruções geralmente ocorre mediante a duas técnicas amplamente aplicadas, sendo elas a interpretação e a tradução de binários. Ambas as técnicas serão discutidas nas seções subsequentes.

2.4.1 INTERPRETAÇÃO

A interpretação é o modo mais natural de emulação de um conjunto de instruções. Um interpretador emula e opera por completo o estado da ISA fonte, incluindo os registradores e a memória principal da aplicação (SMITH; NAIR, 2005). Um interpretador simples percorre o programa fonte, buscando, interpretando e executando instrução por instrução e por conseguinte alterando o estado da máquina de forma adequada. Esta abordagem é muito parecida com a arquitetura de sistemas de computação de John von Neumann, onde as instruções são buscadas, decodificadas e executadas.

A implementação deste interpretador geralmente consiste no desenvolvimento de um laço central de tradução, de forma a ler a instrução do programa fonte, decodificar-la, aplicar o procedimento necessário e equivalente para execução desta instrução na máquina hospedeira, de forma a manter o isomorfismo de estados e voltar ao laço central de interpretação para a busca de uma nova instrução, continuando desta forma até o programa terminar.

As instruções aplicadas para a troca de estados do programa fonte, na máquina hospedeira e na máquina convidada, não necessitam ter um mapeamento um-para-um (POPEK; GOLDBERG, 1974). É possível criar funções muito mais complexas para conseguir gerar esse mapeamento. Para uma instrução do conjunto de instruções da máquina convidada, por exemplo, pode ser que sejam necessárias dez instruções do conjunto de instruções da máquina

hospedeira para poder alcançar um estado isomórfico final equivalente.

Esse interpretador descrito acima é chamado de interpretador de decodificação e despacho (*decode and dispatch interpreter*), definido pela existência de um laço de controle de interpretação central. Este interpretador necessita interpretar toda instrução e voltar ao laço de controle para buscar da próxima, o que, apesar de geralmente ser simples de entender e implementar, gera uma certa degradação do desempenho.

Uma outra maneira, que visa aumentar o desempenho removendo o laço central de interpretação, é a interpretação encadeada (*Threaded Interpretation*) proposta por (KLINT, 1981). Desta forma, mantém-se uma tabela apontando para todas as rotinas referentes as instruções do conjunto de instruções utilizado. Por fim, adversamente à necessidade de sempre voltar ao laço de interpretação para a busca e despacho da próxima instrução, o próximo endereço é calculado no fim da execução da instrução e então a próxima instrução é lida. Utilizando a tabela e a instrução como chave, salta-se para a rotina referente a próxima instrução a ser executada, de maneira a melhorar o desempenho.

A Figura 3.a ilustra o funcionamento de um interpretador que utiliza interpretação de decodificação e despacho e ao lado, na Figura 3.b um interpretador que utiliza interpretação encadeada.

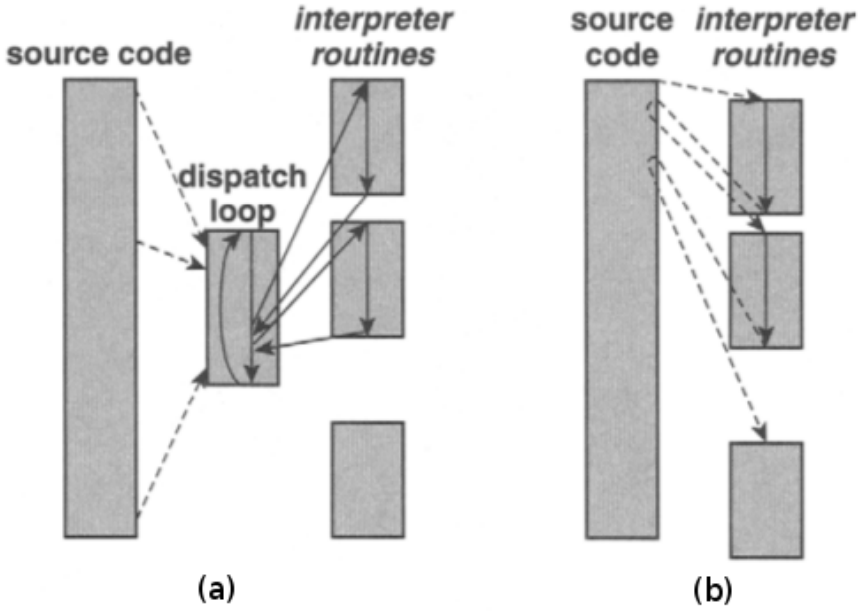

Figura 3: Médodos de interpretação. (a) Interpretação de decodificação e despacho e (b) interpretação encadeada - Adaptado de (SMITH; NAIR, 2005)

2.4.2 TRADUÇÃO DE BINÁRIOS

Divergindo à interpretação, uma outra forma de emulação é a tradução de binários. Em termos conceituais, o processo de tradução consiste em converter as instruções binárias da aplicação fonte para a arquitetura alvo (ALTMAN et al., 2000). Assim, a interpretação consiste num ciclo de busca, decodificação e execução de cada instrução, enquanto a tradução de binários amortiza esse ciclo, traduzindo um bloco de instruções para a arquitetura hospedeira e, por fim, salvando o bloco traduzido para posterior uso. Desta forma, sempre que um bloco traduzido necessitar ser executado novamente, este não precisa passar pelo ciclo todo proposto pela interpretação e pode simplesmente ser executado diretamente, tendo em vista que o bloco já está traduzido para o conjunto de instruções da arquitetura do hospedeiro.

A tradução pode também ocorrer durante a execução da aplicação, traduzindo instruções sob demanda. Tal técnica é chamada de tradução dinâmica de binários (*TDB*). Esta também possui, como sua principal vantagem, o desempenho em relação as outras técnicas de emulação (ALTMAN et al., 2000), em virtude da execução de código traduzido para máquina hospedeira e a menor realização de chamadas ao sistema de gerência de emulação (JONES; TOPHAM, 2009).

A grande desvantagem da tradução de binários é o custo adicional gerado pela tradução. De tal forma, a tradução de toda instrução implicaria em um custo muito alto e consequentemente uma degradação de desempenho considerável (JONES; TOPHAM, 2009). Portanto o reconhecimento de regiões frequentes ou que de alguma forma contribuam para a melhoria do desempenho deve ser levado em consideração. Assim, apenas regiões bastante utilizadas devem ser traduzidas de forma a amortizar o custo adicional gerado pela tradução.

Uma unidade de tradução representa o código que será traduzido e necessita conter todas as informações para que a tradução seja realizada. No caso da tradução dinâmica de binários a unidade de tradução representa um bloco básico ou um agrupamento de blocos básicos (OTTONI et al., 2011). A determinação da região a ser traduzida pode ser uma tarefa que demande demasiado esforço. Um dos problemas da tradução dinâmica é a descoberta de código, pois o fluxo de controle de um programa pode ser alterado dependendo de sua entrada. Instruções de salto, por exemplo, podem depender de valores que são calculados durante a execução do programa, inviabilizando a determinação estática do destino dos saltos (SMITH; NAIR, 2005) (HORSPOOL; MAROVAC, 1980).

2.5 BLOCOS BÁSICOS

Um bloco básico, por definição, é uma sequência maximal de instruções tal que nenhuma instrução exceto a primeira é alvo de um salto e nenhuma instrução, exceto a última, é um salto (MUCHNICK, 1997). Em termos práticos, na tradução dinâmica de binários, um bloco básico é apenas mais uma unidade de tradução que pode ser trabalhada. A vantagem do bloco básico, como mostrado em sua definição, é a ausência de instruções de salto no interior do mesmo, inviabilizando ações dinâmicas dentro deste, com exceção do tratamento de sinais e exceções. Vários perfis podem ser traçados, tendo conhecimento que a última instrução será um salto, por exemplo o perfil de arestas que determina a quantidade de transições existentes entre os blocos. Desta forma, a geração de grafos de fluxo de controle pode se tornar mais simples, podendo reservar mais tempo à otimização tal como construção de super blocos.

3 OTIMIZAÇÃO DINÂMICA

Comparado à interpretação, a tradução de binários pode proporcionar um ganho de desempenho muito alto (SMITH; NAIR, 2005). Otimizações mais complexas utilizam padrões da execução do programa para nortear seu processo de tomada de decisões. Uma das formas para descoberta de padrões (e possivelmente a aplicação de otimizações) baseia-se na geração de perfis de execução dinâmica.

Este capítulo abordará conceitos sobre o perfilamento, que é o processo de coleta de dados para geração de um perfil de execução. A coleta destes dados ocorre de duas formas, sendo por instrumentação ou amostragem. Com estes dados coletados, é possível gerar um grafo de fluxo de controle e executar algoritmos com base neste para aplicação de otimizações.

3.1 PERFILAMENTO

Como mencionado anteriormente, o perfilamento (*profiling*) é um processo que consiste em coletar dados estatísticos para o registro do fluxo de controle de um programa. Os dados do perfil são adquiridos no perfilamento durante a execução do programa e são utilizados em otimizações, que muitas das vezes exploram as características de localidade espacial e temporal da execução do programa devido a previsibilidade do fluxo de execução dos programas, isto é, características do comportamento do programa, devido a execuções passadas, tendem a se repetir com frequência (SMITH; NAIR, 2005). O intervalo e o que é coletado para o perfil é determinado pela técnica utilizada.

Utilizando os dados do perfil é possível identificar de forma mais clara quais unidades de tradução tem melhor propensão a serem traduzidas, com o objetivo de amortizar o custo da tradução por meio de execuções de blocos recorrentes, como citado anteriormente. Podemos também destacar blocos cuja execução é mais frequente e determinar os caminhos que são mais tomados entre os blocos, de forma a reorganizar os blocos ou gerar superblocos (bloco básico composto por vários blocos básicos), aumentando o escopo de otimização e ampliando a unidade de tradução (SMITH; NAIR, 2005; OTTONI et al., 2011; JONES; TOPHAM, 2009).

Os dados coletados durante o processo de perfilamento dependem das otimizações e das análises implementadas no ambiente de execução da máquina virtual. Alguns dados que

são comumente coletados para compor um perfil de fluxo de controle são:

- Endereço dos destinos de salto que partem de um bloco;
- Endereço inicial do bloco, geralmente dado pelo valor do contador de programa (PC); e
- Quantidade de vezes que um bloco foi executado.

Com os valores dos endereços de salto, por exemplo, podemos gerar um grafo (denotado posteriormente de grafo de fluxo de controle) com os blocos básicos e as transições entre eles. Anotando o número de vezes que foram feitos saltos para um bloco básico, é possível determinar, a partir de um determinado limiar, se aquela região de saltos é frequentemente executada ou não.

De forma geral, o perfilamento é realizado durante a execução do código alvo por meio do processo de interpretação. Assim, o tradutor não só é projetado com o objetivo de obter o máximo de desempenho, mas também para executar o perfilador, podendo este gerar uma queda no desempenho. Portanto os dados de perfil coletados devem ser úteis na análise para a realização de otimizações para amortização do custo gerado (FOLEISS, 2012).

3.2 COLETA DE PERFIS

Existem duas formas de coletar perfis de execução: instrumentação e por meio de amostragem. Ambas possuem suas características e peculiaridades e serão discorridas nas subseções subsequentes.

3.2.1 INSTRUMENTAÇÃO

A instrumentação é uma forma de coleta de dados que consiste na inserção de códigos de monitoramento para coleta (*probes*) em pontos específicos do programa emulador de forma a monitorar diversos eventos que ocorrem durante a execução do programa alvo (SMITH; NAIR, 2005). Pode-se, por exemplo, caso seja de interesse coletar instruções de salto, de forma a decidir se um dado salto foi tomado ou não, dispor instruções de controle que registrem essa condição logo após as instruções de salto do programa alvo. A Figura 4 ilustra uma rotina de execução de salto (*branch*) no interpretador (linha 4) e o local onde o código deve ser inserido para computar se o dado salto foi tomado ou não (linha 13).

Uma das máquinas virtuais atual, que utiliza instrumentação para coleta de dados, é a Java Virtual Machine (JVM) (LINDHOLM FRANK YELLIN; BUCKLEY, 2014). Seu

```
1  Instruction function list
   .
   .
4  branch_conditional(inst) {
5       BO = extract(inst,25,5);
6       BI = extract(inst,20,5);
7       displacement = extract(inst,15,14) * 4;
        .
        .
        // code to compute whether branch should be taken
        .
        .
12      profile_addr = lookup(PC);
13      if (branch_taken)
14           profile_cnt(profile_addr, taken);
15           PC = PC + displacement;
16      Else
17           profile_cnt(profile_addr, nottaken);
18           PC = PC + 4;
19 }
```

Figura 4: Uma rotina de interpretação de salto na arquitetura *PowerPC*, com código de instrumentação em itálico (SMITH; NAIR, 2005).

conjunto de instruções é composto por instruções de um e, em alguns casos, dois bytes que representam o *opcode* e os operandos. Este conjunto de instruções foi projetado de maneira à permitir a simplicidade da análise de fluxo de controle e prover informações para a otimização utilizada na JVM. Desta forma, na execução da JVM, códigos de instrumentação são dispostos em locais específicos, já que seu conjunto de instruções é simples e bem conhecido, permitindo demasiada informações para o perfil tornando-o mais conciso e podendo ajudar o processo de otimização e a possibilidade da execução de otimizações agressivas com base nos dados coletados (SUGANUMA et al., 2005).

Com a instrumentação, é possível atingir um nível de detalhamento e precisão muito alto, já que é possível dispor instruções de controle em qualquer ponto do interpretador, levando assim a uma coleta mais rica, sendo essa sua principal vantagem (SMITH; NAIR, 2005). Contudo, uma importante desvantagem da coleta de dados utilizando a instrumentação é o custo adicional gerado pelas instruções de controle dispostas no emulador. Possivelmente várias instruções de instrumentação devem ser colocadas em cada trecho onde deseja-se coletar os dados, degradando o desempenho do processo de emulação. A instrumentação, além de poder realizar desvios para suas rotinas, também pode prejudicar a cache de dados, pela necessidade de manter estruturas de dados adicionais para seu controle. Desta forma, para que a instrumentação seja viável, a otimização utilizada com base no perfil deve ser de extrema utilidade, desmazelando assim o viés do custo adicional gerado.

3.2.2 AMOSTRAGEM

Uma outra forma de obtenção de dados, divergente à instrumentação, é a amostragem. Esta não insere nenhum trecho de controle no código alvo, tanto quanto no interpretador, executando-o de forma imutável. A coleta de dados é realizada interrompendo o programa em intervalos periódicos ou aleatórios e capturando uma instância do evento de execução do programa (SMITH; NAIR, 2005). O custo adicional para perfiladores que se baseiam em técnicas de amostragem é muito baixo (em escala de ordens de magnitude), comparado ao custo para perfiladores que baseiam-se em técnicas de instrumentação, sendo esta sua principal vantagem (BURROWS et al., 2000); (ANDERSON et al., 1997). A ausência de instruções de controle e o intervalo disposto para realização das amostras são pontos que embasam a vantagem da técnica.

A amostragem pode ser feita em hardware, como em Sastry et al. (2001), que utiliza mecanismos especiais para fazer a contagem de instruções desejadas. Após o número de instruções contadas ultrapassar um limiar definido pelo programador (por software), os dados coletados são passados ao software para geração de seu perfil. Esta abordagem, porém, diminui a flexibilidade do amostrador à vista que fica-se atado as dependências do hardware.

A amostragem baseada em tempo (*time-based sampling*) utiliza *timers* para disparar o amostrador, como intervalo de amostragens. Este mecanismo possui um custo de aplicação muito baixo além da simplicidade da implementação. Todavia, como possivelmente executa-se o emulador e o amostrador como programas em um sistema operacional, não temos uma precisão muito alta quanto ao intervalo de tempo[1], tendo em vista que o sistema operacional precisa gerenciar todo o contexto dos processos existentes (ARNOLD; GROVE, 2005); (BURROWS et al., 2000).

Outra maneira de determinar o intervalo de amostragem é utilizando contagem (*count-based profiling*) que, após um limiar definido pelo programador, da quantidade de instruções do programa executadas no emulador, o perfilador é disparado e a amostra é coletada (ARNOLD; GROVE, 2005). O perfilador muitas vezes executa em uma *thread* separada ao emulador, tornando-o um pouco mais complexo, comparado à técnica que se utiliza de *timers* (em termos de implementação), devido a necessidade de se conhecer o número de instruções que foram executadas, que em algumas aplicações podem ser de difícil conhecimento.

Um exemplo de perfilador de propósito geral é o DCPI (Digital Continuous Profiling Infrastructure). Desenvolvido pela Digital Equipment Corporation, o DCPI é um sistema de

[1] Geralmente, em sistemas que utilizam o Linux padrão, a granularidade mínima de interrupção do *timer*, para processos não privilegiados, é restringida à 1ms.

perfilador, baseado em amostragem, designado para executar continuamente em sistemas de produção (ANDERSON et al., 1997). Este prove um eficiente coletor de dados, utilizando interrupções periódicas disparadas por *timers*. Ele também permite a coleta de diferentes tipos de dados para geração de diferentes perfis de execução e a saída destes dados geralmente é utilizada por programadores para análise do programa ou por compiladores otimizadores estáticos. Por fim, o DCPI também provê uma série de ferramentas para a análise destes dados que são utilizadas por diversos pesquisadores da área.

A maior desvantagem da amostragem é sua falta de acurácia. Técnicas que usam tal abordagem não conseguem capturar todos os valores observados ao longo da execução de uma aplicação, já que os dados são coletados em intervalos, podendo deixar lacunas nas coletas e fazendo com que não se obtenha os dados realmente desejados. Essa desvantagem diverge à instrumentação, onde seus códigos são colocados em locais específicos para obtenção de dados desejados (SMITH; NAIR, 2005);(BURROWS et al., 2000).

3.3 GRAFO DE FLUXO DE CONTROLE

Um grafo de fluxo de controle (*Control Flow Graph - CFG*) é um grafo dirigido onde os nós representam blocos básicos e as arestas representam as transições realizadas entre os blocos básicos (ALLEN, 1970). Utilizando dados coletados no perfilamento, é possível determinar os blocos básicos e as transições existentes entre eles. O grafo de fluxo de controle é gerado com base nas informações extraídas do perfilamento, durante a execução do programa. Otimizações operam sobre este, já que denota o comportamento dinâmico do programa (SMITH; NAIR, 2005).

Em termos práticos, com instrumentação, pode-se facilmente determinar o início, o fim e as instruções intermediárias que compõe um bloco básico, simplesmente marcando instruções de salto como o final de um bloco básico e o endereço de destino deste como o inicio de outro. Desta forma, é possível montar uma estrutura de bloco básico, populá-la com estas informações e posteriormente utilizar uma estrutura de grafo, usando o bloco básico como entrada para desenvolver o grafo de fluxo de controle. O grafo gerado e suas peculiaridades dependem dos dados que são coletados no perfilamento.

A Figura 5.a mostra um grafo de fluxo de controle onde cada nó possui um contador correspondente a quantidade de execuções de cada bloco. Este perfil é chamado de perfil de bloco básico (*basic block profile*) ou também perfil de nó (*node profile*). Já na Figura 5.b, cada aresta possui um contador correspondente a quantidade de vezes que aquele caminho foi

tomado. Este perfil é conhecido como perfil de arestas (*edge profile*). Em geral, este perfil é bastante utilizado por perfiladores e, a partir deste, pode-se gerar um perfil de nó, atribuindo ao nó o resultado da soma das arestas que incidem nele. Por exemplo, a quantidade de execuções do nó D, na figura 5.b pode ser calculado somando a quantidade de execução das arestas que incidem nele (12 mais 13), que é 25 (SMITH; NAIR, 2005).

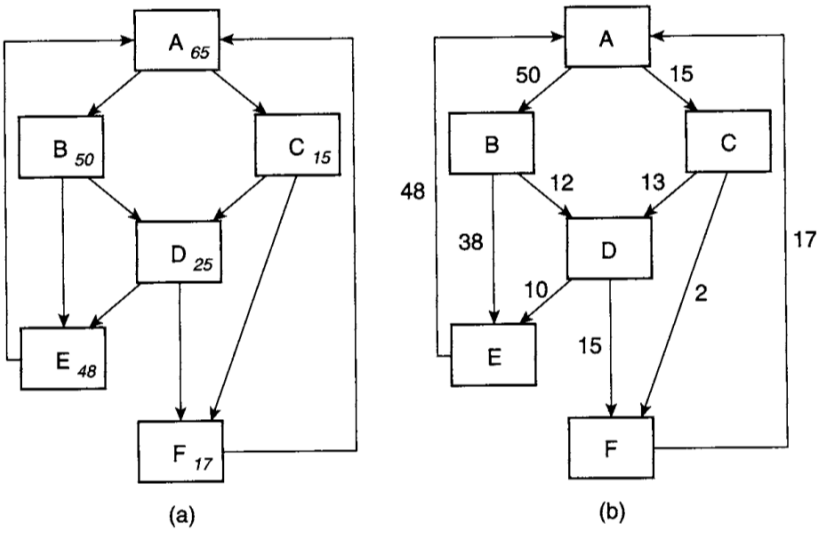

Figura 5: Perfilamento no grafo de fluxo de controle. (a) Perfil de nó e (b) Perfil de arestas - Adaptado de (SMITH; NAIR, 2005).

A tradução do grafo todo pode ser muito custoso, a ponto de degradar o desempenho da emulação. Uma otimização possível, utilizando o grafo de fluxo de controle gerado com um perfil de nó, consiste em traduzir apenas sequências de código consideradas realmente quentes, ou seja, sequências de nós executados com frequência. Desta forma, a execução de código traduzido para regiões quentes amortiza o custo de sua tradução, aumentando o desempenho do processo de emulação. (SMITH; NAIR, 2005).

No capítulo seguinte é apresentada uma técnica de amostragem de baixo custo e um algoritmo para construção de grafo de fluxo de controle também de baixo custo, que gera um grafo de fluxo de controle apenas das regiões quentes.

4 DESCRIÇÃO DAS TÉCNICAS

Neste capítulo será apresentada a técnica de amostragem utilizada neste trabalho. Esta técnica é baseada em contagem e tempo e os lotes de amostras são utilizados pelo algoritmo de geração de grafo de fluxo de controle. O algoritmo GFCGaussiano também será apresentado, na seção 4.2, juntamente com suas premissas e considerações.

4.1 TÉCNICA DE AMOSTRAGEM

A coleta de dados utilizando instrumentação é bastante custosa devido a necessidade de inserir códigos no interpretador para o registro do perfil de execução (SMITH; NAIR, 2005). Por outro lado, as técnicas baseadas em amostragem possuem custo menor, uma vez que a coleta dos dados geralmente é realizada de forma espaçada no tempo e obtida de forma indireta, por exemplo, com uso de hardware especial (SASTRY et al., 2001) ou com *threads* de execução separadas.

Embora o custo da amostragem seja menor que o custo da instrumentação, existe um $trade-off$ em relação à precisão dos grafos de fluxo de controle que são obtidos em relação às duas técnicas. Com instrumentação é possível rastrear todas as transições entre os blocos básicos do programa inserindo código no interpretador ou em seções de código traduzidas para as instruções de fluxo de controle. Portanto, neste caso, é possível obter o grafo de fluxo de controle exato (SMITH; NAIR, 2005). No entanto, quando o perfilamento por amostragem é usado, apenas parte das instruções de salto são conhecidas, portanto, é necessário desenvolver algoritmos capazes de inferir o grafo de fluxo de controle e o conteúdo dos bocos básicos considerando que as amostras são coletadas de forma espaçada no tempo.

Contudo, a construção de grafos de fluxo de controle é viável baseando-se no principio de localidade espacial presente nos programas (SMITH; NAIR, 2005), o qual diz que um programa tende a ficar grande parte de seu tempo de execução em uma fração relativamente pequena de seu código. Assim, quando uma amostragem é realizada, possivelmente capturará alguma região bastante recorrente, também conhecida como região quente. Desta forma, a amostragem é suficiente para determinar os grafos de fluxo de controle destas regiões, que são as regiões de interesse para a maioria das aplicações de grafos de fluxo de controle em máquinas virtuais, como, por exemplo, técnicas de otimização dinâmica, que muitas das vezes utilizam

apenas das regiões quentes.

4.1.1 AMOSTRAGEM BASEADA EM CONTAGEM E TEMPO

A técnica de amostragem utilizada neste trabalho para o desenvolvimento do algoritmo para construção do grafo de fluxo de controle foi proposta por Arnold e Grove (2005). Esta técnica consiste em utilizar amostragem baseada em contagem e tempo, de forma que a máquina virtual sofre interrupções periódicas para sinalizar o perfilador que, quando iniciado, faz a amostragem de um determinado número de instruções em sequência e volta ao seu estado de espera para reinicialização do processo.

A Figura 6 ilustra esta técnica de amostragem baseada em contagem e tempo. Os traços correspondem as instruções sendo executadas pelo emulador e a cada intervalo de `X` milissegundos, é realizada uma amostra de `N` dados, em sequência, representado pelos traços pretos.

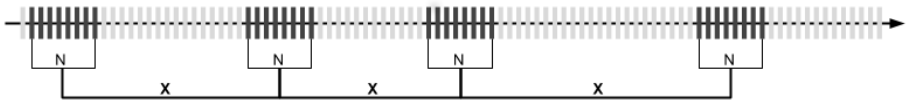

Figura 6: Técnica de perfilamento por amostragem baseada em tempo e contagem

Em termos de implementação, esta técnica pode ser realizada utilizando uma *thread* separada do emulador, com o objetivo de não interromper a execução da emulação e possuir visibilidade à região de memória do processo, em que a cada `X` milissegundos é amostrado `N` dados sequencialmente e armazenado em um *buffer*. Alguns dados preferíveis para a coleta geralmente são os contextos dos registradores, em especial o contador de programa (PC). A coleção de `N` amostras sequenciais é denominada **lote de amostras**, ou simplesmente, **lote**.

Os dados, quando coletados e armazenados, podem ser despachados para o algoritmo GFCGaussiano, que constrói o grafo de fluxo de controle do programa incrementalmente a cada lote. Este algoritmo é apresentado a seguir.

4.2 O ALGORITMO GFCGAUSSIANO

O algoritmo GFCGaussiano constrói um grafo de fluxo de controle incrementalmente, ou seja, a cada lote coletado o algoritmo é invocado. Com efeito, cada execução do algoritmo é uma iteração do processo de construção do GFC. A seguir são apresentadas as premissas e as considerações que serviram como alicerces para o desenvolvimento do algoritmo, na

seção 4.2.3, o algoritmo GFCGaussiano é apresentado, bem como algumas considerações de implementação.

4.2.1 PREMISSAS

Conforme discutido anteriormente, grande parte das instruções executadas pertencem a uma pequena parcela do programa. O fluxo de controle de um programa é dado pela sequência de valores assumidos pelo registrador PC, ou seja, o endereço do PC varia de acordo com cada instrução executada. Desta forma, uma amostragem periódica tende a capturar as transições mais recorrentes (transições quentes) no fluxo de execução do programa. Em determinado lote, **transições** são caracterizadas por instruções que alteram o fluxo de controle do programa, tais como instruções de salto, desvios condicionais, chamadas ao sistema operacional e sinais.

A variação no fluxo de controle de um programa pode ser vista em dois níveis: nível local e nível global.

Nível Local: Neste nível estão as transições entre blocos básicos que são localmente próximos, ou seja, entre blocos que possuem endereços absolutos próximos. Usualmente estes blocos fazem parte de estruturas relacionadas a subprogramas. Transições quentes neste nível são características de transições com alta localidade temporal e representam os núcleos de execução dos programas que ocupam a maior parte das instruções executadas. Em alto nível, transições relacionadas a laços de repetição muitas vezes são exemplos de transições em nível local.

Nível Global: Neste nível estão as transições entre blocos básicos que estão em rotinas com endereços distantes. Em alto nível, estas transições correspondem a chamadas de sistema. Transições quentes neste nível caracterizam fluxos entre rotinas que são altamente relacionadas. No entanto, não é possível concluir que sua localidade temporal é proporcionalmente alta.

A Figura 7 ilustra um exemplo de transições em nível local, representada pelas setas tracejadas escuras que denotam um laço de repetição entre os blocos 1, 2 e 3, e em nível global, representado pela seta pontilhada clara denotando uma chamada de sistema do bloco 2 a um bloco distante (bloco 15).

Um algoritmo de construção de grafo de fluxo de controle necessita detectar as transições que estão em ambos os níveis. Lotes de amostras que possuem somente transições de nível local são úteis para determinar a sequência das instruções que compõe os blocos básicos. Isto é útil pois usualmente as aplicações que utilizam grafos de fluxo de controle necessitam não somente das transições entre blocos, mas também das instruções que os compõe.

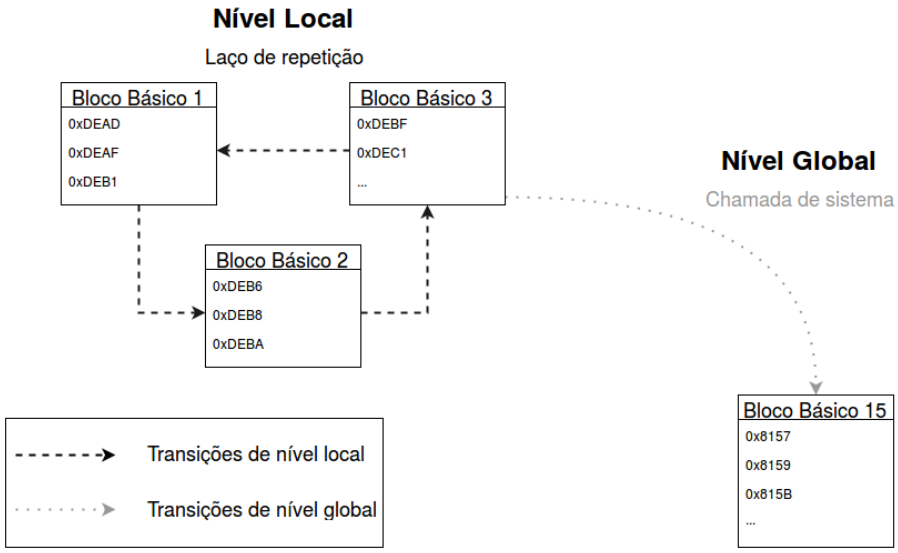

Figura 7: Exemplo de níveis do fluxo de controle de um programa

Lotes de amostras que possuem transições de nível global são úteis para determinar transições entre rotinas que são relacionadas. Para distinguir entre as transições que pertencem ao nível local ou ao nível global, é necessário analisar o lote de amostras e determinar se existem transições para blocos que são formados de instruções cujos endereços são significativamente diferentes dos demais endereços do lote. Uma vez que os blocos básicos são caracterizados pelos endereços de suas instruções, o desvio padrão da média aritmética entre os valores dos endereços das instruções de um lote (σ_l) pode ser utilizado para determinar se as transições contidas no lote são globais ou locais. Desta forma,

- σ_l `baixo` indica que as transições que pertencem ao lote são locais, pois a variação entre os endereços do lote é baixa.
- σ_l `alto` indica que a variação entre os valores dos endereços é muito maior, uma vez que pelo menos um endereço, que corresponde ao destino da instrução de salto global, é estatisticamente diferente dos demais endereços do lote. Neste caso, podem existir transições locais também, no entanto existe pelo menos uma transição global.

Outra medida gaussiana de interesse é a média aritmética dos endereços das instruções de um lote (μ_l) que pode ser usada para determinar quais regiões do código são quentes: médias recorrentes indicam regiões do código que são executadas repetidas vezes.

Portanto, é possível utilizar medidas gaussianas (médias e desvio padrão) sobre a sequência de endereços de cada lote (l) para determinar o grafo de fluxo de controle das regiões quentes do programa. A seguir são apresentadas algumas considerações práticas que viabilizam

o desenvolvimento de um algoritmo das premissas apresentadas.

4.2.2 CONSIDERAÇÕES

Em determinado lote é possível que existam diversas transições entre blocos básicos. Assim, as medidas gaussianas aplicadas em granularidade de lote (sobre todas as instruções de uma vez só), σ_l e μ_l, não permitem detectar de forma precisa os pontos do lote que as transições ocorrem. Além disso, em granularidade de lote, a contagem de recorrência de médias não é suficiente para aproximar a recorrência de blocos básicos, uma vez que a média dos endereços de todas as instruções do lote identificam uma região potencialmente maior que o tamanho dos blocos básicos.

Desta forma, as medidas gaussianas podem ser avaliadas utilizando janelas deslizantes de tamanho menor que o tamanho do lote. Assim, para um lote contendo M amostras e uma janela deslizante de tamanho N, as medidas gaussianas são calculadas para $n_w = M - N + 1$ janelas deslizantes, resultando em médias $\mu_w, w = 1, .., n_w$. Com efeito, o uso de janelas deslizantes aumenta a resolução das medidas, o que contribui com maior precisão na detecção de regiões quentes do código.

A Figura 8 ilustra um lote que contém 24 instruções (denotado pelos retângulos internos) e uma janela de tamanho 6. As médias das janelas são calculadas com os endereços das instruções que compõem cada janela.

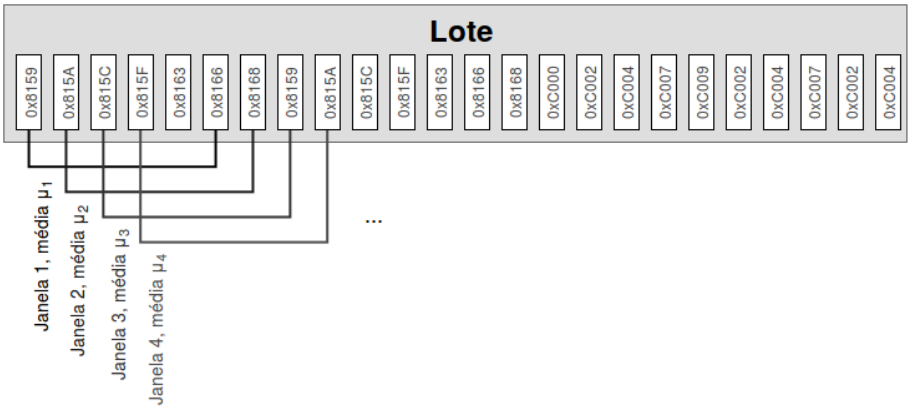

Figura 8: Calculo de médias com janelas deslizantes

Considerando a estratégia de utilizar janelas deslizantes para aumentar a resolução espacial das medidas gaussianas, o nível das transições de um lote pode ser determinado a partir do cálculo do desvio padrão das médias das janelas deslizantes do lote. As transições de um lote são marcadas como locais se o desvio padrão for menor que um limiar σ_t

(stdevThreshold). Desta forma, um lote contém apenas transições locais se e somente se $\sigma(\mu_w) < \sigma_t, w = 1, \ldots, n_w$.

As médias das janelas deslizantes μ_w em lotes que apresentam transições locais entre blocos básicos possuem valores apenas ligeiramente diferentes. Idealmente, todas as instruções que pertencem a um mesmo bloco básico devem ser **agrupadas** em um único grupo para que as execuções recorrentes de uma mesma região de código estejam de acordo, e seja possível detectar regiões quentes. Em outras palavras, isto permite que a contabilização das execuções de janelas que pertencem a um mesmo grupo seja acumulada no mesmo contador. Além disto, é importante que este agrupamento possa ser relacionado com o endereço de cada instrução do bloco, não somente a μ_w. Este mapeamento também facilita a construção do grafo de fluxo de controle, uma vez que é possível inferir o bloco básico que contém determinada instrução a partir de seu endereço, facilitando a implementação das estruturas subjacentes ao grafo de fluxo de controle.

Uma forma eficiente de agrupar as janelas com médias próximas se dá pela utilização de "baldes"(bins) auto-organizáveis em função das médias dos endereços das instruções das janelas. Os baldes B_k são identificados por rótulos (centróides) R_b, tal que $R_b = \mu(\mu_{b,i}), i = 1, \ldots, n_{bw}$, tal que b é o balde, $\mu_{b,i}$ é a média da janela i associada ao balde b e n_{bw} é o número de janelas associadas a um balde B_b. Além do rótulo, informações adicionais podem ser associadas aos baldes, tal como a contagem da quantidade de janelas que já foram executadas com médias suficientemente próximas ao centróide dos baldes (seja esta contagem C_b, para determinado balde b).

Em termos de implementação, os baldes podem ser colocados em um vetor $\mathbb{V}$ ordenado por R_b. Quando uma nova janela com média μ_w é considerada para inserção em um balde, μ_w é comparada aos centróides dos baldes já presentes. Para diminuir o espaço de busca para inserção, os baldes candidatos à inserção são encontrados, o que pode ser feito de forma eficiente utilizando um algoritmo de busca binária, uma vez que $\mathbb{V}$ está ordenado por R_b. Para considerar se os centróides mais próximos (R_{b-1}, R_b, R_{b+1} tal que R_b é o centróide do balde retornado pela busca binária) de μ_w são suficientemente próximos, a distância euclidiana entre cada centróide R_{b-1}, R_b, R_{b+1} e μ é calculada. Caso a distância ao centróide mais próximo seja menor que determinado limiar δ **(binSize)**, a janela é inserida no balde e o centróide correspondente é recalculado. Em outras palavras, C_k é incrementado, w é inserida em B_k e R_k é recalculado se e somente se

$$\left[\min_{b-1 \leq k \leq b+1} d(\mu_w, R_k) \right] \geq \delta$$

Caso contrário, um novo balde é inserido na posição acusada pela busca binária.

A Figura 9 ilustra a inserção de uma janela com média 33120. Os baldes estão ordenados pelo centróide (rótulo) e então é realizada uma busca binária utilizando a média da janela, 33120, para localizar o possível balde em que esta será inserida. Após a localização, neste caso o balde com centróide 33115, distância euclidiana entre a média da janela sendo inserida e os centróides dos baldes próximos ao rótulo 33115 (28036, 33115 e 33131) é calculada. A menor distância é então comparada com o limiar binSize. No caso da Figura 9, a menor distância da janela 33120 é para o balde de centróide 33115 (distância de 5) e esta distância ainda é menor que o limiar binSize (que é de tamanho 10). Logo esta janela é adicionada ao balde de centróide 33115 e o centróide é recalculado (para 33116). Caso a distância fosse maior que o limiar binSize, um novo balde balde seria criado e inserido na posição correta.

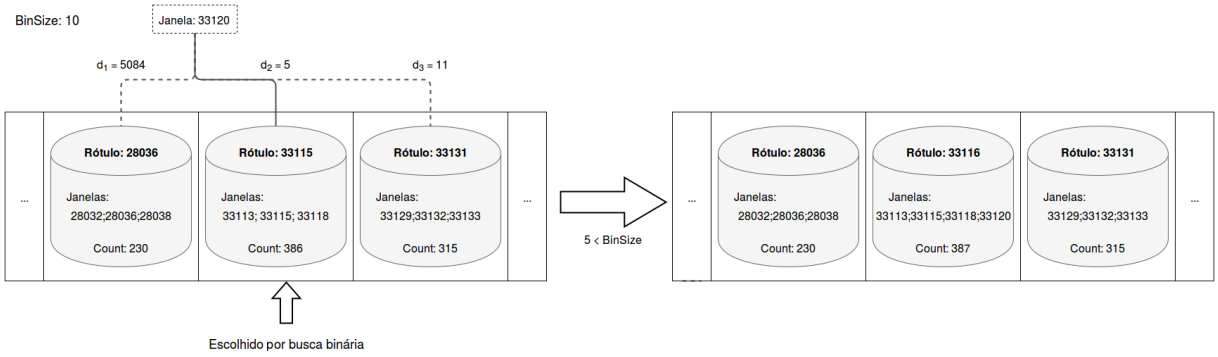

Figura 9: Exemplo de inserção de uma nova média

Como os centróides são construídos baseando-se nas médias das janelas deslizantes, dado o endereço de uma instrução, o balde que deve conter esta instrução pode ser encontrado aplicando uma busca binária na estrutura que armazena os baldes. É importante ressaltar que isto acontece quando somente janelas com desvio padrão baixo são inseridas nos baldes. Isto se deve ao fato que, janelas com desvio padrão baixo implicam em grupos de instruções com endereços mais próximos. Desta forma, o centróide dos baldes que agrupam estas janelas tendem a ser mais próximos dos endereços das instruções que compõe um determinado balde do que do centróide de outros baldes.

Desta forma, para determinar se uma determinada região é quente, a contagem de execuções de um determinado balde é avaliada. Se for maior que determinado limiar λ **(recurrentThreshold)**, é considerada quente. Portanto, a região de código representada pelo balde B_k é considerada quente se e somente se $C_k \geq \lambda$.

4.2.3 GFCGAUSSIANO

O algoritmo GFCGaussiano é um algoritmo de baixo custo que constrói um grafo de fluxo de controle incrementalmente, em tempo de execução. O algoritmo analisa as médias e os desvios padrão, mantém controle dos baldes, determina os blocos básicos e constrói o grafo de fluxo de controle levando em consideração as premissas e as considerações discorridas anteriormente. A ilustração de seu funcionamento é mostrada no fluxograma da Figura 10, seguido da explanação das rotinas e então da explanação do funcionamento em si. O pseudo-código deste algoritmo é apresentado no Apêndice 7.

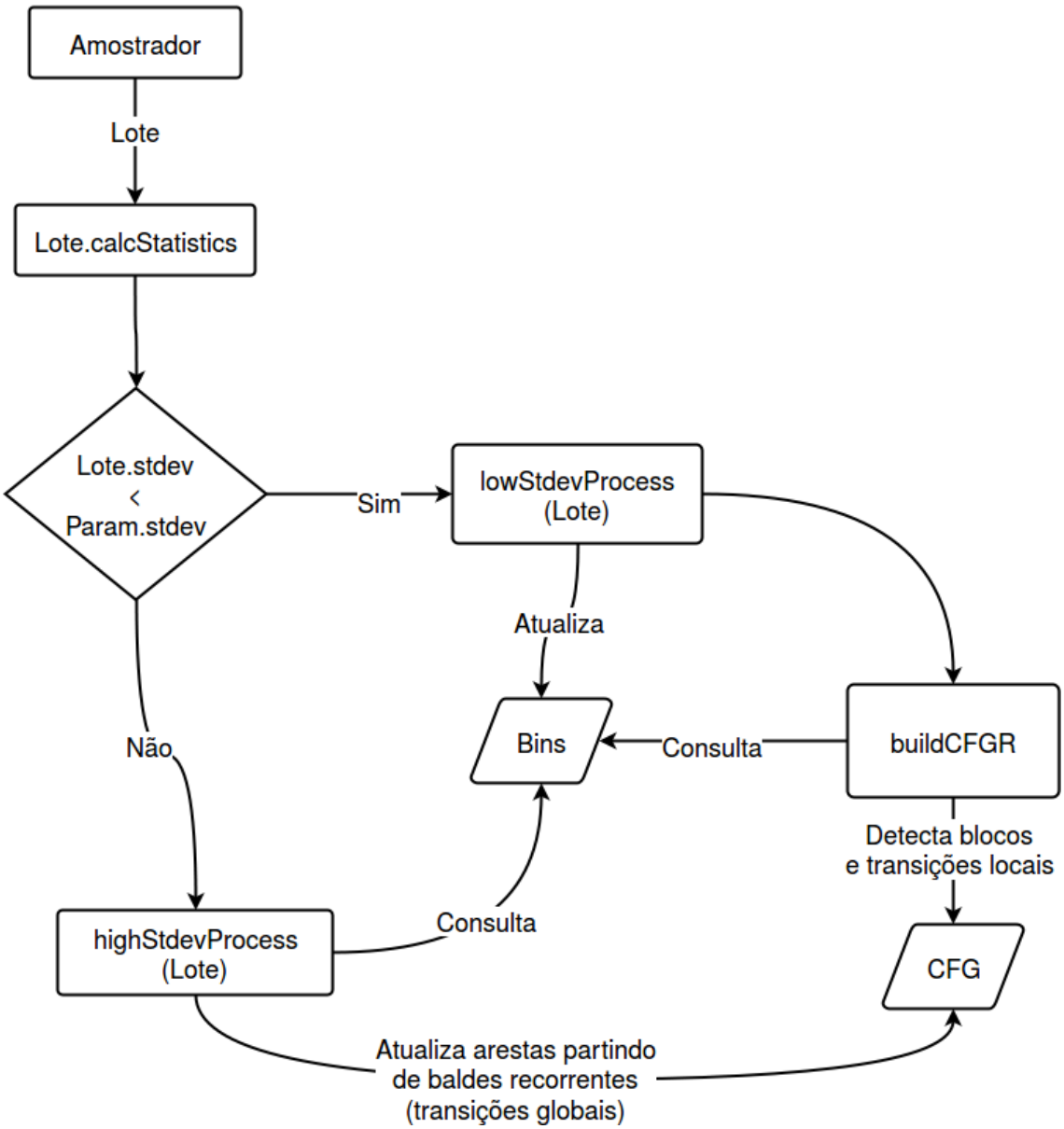

Figura 10: Ilustração do funcionamento do algoritmo

O algoritmo GFCGaussiano é um algoritmo iterativo e recursivo que possui três rotinas

principais para execução, explanadas abaixo.

buildCFGR: esta é a principal rotina do algoritmo. Esta rotina constrói os blocos básicos e as transições, de forma recursiva, analisando as instruções contidas no lote. Ela, basicamente, analisa sequencialmente cada instrução do lote e consulta a existência e a recorrência de um balde para esta instrução. O algoritmo é continuado se, e somente se, um balde para a instrução existir e se a recorrência dele for maior que o `recurrentThreshold`. Então, a instrução é analisada e caso não seja uma instrução de salto um bloco básico é gerado ou coletado (se a instrução já pertencer à algum bloco básico). Caso esta seja uma instrução de salto, o ponto final de um bloco básico é definido e um bloco básico é gerado e então esta rotina é chamada novamente (dando inicio à recursão) para que um novo bloco básico seja gerado. A rotina retorna assim que acabarem as instruções do lote e ao retornar uma aresta entre o bloco básico construído anteriormente (antes de chamar a rotina novamente) e o bloco básico atual é ligada e o grafo de fluxo de controle vai sendo construído incrementalmente.

lowStddevProcess: é a rotina executada caso o desvio padrão do lote seja baixo, indicando que as transições do fluxo de controle contidas no lote podem estar no nível local. Esta rotina é responsável pelo gerenciamento dos baldes, criando novos baldes, caso não exista ou caso a distância dos centróides forem maior que o limiar `binSize`, adicionando janelas e recalculando centróides e inserindo novos baldes em suas devidas posições. Uma vez que os baldes são gerenciados a rotina de buildCFGR é chamada para construção do grafo de fluxo de controle.

highStddevProcess: é a rotina executada caso o desvio padrão do lote seja alto, indicando que as transições do fluxo de controle do programa podem estar no nível global. Basicamente essa rotina faz com que apenas transições entre blocos básicos recorrentes sejam criadas ou atualizadas. As instruções são analisadas sequencialmente e o balde que a instrução pertence é coletado. Caso o balde seja recorrente, o balde da próxima instrução é coletados e caso também seja recorrente a rotina buildCFGR é chamada e são criadas ou incrementadas as arestas entre os blocos básicos que pertencem estas instruções.

O algoritmo possui o funcionamento como mostrado na Figura 10 e é executado sempre que o amostrador realiza a coleta de um lote. Sempre que uma amostragem é realizada, as estatísticas necessárias são calculadas (médias e desvio padrão das janelas). E então o desvio padrão do lote é analisado e o fluxo da execução do algoritmo é determinado por seu valor (esta condição é denotada pelo losango na Figura 10). Caso o desvio padrão seja baixo a rotina lowStddevProcess é chamada e caso o desvio padrão seja alto a rotina highStddevProcess é chamada. Ao fim, o grafo de fluxo de controle das regiões quentes é construído.

O pseudo-código mostrado no 7 mostra, de forma comentada, algoritmo e suas rotinas tão quanto algumas estruturas auxiliares que podem ser utilizadas para sua implementação. A rotina GFCGaussiano é a rotina executada sempre que uma amostragem é realizada.

5 RESULTADOS

O algoritmo GFCGaussiano tem como principal objetivo construir um grafo de fluxo de controle das regiões quentes de um programa baseando-se em lotes de amostras coletadas ao longo da execução. A principal vantagem do uso de amostragem é que o custo do processo de amostragem na emulação é mais baixo que o custo da instrumentação. No entanto, o grafo obtido a partir de técnicas baseadas em amostragem tende a ser menos preciso do que um grafo obtido por técnicas de instrumentação. Desta forma, para avaliar o algoritmo proposto é necessário comparar os grafos gerados por ele e os grafos exatos das execuções construídos via instrumentação. Neste Capítulo, são apresentados os resultados obtidos pelo algoritmo em um ambiente de emulação, bem como uma discussão sobre a escolha de parâmetros e seus respectivos efeitos nos resultados.

5.1 PROVA DE CONCEITO

A técnica de perfilamento por amostragem proposta na seção 4.1 foi implementada no Jrynes (FOLEISS, 2012), um emulador do *videogame* Nintendo Entertainment System (NES), escrito na linguagem C, que possui um interpretador e um tradutor de binários dinâmico. O *videogame* NES conta com 13 modos de endereçamento distintos e um processador MOS 6502 de 8 bits com um conjunto de instruções CISC, onde as instruções podem variar de 1 a 3 bytes (DISKIN, 2004). A arquitetura do NES foi escolhida devido a sua simplicidade, boa documentação e conhecimento prévio pelos membros da equipe e, apesar de antiga, possui muitos desafios semelhantes aos dos dias atuais.

O amostrador foi implementado na linguagem C em uma *thread* separada do emulador com o intuito de não interromper o processo de emulação e possuir visibilidade aos dados, uma vez que compartilha o espaço de endereçamento com as rotinas de emulação. A implementação faz com que a cada `X` milissegundos o amostrador sinalize o emulador para iniciar a coleta sequencial dos `N` valores assumidos pelo registrador PC e então os armazenar em um *buffer*. Ao término da coleta do lote, o conteúdo do *buffer*, juntamente com algumas informações adicionais como a desmontagem e o tipo da instrução, são registrados em um arquivo de texto para posteriormente servirem como entrada para o algoritmo de construção de grafo de fluxo de controle.

O algoritmo de construção de grafo de fluxo de controle GFCGaussiano proposto na Seção 4.2 foi implementado na linguagem Python por ser mais simples e mais rápido de implementar, dado o tempo disponível para implementação e testes. Embora importante, decidimos por deixar a análise de tempo de execução do algoritmo para trabalhos futuros, uma vez que isto requer que o algoritmo seja implementado diretamente no emulador para avaliar o custo adicional que a tarefa de construção do grafo de fluxo de controle acarreta no tempo total de execução. Portanto, o foco da análise de resultados deste trabalho está na semelhança dos grafos, conforme explicado anteriormente.

A implementação toma como entrada o arquivo gerado pelo amostrador contendo as amostras e também todos os parâmetros do próprio algoritmo discorridos na seção 4.2.2. O grafo de fluxo de controle construído a partir das amostras é gerado incrementalmente ao longo da execução do algoritmo, que atualiza o grafo a cada lote processado.

5.2 AMBIENTE EXPERIMENTAL

Um conjunto de parâmetros são necessários para o funcionamento do algoritmo GFCGaussiano e cada um deles impactam no processo de construção do grafo de fluxo de controle. Os valores inferidos para estes parâmetros foram determinados a partir da análise do comportamento dos jogos e da análise das características do grafo de fluxo de controle gerado pela instrumentação. A escolha destes parâmetros está documentada na Seção 5.4. O grafo instrumentado foi obtido executando os jogos no Jrynes e habilitando o Tradutor Dinâmico de Binários, implementado no próprio emulador, que instrumenta toda a execução e produz um grafo de fluxo de controle completo da execução do programa. No entanto, nenhuma tradução de binários é realizada, uma vez que o tradutor implementado não realiza perfilamento em rotinas traduzidas. Cada jogo utilizado neste trabalho foi gravado e suas gravações foram executadas no Jrynes por aproximadamente cinco minutos. No total foram executados oito jogos, sendo eles:

- 1942
- Ballon Fight
- Dig Dug
- Donkey Kong
- Mario Bros

- Pac-Man
- Popeye
- Super Mario Bros

Como a similaridade entre os grafos gerados pelo GFCGaussiano e os grafos gerados por instrumentação é a principal métrica de avaliação do desempenho do algoritmo proposto neste trabalho, a função de similaridade é apresentada a seguir. Sejam $G = \{V_G, A_G\}$ e $H = \{V_H, A_H\}$ grafos dirigidos, tal que G é o grafo gerado pelo algoritmo GFCGaussiano e H é o grafo gerado por meio de instrumentação pelo emulador. A Similaridade entre G e H é dada pela razão entre as arestas em comum em ambos os grafos e a quantidade de arestas em H. Em outras palavras, a similaridade dos grafos é função da quantidade de arestas em comum nos dois grafos em razão da quantidade de arestas no grafo H. Formalmente,

$$\text{Similaridade}(G,H) = \frac{\#(A_G \cap A_H)}{\#A_H}$$

considerando "#"como operador *cardinalidade de conjunto*.

Além disso, dado a premissa de localidade espacial, a amostragem tende a registrar apenas regiões recorrentes do programa. Como o grafo de fluxo de controle instrumentado é completo, ele abrange tanto regiões recorrentes quanto regiões não recorrentes. Como o objetivo do algoritmo GFCGaussiano e deste trabalho é construir um grafo a partir dos dados amostrados apenas das regiões recorrentes, a função de similaridade deve ser aplicada somente sobre o grafo amostrado e a região recorrente do grafo instrumentado. Desta forma, para medir a similaridade foi utilizado o grafo instrumentado quente, que é composto apenas pelas arestas (e seus blocos básicos) com maior número de transições e que a soma destas transições correspondem a 90% do total de transições de todas as arestas do grafo instrumentado.

5.3 DISCUSSÃO DO AMOSTRADOR

O número de amostras no lote e a frequência de amostragem influencia no trabalho do algoritmo GFCGaussiano, já que com mais instruções amostradas é possível analisar melhor o fluxo de controle do programa. Como um bloco básico possui em média 6 instruções (FOLEISS, 2012), o número de amostras por lote foi definido como 25, desta forma é possível determinar cerca de 4 blocos básicos por lote no melhor caso. Todavia, quanto maior o número de instruções por lote, maior o tempo de execução do amostrador. Naturalmente, quanto maior

o intervalo de amostragem entre lotes, mais tempo será necessário para a construção do grafo de fluxo de controle, uma vez que a rotina de construção é invocada ao término da coleta de cada lote.

A Tabela 1 mostra a quantidade de lotes, o número total de instruções executadas e a taxa de instruções amostradas para cada jogo executado. As amostras foram coletadas com intervalo entre lotes de aproximadamente $3ms$, cada lote contendo 25 amostras. Nota-se que, embora todos os jogos tenham sido executados por exatamente 5 minutos, a quantidade de lotes coletados variou significativamente, média de 104.757 ± 23.436. Isto é resultado dos testes serem executados em um sistema operacional interativo, onde os temporizadores disponíveis ao usuário não necessitam garantir resolução temporal. No entanto, isto não impactou de forma negativa os resultados obtidos pelo GFCGaussiano, apresentado na próxima Seção. Estes parâmetros foram escolhidos baseando-se em alguns experimentos não mostrados neste trabalho. Um estudo mais rigoroso dessa parametrização deve ser realizado para tirar conclusões mais assertivas em relação à escolha destes parâmetros.

Tabela 1: Perfil das amostragens

Jogo	Qtde. lotes	Tam. lote	Total instruções executadas	(Qtde. * Tam.) / Total (%)
1942	100.678	25	206.669.737	1,21
Ballon Fight	96.043	25	206.182.658	1,16
Dig Dug	94.325	25	187.979.198	1,25
Donkey Kong	87.918	25	153.165.705	1,43
Mario Bros	129.890	25	163.787.250	1,98
Pacman	92.582	25	184.049.378	1,25
Popeye	151.572	25	165.474.083	2,28
Super Mario	85.050	25	181.871.018	1,16

É importante ressaltar que no Jrynes, utilizando o tradutor de binários dinâmico, toda instrução é instrumentada e a instrumentação corresponde a 49% da execução do emulador (FOLEISS, 2012), considerada alta, uma vez que o emulador também realiza muitas outras tarefas além da emulação de instruções. Como mostrado na Tabela 1 a amostragem utiliza em média 1.5% das instruções executadas para posteriormente serem analisadas pelo algoritmo GFCGaussiano. Como o esforço por instrução da amostragem e da instrumentação é comparável, uma vez que consistem em anotar o estado da máquina em uma estrutura de dados interna, é esperada redução média de 98.53% no tempo necessário para registro dos caminhos de execução ao utilizar a abordagem baseada em amostragem.

5.4 DISCUSSÃO DA CONSTRUÇÃO DO GRAFO DE FLUXO DE CONTROLE

Para a construção do grafo de fluxo de controle utilizando o algoritmo GFCGaussiano, os parâmetros foram inferidos com base na avaliação do comportamento dos jogos e da teoria apresentada na Seção 4.2.2. Cada combinação de parâmetros foi executada a fim coletar os resultados e realizar uma análise das similaridades entre os grafos instrumentados e amostrados, tanto quanto uma analise dos próprios parâmetros. Os parâmetros utilizados pelo algoritmo são mostrados na Tabela 2.

Tabela 2: Parâmetros do algoritmo de construção de grafo de fluxo de controle

binSize (bytes)	8, 10, 13, 16
stdevThreshold	8, 10, 12, 15
windowSize (instruções)	6, 10 ,13
recurrentThreshold	5, 8, 12

O parâmetro `stdevThreshold` é utilizado para decidir se um dado lote possui transições de níveis locais ou globais. Este parâmetro é complexo de ser inferido pois é intrínseco ao programa em execução. O desvio padrão varia de acordo com a forma que as rotinas e saltos do programa foram escritos. Desta forma, um programa pode ser composto de diversas transições de nível global, e pouquíssimas transições de nível local, dificultando a geração e manutenção dos baldes. Os valores inferidos para este parâmetro foram determinados executando os programas com combinações diversas e observando a média dos desvios padrão que foram utilizados para criação dos baldes. De tal forma, foi possível notar que esta média varia de 1 a 7, portanto valores maiores que 8 foram utilizados para este parâmetro. Contudo, este parâmetro ainda deve ser melhor estudado para ser devidamente usado.

Para o `recurrentThreshold` os valores foram inferidos analisando os melhores resultados em experimentos prévios. Com isso foi possível determinar o valor do parâmetro para 5. Para valores muito menores, blocos básicos não recorrentes estavam sendo gerados, degradando o grafo de fluxo de controle. Desta maneira, alguns valores maiores geravam resultados mais consistentes e desta forma os valores 8 e 12 foram selecionados. Os valores do parâmetro `binSize` foram inferidos calculando o número de instruções por lote juntamente com a média do tamanho das instruções. Desta forma, para o caso onde um bloco básico contenha apenas instruções de 3 bytes, a média do tamanho do bloco é de 18 bytes. Analogamente, caso um bloco básico contenha apenas instruções de 1 byte, a média do tamanho do bloco é de 6. Portanto, valores dentro deste intervalo, de 6 a 18, como 8, 10, 13 e 16 foram inferidos a este parâmetro.

Ao todo, são 144 combinações de parâmetros distintas que foram executadas para cada

jogo. Os resultados são apresentados na próxima Seção, juntamente com a discussão sobre os resultados obtidos.

5.4.1 ANÁLISE DOS RESULTADOS

Os resultados dos experimentos utilizando as combinações de parâmetros são dispostos nas Tabelas 3 e 4, onde a primeira mostra as três melhores taxas de similaridade de cada jogo e a outra mostra as três piores taxas de similaridade de cada jogo, respectivamente. Na Tabela 3 é possível notar a acurácia do grafo de fluxo de controle amostrado. Para grande parte dos jogos executados a similaridade dos grafos é de mais de 70%, obtendo as taxas mais altas nos jogos Mario Bros e Popeye, que são de 83%. É também possível notar que, para os melhores resultados, o tamanho da janela (`windowSize`) e do `binSize` tendem aos valores mais altos entre os testados, enquanto `recurrentThreshold` tende a assumir valores menores entre os testados.

Tabela 3: Melhores resultados das similaridades para os jogos

Jogo	binSize	stdevThres.	windowSize	recurrentThres.	Baldes	Similaridade
1942	10	15	13	5	252	28%
1942	13	15	13	5	208	28%
1942	10	15	13	8	252	28%
Ballon Fight	13	8	13	5	57	80%
Ballon Fight	16	8	13	5	47	80%
Ballon Fight	16	8	13	8	47	80%
Dig Dug	13	15	13	5	279	75%
Dig Dug	8	15	13	5	238	76%
Dig Dug	10	15	13	5	192	77%
Donkey Kong	16	10	13	12	56	67%
Donkey Kong	16	15	13	5	84	68%
Donkey Kong	16	15	13	8	84	68%
Mario Bros	16	10	10	12	54	83%
Mario Bros	16	10	10	8	54	83%
Mario Bros	16	12	10	8	61	83%
Pac-Man	8	15	13	5	211	69%
Pac-Man	10	15	13	5	186	69%
Pac-Man	16	15	13	8	129	69%
Popeye	16	12	10	5	79	83%
Popeye	16	15	10	5	95	83%
Popeye	10	15	13	5	145	83%
Super Mario	10	15	13	5	165	63%
Super Mario	13	15	13	5	228	63%
Super Mario	13	12	13	5	186	64%

Conforme discutido na Seção 4.2.2, `recurrentThreshold` é o parâmetro utilizado para determinar se os blocos básicos são recorrentes. Quanto maior este limiar, mais recorrente

um determinado balde deve ser para ser considerado quente. Caso este parâmetro seja muito baixo, blocos básicos com instruções não recorrentes são gerados. Nos resultados dispostos na Tabela 3 é possível notar que o melhor valor assumido para este limiar foi quase sempre de 5. Seu impacto quando um valor grande é assumido é mostrado na Tabela 4 onde os grafos são menos similares. Isto se deve ao fato que, devido à natureza do perfilamento por amostragem, mesmo regiões recorrentes podem ser amostradas poucas vezes, embora isto seja menos provável de acontecer do que regiões não-recorrentes serem amostradas regularmente. Portanto, um valor alto para `recurrentThreshold` pode recusar blocos básicos recorrentes que não tenham sido amostrados o suficiente.

Tabela 4: Piores resultados das similaridades para os jogos

Jogo	binSize	stdevThres.	windowSize	recurrentThres.	Baldes	Similaridade
1942	8	8	6	12	83	16%
1942	8	8	6	12	83	16%
1942	10	8	6	12	73	16%
Ballon Fight	8	12	10	12	79	39%
Ballon Fight	8	15	10	12	127	45%
Ballon Fight	8	10	10	12	162	46%
Dig Dug	8	8	6	12	85	28%
Dig Dug	8	8	6	12	85	28%
Dig Dug	8	8	6	8	85	29%
Donkey Kong	10	8	6	5	47	43%
Donkey Kong	13	8	6	5	41	43%
Donkey Kong	13	8	6	8	41	43%
Mario Bros	8	8	6	8	29	53%
Mario Bros	10	8	6	8	26	53%
Mario Bros	13	8	6	8	23	53%
Pac-Man	10	8	6	8	70	28%
Pac-Man	10	8	6	12	70	28%
Pac-Man	10	8	6	5	70	30%
Popeye	8	15	6	12	116	66%
Popeye	8	12	10	12	122	68%
Popeye	8	15	10	12	150	69%
Super Mario	8	8	10	12	72	26%
Super Mario	8	10	10	12	100	26%
Super Mario	8	8	6	5	143	30%

O `binSize` determina a distância que uma janela deve possuir para ser anexada a um determinado balde ou para a criação de um novo. Ele também determina, indiretamente, intervalo de abrangência dos endereços representados pelos baldes. Na tabela 3 `binSize` maiores foram utilizados para obter melhores resultados, quase sempre sendo o valor 16, o maior testado, enquanto que na Tabela 4 o `binSize` quase sempre assume valores menores como 8 e 10 e a similaridade é baixa. Neste caso, os resultados menos favoráveis estão relacionados a `binSizes` baixos pois valores baixos podem implicar na separação de blocos

básicos maiores em mais de um balde, implicando na desclassificação de regiões quentes.

Conforme a Tabela 3, a grande maioria dos melhores resultados obtidos está relacionada à maior janela testada, contendo 13 instruções. Isto está relacionado ao fato que janelas maiores tendem a capturar mais transições por janela, aumentando a confiança do desvio padrão. Portanto, nestes casos, um desvio padrão baixo indica que as transições locais são realmente recorrentes, implicando em uma escolha melhor das regiões com localidade espacial.

Jogos como 1942 podem possuir a estrutura de código pouco diferente das demais (devido as limitações da época), talvez com muitas transições de nível local ou blocos básicos muito pequenos, tendo muitos blocos básicos pouco recorrentes e degradando a construção do grafo de fluxo de controle. O melhor resultado do jogo 1942 é de 28%, como mostra na Tabela 3. No entanto, um estudo mais criterioso em relação ao código e à estrutura dos programas é necessário para explicar os fenômenos relacionados ao baixo desempenho do algoritmo em alguns casos.

Apesar da explicação do possível impacto de cada parâmetro em específico na melhoria ou na degradação do resultado dos experimentos, dentre as 144 combinações de parâmetros distintas, algumas delas em especial foram, em média, as que geraram os melhores e os piores resultados entre as demais. Essa análise em conjunto pode facilitar a visualização de alguns padrões implícitos existentes. Nos experimentos, as 144 combinações (chamadas de configurações) foram enumeradas de 0 a 143 e testadas para cada jogo.

A configuração que obteve os melhores resultados, entre o maior número de jogos, foi a configuração #45 composta pelos valores dos parâmetros: `binSize` = 10; `stdevThreshold` = 15; `Tamanho da janela` = 13; `recurrentThreshold` = 5. Na Tabela 3 é possível notar esta configuração para 5 dos 8 jogos, sendo eles: 1942, Dig Dug, Pac-Man, Popeye e Super Mario. No jogo Popeye esta configuração obteve 83% de similaridade. Em contrapartida, configurações como a #97, composta pelos valores dos parâmetros: `binSize` = 8; `stdevThreshold` = 8; `Tamanho da janela` = 6; `recurrentThreshold` = 12, e também como a #128, composta pelos valores: `binSize` = 8; `stdevThreshold` = 8; `Tamanho da janela` = 6; `recurrentThreshold` = 12, foram as que obtiveram os piores resultados, sendo possível observá-las na Tabela 4 em jogos como 1942, Pac-Man e Dig Dug onde 16%, 28% e 28% foram suas taxas de similaridade, respectivamente.

É possível constatar que as explicações dos parâmetros condizem com as configurações degradando e melhorando o desempenho, porém uma análise mais criteriosa deve analisar de forma conjunta estas configurações e determinar quais as relações existentes entre os parâmetros e o impacto gerado nos resultados.

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou um algoritmo de construção de grafos de fluxo de controle iterativo baseado em perfilamento por amostragem de lotes denominado GFCGaussiano. As principais decisões de projeto para a construção do algoritmo estão embasadas em propriedades estatísticas relacionadas a medidas estatísticas clássicas como médias e desvio padrão, juntamente com observações sobre a natureza da execução de programas, como a localidade temporal.

Uma vez selecionados os parâmetros corretamente, o algoritmo proposto mostrou-se preciso, sendo possível construir grafos de fluxo de controle das regiões recorrentes (as regiões de real interesse) com similaridade média entre 60% e 80% ao grafo de fluxo de controle quente gerado por instrumentação, como mostrado na Tabela 3. O algoritmo também se mostrou eficiente, já que foi possível construir grafos com alta similaridade com relativamente poucas amostras (apenas 2% das instruções totais executadas).

Em função da natureza complexa da execução de programas e da modelagem baseada em medidas estatísticas simples, o algoritmo utiliza alguns parâmetros como entrada para guiar o processo de construção dos grafos. Portanto, alguns estudos foram realizados para estimar valores iniciais para estes parâmetros. Os resultados destes estudos estão inclusos nos Capítulos 4 e 5, também representam parte das contribuições deste trabalho.

O principal diferencial do algoritmo desenvolvido é a construção de grafos de fluxo de controle de forma incremental, sendo possível executá-lo em tempo de execução, baseado em amostragem de lotes, com baixo custo, utilizando-se apenas de medidas gaussianas juntamente com o conceito de janelas deslizantes para determinação dos blocos básicos e as transições entre eles.

Em virtude do tempo alocado para o TCC, do tamanho da equipe e da complexidade do trabalho, alguns tópicos interessantes não foram abordados neste trabalho. Como trabalhos futuros algumas tarefas para melhoria e maior abrangência da técnica de perfilamento e do algoritmo GFCGaussiano são listadas abaixo.

- Estudar os parâmetros de amostragem e determinar qual a melhor proporção entre o tempo da amostragem e a quantidade de amostras por lote, tanto para diminuir o custo da

técnica quanto para aumentar a precisão do algoritmo GFCGaussiano;

- Realizar uma análise formal do algoritmo GFCGaussiano;
- Realizar mais experimentos para determinar as relações entre os parâmetros, as características dos programas e as arquiteturas.
- Implementar o GFCGaussiano em alguma máquina virtual para avaliar o custo não somente da amostragem, mas também o custo da construção do grafo de fluxo de controle;
- Executar o GFCGaussiano em outras arquiteturas e avaliar sua eficiência e precisão.

7 APÊNDICE

Neste apêndice é disposto o pseudo-codigo do algoritmo GFCGaussiano, mostrado na Secao 4.2.3. Este pseudo-codigo esta é similar a um codigo na linguagem Python, foi separado por suas rotinas principais e cada uma delas foi comentada visando o melhor entendimento. Os parametros do algoritmo, discorridos na Secao 4.2.2, estao contidos na variavel `param`.

```
/* Algumas estruturas utilizadas para armazenar os dados para construcao do GFC */
Targets = dictionary()
Bins = ordered_array()
BBRepository = dictionary()

/* Esta rotina Ã© executada apos a coleta de cada lote (batch) */
GFCGaussiano (batch):
    /* Calcula as mÃ©dia das janelas e o desvio padrao. As mÃ©dias das janelas ficam
        armazenadas em batch.windows e o desvio padrao das mÃ©dias das janelas em batch.
        stddev */
    batch.calcStatistics (param.windowSize)

    /* Desvio padrao baixo (nÃvel local), executa a rotina para gerenciar os baldes e
        entao a rotina para construir o GFC */
    if batch.stddev <= param.stddevThreshold:
        lowStddevBatchProcess(batch)
        buildCFGR(iterator(batch), batch, false)

    /* Desvio Padrao alto (nÃvel global). Cria arestas no grafo caso encontre alguma
        instrucao de salto recorrente. */
    else:
        highStddevBatchProcess(batch)


/* Funcao auxiliar que cria uma aresta entre dois blocos basicos */
createEdge(sourceBB, targetBB):
    sourceBB.addTarget(targetBB)
    targetBB.addSource(sourceBB)
    CFG.createOrIncrementEdge(sourceBB, targetBB)
```

Listing 7.1: Rotina lowStddevProcess

```
/* Esta rotina Ã© executada caso o desvio padrao seja baixo. Ela Ã© responsavel pelo
    gerenciamento dos baldes */
lowStddevBatchProcess (batch):
    for all W in batch.windows:
        /* Encontra a distancia do centroide mais proximo da mÃ©dia W */
        distanceToNearest = Bins.getNearstBinDist(W)

        /* Caso nao exista nenhum balde: cria-se um novo balde e adiciona W Ã s janelas
            existentes deste balde. O endereco do centroide Ã© recalculado a cada
            insercao de janelas no balde */
        if distanceToNearest == infinity:
            Bins.append(createBin(W))
            Bins[0].windows.append(W)

        else:
```

```
13              /* Se a distancia do centroide encontrada for menor que o limiar binSize, o
                    balde do centroide Ã© localizado e a janela Ã© inserida (caso nao esteja
                    no balde) */
14              if distanceToNearest <= param.binSize:
15                  positionNearestBin = Bins.getNearstBinPos(W)
16                  Bins[positionNearestBin].count += 1
17
18                  if W not in Bins[positionNearestBin].windows:
19                      Bins[positionNearestBin].windows.append(W)
20
21              /* Se a distancia do centroide for maior que o tamanho do limiar binSize,
                    procura-se o local de insercao, cria-se um novo balde e adiciona a janela
                     W ao novo balde  */
22              else:
23                  finalPos = Bins.getBinPosition(W)
24                  Bins.insert(createBin(W), finalPos)
25                  Bins[finalPos].windows.append(W)
```

Listing 7.2: Rotina highStddevProcess

```
1  /* Esta rotina Ã© executada caso o desvio padrao seja alto. */
2  highStddevBatchProcess (batch):
3      for all inst in batch:
4          /* Se for uma instrucao de salto, cria uma nova chave ou incrementa a existente
               em Targets e pega o balde que possui o endereco */
5          if isBranchOrCall(inst):
6              Targets.createOrIncrementEdge(inst.target)
7              bin = Bins.getBinFromAddress(inst.pc)
8
9              if bin not exists : continue
10
11             /* Pega o bloco basico que contÃ©m o pc */
12             basicBlock = BBrepository.getBB(inst.pc)
13
14             /* Se existir o bloco basico e a sua contagem de execucoes do balde for maior
                    que recurrentThreshold, pega a proxima instrucao apos instrucao */
15             if  basicBlock exists and bin.count >= param.recurrentThreshold
16                 nextInst = batch.getInstAfter(inst)
17
18                 if nextInst not exists : continue
19
20                 nextInstBB = BBrepository.getBB(nextInst.pc)
21
22                 /* Se o bloco basico da proxima instrucao nao existir, chama a funcao de
                       criacao de grafo recursiva, com o parametro justBuild sendo
                       verdadeiro */
23                 if nextInstBB not exists:
24                     buildCFGR(iterator(batch), batch, true)
25
26                 /* Se o bloco basico existe, cria uma aresta de basicBlock para anotherBB
                        */
27                 else:
28                     createEdge(basicBlock, nextInstBB)
```

Listing 7.3: Rotina recursiva para criacao do GFC (buildCFGR)

```
1  /* Esta funcao recursiva Ã© responsavel pelo reconhecimento de transicoes em nÃ vel local
       entre blocos basicos quentes. O parametro justBuild Ã© utilizado para forcar a
      construcao de um bloco basico, mesmo que nao seja quente. */
2  buildCFGR (batch-iterator, batch, justBuild):
3      /* Itera sobre as instrucoes do batch */
4      for inst em batch-iterator:
5          if isBranchOrCall(inst):
```

```
6               Targets.createOrIncrementTargets(inst.target)
7
8           /* Pega o balde que possui o endereco. */
9           bin = Bins.getBinFromAddress(inst.pc)
10
11          if bin not exists and justBuild == false:
12              return
13
14          /* O balde Ã© recorrente se ele existir e sua contagem for maior que o
                recurrentThreshold */
15          recurrent = (bin exists and bin.count > param.recurrentThreshold)?
16
17          if recurrent or justBuild:
18              /* Se Targets ja possuir a chave ou o bloco estiver criado, pega ou cria (
                    caso nao exista) um novo bloco basico. Isto faz com que novos blocos
                    basicos so sejam construÃdos a partir de enderecos conhecidos por serem
                    alvos de instrucoes de fluxo de controle. */
19              if Targets.has_key(inst.pc) or justBuild:
20                  basicBlock = BBrepository.getBBOrCreate(inst.pc)
21
22                  /* Enquanto o bloco nao estiver terminado, itera-se sobre as instrucoes
                        para localizar uma instrucao que termina um bloco basico. Neste caso,
                         consideramos que as instrucoes de fluxo de controle desempenha este
                        papel. */
23                  while not bb.done:
24                      anotherBB = BBrepository.getBB(inst.pc)
25
26                      /* Se iterando sobre as instrucoes, localizar um bloco basico
                            diferente (com pontos de entrada diferentes), o bloco basico Ãé
                            concluÃdo e cria-se uma aresta de basicBlock para anotherBB */
27                      if anotherBB exists:
28                          if anotherBB.entryAddress != basicBlock.entryAddress
29                              basicBlock.done = true
30                              createEdge(basicBlock, anotherBB)
31                              break /*termina a criacao do bloco basico atual, uma vez que
                                    um bloco basico ja inicia na proxima instrucao.*/
32
33                      /* Se o bloco basico nao tiver a instrucao, adicione-a */
34                      if not basicBlock.has_instruction(inst.pc):
35                          basicBlock.addInstruction(inst)
36
37                      /* Se iterando sobre o bloco encontrar uma instrucao de salto,
                            localiza-se o fim do bloco basico, marcando-o como feito e entao
                            verifica-se a instrucao apos o salto para saber o alvo */
38                      if isBranchOrCall(inst):
39                          basicBlock.done = true
40                          nextInst = batch.getInstAfter(i)
41
42                          /* Se nao existir a proxima instrucao (ja estiver no fim do batch
                                ), termina o while e o bloco fica sem o alvo */
43                          if nextInst not exists: break
44
45                          /* Se a proxima instrucao for a mesma que o alvo do salto, o
                                caminho especificado pelo salto foi tomado, portanto chama-se
                                 o buildCFGR, para construir o proximo bloco. Ao termino, o
                                targetBB tera sido construÃdo. */
46                          se nextInst.pc == inst.target
47                              Targets.createOrIncrementTargets(inst.target)
48                              buildCFGR(batch-iterator, batch, false)
49                              targetBB = BBRepository.getBB(inst.target)
50
51                          /* Se o caminho do salto nao tiver sido tomado, Ãé construÃdo o
                                bloco basico utilizando a proxima instrucao */
52                          senao:
```

```
53                              createOrIncrementTargets(nextInst.target)
54                              buildCFGR(batch-iterator, batch, false)
55                              targetBB = BBRepository.getBB(nextInst.pc)
56
57                          /* Se o targetBB existir (o buildCFGR construiu o bloco e o bloco
                                ficou pronto), cria-se uma aresta de basicBlock para
                               targetBB. O targetBB pode nao existir, caso durante a criacao
                                do Bloco Basico o lote terminar */
58                          se targetBB exists:
59                              createEdge(basicBlock, targetBB)
60
61                      /* Iterar sobre o lote. Sai do laco quando nao houver mais instrucoes
                            */
62                      inst = batch-iterator.getNext()
63                      if inst not exists:
64                          break;
65
66          /* Se a instrucao atual for um branch e a proxima instrucao existir */
67          if isBranchOrCall(inst):
68              nextInst = batch.getInstAfter(inst)
69
70              /* Se existir blocos basicos ja criados para instrucao atual e para proxima
                    instrucao, cria-se uma aresta de thisBB para targetBB */
71              if nextInst exists:
72                  targetBB = BBRepository.getBB(nextInst.pc)
73                  thisBB = BBRepository.getBB(inst.pc)
74
75                  if targetBB exists and thisBB exists:
76                      createEdge(thisBB, targetBB)
```

REFERÊNCIAS

ALLEN, F. E. Control flow analysis. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 5, n. 7, p. 1–19, jul. 1970. ISSN 0362-1340. Disponível em: http://doi.acm.org/10.1145/390013.808479.

ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Guest editors' introduction: Welcome to the opportunities of binary translation. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 33, n. 3, p. 40–45, mar. 2000. ISSN 0018-9162. Disponível em: http://dx.doi.org/10.1109/2.825694.

ANDERSON, J. M.; BERC, L. M.; DEAN, J.; GHEMAWAT, S.; HENZINGER, M. R.; LEUNG, S.-T. A.; SITES, R. L.; VANDEVOORDE, M. T.; WALDSPURGER, C. A.; WEIHL, W. E. Continuous profiling: Where have all the cycles gone? **ACM Trans. Comput. Syst.**, ACM, New York, NY, USA, v. 15, n. 4, p. 357–390, nov. 1997. ISSN 0734-2071. Disponível em: http://doi.acm.org/10.1145/265924.265925.

ARNOLD, M.; GROVE, D. Collecting and exploiting high-accuracy call graph profiles in virtual machines. p. 51–62, 2005.

BURROWS, M.; ERLINGSSON, Ú.; LEUNG, S.-T.; VANDEVOORDE, M. T.; WALDSPURGER, C. A.; WALKER, K.; WEIHL, W. E. Efficient and flexible value sampling. ACM, v. 34, n. 5, 2000.

DISKIN, P. Nintendo entertainment system documentation. 2004.

FOLEISS, J. H. Profiling continuo para determinacao de unidades de traducao em traducao dinamica de binarios. 2012.

HORSPOOL, R. N.; MAROVAC, N. An approach to the problem of detranslation of computer programs. **The Computer Journal**, Br Computer Soc, v. 23, n. 3, p. 223–229, 1980.

JONES, D.; TOPHAM, N. High speed cpu simulation using ltu dynamic binary translation. Springer, p. 50–64, 2009.

KLINT, P. Interpretation techniques. **Software: Practice and Experience**, Wiley Online Library, v. 11, n. 9, p. 963–973, 1981.

LINDHOLM FRANK YELLIN, G. B. T.; BUCKLEY, A. The java virtual machine specification, java se 8 edition (1st ed.). Addison-Wesley Professional, 2014.

MUCHNICK, S. S. **Advanced compiler design implementation**. [S.l.]: Morgan Kaufmann, 1997.

OTTONI, G.; HARTIN, T.; WEAVER, C.; BRANDT, J.; KUTTANNA, B.; WANG, H. Harmonia: a transparent, efficient, and harmonious dynamic binary translator targeting the intel® architecture. p. 26, 2011.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. **Commun. ACM**, ACM, New York, NY, USA, v. 17, n. 7, p. 412–421, jul. 1974. ISSN 0001-0782. Disponível em: http://doi.acm.org/10.1145/361011.361073.

SASTRY, S. S.; BODÍK, R.; SMITH, J. E. Rapid profiling via stratified sampling. In: IEEE. **Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on**. [S.l.], 2001. p. 278–289.

SMITH, J.; NAIR, R. **Virtual machines: versatile platforms for systems and processes**. [S.l.]: Elsevier, 2005.

SUGANUMA, T.; YASUE, T.; KAWAHITO, M.; KOMATSU, H.; NAKATANI, T. Design and evaluation of dynamic optimizations for a java just-in-time compiler. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 27, n. 4, p. 732–785, 2005.