UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Juliano Henrique Foleiss

Profiling contínuo para determinação de Unidades de Tradução em Tradução Dinâmica de Binários

Maringá
2012

Juliano Henrique Foleiss

Profiling contínuo para determinação de Unidades de Tradução em Tradução Dinâmica de Binários

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.
Área de concentração: Ciência da Computação

Orientador: Prof. Dr. Anderson Faustino da Silva

Maringá
2012

"Dados Internacionais de Catalogação-na-Publicação (CIP)"
(Biblioteca Setorial - UEM. Nupélia, Maringá, PR, Brasil)

F667p

Foleiss, Juliano Henrique, 1987-
Profiling contínuo para determinação de Unidades de Tradução em Tradução Dinâmica de Binários / Juliano Henrique Foleiss. – Maringá, 2012.
82 f. : il. (algumas color.).

Dissertação (mestrado em Ciência da Computação)--Universidade Estadual de Maringá, Dep. de Informática, 2012.
Orientador: Prof. Dr. Anderson Faustino da Silva.

1. Linguagem de programação (Tradutores). 2. Tradutores de linguagem de programação. 3. Máquinas virtuais - Tradução Dinâmica de Binários. 4. Tradução Dinâmica de Binários - *Profiling* contínuo. I. Universidade Estadual de Maringá. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação.

CDD 22. ed. -005.45
NBR/CIP - 12899 AACR/2

Maria Salete Ribelatto Arita CRB 9/858
João Fábio Hildebrandt CRB 9/1140

JULIANO HENRIQUE FOLEISS

Profiling contínuo para determinação de Unidades de Tradução em Tradução Dinâmica de Binários

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação pela Comissão Julgadora composta pelos membros:

COMISSÃO JULGADORA

Prof. Dr. Anderson Faustino da Silva
Universidade Estadual de Maringá

Prof. Dr. Ronaldo Augusto de Lara Gonçalves
Universidade Estadual de Maringá

Prof. Dr. Marcelo Lobosco
Universidade Federal de Juiz de Fora

Aprovada em: 31 de Outubro de 2012.
Local de defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá.

AGRADECIMENTOS

Primeiramente agradeço a Deus pela força. Em alguns momentos não foi nada fácil lidar com todos os problemas que apareceram na elaboração de cada passo deste trabalho. No entanto, Ele me deu forças pra continuar e coragem para enfrentar os inúmeros desafios apresentados.

Agradeço a todos os meus amigos que, direta ou indiretamente me apoiaram para que este trabalho fosse concluído.

Agradeço à minha família que sempre me apoiou a estudar e a fazer coisas que sejam importantes na vida das pessoas. Em especial agradeço a meus Pais, que me apoiaram e me deram muito mais que condições financeiras para chegar até aqui: me deram dignidade, educação e cidadania. Ao Gabriel, meu irmão camarada que está sempre junto em todos os momentos! Muito obrigado mesmo!

Muito obrigado à minha namorada, que com certeza me deu muitas forças para o desenvolvimento deste trabalho. Com certeza a conclusão deste trabalho não teria a mesma graça se não fosse a sua frase "e a dissertação, já terminou?"

Aos meus amigos de laboratório que fizeram ótima companhia em todas as horas! Até de madrugada! Em especial ao André D'Amato e Danilo Egêa.

E, com destaque especial, gostaria de agradecer ao meu Orientador, o Prof. Anderson Faustino da Silva. Com certeza aprendi muitas coisas importantíssimas com ele. Muito obrigado pelo conhecimento abundante, pelos conselhos, pela força, pelo rigor e pela amizade.

Ao CNPq pelo apoio financeiro concedido a este trabalho.

Profiling contínuo para determinação de Unidades de Tradução em Tradução Dinâmica de Binários

RESUMO

Máquinas virtuais eficientes estão se tornando cada vez mais importante no dia-a-dia da academia e da indústria em geral. Portanto, é importante o desenvolvimento de técnicas para a execução eficiente de programas nestes ambientes. Um problema específico em máquinas virtuais de sistema é a necessidade de executar o conjunto de instruções da arquitetura original na arquitetura hospedeira de maneira eficiente. Uma das técnicas desenvolvidas é a tradução dinâmica de binários (TDB), que permite a execução de programas em formato binário por meio de tradução em tempo de execução. Trabalhos anteriores constataram que traduzir o programa todo não é a melhor escolha, devido ao custo elevado da tradução. Desta forma, é necessário detectar quais regiões devem ser traduzidas, de maneira que não haja traduções excessivas ao mesmo tempo mantendo a execução das instruções a maior parte do tempo por meio de traduções. Trabalhos anteriores apresentam abordagens de monitoramento, ou *profiling*, da execução dos programas para determinar seu fluxo de execução, obtendo assim regiões candidatas a tradução. A principal contribuição deste trabalho é a proposta de um sistema TDB que utilize *profiling* contínuo de maneira que seu custo seja baixo frente ao benefício da execução eficiente oriunda da detecção de unidades de tradução quentes. Em específico, o mecanismo apresentado para o monitoramento da execução, os algoritmos de análise de fluxo de controle eficientes, juntamente com o mecanismo de controle de retraduções formam o conjunto de contribuições deste trabalho. Os resultados obtidos utilizando um emulador do NES (*Nintendo Entertainment System*) mostram que a abordagem sugerida permite a emulação eficiente de programas, com 85,21% das instruções executadas por meio de traduções, sendo até 6,29 vezes mais rápido que interpretação tradicional e 2,34 vezes mais rápido que a abordagem interpretativa.

Palavras-chave: Tradução Dinâmica de Binários. Detecção de código quente. Máquinas virtuais eficientes. *Profiling* contínuo. Instrumentação.

Continuous Profiling for Translation Unit discovery in Dynamic Binary Translation

ABSTRACT

Efficient virtual machines are becoming increasingly important in the academy and industry in general. Thus, it is important to develop new techniques for efficient program execution in such runtime environments. A specific problem in system virtual machines is the need to execute the guest architecture instruction set by the host architecture in an efficient manner. Dynamic Binary Translation (DBT) allows such execution by allowing the execution of programs in binary format by translating the guest machine language program into host machine language program during runtime. Previous work shows that translating the entire program is not the best option, since it incurs in excessive translation overhead. Therefore, it is necessary to detect which regions of code should be translated in such a way that it does not cause excessive translations at the sime time executing most instructions by means of translation. Previous work presents profiling approaches that are used to collect information about a program's control flow, which is then inspected for hot regions deemed for translation. This dissertation proposes a DBT system that uses low-overhead continuous profiling techniques that allows the detection of hot translation units. More specifically, the execution profiling techniques, efficient control flow analysis algorithms along with a retranslation control mechanism are the main contributions of this work. Results obtained by implementing such techniques in a NES (*Nintendo Entertainment System*) emulator shows that the proposed system allows efficient program execution, with an average of 85,21% of instructions executed by means of translation, leading to a 6,29 speedup over traditional interpretation, and 2,34 speedup over an interpretative profiling approach.

Keywords: Dynamic Binary Translation. Hot code detection. Efficient virtual machines. Continuous Profiling. Instrumentation.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1

INTRODUÇÃO

Embora o conceito de máquinas virtuais não seja novo, ultimamente estas vem recebendo atenção especial da comunidade científica e da indústria em geral. Máquinas virtuais são utilizadas para permitir novas possibilidades e para resolver ampla gama de problemas relacionados à estruturação e execução de sistemas em várias áreas da computação, como em sistemas operacionais, linguagens de programação e compiladores e arquitetura de computadores (Smith e Nair, 2005).

Em especial, o uso de máquinas virtuais no contexto de arquitetura de computadores permite a execução de conjuntos de instrução distintos do conjunto de instrução do *hardware* real. Isto permite, por exemplo, que sistemas operacionais e suas aplicações sejam executadas em *hardware* diferente para o qual foram desenvolvidos inicialmente. Esta possibilidade possui aplicações importantes tanto para a academia quanto para a indústria, pois permite a execução de sistemas legados e de sistemas proprietários cujo *hardware* real pode ser de difícil aquisição ou economicamente inviável (Smith e Nair, 2005). Outro uso apontado em trabalhos recentes (Hong et al., 2012; Ottoni et al., 2011) no contexto de arquitetura de computadores é a possibilidade de utilizar máquinas virtuais para a execução de programas para sistemas embarcados em outros sistemas embarcados. Em geral, o uso de máquinas virtuais em ambientes com recursos relativamente escassos, como no caso de *smartphones*, por exemplo, é impedido pela complexidade deste processo.

Uma questão essencial neste cenário é a necessidade de que a execução da máquina virtual seja eficiente, ou seja, que sua execução aconteça de maneira transparente e que forneça os mesmos resultados que seriam obtidos caso o programa estivesse sendo executado na máquina original, dentro do tempo esperado pelo usuário. Para este fim,

foram desenvolvidos ao longo dos anos diversas técnicas de tradução e otimização dinâmica para redução do consumo de energia e para aumentar o desempenho da execução (Smith e Nair, 2005).

As técnicas utilizadas para executar programas que são apresentados em formato binário, ou seja, sem código fonte disponível, são chamadas coletivamente de Tradução de Binários (TB). Quando estas técnicas são empregadas em tempo de execução, são denominadas de Tradução Dinâmica de Binários (TDB). Uma das vantagens do emprego das máquinas virtuais é que estas permitem o monitoramento da execução do programa, denominado *profiling*. Os dados coletados durante o monitoramento podem ser utilizados nas fases de otimização dinâmica com o objetivo de aprimorar a qualidade do código gerado para tradução. Em específico, estas informações permitem a descoberta de fluxos de execução quentes de código, indicando quais regiões são boas candidatas à tradução.

Este monitoramento pode ser feito basicamente de duas formas, sendo elas: parcialmente durante a execução ou continuamente. Em abordagens parciais (Böhm et al., 2010; Jones, 2010) somente parte da execução é monitorada. Embora esta abordagem tenha a vantagem de reduzir o custo do monitoramento, ela normalmente faz com que alguns fluxos deixem de ser reconhecidos, deixando de lado algumas oportunidades de tradução e otimização. Por outro lado, existe a abordagem contínua, que permite a detecção de mudanças no padrão de execução dos programas para determinar unidades de tradução cada vez melhores. Embora não seja uma idéia nova (Banerjia et al., 1997; Ebcioglu et al., 2001), esta estratégia ocasiona perda de desempenho, principalmente pelo custo da instrumentação necessária e pela quantidade excessiva de traduções (Smith e Nair, 2005).

Neste contexto, o objetivo principal deste trabalho é propor, implementar e testar um sistema TDB que utilize *profiling* contínuo de maneira eficiente: balanceando o custo da instrumentação, a quantidade de traduções necessárias, o custo da análise de fluxo de dados e o desempenho da execução do programa original. Para cumprir este objetivo, os seguintes objetivos específicos foram traçados:

- Oferecer mecanismos eficientes de *profiling* contínuo, visando equilíbrio entre seu custo e a acurácia das informações coletadas;

- Oferecer algoritmos de análise de fluxo de dados eficientes que utilizem as informações obtidas durante o *profiling*, de forma a permitir a detecção satisfatória de unidades de tradução quentes, minimizando assim a quantidade de traduções e retraduções necessárias. A partir da detecção de tais unidades é possível manter as instruções do programa sendo executadas mais tempo por meio de tradução;

- Implementar um emulador como prova de conceito que contenha ambas abordagens de *profiling*: contínua e parcial;
- Avaliar os resultados obtidos e compará-los aos objetivos esperados.

Com base nestes objetivos, a principal contribuição deste trabalho é a proposta de um sistema TDB que utilize *profiling* contínuo de maneira que seu custo seja baixo frente ao benefício da execução eficiente oriunda da detecção de unidades de tradução quentes. Em específico, o mecanismo apresentado para o monitoramento da execução, os algoritmos de análise de fluxo de controle eficientes, juntamente com o mecanismo de controle de retraduções formam o conjunto de contribuições deste trabalho.

Desta forma, o presente trabalho justifica-se pela necessidade de técnicas cada vez mais aprimoradas para a execução eficiente de máquinas virtuais eficientes, as quais estão se tornando cada dia mais importantes e necessárias no dia-a-dia da academia e da indústria.

As técnicas apresentadas neste trabalho foram avaliadas em um emulador real e seus resultados podem ser utilizados como base na implementação de outros sistemas que utilizem técnicas relacionadas. A abordagem sugerida neste trabalho atinge a execução de, em média, 85% das instruções por meio de tradução, sendo até 6 vezes mais rápido que interpretação tradicional e 2 vezes mais rápido que a abordagem interpretativa proposta por Jones (Jones, 2010). Além destes números, os resultados apresentam outras contribuições que devem ser levadas em consideração durante o projeto de um sistema TDB, que são discutidas no Tópico 5.

Esta dissertação está organizada como segue: O Tópico 2 contém a teoria relacionada à tradução dinâmica de binários, estratégias de *profiling* e unidades de tradução. O Tópico 3 apresenta trabalhos relacionados atuais, cujos objetivos são a execução eficiente do código emulado. O Tópico 4 apresenta a proposta de um sistema de TDB que utiliza *profiling* contínuo de maneira eficiente, a principal contribuição deste trabalho. O Tópico 5 apresenta os resultados obtidos, juntamente com uma comparação direta entre a abordagem contínua e parcial, além de uma comparação de desempenho entre a implementação da proposta e uma versão puramente interpretada. Por fim, o Tópico 6 apresenta as conclusões e trabalhos futuros.

2

TRADUÇÃO DINÂMICA DE BINÁRIOS

Tradução Dinâmica de Binários (TDB) é um conjunto de técnicas que permitem a execução de código de máquina de uma arquitetura alvo em uma arquitetura hospedeira. Em específico, TDB trabalha traduzindo sob demanda as instruções da máquina alvo em uma sequência equivalente de instruções da máquina hospedeira durante a execução do programa (Altman et al., 2000).

A principal vantagem em utilizar tradução dinâmica de binários é o desempenho proporcionado em relação a outras técnicas de execução, como por exemplo interpretação e tradução estática de binários (Altman et al., 2000). Esse aumento no desempenho está relacionado ao fato que, ao executar o código traduzido, o sistema passa a realizar menos chamadas ao sistema que gerencia a execução (Jones, 2010). Outro fator que leva ao aumento do desempenho está relacionado ao fato que, como a tradução acontece durante a execução do programa, é possível coletar informações em tempo de execução que permitam otimizar o código de maneira antes impossível ao compilador estático originalmente utilizado para gerar o código sendo executado (Bala et al., 2011).

Além do desempenho, a tradução dinâmica de binários herda as principais vantagens das técnicas de tradução dinâmica, como a capacidade de executar código desenvolvido para outras arquiteturas. Isso permite a execução de sistemas legados, cujos equipamentos não são mais produzidos, ou cujo código fonte não esteja mais disponível. Além de sistemas legados, atualmente TDB vem sendo utilizada para execução de programas modernos desenvolvidos para outras arquiteturas, visando redução de custos de desenvolvimento e

compatibilidade com *software* desenvolvido para arquiteturas incompatíveis de empresas concorrentes (Altman et al., 2000; Ottoni et al., 2011).

De fato, a necessidade do desenvolvimento de TDB não é algo novo. Trabalhos como Shade (Cmelik e Keppel, 1994), de meados dos anos 90, já visavam o uso de TDB para acelerar a simulação de conjuntos de instrução. Empresas como a IBM e DEC também investiram em sistemas de tradução dinâmica de binários para executar programas desenvolvidos para x86 em arquiteturas como PowerPC (Ebcioglu et al., 2001) e Alpha (Chernoff et al., 1998).

Atualmente, o foco em pesquisas relacionadas à TDB está voltado ao desempenho. Embora inúmeras técnicas tenham sido propostas para melhorar o desempenho de TDB, a complexidade das arquiteturas-alvo modernas requer que a tecnologia de simulação seja cada vez mais aprimorada. Como um exemplo prático, um programa moderno de compressão e descompressão de imagens no formato JPEG requer entre $10x10^9$ e $16x10^9$ instruções para ser executado (Böhm et al., 2010), enquanto um simulador moderno com TDB é capaz de executar até $5x10^8$ instruções por segundo (Jones, 2010).

Além do desempenho, empresas como a Intel e AMD, que são voltadas tradicionalmente ao mercado *desktop*, pretendem entrar mais agressivamente no mercado de processadores para dispositivos móveis (Ottoni et al., 2011). Para que ingressem no mercado de forma competitiva, é necessário que seja possível executar aplicações existentes desenvolvidas para processadores embarcados tradicionais, como ARM e compatíveis. Embora os processadores para dispositivos móveis sejam eficientes, os custos para utilizar técnicas de TDB ainda são proibitivos. Portanto, é necessário pesquisar técnicas que possam ser eficientemente utilizadas em dispositivos com recursos relativamente limitados.

A principal desvantagem em utilizar técnicas de tradução dinâmica de binários está ligada ao fato que a tradução ocorre em tempo de execução, utilizando processamento útil que poderia estar sendo utilizado na execução do programa (Altman et al., 2000; Bala et al., 2011; Böhm et al., 2011a). Portanto, é necessário que as técnicas utilizadas em TDB sejam sensíveis ao *trade-off* entre o tempo gasto em tradução e otimização em relação ao ganho de desempenho proporcionado.

A determinação de unidades de tradução é um dos principais pontos que determinam o sucesso das técnicas de TDB (Jones, 2010). É prática comum entre as técnicas de TDB que a seleção das unidades de tradução seja feita de forma a escolher regiões a serem traduzidas de forma que o custo da tradução seja amortizado durante a execução devido ao ganho de desempenho que proporciona (Ottoni et al., 2011).

2.1 UNIDADES DE TRADUÇÃO

Tradicionalmente, uma Unidade de Tradução (UT) representa o código a ser traduzido e deve conter todas as informações necessárias para que a tradução possa ser realizada. Em TDB, uma UT representa um trecho de código da arquitetura-alvo, juntamente com informações que a identifiquem no sistema. Usualmente, uma UT contém o endereço inicial do trecho, o tamanho do trecho, metadados contendo informações sobre as instruções e outros detalhes que podem ser utilizados durante a tradução (Ottoni et al., 2011).

É prática comum em TDB que nem todas as unidades de tradução reconhecidas sejam traduzidas. Normalmente, a quantidade de unidades de tradução de um programa é alta. Portanto, traduzir todas as unidades de tradução implica em usar muito do tempo disponível para a aplicação em tradução. Além disto, muitas das unidades de tradução reconhecidas são raramente utilizadas, e, portanto, seu custo de tradução não é amortizado durante sua execução. Desta forma, é necessário decidir se determinada UT deve ser traduzida e se sua tradução contribui para a melhora do desempenho ao longo da execução. A Definição 1 apresenta o conceito de Função Traduzida.

Definição 1. Função Traduzida – Resultado da tradução de uma UT (Jones, 2010).

Um dos desafios na determinação de UTs é o fato do código-fonte do programa sendo executado não estar disponível. Portanto, toda informação estática sobre o código sendo traduzido deve ser adquirida durante a execução. Por conta disso, os detalhes do conjunto de instrução devem ser levados em consideração durante a geração de unidades de tradução.

Outro desafio está relacionado ao problema de separar dados de código em um programa em formato binário. Horspool e Marovac(Horspool e Marovac, 1980) argumentam que é necessário conhecer o endereço inicial de um trecho executável para que seja possível a identificação de sequências de instruções. Além disto, argumentam que é necessário conhecer informações específicas sobre a arquitetura-alvo em questão, como a codificação das instruções e modos de endereçamento.

Entretanto, a determinação de unidades de tradução vai além da análise das instruções do código de máquina sendo executado. O maior desafio está relacionado a determinar quais regiões do código alvo devem ser executadas com o objetivo de escolher aquelas que venham compensar o tempo investido em sua tradução.

Tradicionalmente, as unidades de tradução são determinadas por instruções individuais ou blocos básicos (Hecht, 1977). Trabalhos recentes (Bellard, 2005; Böhm et al., 2011a; Jones, 2010) apontam que tais unidades são inviáveis, principalmente por serem muito

pequenas, ocasionando muitas traduções e excessivas chamadas ao sistema que gerencia a execução.

2.1.1 Determinação de Unidades de Tradução

Em TDB as unidades de tradução são determinadas em tempo de execução, e podem ser determinadas por meio de uma estratégia estática ou baseada em *profiling*.

Determinação Estática

Nas técnicas de determinação de unidades de tradução estáticas, nenhuma informação sobre o fluxo de controle da execução do programa é utilizado. A primeira vez que o sistema encontra uma instrução que ainda não foi selecionada para tradução, esta e as instruções que seguem são adicionadas a uma UT. Determinar quantas ou quais instruções são inseridas na UT depende da política adotada pelo sistema em questão. Sistemas como QEMU (Bellard, 2005) e Harmonia (Ottoni et al., 2011) utilizam blocos básicos como UT.

Definição 2. Bloco Básico – Uma sequência maximal de instruções tal que nenhuma instrução exceto a primeira é alvo de um salto, e nenhuma exceto a última é um salto (Muchnick, 1997)

Nem todos os sistemas TDB utilizam a definição de bloco básico de acordo com a Definição 2. Na prática, blocos básicos caracterizam apenas uma unidade elementar utilizada para descrever unidades de tradução. De fato, unidades de tradução são tipicamente agrupamentos de blocos básicos. Este agrupamento é realizado com base em políticas determinadas pelo projeto do sistema TDB.

Nem toda UT gerada é traduzida. Normalmente, quando o sistema adota determinação estática, UTs somente são traduzidas após sua quantidade de execuções em modo interpretado ultrapassar um limiar pré-determinado (Ung e Cifuentes, 2000). Tais blocos são denominados *hotspots*, ou seja, blocos "quentes", potencialmente utilizados com frequência. Tal política tem como objetivo traduzir somente os blocos cuja execução é recorrente. Desta forma, blocos que raramente são executados não são traduzidos, economizando tempo de execução. Independentemente da UT ser escolhida ou não para tradução, um registro dela normalmente é mantido, com o objetivo de impedir que seja determinada toda vez que o contador de programa aponte para sua primeira instrução.

Determinação Baseada em Profiling

Nas técnicas de determinação de unidades de tradução baseadas em *profiling*, o sistema de TDB utiliza informações sobre o fluxo de controle da execução para determinar como as unidades de tradução são reconhecidas. A Definição 3 apresenta o conceito de *profiling* utilizado neste trabalho.

Definição 3. ***Profiling*** – Processo de registro do fluxo de controle de um programa.

As informações de um *profile* são recolhidas durante a execução do programa. Desta forma, os registros refletem o comportamento da execução do programa. O intervalo e o que é registrado em um *profile* é determinado pela técnica sendo utilizada. Os registros podem ser realizados em relação à instrução ou ao agrupamento de instruções, como, por exemplo, blocos básicos. Informações geralmente encontradas em um registro são:

- Endereço inicial do registro (valor do contador de programa);
- Endereço dos destinos de saltos que partem do registro; e
- Quantidade de vezes que o registro foi executado.

Ao utilizar *profiles* recolhidos durante a execução atual é possível detectar mais claramente quais unidades de tradução propostas são mais adequadas para serem traduzidas. Além de evidenciar os blocos básicos cuja execução é mais recorrente, é possível determinar quais os caminhos que são tomados entre os blocos básicos, de maneira a aumentar o escopo das otimizações e englobar mais instruções em uma única unidade de tradução (Bala et al., 2011; Ebcioglu et al., 2001; Jones, 2010; Ottoni et al., 2011).

De maneira geral, o processo de *profiling* é realizado durante a execução do código-alvo por meio de interpretação. Isso se deve ao fato de que, embora o interpretador também seja projetado para obter máximo desempenho, o interpretador é projetado para realizar *profiling* durante a execução da aplicação com o objetivo de retratar o fluxo de execução com fidelidade, sacrificando um pouco de seu desempenho.

No entanto, em geral, ao executar funções traduzidas, o processo de *profiling* é interrompido. Isto ocorre para que o código execute o mais rápido possível, sem a necessidade de coletar nem atualizar estruturas de dados alheias à execução do código correspondente ao programa traduzido.

Entretanto, sacrificar *profiling* para obter desempenho implica na desaceleração do processo de identificação das regiões mais frequentemente utilizadas durante a execução de um programa. Com isto, o sistema perde a capacidade de se adequar à mudanças no

fluxo de controle em programas complexos. Portanto, programas como compiladores e jogos são prejudicados pela abordagem de *profiling* interpretativo. O processo de *profiling* pode ser categorizado de acordo com o escopo do código sendo monitorado. Desta forma, abordagens de *profiling* podem ser categorizadas em *Profiling interpretativo* e *Profiling Contínuo*.

Definição 4. *Profiling* interpretativo (PrI) – *Profiling* realizado somente durante a execução do interpretador (Jones, 2010).

Na abordagem de *profiling* contínuo (Definição 5), o processo de *profiling* ocorre durante toda a execução do programa. Desta forma, é possível detectar mudanças no fluxo de controle durante a execução.

Definição 5. *Profiling* contínuo (PrC) – *Profiling* realizado durante toda a execução do programa, tanto na execução de unidades traduzidas quando interpretadas (Jones, 2010).

PrC também pode ser utilizado para amenizar o problema de fragmentação de unidades de tradução em sistemas que utilizam unidades de tradução longas. Os conceitos de unidades de tradução longas e fragmentação de unidades de tradução são discutidos nas Seções 2.2 e 2.2.2 e são os principais pontos de interesse desta pesquisa.

2.2 UNIDADES DE TRADUÇÃO LONGAS

Durante o desenvolvimento do simulador EHS (Jones, 2010) foi desenvolvido o conceito de unidade de tradução longa, que é formalmente definido como descrito na Definição 6.

Definição 6. Unidade de Tradução Longa (UTL) – Uma UT composta por um grupo de blocos básicos do programa alvo que são conectados por arestas em um grafo de fluxo de controle que podem ter um ou mais pontos de entrada ou saída.

A tese defendida por Jones admite que, quanto maior a unidade de tradução, maior a velocidade da simulação. Além disso, Jones alega que os principais motivos apontados para o aumento no desempenho são:

1. UTL fornece ao tradutor dinâmico escopo maior para aplicação de otimizações; e

2. Maiores seções de código alvo são executadas. Isto resulta em menos retornos ao ambiente de execução, sendo maior proporção da execução gasto em execução de instruções por meio de tradução.

A proposta de Jones visa aumentar o desempenho de simuladores baseados em TDB de duas maneiras. Primeiramente, usando UTL para obter desempenho como descrito acima. Além disso, propõe a determinação de unidades de tradução por meio da descoberta de caminhos frequentemente utilizados entre blocos, ao invés da detecção de blocos ou páginas frequentemente executadas isoladamente, que podem conter blocos básicos raramente executados ou que sejam, em sua maioria, dados.

2.2.1 Profiling e Classificação de UTL

A tese de Jones apresenta uma estratégia para reconhecer UTLs utilizando *profiling* interpretativo. Em sua abordagem, o tempo de simulação é dividido em *épocas*, que são determinadas pelo intervalo correspondente a uma quantidade pré-estabelecida de execuções de blocos básicos.

Definição 7. Época – Unidade de tempo de simulação que consiste no intervalo correspondente a uma quantidade pré-estabelecida de execuções de blocos básicos.

Em cada época e para cada página de memória física, é mantido um grafo de fluxo de controle (Muchnick, 1997) que descreve os caminhos percorridos até a época atual. Durante uma mesma época, novas UTs são interpretadas, UTs previamente conhecidas e não traduzidas são interpretadas, UTs podem ser descartadas devido à código auto-modificável e funções de tradução são executadas (Jones, 2010).

Durante cada época de simulação, os fluxos de execução são coletados e armazenados nos grafos de fluxo de controle de cada página física. Dependendo da natureza das unidades de tradução, existem procedimentos para a construção dos mesmos. Em UTs baseadas em blocos básicos, é realizada a contagem de quantas vezes cada bloco básico foi interpretado. Para UTLs, o procedimento é um pouco mais complexo. Primeiramente, o bloco básico deve ser inserido no grafo de fluxo de controle da página, caso ainda não tenha sido. Além disto, a contagem da execução de cada bloco e as arestas percorridas são atualizadas no *profile* da época atual conforme forem sendo executados.

Classificação de UTL

Foram propostos quatro tipos de UTL (Jones e Topham, 2009): blocos básicos, blocos fortemente conexos, grafos de fluxo de controle e páginas. A seguir uma breve descrição dos quatro tipos propostos.

- **Blocos Básicos (BB)** – Blocos frequentemente identificados são traduzidos. As funções traduzidas resultantes são executadas quando o contador de programa (PC) atinge o endereço inicial do bloco;

- **Blocos Fortemente Conexos (BFC)** – Uma análise de componentes fortemente conexos do grafo de fluxo de controle obtido durante o *profiling* é realizada, e cada componente resultante é considerado uma UTL. Quando PC atinge o endereço inicial da raíz de um componente, a função traduzida correspondente é executada;

- **Grafos de Fluxo de Controle (GFC)** – Uma análise dos possíveis caminhos a partir das raízes do grafo de fluxo de controle obtido durante o *profiling* é realizada e cada caminho percorrido é considerado uma UTL. Quando PC atinge o endereço inicial da raíz de um GFC, a função traduzida correspondente é executada; e

- **Páginas** – O grafo de fluxo de controle é gerado dentro das páginas físicas atribuídas ao espaço de endereçamento do programa. Neste tipo de UTL, o grafo inteiro é utilizado como UTL. Quando PC atinge o endereço de qualquer bloco em uma função traduzida a partir de uma UTL de página, a função traduzida do bloco correspondente é executada.

Ao final de cada época, os grafos de fluxo de controle, juntamente com os *profiles* gerados são analisados para recuperar unidades de tradução. Esta análise é feita por meio de algoritmos conhecidos na literatura de computação em geral.

- Para identificar **BFCs**, é utilizado o algoritmo de Tarjan (Tarjan, 1971);

- Para encontrar caminhos em **GFC**, basta utilizar o algoritmo de travessia de grafos descrito em (Muchnick, 1997) a partir do nó raíz; e

- Nenhuma análise é necessária para a abordagem de **Páginas**. O grafo correspondente ao *profile* todo é considerado como UTL.

No final de cada época, todas as UTLs são avaliadas para determinar se devem ou não ser traduzidas. A Tabela 2.1 indica as métricas utilizadas em cada abordagem.

As UTLs cuja quantidade de execuções atinge o limiar de tradução na época atual são marcadas para tradução durante a fase de tradução. Embora esta técnica de determinação de UTLs funcione bem, existe um problema que pode prejudicar o desempenho de maneira significativa em algumas aplicações. A próxima Seção detalha este problema, conhecido como Fragmentação de UTLs.

Tabela 2.1: Métricas para avaliar a decisão de tradução de UTLs – Adaptado de (Jones, 2010)

Abordagem	Métrica
BB	Número de execuções
BFC	Número de execuções do nó raiz
GFC	Número de execuções do nó raiz
Página	Sempre traduzir

2.2.2 Fragmentação de UTLs

A fragmentação de UTLs é a geração de múltiplas unidades de tradução pequenas e ocorre quando o processo de *profiling* é interrompido durante determinada época (Jones, 2010). Esta interrupção pode ocorrer por diversos fatores, mas é mais comumente atribuída ao tamanho da página física, a época da simulação e a execução de funções traduzidas.

O simulador EHS desenvolvido por Jones (Jones, 2010; Jones e Topham, 2009) utiliza *profiling* interpretativo. Portanto, enquanto as funções de tradução são executadas, o processo de *profiling* é interrompido, e, por consequência, grafos pequenos de fluxo de controle são gerados. A partir de grafos pequenos, UTs pequenas são reconhecidas e traduzidas. Com isto, o tempo de execução é consideravelmente prejudicado devido ao fato que unidades de tradução pequenas degradam o desempenho (Jones, 2010). Esta degradação acontece devido ao fato que o escopo de otimizações possíveis para o compilador dinâmico diminui, além de blocos menores causarem mais chamadas ao sistema que gerencia a execução como um todo. Quanto mais chamadas ao sistema, mais tempo é gasto em gerência e, portanto, o programa demanda mais tempo para ser executado.

A Figura 2.1 - mostra um exemplo de fragmentação de unidades de tradução ocorrendo durante TDB utilizando a abordagem GFC. A Figura 2.1 - (a) mostra o grafo de fluxo de controle montado durante a execução até a época N. Os nós representam blocos básicos e os arcos representam possíveis fluxos de execução entre eles. Os arcos vermelhos na Figura 2.1 - (b) representam os caminhos interpretados durante a época N, que são identificados como uma UTL e é então traduzida. Após algumas épocas, outros fluxos de execução são tomados, como os representados com arcos verdes na Figura 2.1 - (c). A partir daí, os nós sombreados representam os candidados a UTL. Nota-se que mesmo que os dois da direita se tornem uma UTL e o da esquerda outra UTL, as regiões se tornam muito pequenas, levando à degradação do desempenho.

Outro ponto negativo relacionado à fragmentação de UTLs está no fato que a execução do programa torna-se inflexível em função de mudanças de fase. Uma mudança de fase

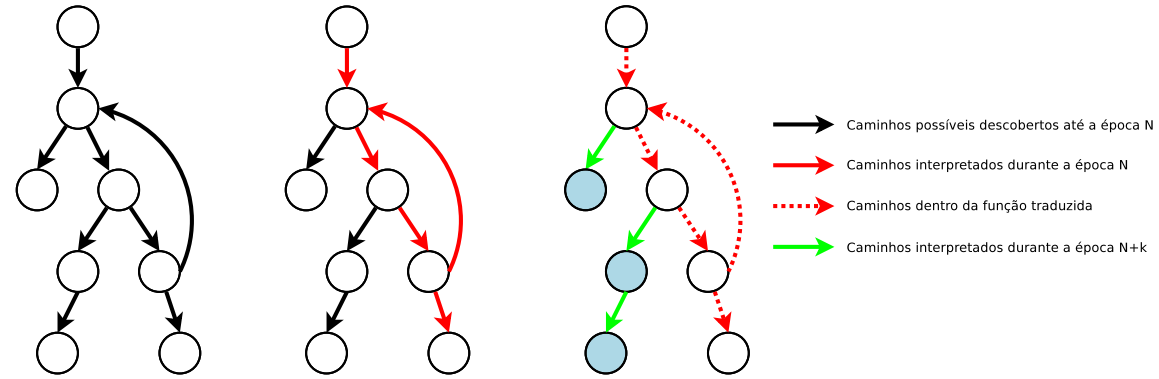


Figura 2.1: Fragmentação de UTLs – Adaptado de (Jones, 2010)

ocorre quando o fluxo de controle do programa é alterado abruptamente, e normalmente ocorre em programas que são constituídos de várias fases com código compartilhado mas com caminhos de execução distintos. A mudança de fase também é um comportamento que influencia as estruturas de *cache*, tanto de dados quanto código. Portanto, o problema de fragmentação de UTL é prejudicial ao desempenho da simulação.

Este trabalho tem como objetivo atacar o problema de fragmentação de UTLs utilizando *profiling* contínuo e fusão de UTLs temporalmente próximas dirigida por fluxo de controle. O Tópico 4 apresenta a proposta desenvolvida.

3

TRABALHOS RELACIONADOS

Neste capítulo são apresentados trabalhos que implementam técnicas de TDB com o objetivo de obter máximo desempenho na execução. Embora os trabalhos apresentados sejam de várias épocas diferentes, foram escolhidos os trabalhos de acordo com o nível de proximidade à proposta deste trabalho.

3.1 EHS

O *Edinburgh High-Speed Simulator* (EHS) é um simulador desenvolvido por Jones como parte de sua tese de doutorado na Universidade de Edimburgo. A tese defendida pelo autor é que o uso de unidades de tradução longas beneficia a execução em termos de desempenho. De fato, o ganho de desempenho obtido utilizando UTLs em relação à blocos básicos como UT é de aproximadamente 63% (Jones, 2010) no modo rápido. Além disso, Jones afirma que a execução em modo TDB é pelo menos $14,8$ vezes mais rápida que o interpretador utilizado como base. Em simulações que visam precisão, a utilização de unidades de tradução longas executa 32% mais rápido, sendo o modo TDB $8,37$ vezes mais rápido que o interpretador.

A principal contribuição do EHS é o conceito de Unidades de Tradução Longas, apresentado na Seção 2.2. A Figura 3.1 - mostra o fluxo de execução do simulador. O tempo de execução é dividido em épocas, onde em cada época um *profile* do fluxo de execução do programa é criado. No final de cada época, o fluxo de execução registrado em forma de grafo de fluxo de controle é analisado e então as UTLs são extraídas. Em seguida, as UTLs são analisadas e, aquelas que forem consideradas vantajosas são traduzidas.

O simulador utiliza um nível de cache de tradução para manter as funções traduzidas mais recentemente utilizadas, além de um mapa de traduções que mantém a relação de cada ponto de entrada das unidades de tradução e o respectivo endereço de cada função traduzida.

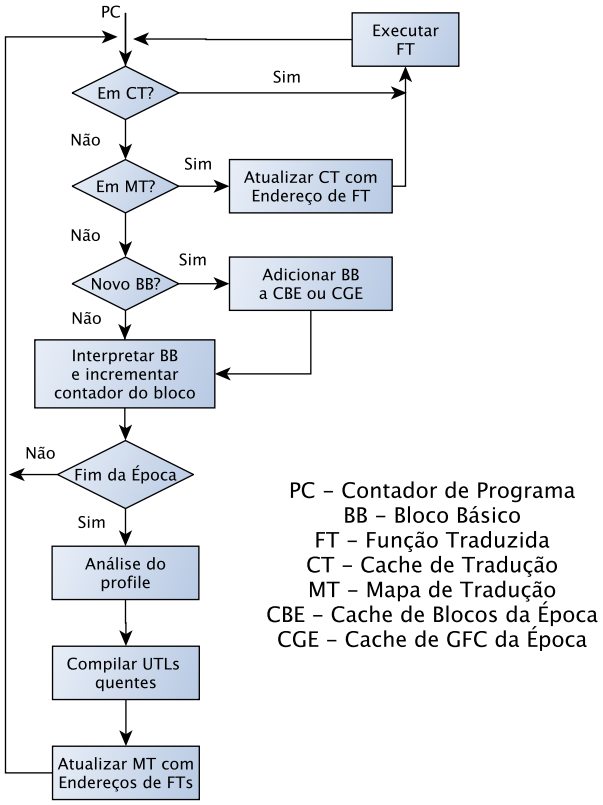


Figura 3.1: Fluxo de Execução do EHS - Adaptado de (Jones, 2010)

O processo de tradução ocorre primeiramente com a conversão do código de cada instrução da UTL em código C por meio de macros. Em seguida, este código é compilado usando o `gcc` em uma biblioteca compartilhada, que é então ligada utilizando o *linker* dinâmico. Por fim, a função traduzida é então instalada em um mapa de traduções que fica à disposição do simulador para execução.

Conforme descrito na Seção 2.2.2, a principal deficiência do EHS é a fragmentação das unidades de tradução. O principal fator que corrobora para este fenômeno no EHS é o fato do processo de *profiling* acontecer apenas durante a interpretação de blocos básicos. Uma das alternativas, também brevemente descrita por Jones (Jones, 2010) é a realização de *profiling* contínuo, ou seja, que ocorra durate toda a execução do programa, não somente durante a interpretação. O Capítulo 4 apresenta a abordagem de *profiling*

contínuo desenvolvida neste trabalho, que utiliza mecanismos originais para diminuir o custo necessário para o *profiling* contínuo, ao mesmo tempo proporcionando a redução significativa na fragmentação de unidades de tradução.

3.2 QEMU

QEMU (Bellard, 2005) é um emulador de sistema que foi desenvolvido para ser rápido e portável. Atualmente, é capaz de executar código de várias arquiteturas-alvo, como PowerPC, x86, ARM e Sparc. Sua portabilidade é seu grande diferencial, que permite a emulação em diversas arquiteturas hospedeiras, tais como PowerPC, x86, ARM, Alpha, Sparc e MIPS.

O ciclo de execução adotado pelo QEMU segue o padrão de sistemas TDB. A cada iteração, o endereço atual do PC é verificado. Caso ainda não haja tradução disponível do bloco apontado pelo endereço é realizada a tradução, um bloco de cada vez. No QEMU, nenhuma análise sobre fluxo de execução é utilizada para obter unidades de tradução maiores. Sua política de determinação de unidades de tradução é estática e é realizada a partir do endereço inicial que causou a tradução até o primeiro salto incondicional ou até uma instrução que modifique o *estado estático da CPU*. O autor define *estado estático da CPU* como instruções que modificam o estado da CPU de maneira que o resultado seja imprevisível em tempo de tradução. As unidades de tradução resultantes são denominadas Blocos Traduzidos (BT). No QEMU não acontece interpretação. Dessa forma, o tradutor dinâmico fica responsável pela execução de toda aplicação, o que implica que seu código deve ser altamente eficiente para que seu custo de tradução seja amortizado.

Desta forma, instruções de salto condicional não terminam a determinação dos BT. Em específico, instruções de salto cujo destino pode ser definido em tempo de execução podem acompanhar um salto direto a outros blocos já traduzidos, evitando o retorno desnecessário ao *loop* principal do QEMU. Tal técnica é denominada *block-chaining*. Embora esta técnica seja utilizada no QEMU, nenhuma verificação posterior é realizada para conectar blocos previamente traduzidos a novos blocos.

O processo de tradução do QEMU reflete seu objetivo referente à portabilidade. Ao invés de traduzir as instruções da arquitetura-alvo diretamente para código da máquina hospedeira, cada instrução é dividida em micro operações. Os padrões que determinam quais micro operações são necessárias para uma determinada instrução são construídos à mão. As micro operações são implementadas em C, seguindo convenções para acessar o contexto emulado da CPU e seus demais componentes.

O conjunto de micro operações é utilizado como entrada para uma ferramenta denominada *dyngen*, que gera um gerador de código dinâmico. Este processo é feito em tempo de compilação do QEMU, não durante sua execução. Durante a execução, este gerador dinâmico é invocado com o BT a ser traduzido. Em seguida, o gerador concatena as micro operações de todas as instruções do BT em sequência.

No entanto, o gerador não produz código executável diretamente. Primeiramente são realizadas duas adequações do código gerado, a saber: alocação de registradores e otimização de códigos de condição. O alocador de registradores utiliza a abordagem de alocação fixa, onde cada registrador da máquina alvo fica armazenado em um registrador da máquina hospedeira, quando possível. Quando a máquina alvo possui mais registradores que a hospedeira, endereços de memória fixos são utilizados para representá-los. Esta abordagem, apesar de sua simplicidade, pode prejudicar o tempo de execução, pois muitos acessos à memória podem ser descartados se uma política de alocação de registradores mais eficiente for utilizada (Probst et al., 2002).

A otimização de código de condição utilizada avalia os códigos de condição de maneira tardia. Nesta abordagem, os códigos não são recomputados imediatamente após cada instrução executada. Os valores dos operandos e a operação são guardados em variáveis auxiliares, e portanto, podem ser computados facilmente em instruções que realmente necessitem dos códigos de condição. Além disso, a otimização utiliza-se do fato que o código do BT inteiro é conhecido durante o processo de otimização. Isto permite a verificação se as instruções subsequentes utilizam ou não as informações dos códigos de condição. Caso não utilizem, eles só precisam ser atualizados no final do bloco. De fato, a otimização de códigos de condição é algo que reflete diretamente na eficiência em um processo de emulação (Hu et al., 2009; Ottoni et al., 2011).

A principal contribuição do QEMU é o fato de ser uma plataforma de emulação projetada para ser portável. Em consequência disto, suporte a diversas arquiteturas alvo e hospedeiras foi implementado. Além disso, o QEMU serve como base para trabalhos de TDB mais avançados, como Harmonia (Ottoni et al., 2011) e HQEMU (Hong et al., 2012).

3.3 HQEMU

O HQEMU (Hong et al., 2012) é um aprimoramento do QEMU que utiliza tecnologias como LLVM (Lattner e Adve, 2004), processamento paralelo e HPM (*Hardware Performance Counters*) (Schneider et al., 2007) para obter maior desempenho. Os resultados do protótipo construído, que traduz x86 em x86_64, mostram ganho de desempenho no

benchmark SPEC2006 (Li et al., 2009) para números inteiros e em ponto flutuante de 2, 4 a 4 vezes, respectivamente, em relação ao QEMU. Além disso, os autores afirmam que a execução nos mesmos *benchmarks* é apenas 2, 5 e 2, 1 vezes mais lentas que as versões nativas. No tocante à emulação ARM em um x86_64, a execução foi 2, 4 vezes mais rápida que o QEMU no *benchmark* SPEC2006 para números inteiros. Nota-se que não há *benchmark* SPEC2006 para operações em ponto flutuante para ARM.

Segundo o autor, o objetivo do trabalho foi projetar um sistema TDB capaz de emitir código de alta qualidade sem sobrecarregar a execução. Para isto, o fluxo de execução do HQEMU foi dividido em duas *threads*. A primeira abriga uma versão modificada do QEMU, que continua responsável por reconhecer os BTs e gerar uma tradução rápida de forma que a execução não fique ociosa. Assim como no QEMU, é realizada a tradução de somente um bloco básico por vez. As modificações realizadas inserem instrumentação no início de cada bloco com o objetivo de realizar *profiling* da execução. Portanto, HQEMU utiliza *profiling* contínuo na determinação de suas unidades de tradução longas.

Embora a primeira tradução seja feita de forma independente para cada bloco, as informações utilizadas pelo QEMU para gerar o código são armazenadas. Estas informações, juntamente com o *profile* obtido com a instrumentação, são utilizadas para determinar unidades de tradução maiores que sejam frequentemente utilizadas.

O HQEMU utiliza um algoritmo simplificado baseado no algoritmo NET (Duesterwald e Bala, 2000) para obter unidades de otimização. A instrumentação adicionada ao início de cada bloco é dividida em dois pequenos trechos de código. O primeiro deles é denominado *profiler*, que é responsável por detectar quais os blocos cuja execução é recorrente. O *profiler* contém um contador que é incrementado toda vez que o bloco é executado. Assim que este contador atinge determinado limiar, o segundo trecho da instrumentação é habilitado

O segundo trecho na instrumentação é denominado preditor. O preditor é responsável por procurar um ciclo no fluxo de execução, tornando-o uma unidade de tradução. A partir do momento que o preditor está ativo, todos os blocos percorridos são colocados em uma lista em ordem de travessia. O preditor termina seu trabalho quando encontra um bloco que já foi visitado anteriormente. Neste momento, um ciclo é identificado. A preferência por ciclos está no fato de que ciclos normalmente indicam a presença de uma estrutura de repetição. De maneira geral, estruturas de repetição caracterizam boa parte da execução e, portanto, é vantajoso otimizar tais regiões (Duesterwald e Bala, 2000).

Assim que um ciclo é identificado, ele é inserido em uma fila FIFO que alimenta a segunda *thread* do sistema. Esta *thread* contém o otimizador e um tradutor baseado em LLVM. Primeiramente, o otimizador utiliza os ciclos identificados pelo *profiler* e pelo

preditor com o objetivo de fundir ciclos e identificar unidades de tradução maiores, que possam ser otimizadas para reduzir os custos de execução e acelerar o desempenho. Esta otimização é feita em três estágios. No primeiro, as unidades de otimização identificadas são acumuladas para determinar quais são mais frequentemente executadas. Assim que alguma unidade de otimização é reconhecida como frequente, o processo chega ao segundo estágio. Neste estágio, é verificado se a unidade de otimização está apta a ser otimizada com outras que foram executadas em um mesmo período de tempo. Depois de escolher as unidades de otimização, a fusão entre elas ocorre no terceiro estágio, resultando em unidades de tradução contendo código cujo fluxo de execução está altamente relacionado.

Tanto o processo de tradução rápida quanto a tradução de unidades de otimização é realizado por meio do uso da LLVM. Para isso, o código intermediário gerado pelo QEMU é traduzido para a representação intermediária da LLVM, LLVM-IR. Com esta representação, é possível aplicar várias otimizações agressivas com o objetivo de melhorar o desempenho do código ao máximo. Assim que o código é otimizado, a tradução final para código nativo é efetivada e registrada para ser chamada corretamente.

Embora o uso de um compilador otimizador seja desejável por gerar código de alta qualidade, o tempo necessário para realizar as otimizações pode ser longo. Desta forma, ao incluir o tradutor e o otimizador binário dinâmico em uma *thread* separada de execução, o autor demonstra que seu tempo de execução pode ser mascarado, e, assim que a tradução torna-se disponível é possível continuar a execução sem mecanismos de sincronização.

Outro fator que pode prejudicar o desempenho é o processo de *profiling* dinâmico. Para determinar as unidades de otimização aptas a serem fundidas, é necessária a inclusão de rotinas no código que determinam quais foram os caminhos executados em um determinado período da execução. Desta forma, parte do tempo de execução da aplicação é tomado na execução destas rotinas. Os autores do HQEMU e de outros trabalhos (Buytaert et al., 2007; Cuthbertson et al., 2009; Schneider et al., 2007) sugerem a utilização de contadores de eventos em *hardware* para constatar o fluxo de execução de maneira indireta. Esta técnica, chamada de HPM (*Hardware Performance Monitoring*), deixa a cargo do próprio processador e sistema operacional da máquina hospedeira fazer o registro do contador de programa de forma periódica com o objetivo de rastrear o caminho da execução. De acordo com o autor, o custo do uso das rotinas de interpretação chega a ser de $24,9\%$ do tempo de execução de aplicações do SPEC2006 para inteiros e $11,7\%$ do tempo de execução de aplicações do SPEC2006 para ponto flutuante, enquanto o custo de utilizar HPM é de apenas $1,3\%$ em média.

3.4 HARMONIA

Harmonia (Ottoni et al., 2011) é um emulador experimental baseado no QEMU cujo objetivo é executar programas ARM em aquiteturas x86_64. Em média, o Harmonia executa programas do *benchmark* SPEC2000-INT ARM com 55% da eficiência das execuções nativas. Os autores também afirmam que a execução é 2, 2 vezes mais rápida que a execução com QEMU. O objetivo deste emulador é aprimorar a execução de arquiteturas utilizadas em dispositivos embarcados em arquiteturas Intel. O objetivo é desenvolver um emulador que possa ser executado em futuros processadores Intel embarcados e que possam executar aplicações já existentes para arquiteturas populares como ARM. Entretanto, por se tratarem de processadores menos poderosos que os IA tradicionais, é necessário desenvolver técnicas de TDB eficientes. Desta forma, o usuário pode desfrutar de toda biblioteca de títulos disponíveis para ARM no lançamento dos processadores embarcados Intel.

A principal contribuição do trabalho é a investigação sobre problemas específicos na geração de código para arquiteturas baseadas na arquitetura IA, da Intel, tais como x86 e x86_64. Em específico, o trabalho trata de dois problemas que tornam a geração de código eficiente para IA mais difícil, a saber: o problema da alocação de registradores e o problema dos códigos de condição. Além disto, o trabalho apresenta algumas técnicas de tradução e fusão de blocos eficientes.

O Harmonia utiliza um esquema de tradução de duas marchas, onde na primeira marcha o código é gerado utilizando o QEMU padrão, sem modificações no processo de tradução. De fato, a única modificação realizada no QEMU é a inserção de instrumentação no início de cada unidade de tradução para detectar se a mesma é um *hotspot*. Assim que uma unidade de tradução torna-se um *hotspot*, sua tradução é passada para a segunda marcha. O objetivo da segunda marcha é gerar código mais eficiente, baseando-se em três técnicas principais, a saber:

- Tradução direta das instruções alvo em instruções hospedeiras;
- Otimizações para mapeamento de registradores; e
- Otimizações de códigos de condição.

No entanto, antes de aplicar as três técnicas acima, é identificada uma região de BTs quentes para que mais otimizações possam ser aplicadas. Para isto, é utilizada uma técnica similar àquela utilizada no simulador HQEMU, explicada anteriormente. No entanto, as regras para decidir a fusão dos blocos é diferente. Como não há *profiling* contínuo, outra

abordagem é utilizada. Nesta abordagem, é utilizado um limiar de afinidade, onde somente os blocos vizinhos que também forem *hotspots* são fundidos. Quando uma unidade de tradução é gerada a partir desta análise, as técnicas citadas anteriormente são utilizadas.

O autor argumenta que parte da ineficiência do QEMU é inerente ao uso do GCC na geração de código para as micro operações. Em específico, ele aponta que os custos de carga de operandos imediatos e acessos à memória podem ser minimizados ao emitir código da máquina hospedeira diretamente. No entanto, para seguir esta abordagem é necessário deixar de lado a portabilidade do QEMU em função do benefício da tradução direta. O autor argumenta que a geração de código nativo diretamente permite a emissão de operandos imediatos diretamente nas instruções e melhores sequências de instruções nos casos que a geração de código do GCC não é ótima. Além da emissão direta, o Harmonia também emprega várias otimizações tradicionais em sistemas TDB para alcançar melhor desempenho, além de otimizações propostas no trabalho para alocação de registradores e códigos de condição.

Ottoni também discute os principais desafios enfrentados ao desenvolver TDB com arquitetura Intel como hospedeira. O baixo número de registradores e a escassez de registradores de códigos de condição apresentam desafios que, se não forem abordados corretamente, causam sérios prejuízos ao tempo de execução das aplicações. Os algoritmos apresentados no trabalho, juntamente com as estratégias de determinação e tradução de unidades de tradução foram projetados para serem executados em dispositivos embarcados. No entanto, embora a tradução direta resulte em código mais enxuto, o que faz a abordagem do GCC menos eficiente é o projeto das micro operações, não o uso do GCC em si. Como o GCC não possui informações sobre as micro operações adjacentes durante a compilação, por serem compiladas separadamente, não é possível gerar código que seja ótimo em todas as situações. Desta forma, como a tradução das instruções é implementada a partir da concatenação de código de micro operações já compiladas separadamente, resta código desnecessário na tradução. No entanto, não é necessário utilizar tradução direta para sanar estes problemas. Outros trabalhos, como HQEMU e o EHS realizam tradução com o auxílio de compiladores de propósito geral, como LLVM e o próprio GCC, respectivamente.

3.5 ARCSIM

O ARCSim (Böhm et al., 2011a,b; Powell e Franke, 2009) é um simulador de sistema projetado para executar programas desenvolvidos para a arquitetura ARCompact, uma arquitetura RISC para processadores de baixo consumo de energia desenvolvido na Uni-

versidade de Edimburgo. A principal contribuição do ARCSim está no desenvolvimento de um sistema de compilação em paralelo capaz de esconder a latência da tradução dinâmica, ao mesmo tempo utilizando as características de otimização da LLVM para gerar código mais eficiente.

O simulador é um aprimoramento do EHS, proposto por Jones (Jones, 2010). Para determinar as unidades de tradução longas, é utilizada a abordagem baseada em grafos de fluxo de controle descrita na Seção 2.2. O processo de *profiling* continua sendo executado apenas durante a interpretação. Portanto, os problemas relacionados à fragmentação de UTLs e à insensibilidade a alteração nas fases do programa permanecem.

No entanto, a novidade está no processo de compilação, que, ao invés de acontecer sequencialmente à execução da aplicação, acontece em *threads* separadas. Desta forma, a execução prossegue na *thread* principal com a interpretação. Com efeito, a execução da compilação em outra *thread* esconde a latência da tradução, ao mesmo tempo que permite que mais otimizações sejam aplicadas em tempo de execução.

Assim que as UTLs são identificadas, são colocadas em uma fila de prioridade. Esta fila de prioridades é acessada pelas *threads* de tradução sempre que alguma tradução é finalizada. Para obter o melhor desempenho, a fila está organizada de forma que dê prioridades às UTs mais frequentemente utilizadas durante a execução.

No entanto, ao invés de traduzir as UTLs durante a execução com GCC, as UTLs são convertidas para a representação intermediária da LLVM. Com isto é possível utilizar as otimizações existentes na LLVM para gerar código eficiente. Uma das principais otimizações ao gerar código para arquiteturas IA é a otimização de promoção de registradores, que servem para eliminar acessos redundantes à memória (Ottoni et al., 2011). Além disso, a LLVM também dispões de técnicas de seleção de instruções poderosas, que utilizam instruções SIMD avançadas para gerar código eficiente para uma sequência de operações escalares (Hong et al., 2012). Esta série de otimizações, embora melhorem o código significativamente, são relativamente demoradas, e, caso não fossem executadas em paralelo em relação à simulação, se tornariam impraticáveis. Após a conversão em LLVM-IR e os passos de otimização, o compilador JIT da LLVM é utilizado para gerar código nativo. O código então é gerado e colocado no mapa de traduções. A partir deste momento, o código fica disponível para execução posterior.

A contribuição principal do ARCSim está ligada à forma com que a tradução é conduzida utilizando *threads* distintas. Desta forma, é possível aproveitar as facilidades e os passos de otimização disponibilizados pela LLVM para gerar código eficiente utilizando os recursos computacionais disponíveis, sem degradar o desempenho da simulação.

3.6 CONSIDERAÇÕES FINAIS

Os trabalhos apresentados neste Capítulo utilizam diferentes técnicas de determinação de unidades de tradução. No entanto, apenas um deles, HQEMU, utiliza *profiling* contínuo para determinar fluxos de execução quentes. No entanto, a abordagem sugerida no HQEMU requer a utilização de HPM como alternativa à instrumentação de instruções interpretadas e traduzidas. O objetivo deste trabalho é utilizar instrumentação em instruções interpretadas apenas para determinar blocos básicos. O restante do processo de *profiling* é realizado apenas no laço do emulador, não sendo necessária instrumentação nem HPM para monitorar o fluxo de execução dentro de fluxos traduzidos. O próximo Capítulo apresenta a proposta que caracteriza a principal contribuição deste trabalho: *profiling* contínuo para determinação de unidades de tradução utilizando instrumentação de transições.

4

PROFILING CONTÍNUO PARA DETERMINAÇÃO DE UNIDADES DE TRADUÇÃO

A contribuição esperada com este trabalho é a elaboração de técnicas que devem proporcionar a identificação de unidades de tradução longas quentes utilizando *profiling* contínuo, evitando a fragmentação de UTLs. Embora a utilização de *profiling* contínuo seja uma solução discutida anteriormente por Jones (Jones, 2010), o autor explica que sua implementação potencialmente leva a problemas de desempenho devido ao fato da necessidade de introduzir código de monitoramento dentro das unidades traduzidas.

Por outro lado, o presente trabalho propõe a utilização de *profiling* contínuo, contudo monitorando apenas as transições entre as unidades de tradução do programa. Desta forma, não é necessário introduzir código dentro das unidades traduzidas. No entanto, esta não é a única contribuição. A proposta atual também integra a utilização de análises no grafo de fluxo de controle para determinar quais unidades de tradução podem ser combinadas, inclusive fluxos de execução previamente traduzidos. Embora as técnicas de fusão sejam promissoras, existem problemas que podem degradar a execução, principalmente em relação à quantidade de chamadas necessárias ao sistema de síntese e tradução. Portanto, também foram definidos mecanismos que regulam estas chamadas, permitindo apenas traduções que levem a ganhos significativos no desempenho da execução.

O restante deste tópico está dividido como segue: A Seção 4.1 apresenta uma visão geral das estratégias utilizadas na proposta. A Seção 4.2 mostra detalhes sobre o projeto

da proposta, indicando as principais diretivas utilizadas na elaboração das estratégias apresentadas.

4.1 VISÃO GERAL DA PROPOSTA

A proposta é composta de uma cooperação entre três fases que operam em conjunto para determinar fluxos de execução quentes entre as unidades de tradução do programa alvo. As unidades de tradução correspondem a conjuntos de instruções que constituem o programa sendo executado. Dois tipos de unidades de tradução são considerados na proposta, a saber: blocos básicos, de acordo com a Definição 2 e fluxos. A Figura 4.1 - apresenta a relação entre as três fases envolvidas no sistema de TDB proposto.

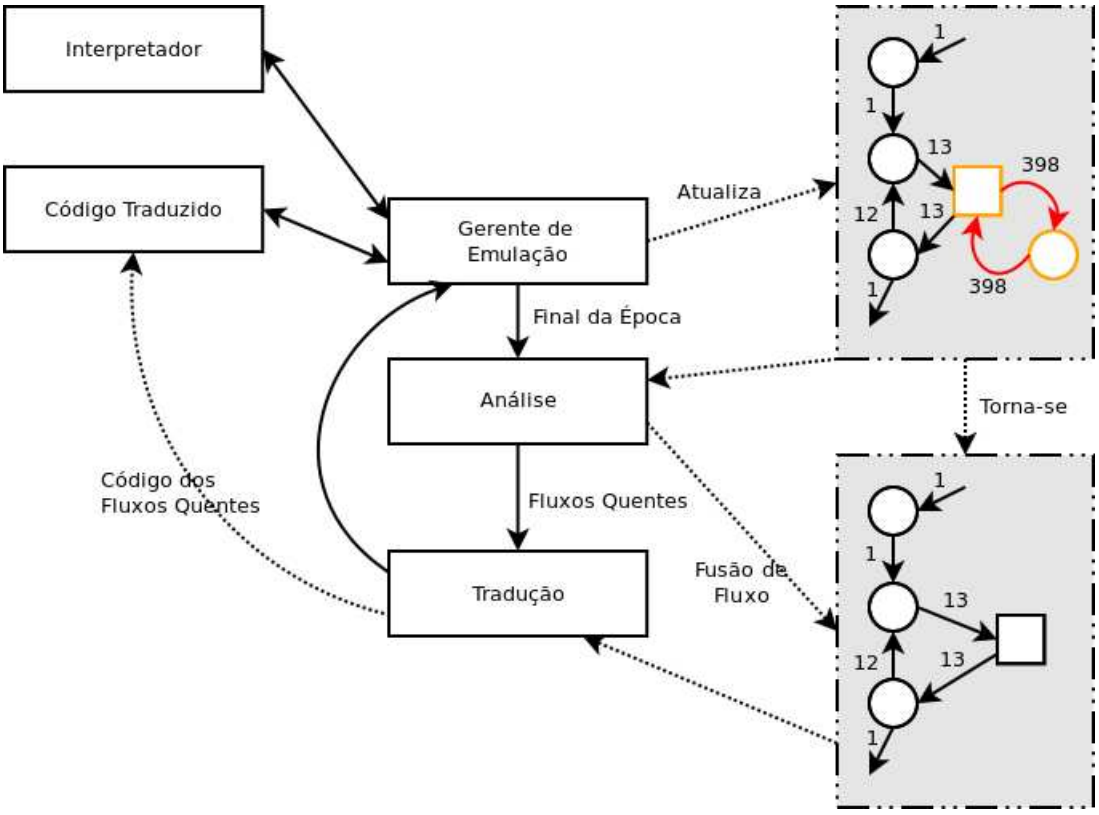


Figura 4.1: Visão Geral da Proposta

Durante o processo de *profiling* e execução são realizadas as tarefas de monitoramento do fluxo de controle entre as unidades de tradução. O tempo de execução é discretizado e dividido em épocas, onde cada época dura uma quantidade pré-estabelecida de transições entre unidades de tradução. A execução pode acontecer de duas maneiras: por interpretação ou pela execução de fluxos traduzidos. A interpretação ocorre quando

o contador de programa aponta para uma instrução que não está vinculada a qualquer fluxo traduzido.

Definição 8. Fluxo – Conjunto de blocos básicos e outros fluxos, juntamente com suas respectivas transições.

Durante a interpretação rotinas de *profiling* são responsáveis por descobrir os blocos básicos que constituem o programa. Esta tarefa não é simples, pois nem todos os alvos de saltos podem ser descobertos estaticamente. Portanto, a abordagem utilizada não emprega informações estáticas para determinar blocos básicos, que são demarcados como na Definição 2.

Um problema específico na descoberta de blocos básicos está relacionado à descoberta de um alvo de salto no meio de um bloco básico previamente delineado. Quando isto acontece, é necessário que haja a divisão do bloco básico para que a semântica do grafo de fluxo de controle permaneça coerente. Caso a divisão ocorra em um bloco básico que já tenha sido incluído em algum fluxo, é necessário que o fluxo seja atualizado para acolher a informação sobre o novo ponto de entrada.

Quando o contador de programa aponta para uma instrução vinculada a um fluxo traduzido, o código correspondente é chamado e a execução permanece até que o fluxo de controle atinja uma instrução que não está contida no fluxo. Neste momento, o fluxo da execução do emulador é retornado ao laço principal da emulação, onde o monitoramento de transições é realizado.

A tarefa de monitorar transições consiste em acompanhar o fluxo de execução entre as unidades de tradução. Cada vez que uma transição é detectada, um grafo de fluxo de controle que contém o *profile* da época atual é atualizado. O grafo de fluxo de controle (GFC) é uma estrutura de dados que contém as informações sobre o fluxo de execução do programa, identificando quantas vezes cada unidade de tradução foi executada, bem como quantas trasições entre unidades de tradução ocorreram na última época. Formalmente, o grafo de fluxo de controle $G = (V, E)$ é constituído por um conjunto de vértices V e um conjunto de arestas direcionadas E, cujos vértices representam unidades de tradução e as arestas representam as transições na execução entre unidades de tradução. Cada aresta também possui a informação de quantas vezes a transição correspondente foi executada, que é utilizada posteriormente para determinar os fluxos de execução quentes. Ao final de cada época, o grafo de fluxo de controle da época é analisado com o objetivo de determinar fluxos quentes de execução.

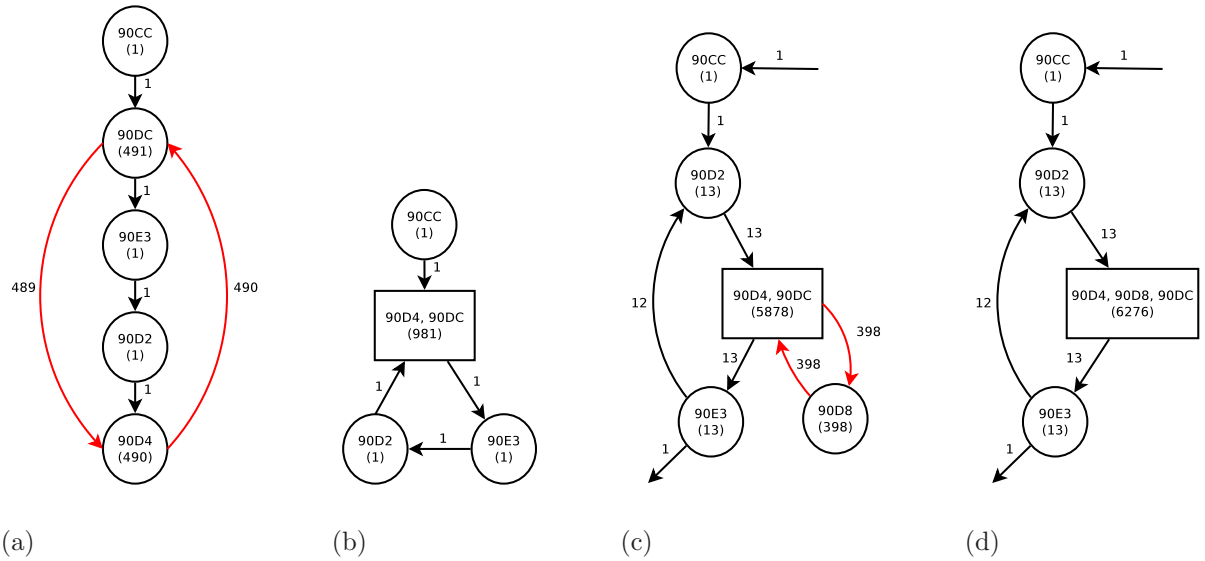
Com o *profile* gerado durante a fase de *profiling*, é possível determinar quais fluxos de execução foram os mais recorrentes durante a época atual. Trabalhos anteriores (Bala et

al., 2011; Hong et al., 2012) sugerem que fluxos de execução baseados em estruturas de repetição tendem a serem quentes, pois indicam uma região de alta localidade temporal. Em um grafo de fluxo de controle, tais fluxos são representados por ciclos entre as unidades de tradução correspondentes. Portanto, a principal tarefa da fase de análise é determinar tais ciclos utilizando um algoritmo capaz de detectá-los, utilizando as informações sobre transições entre unidades de tradução como heurística para determinar fluxos quentes de execução. Para determinar tais ciclos de maneira eficiente e satisfatória durante a execução da fase de análise, foi desenvolvido um algoritmo, denominado HotDFS, que é apresentado posteriormente.

No entanto, nem sempre é vantajoso traduzir todos os fluxos detectados pelo HotDFS. Portanto, antes de serem traduzidos, o calor dos fluxos é avaliado para determinar se o custo da tradução compensa o ganho posterior em sua execução. Caso o calor de determinado fluxo exceda um limiar dinâmico baseado no fluxo da execução da aplicação, o fluxo é escalonado para síntese e tradução.

No entanto, antes de serem traduzidos, os ciclos quentes são fundidos em uma única unidade de tradução, um fluxo. Neste processo, o grafo de fluxo de controle é atualizado e os fluxos gerados substituem suas partes constituintes. Desta forma, toda vez que o laço principal do emulador detecta a transição entre qualquer outra unidade de tradução e uma instrução dentro de um fluxo, a transição registrada é entre a unidade de tradução e o fluxo. Tal característica contribui para a determinação de fluxos quentes de maneira recursiva, que constitui um mecanismo uniforme e eficiente para determinar fluxos quentes com o objetivo de balancear a necessidade de compilação com a determinação de unidades que sejam realmente executadas recorrentemente.

A Figura 4.2 - mostra um exemplo de fusão de fluxo. Os ovais representam blocos básicos e as caixas representam fluxos. As arestas pretas são consideradas frias, enquanto as arestas vermelhas são consideradas quentes em relação ao limiar atual vigente nas épocas correspondentes, portanto indicam quais nós devem ser fundidos em um único fluxo de execução. As Figuras (a) e (b) ilustram a fusão dos blocos básicos iniciados nos endereços 90DC e 90D4 em um único fluxo. Nota-se que, no processo de fusão as arestas externas devem ser remanejadas para apontarem para o novo fluxo, e não para suas unidades de tradução constituintes. Além disso, a fusão é realizada apenas no fluxo de execução quente, marcado em vermelho. Ao final de outra época posterior, as Figuras (c) e (d) representam a fusão de dois tipos de unidades de tradução distintas. Uma delas é a unidade de tradução resultante da fusão mostrada nas Figuras (a) e (b). Portanto, o algoritmo HotDFS reconhece corretamente que agora há outro fluxo de execução quente



(a) (b) (c) (d)

Figura 4.2: Fusão de Fluxo

envolvendo estas unidades de tradução, além de uma terceira, um bloco básico iniciado no endereço 90D8.

Conforme mais unidades quentes são fundidas, mais tempo é despendido na execução de unidades traduzidas. Desta forma, o grafo de fluxo de controle acompanha a transição do fluxo de controle entre os vários níveis da aplicação sendo executada. Em outras palavras, a proposta atual visa identificar os fluxos de execução com granularidade crescente em função à estrutura do programa e de seu respectivo tamanho. Portanto, em situações diferentes da execução do programa, o *profile* gerado durante determinada época pode representar níveis de granularidade variáveis em relação ao tamanho das unidades de tradução.

Após a análise, os fluxos quentes são traduzidos para código C, em uma tarefa denominada síntese. Durante a síntese, informações sobre as transições e as unidades de tradução que compõem determinado fluxo são utilizadas para gerar código eficiente. Embora o foco da proposta não tenha sido a fase de síntese e tradução, são utilizadas algumas técnicas de TDB conhecidas para geração de código eficiente. Entre elas estão a utilização de tabelas de salto para conectar blocos básicos, predição de saltos em desvios indiretos e a utilização de macros na descrição das instruções da máquina alvo, com o objetivo de oferecer oportunidades de otimização mais agressivas ao compilador dinâmico. A prova de conceito concebida utiliza o compilador GCC como compilador dinâmico, juntamente com a biblioteca de ligação dinâmica oferecida pela plataforma Linux para ligar o código gerado dinamicamente ao emulador.

As próximas seções descrevem detalhes sobre o projeto da proposta. O objetivo é mostrar quais foram as principais diretivas consideradas durante o projeto, evidenciando as decisões que influenciam de maneira direta nas contribuições oferecidas por este trabalho.

4.2 DETALHES DA PROPOSTA

4.2.1 Profiling

O diferencial da proposta apresentada está no fato que o processo da aquisição das informações sobre a execução do programa, chamado *profiling*, é realizado não somente durante a interpretação (Jones, 2010; Jones e Topham, 2009), mas também durante a execução de código traduzido. Isto permite avaliar o fluxo de execução não somente entre unidades de tradução interpretadas, mas também entre unidades de tradução interpretadas e traduzidas.

A proposta de TDB apresentada neste trabalho se baseia no fato que laços de repetição representam bons candidatos a serem traduzidos, devido ao fato que normalmente representam estruturas programáticas que apresentam alta localidade temporal (Bala et al., 2011). Com a utilização de *profiling* interpretativo, a tendência é que somente os laços mais internos tornem-se reconhecidos como quentes. Isso se deve ao fato que, uma vez que são traduzidos, não é possível determinar as transições entre o laço e as demais unidades de tradução. Desta forma, torna-se difícil detectar estruturas mais complexas, como estruturas de repetição aninhadas. Com efeito, parte do problema de fragmentação de unidades de tradução longas, como apresentado na Seção 2.2.2, é resultado da estratégia de *profiling* intepretativo.

Um dos objetivos deste trabalho é apresentar uma solução para o problema de fragmentação de unidades de tradução longas com o uso de *profiling* contínuo. A abordagem de Jones no EHS (Jones, 2010; Jones e Topham, 2009) utiliza *profiling* interpretativo, coletando a quantidade de execuções de cada bloco básico executado em uma época. Com base nesta informação, Jones analisa seus grafos de fluxo de controle e determina unidades de tradução longas, como descrito na Seção 3.1. Dessa forma, nenhuma informação sobre a execução das UTLs é coletada. Além disso, Jones também não coleta a contagem das transições entre as unidades de tradução, dificultando a determinação dos fluxos de execução quentes. A partir das características dos *profiles* gerados pelo EHS, o problema de fragmentação de unidades de tradução longas é agravado: unidades de tradução são determinadas usando apenas informações sobre a quantidade

de execuções de cada bloco básico. Além disso, por não recolher informações sobre as transições entre unidades de tradução, exclui a possibilidade de fundir unidades de tradução adjacentes, além de impedir a aplicação de algumas otimizações relacionadas ao fluxo de execução.

A abordagem utilizada neste trabalho é baseada em *profiling* contínuo, que é utilizado para contar a quantidade de execuções de cada unidade de tradução, bem como a quantidade de transições entre duas unidades de tradução distintas. Primeiramente, o tempo de execução é discretizado em Épocas, onde cada época dura uma quantidade pré-estabelecida de transições. O Algoritmo 1 sintetiza o processo de *profiling* contínuo adotado na abordagem atual.

Algoritmo 1 *Profiling* Contínuo

```
Transições := TAMANHO_ÉPOCA

REPITA

      UT_ANTERIOR := UT_ATUAL
      SE Fluxo(PC) ENTÃO
            UT_ATUAL := Fluxo(PC)
            UT_ATUAL.Execuções++
            CALL Fluxo(PC)
      SENÃO
            CALL TratarBlocoBásico(PC)
            UT_ATUAL = BB(PC)
            SE PC = UT_ATUAL.EndereçoInicial ENTÃO
                  UT_ATUAL.Execuções++
            FIM SE
            CALL Interpretador(PC)
      FIM SE

      SE UT_ANTERIOR != UT_ATUAL ENTÃO
            CALL IncrementarAresta(UT_ANTERIOR, UT_ATUAL) EM GFC
            Transições--
      FIM SE

ENQUANTO Transições > 0
```

Toda vez que o laço principal do emulador é invocado, a unidade de tradução anterior é considerada como sendo a unidade de tradução atual. Em seguida, o valor do contador de programa é analisado. Caso o valor atual esteja vinculado a um fluxo traduzido, a unidade de tradução atual se torna o fluxo traduzido, que então é invocado. Caso

contrário, PC está apontando para uma instrução que ainda não está associada a nenhum fluxo. Desta forma, é necessário determinar o bloco básico a qual esta instrução pertence. Primeiramente, é verificado se a instrução atual já está inserida em algum bloco básico previamente reconhecido. Neste caso, basta incrementar o contador de execuções do bloco básico correspondente e atualizar a unidade de tradução atual para o bloco básico atual. Caso contrário, é necessário vincular a instrução atual a um bloco básico.

Caso já haja algum bloco básico em construção, basta adicionar a instrução atual no bloco básico em construção. Além disso, é necessário determinar se a instrução atual corresponde a uma instrução que determina o final de um bloco básico. Por outro lado, caso não haja nenhum bloco básico em construção, é necessário determinar o início de um novo bloco básico. Independentemente de estar vinculada a um bloco básico previamente existente, a instrução atual é interpretada normalmente sem necessidade adicional de monitoramento.

De qualquer forma, a instrução atual sempre pertence a uma unidade de tradução antes de sua execução. Desta forma, a unidade de tradução anterior, que determina qual unidade de tradução a instrução ou fluxo de instrução anterior se encontra, é salva antes da execução da próxima instrução. Assim, a instrução atual pode estar em outra unidade de tradução, o que caracteriza uma transição no fluxo de controle do programa. Assim que uma transição é encontrada, o grafo de fluxo de controle é atualizado de acordo. Além disto, o controle da época é realizado, decrementando o contador de transições. Uma vez que o contador de transições atinge o valor zero, o processo de *profiling* da época atual é finalizado e a análise do grafo de fluxo de controle é realizada para determinar fluxos quentes.

A principal desvantagem apontada por outros trabalhos (Bala et al., 2011; Jones, 2010; Ottoni et al., 2011) em relação à *profiling* contínuo está relacionado ao custo de realizar a medição das métricas de execução dentro dos fluxos traduzidos. A abordagem apresentada nesse trabalho minimiza este custo, realizando toda a medição das transições dentro do laço principal do emulador, ou seja, não utiliza tempo de execução do fluxo traduzido para manter informações sobre o fluxo de execução do programa. Como as unidades de tradução constituintes de um dado fluxo são fundidas em uma única unidade de tradução no grafo de fluxo de controle, a abordagem apresentada neste trabalho permite o monitoramento das transições entre todas as entradas e saídas do fluxo traduzido. O principal diferencial deste trabalho em relação ao trabalho de Hong (Hong et al., 2012) está no fato que as rotinas de *profiling* contínuo são inseridas no código do fluxo traduzido, que, segundo o autor, constituem até $24,9\%$ do tempo de execução da aplicação. Por outro

lado, o presente trabalho realiza *profiling* apenas durante a execução do laço principal do emulador, reduzindo o custo significativamente.

4.2.2 Análise

Ao final de cada época o grafo de fluxo de controle é analisado para determinar os fluxos quentes de execução. A principal heurística utilizada na determinação de tais fluxos baseia-se na premissa que estruturas de repetição apresentam alta localidade temporal, e consequentemente, conjuntos pequenos de unidades de tradução que possuem alta taxa de transição entre si. Em um GFC, estruturas de repetição são representadas por ciclos. Portanto, é possível utilizar algoritmos de detecção de ciclos para determinar possíveis estruturas de repetição, e consequentemente, determinar fluxos de execução potencialmente quentes.

Um dos principais objetivos de um sistema de TDB é o desempenho. Como todas as tarefas que não fazem parte da execução do programa alvo são considerados como custos, é necessário que tais tarefas sejam executadas de maneira eficiente. Portanto, os algoritmos subjacentes devem ser eficientes, de maneira que seu tempo de execução torne-se desprezível em função do benefício que proporcionam. Desta forma, os algoritmos utilizados na fase de análise dos grafos de fluxo de controle devem ser eficientes, ao mesmo tempo sendo capazes de determinar fluxos de execução relevantes.

Embora o algoritmo de detecção de ciclos, que é apresentado a seguir, tenha tempo de execução linear em função da quantidade de arestas do grafo, sua execução pode tornar-se onerosa, principalmente se o GFC da época for composto de muitos nós. Além disso, o algoritmo de detecção de ciclos precisa ser executado mais de uma vez, aumentando ainda mais o problema do desempenho. A Figura 4.3 - apresenta o processo de análise do grafo de fluxo de controle.

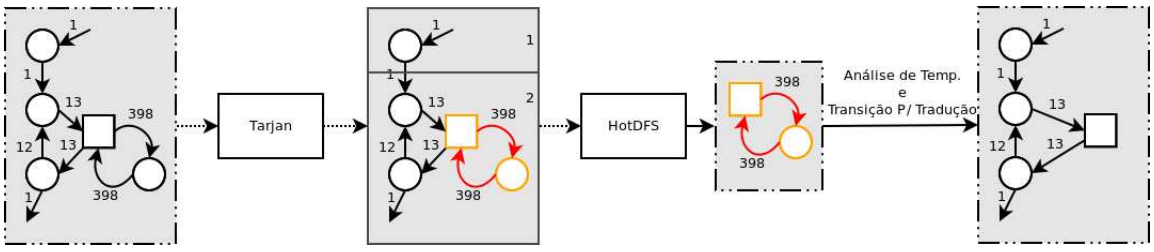


Figura 4.3: Análise de Fluxo de Controle

Com o objetivo de minimizar este problema foi utilizada a estratégia de particionar o grafo para minimizar o espaço de busca por ciclos. Para isto, foi utilizado o conceito de componentes fortemente conexos (CFC). Um componente fortemente conexo é um

grafo dirigido tal que todos os nós possuem pelo menos um caminho entre todos os outros (Cormen et al., 2001). Thulasiraman e Swamy(Thulasiraman e Swamy, 1992) provam que componentes fortemente conexos de um grafo dirigido formam, entre si, um grafo acíclico dirigido. Portanto, um ciclo pode estar somente em um componente fortemente conexo por vez. Desta forma, ao particionar o grafo em componentes fortemente conexos é possível diminuir o espaço de busca por ciclos. Assim, diminuir o espaço de busca necessário para determinar ciclos não diminui o poder de encontrar ciclos, consequentemente fluxos quentes, devido ao fato que a partição em CFCs garante que os ciclos estejam contidos em um mesmo componente.

Outro problema na determinação de ciclos está na necessidade de existir um nó como ponto inicial na busca. Mesmo que o particionamento em componentes fortemente conexos diminua o espaço de busca por ciclos, é interessante obter informações sobre os componentes de maneira a iniciar a busca por ciclos em nós estratégicos, de maneira que não seja necessário passar por todos os nós do grafo. Além disto, somente os ciclos que possuem alta quantidade de transições entre seus nós constituintes são interessantes na detecção de fluxos quentes. Portanto, nem todos os ciclos são interessantes. Em relação a este problema, os nós do grafo podem ser analisados para determinar qual o maior valor entre todas suas arestas, ou seja, qual a transição mais recorrente a partir de dado nó. Com esta informação é possível determinar quais nós são mais propensos a retornarem fluxos quentes, possivelmente tornando-se pontos iniciais interessantes para o algoritmo de detecção de ciclos.

Para o particionamento do grafo em CFCs foi utilizado o algoritmo de Tarjan (Tarjan, 1971), que possui complexidade linear em função das arestas $O(|V|+|E|)$. A determinação de transições recorrentes pode ser realizada durante a execução do algoritmo de Tarjan, descartando a necessidade de outra travessia posterior para este fim. Antes de determinar os ciclos dos componentes é necessário determinar os nós pelos quais as buscas devem ser iniciadas. Para isto, são analisadas as transições recorrentes de cada nó dos CFCs. Caso as transições recorrentes sejam maiores que determinado limiar, denominado de Limiar de Calor do Nó (LCN), o Nó que inicia a transição correspondente é marcado como Nó Quente. Tais nós são pontos iniciais de busca razoáveis, pois direcionam a busca por ciclos, consequentemente fluxos, que contenham quantidade satisfatória de transições.

Definição 9. Nó Quente – Nós que são considerados como pontos iniciais para a busca de fluxos quentes de execução por originarem uma quantidade satisfatória de transições entre unidades de tradução.

Definição 10. Limiar de Calor do Nó (LCN) – Limiar que representa a quantidade de transições entre unidades de tradução que deve ser considerada para determinar o calor de dado nó. Nós cuja aresta de maior rótulo de transições ultrapassa essa valor são considerados quentes.

O principal desafio em determinar nós quentes é determinar o limiar de calor do nó de maneira apropriada. Isto se deve ao fato que este parâmetro é estreitamente ligado à estrutura do programa sendo executado. Além disto, o padrão de execução do programa pode ser alterado dinamicamente durante sua execução. Desta forma, o LCN deve ser determinado durante a execução do programa. Na abordagem atual, o LCN é recalculado a cada época e deve refletir o fluxo de execução atual. Caso o valor do limiar se torne muito baixo, a quantidade de pontos iniciais se torna muito elevada, levando a excessivas buscas por ciclos que posteriormente são excluídas por sua baixa temperatura. Por outro lado, caso o LCN se torne muito alto, não haverão pontos iniciais para busca por fluxos quentes, e a fusão de unidades de tradução pode tornar-se estagnada. Portanto, é necessário determinar uma fórmula que equilibre o limiar de maneira a predizer apenas chutes que representam boas chances de encontrar fluxos quentes. A fórmula utilizada para computar o limiar de calor do nó na próxima época, $k+1$, é dada por:

$$LCN_{k+1} = \frac{\left(\frac{\sum_{\epsilon \in E_k | \epsilon > 1} \epsilon}{\sum_{\epsilon \in E_k | \epsilon > 1} 1} \right) + LCN_k}{2}$$

Sendo E o conjunto de arestas do grafo de fluxo de controle, ϵ a quantidade de transições de determinada aresta e L_n o limiar de calor do nó na época n. O objetivo desta fórmula é considerar os nós cujo valor da maior transição seja pelo menos a média das transições não-unitárias da época atual. Nota-se que o conjunto de arestas atual é utilizado para calcular o limiar da próxima época. Embora este cálculo se torne menos preciso ao ser realizado com dados baseados em uma época anterior, é possível computar o limiar durante a execução do algoritmo de Tarjan, sem a necessidade de outra travessia. Além disso, a fórmula utiliza também o limiar da época anterior para suavizar mudanças abruptas que podem ocorrer entre uma época e outra, mantendo o limiar no valor médio das transições.

Ao final da execução do algoritmo de Tarjan o GFC torna-se particionado em CFCs, que são explorados para determinação de fluxos de execução quentes. Além disso, cada

componente possui uma lista de seus respectivos nós quentes, que são utilizados como pontos iniciais para determinação de fluxos de execução quentes.

O Algoritmo 2 apresenta o algoritmo utilizado para determinar os ciclos no GFC, denominado HotDFS. O algortimo é uma versão modificada da busca em profundidade que utiliza uma heurística gulosa para determinar fluxos de execução quentes em relação à quantidade de transições entre as unidades de tradução. O algoritmo é aplicado para cada nó quente determinado durante a execução do algoritmo de Tarjan. É importante perceber que o algoritmo somente faz a travessia entre nós do mesmo componente fortemente conexo, direcionando a busca com o objetivo de tornar sua implementação eficiente, não necessariamente reduzindo sua complexidade assintótica.

Para detectar ciclos em um grafo dirigido qualquer, apenas o encontro de um nó anteriormente visitado não é suficiente para garantir que um ciclo foi encontrado. A literatura (Cormen et al., 2001; Thulasiraman e Swamy, 1992), no entanto, apresenta um algoritmo baseado em categorizar o estado da visita de um nó e seus descendentes com um esquema de cores. Os autores destes trabalhos afirmam que é possível dizer que há um ciclo em um grafo dirigido desde que, durante a travessia, um nó seja visitado novamente antes que todos seus descendentes tenham sido visitados.

Desta forma, todos os nós são inicialmente marcados de branco, indicando que ainda não foram visitados. Quando visitado, o nó é marcado de cinza, permanecendo nesta cor até que todos os seus descendentes tenham sido visitados, oportunidade em que é marcado na cor preta. Portanto, se durante a travessia algum nó cinza for encontrado, significa que há algum caminho entre seus descendentes que leva novamente ao nó cinza, fechando assim, um ciclo.

Com o objetivo de encontrar ciclos que correspondam a fluxos de execução recorrentes, as arestas são visitadas em ordem descrescente de transições entre as unidades de tradução adjacentes em um mesmo componente fortemente conexo. Desta forma, o algoritmo tenta determinar ciclos com altas taxas de transição, levando assim à determinação de um fluxo de execução potencialmente quente para tradução.

De fato, como se trata de uma heurística gulosa para determinação de fluxos quentes, o algoritmo não garante encontrar fluxos de execução maximais, nem mesmo fluxos ótimos não-maximais. No entanto, cumpre o objetivo de encontrar fluxos coesos e recorrentes na execução da época atual. Além disso, caso o fluxo seja realmente recorrente, virá novamente a ser candidato à fusão em épocas posteriores, permitindo que o restante do fluxo previamente descartado seja fundido à fluxos já traduzidos. Além disso, outro objetivo no projeto deste algoritmo foi o desempenho. É trivial mostrar que sua

Algoritmo 2 HotDFS - Algoritmo para Determinação de Fluxos Quentes

```
GLOBAL Ciclo, PilhaVisita

VISITAR(NoAtual, NoInicial)
    SE Ciclo ENTÃO
            RETORNAR
    FIM SE

    NoAtual.Cor := CINZA
    PilhaVisita.PUSH(NoAtual)

    PARA CADA e EM (NoAtual, Alvo) EM
            Ordem Decrescente de Transições EM CFC FAÇA
        NoAlvo := e.NoAlvo
        SE NoAlvo.Cor = CINZA ENTÃO
            Ciclo := NOVO Ciclo
            PARA CADA NoDoCiclo em PilhaVisita FAÇA
                SE NoDoCiclo != NoInicial
                    Ciclo.INSERT(NoDoCiclo)
                SENÃO
                    BREAK
                FIM SE
            FIM PARA
            Ciclo.INSERT(NoInicial)
        SENÃO
            VISITAR(NoAlvo, NoInicial)
        FIM SE

        SE Ciclo ENTÃO
            RETORNAR
        FIM SE
    FIM PARA

    PilhaVisita.POP()

    NoAtual.Cor = PRETO
FIM

HOTDFS(NoInicial)
    /*InicializarCFC: Marca de branco todos os nós do CFC atual
    que ainda não foram inseridos em um ciclo*/
    InicializarCFC(NoInicial)
    Ciclo := NIL
    PilhaVisita := NIL
    VISITAR(NoInicial, NoInicial)
FIM
```

complexidade de tempo é $O(|V_{CFC}| + |E_{CFC}|)$, sendo assim, um algoritmo eficiente para as restrições da aplicação em TDB.

Assim que todos os ciclos são determinados a partir do algoritmo HotDFS com base nos nós quentes, é necessário analisar se tais ciclos formam fluxos de execução quentes. A temperatura do fluxo é dada pela soma da quantidade de execuções de todas as unidades de tradução que constituem o fluxo. Somente fluxos considerados quentes são fundidos, e posteriormente, traduzidos em código da arquitetura hospedeira.

Definição 11. Grau de Calor do Fluxo – Medida que determina o impacto do fluxo na execução do programa.

Definição 12. Limiar de Tradução – Limiar utilizado para determinar se o fluxo de execução deve ser fundido e traduzido. O objetivo é determinar um limiar de tradução que não cause traduções excessivas, ao mesmo tempo que permita traduções com impacto modesto para evitar estagnação.

Definição 13. Fluxo Quente – Fluxo cuja medida de grau de calor excede o limiar de tradução. Fluxos quentes são traduzidos em instruções da arquitetura hospedeira.

Para determinar se um fluxo é quente, é necessário determinar se o ciclo subjacente é constituído apenas de blocos básicos ou se há algum fluxo traduzido entre os componentes. Ciclos constituídos apenas por blocos básicos são sempre traduzidos. Desta forma, todas as estruturas de repetição executadas apenas por interpretação são traduzidas. Por outro lado, caso haja algum fluxo traduzido entre os componentes é necessário verificar se a nova fusão acarreta em um fluxo mais quente do que os fluxos que já fazem parte de outras traduções. Desta forma, somente traduções que levam a ganhos significativos são efetivadas.

Para determinar o ganho desejado, um limiar denominado limiar de tradução é utilizado. O Limiar de Tradução é um limiar estático, definido antes da execução do programa e, perante a realização de experimentos, foi detectado que um limiar de 10% leva a um equilíbrio eficiente entre a quantidade de traduções efetivadas e a proporção da execução por meio de traduções.

Para decidir se um fluxo é quente, o limiar de tradução é comparado à razão entre a temperatura do fluxo e a temperatura do fluxo mais quente que constitui o novo fluxo. Caso a razão exceda o limiar de tradução, o fluxo é considerado quente e pode ser traduzido no final da época.

A análise de temperatura tem o objetivo de decidir se o novo fluxo encontrado irá impactar positivamente a execução, sendo viável e vantajoso o custo de fusão e tradução

inerente. Vale lembrar que um dos objetivos da proposta é detectar fluxos que levam ao equilíbrio entre a execução de fluxos efetivamente executadas e a quantidade de chamadas ao sistema de síntese e tradução.

Uma vez determinados os fluxos cujas temperaturas estão além do limiar de tradução, o GFC é atualizado em um procedimento denominado fusão de fluxo, descrito anteriormente. A fusão de fluxo consiste em substituir as unidades de tradução constituintes de um fluxo por um único nó que representa o fluxo como um todo no grafo de fluxo de controle, como mostrado na Figura 4.2 - . Dessa forma, o nó representa todas as unidades de tradução do fluxo, permitindo monitorar as transições o fluxo e as demais unidades de tradução do programa.

Unidades de tradução resultantes de fusões de fluxo são representadas internamente como CFGs adicionais, preservando informações sobre as transições entre suas partes constituintes. No CFG original, os nós são agrupados em um único nó e as arestas são recalculadas. Todas as arestas incidentes são recomputadas para apontar para o novo nó que representa o novo fluxo. Arestas novas que ligam o novo fluxo são criadas para apontar para os nós sucessores, enquanto as arestas entre as constituintes e os sucessores são removidas.

A contribuição apresentada no processo de análise está no fato que a fusão de unidades de tradução em um único nó no GFC permite que as transições entre unidades de tradução possam ser monitoradas de maneira simplificada durante o processo de *profiling*. Além disso, esta abordagem permite a detecção de fluxos de execução complexos, como por exemplo, estruturas de repetição e seleção aninhadas.

Para cumprir o objetivo de minimizar o custo do processo de análise, foram apresentados mecanismos utilizados para auxiliar a busca de ciclos em grafos de fluxo de controle potencialmente grandes. A estratégia apresentada une algoritmos simples, porém eficientes, que permitem a identificação de fluxos de execução recorrentes. A proposta também prevê mecanismos para prevenir a compilação excessiva de fluxos de tradução. Antes de serem traduzidos, os fluxos são avaliados para determinar se o ônus de sua tradução é compensado pelo ganho de sua execução posterior.

Ao final do processo de análise, os fluxos quentes são sintetizados em código C e depois traduzidos em código de máquina da arquitetura hospedeira. A próxima Seção apresenta as técnicas de síntese e tradução utilizadas.

4.2.3 Síntese e Tradução

O principal foco deste trabalho está no processo de detecção de unidades de tradução quentes. Desta forma, a abordagem utilizada no processo de síntese e tradução é considerada padrão, utilizando técnicas conhecidas da literatura de projeto e implementação de máquinas virtuais. Alguns ajustes foram feitos nos algoritmos para permitir a utilização de múltiplas entradas em unidades de tradução.

A geração de código equivalente para a arquitetura hospedeira é realizada em duas etapas, a saber: síntese e tradução. A etapa de síntese é responsável pela geração de código C que representa as instruções que devem ser executadas para cada bloco básico que constitui o fluxo. Além de gerar código para a execução das instruções é necessário gerar código para manter a execução, como os saltos para instruções de fluxo de controle.

Em específico, é necessário gerar o preâmbulo que é executado na entrada da execução do fluxo, que é responsável por despachar a execução para o bloco básico inicial, que é realizado com base em PC. A partir deste mecanismo é possível permitir múltiplas entradas para um fluxo traduzido. Uma das contribuições deste trabalho é detectar fluxos quentes com múltiplas entradas, portanto é necessário gerar código eficiente que gerencie esta caracteristica.

Após a fusão de fluxos, o grafo de fluxo de controle é atualizado para conter apenas um nó que representa todos os nós contituintes do fluxo. No entanto, as transições entre os nós constituintes são mantidas internamente e são utilizadas durante o processo de síntese. Desta forma, para gerar o código das instruções basta iterar pelos nós que constituem o fluxo e por suas respectivas instruções que já foram decodificadas no processo de construção dos blocos básicos.

A implementação das instruções é realizada utilizando macros em linguagem C. A opção por utilizar macros ao invés de funções está no fato que, geralmente, macros permitem uma variedade de otimizações para o compilador utilizado na etapa de tradução. Além disto, a utilização de macros também exclui a necessidade de chamadas de função durante o processo de execução das instruções, que diminui o desempenho da execução significativamente (Foleiss et al., 2012).

A tradução é a etapa responsável por traduzir o código da linguagem C em código de máquina da arquitetura hospedeira. Geralmente, o custo desta etapa é considerado o maior entre todos os custos envolvidos no processo de TDB (Smith e Nair, 2005). Um dos objetivos deste trabalho é reduzir a quantidade de chamadas ao sistema de tradução, de forma que apenas fluxos vantajosos sejam traduzidos. No entanto, a tradução é necessária para substituir a execução interpretada.

Desta forma, é necessário escolher um compilador que seja rápido e que possa otimizar o código gerado com eficiência. Um dos principais pontos negativos em utilizar um compilador convencional para gerar código em um ambiente com tradução dinâmica é que o código gerado pelo compilador deve ser relocado e ligado dinamicamente ao ambiente de execução. Este processo também é um processo demorado que deve ser levado em consideração (Levine, 1999).

4.3 CONSIDERAÇÕES FINAIS

A proposta apresentada contém mecanismos e algoritmos que suportam a utlização de *profiling* contínuo para determinação de unidades de tradução quentes, denominados fluxos. Os meios apresentados aumentam o desempenho de sistemas que utilizam TDB, fazendo com que a maior parte da execução seja realizada por meio de tradução sem causar chamadas excessivas ao sistema de síntese e tradução. O tópico seguinte apresenta uma avaliação experimental de um emulador implementado como prova de conceito da proposta apresentada.

5 RESULTADOS

Um dos maiores desafios no projeto e implementação de um sistema TDB é o desempenho. O agravante é que, o tempo de execução do programa possui diversos custos adicionais tais como: descoberta de código, otimização, síntese e tradução. Desta forma, é necessário que estas tarefas sejam executadas de forma eficiente, compensando o tempo dispendido nas tarefas de apoio. Assim, é importante analisar o impacto das tarefas de TDB na execução dos programas com o objetivo de avaliar a proposta e suas estratégias.

Este tópico é dividido como segue. A Seção 5.1 apresenta as decisões de projeto e implementação de um emulador de NES utilizado como prova de conceito das estratégias apresentadas nesta proposta. A Seção 5.2 apresenta o método adotado na avaliação dos resultados. A Seção 5.3 discute os experimentos realizados para avaliar a parametrização do sistema. Por fim, a Seção 5.4 apresenta a avaliação de vários aspectos relacionados ao desempenho da execução dos programas avaliados.

5.1 PROVA DE CONCEITO

Com o objetivo de avaliar a proposta apresentada no Tópico 4 um emulador foi desenvolvido como prova de conceito. O emulador, denominado `jrynes`, emula o funcionamento do console NES da Nintendo, lançado no Japão em 1981 (Diskin, 2004). O NES utiliza uma CPU CISC baseada na arquitetura 6502, denominada 2A03, projetada pela MOS Technologies em meados dos anos 70 e modificada pela Ricoh, para incluir temporizadores e outros componentes que permitem o processamento de áudio.

A CPU utilizada no NES possui 56 instruções, incluindo instruções lógicas e aritméticas, transferência de dados, operações de pilha, saltos, desvios condicionais e instruções para

chamadas de procedimentos. O 2A03 possui 5 registradores de 8-*bits*, sendo 3 de própósito geral (um acumulador (A) e dois indexadores(X e Y)), um apontador de pilha (S) e um registrador de *status* (P). O contador de programa (PC) é o único registrador de 16-bits, permitindo programas de até 64KB.

Ao todo, o 2A03 possui 11 modos de endereçamento. Além dos modos mais usuais, como imediato, acumulador, implícito, absoluto, relativo e indireto, existe um modo de endereçamento denominado página-zero. Este modo de endereçamento permite a execução eficiente de instruções cujos operandos estão na primeira página de RAM, entre os endereços $0000 e $00FF. Não somente estas instruções operam mais rapidamente, mas também são codificadas de forma a reduzir seu tamanho.

Embora o espaço de endereçamento do 2A03 seja de 16-bits, o NES não possuía 64KB de memória física. Na realidade, o NES possuía apenas 2KB de memória física disponível para processamento. Os endereços $0000 até $07FF correspondem à memória física do NES, sendo esta espelhada três vezes até o endereço $2000. O espelhamento de memória indica que, por exemplo, os endereços $0000 e $0800 correspondem ao mesmo *byte* na memória principal. A memória física é dividida em três faixas lógicas: entre $0000 e $00FF está a página zero, já citada anteriormente. Entre $0100 e $01FF é a região da pilha. Como o registrador de pilha é um registrador de 8-bits, todas as operações utilizam o registrador de pilha como um deslocamento do endereço $0100. A terceira região da memória física, denominada PRG-RAM, reside entre $0200 e $7FF e é utilizada para manter valores de propósito geral (Fayzullin, 2005).

O programa pode ser acessado na faixa de endereços $8000 e $7FFF, denominada PRG-ROM, contando com apenas 32KB disponíveis. É importante notar que o programa é armazenado em ROM, portanto não ocupa espaço em memória. Desta forma, não é necessário preocupar-se com código auto-mutável durante o processo de TDB. Além disso, o tamanho relativamente pequeno do programa permite que estruturas de dados com acesso direto possam ser utilizadas para melhorar o desempenho do emulador. Por exemplo, determinar se existe tradução disponível para determinado endereço do programa emulado pode ser realizado em tempo constante.

Nem todos os jogos de NES utilizam apenas 32KB de ROM para seus programas. Durante o tempo de vida do NES, que foi entre 1981 e 1995, foram desenvolvidos dispositivos em *hardware*, denominados *Memory Mappers*, que eram acoplados aos cartuchos que permitiam a troca do conteúdo da área de ROM durante a execução dos programas. Na realidade, tais dispositivos apenas multiplexavam diferentes bancos de ROM presentes nos cartuchos para bancos pré-definidos no espaço de endereçamento da memória principal

do NES, sempre na faixa $8000 - $7FFF, podendo esta ser dividida em até 16 bancos independentes.

O restante do espaço de endereçamento do NES é utilizado para acessar periféricos, como a bateria para salvar os jogos (SRAM, $6000 - $7FFFF) e registradores de E/S, mapeados entre $2000 - $2007 para a PPU e entre $4000 - $401F para a APU (*Audio Processing Unit*) e dispositivos de entrada. A PPU, *Picture Processing Unit* (Taylor, 2004), é o processador gráfico do NES, capaz de gerar saída NTSC ou PAL em 60Hz e 50Hz, com resolução de 320x200 e 13 cores simultâneas. Por meio dos registradores mapeados em memória é possível configurar e controlar o funcionamento da PPU, bem como transferir dados entre a memória principal do NES e a memória interna da PPU, de 16KB. Embora simples, a PPU do NES era aclamada na época, por rolagem da tela por *hardware* (*scrolling*), mudança dinâmica de paleta de cores e DMA entre partes da memória da CPU e da PPU.

No emulador `jrynes` foi implementado um conjunto de componentes que permitem a execução de todos os jogos que utilizam o *Mapper 0*, juntamente com alguns que utilizam o *Mapper 3*. Tais componentes são:

- A CPU 2A03, incluindo execução de todas as instruções e tratamento de interrupções;
- A PPU NTSC 2C02, incluindo a renderização em tela em 60Hz com limitação de quadros opcional;
- Controles, que podem ser acionados via teclado;
- Carregador de cartucho;
- Memória Principal do NES, com espelhamento baseado em máscaras de bits;
- Memória da PPU; e
- Suporte para *Mapper* 0 e 3

O conjunto de componentes apresentados permite a execução dos testes necessários para a avaliação da proposta de TDB apresentada neste trabalho. Em relação aos motores de emulação da CPU, foco deste trabalho, foram implementados três modos de execução:

1. Interpretador;
2. Tradução Dinâmica de Binários com *Profiling* Interpretativo; e

3. Tradução Dinâmica de Binários com *Profiling* Contínuo.

O interpretador foi implementado utilizando intepretação encadeada indireta (*Indirect Threaded Interpretation*)(Dewar, 1975). Basicamente, o código de operação das instruções indexa uma tabela contendo o endereço da rotina que interpreta determinada instrução. Além disto, parte do código de desvio é inserido no final de cada rotina de interpretação, que acessa esta tabela para saltar diretamente para a próxima instrução ser executada, sem a necessidade de retornar ao laço do emulador.

A tradução dinâmica de binários com *profiling* interpretativo e com *profiling* contínuo executa o código do programa emulado de forma híbrida: tanto por meio de interpretação quanto por meio de fluxos de execução traduzidos. A principal diferença entre ambos modos de execução é a abordagem utilizada para realizar profiling.

Na implementação do emulador `jrynes`, a interpretação acontece apenas uma instrução por vez, retornando ao laço de emulação ao final da execução de cada instrução. Caso a próxima instrução faça parte de um fluxo traduzido, o código correspondente é chamado. No entanto, durante a interpretação, uma rotina de instrumentação é executada no início de cada instrução, permitindo o reconhecimento das unidades de tradução básicas, os blocos básicos. Além disto, tanto em *profiling* interpretativo quanto em *profiling* contínuo, as transições entre unidades de tradução são registradas, construindo o gráfico de fluxo de controle de maneira incremental. A diferença entre ambos modos de execução está no fato que quaisquer transições que contenham pelo menos um fluxo traduzido não são registradas durante o *profiling* interpretativo.

O principal objetivo da análise de resultados apresentada neste Tópico é determinar, por meio da experimentação utilizando o emulador `jrynes`, a diferença no desempenho entre a proposta que utiliza *profiling* contínuo apresentada no Tópico 4, a proposta de *profiling* interpretativo sugerido por Jones (Jones, 2010), e a interpretação pura. A próxima Seção apresenta o método adotada na realização dos experimentos.

5.2 MÉTODO

Para avaliar a prova de conceito e consequentemente a proposta apresentada, foram executados experimentos para determinar se os objetivos propostos foram cumpridos satisfatoriamente. Em específico, os seguintes fatores foram avaliados:

- Parametrização do Tamanho da Época;
- Desempenho;

1. Tempo de execução em relação à abordagem com *profiling* interpretativo;
2. Proporção do código traduzido na execução;
3. Invocações ao tradutor dinâmico;
4. Fusão entre fluxos de execução;
5. Fragmentação de unidades de tradução; e
6. Custo do *profiling* contínuo.

Embora relacionados, Parametrização e Desempenho são avaliados separadamente, pois representam desafios de projeto e implementação distintos e complementares. De fato, a avaliação da parametrização utiliza métricas de desempenho, enquanto a parametrização afeta diretamente o desempenho. Assim, é necessário determinar a parametrização adequada para que a análise de desempenho possa ser realizada satisfatoriamente.

Nesta avaliação foram utilizados 17 programas-teste, os quais são jogos de NES. Estes programas foram produzidos na década de 80 e foram escritos manualmente em linguagem de montagem do 6502. Cada jogo foi executado por aproximadamente 5 minutos sendo cada execução gravada em macros, possibilitando a repetibilidade das jogadas em todas as execuções posteriores, garantindo a comparação justa entre diferentes parametrizações do emulador. A Tabela 5.1 - apresenta os programas e o perfil básico da execução realizada para cada um deles.

Para que seja possível acompanhar o jogo, um limitador de quadros é utilizado para manter o jogo executando a 60 quadros por segundo. Isto permite que o jogador possa interagir com o jogo sem que ele execute rápido demais. No entanto, para avaliar o ganho de desempenho com as técnicas apresentadas neste trabalho, todos experimentos foram executados com o limitador de quadros desabilitado, utilizando macros para estabelecer o fluxo de controle do programa.

Todos os experimentos foram realizados utilizando a CPU Intel Core i5 como alvo, um sistema com 6GB de RAM e Kernel Linux 3.2. Para coletar os tempos de execução das principais tarefas do sistema foi utilizada instrumentação com a biblioteca PAPI, que proporciona primitivas para medição precisa de tempo na ordem de nanosegundos. O compilador C responsável pela tradução do código gerado em tempo de execução pelo sintetizador é o GCC versão 4.6.1. Tanto o código do emulador quanto o código traduzido em tempo de execução foi compilado sem otimizações habilitadas. Embora este fator aumente consideravelmente o tempo de execução, não foi possível compilar com otimizações habilitadas em função de erros espúrios do próprio GCC.

Tabela 5.1: Perfil dos Experimentos Realizados

Código	Programa	Instruções	Blocos Básicos	Instruções Distintas
1942	1942	174299787	1857	5440
AA	Antarctic Adventure	193676387	1274	3859
AKN	Arkanoid	215928381	1406	5358
BF	Balloon Fight	206953137	1436	4611
BLT	Baltron	176959836	1408	5335
BTC	Battle City	207019298	1165	4013
BMB	Bomberman	194558454	846	2804
DDG	Dig Dug	185730925	1015	4536
DK	Donkey Kong	153145393	1888	5393
GLX	Galaxian	147502027	802	3007
ICE	Ice Climber	178278551	2015	6171
LRU	Lode Runner	210801892	1701	3641
MB	Mario Bros.	160082475	1663	5080
MLP	Millipede	167544959	1536	5187
PAC	Pacman	184757706	870	3678
PEY	Popeye	172636605	1648	5024
SMB	Super Mario Bros.	184452662	2532	8007

Em especial, jogos apresentam desafios interessantes para tradução dinâmica de binários. Primeiramente, existem várias regiões distintas de código que são frequentemente executadas em um curto espaço de tempo. Além disso, a sequência em que estas regiões são executadas nem sempre é previsível, pois depende inteiramente da interação com o jogador, que acontece em tempo real. Frente a isto, é importante que o sistema seja capaz de acompanhar as mudanças no padrão de execução de forma dinâmica e autonômica, sempre que possível. Portanto, antes de avaliar o desempenho das aplicações e das estratégias propostas, a próxima Seção apresenta considerações sobre a parametrização adotada nos experimentos.

5.3 PARAMETRIZAÇÃO

A proposta apresentada no Tópico 4 tem como principal objetivo fornecer estratégias para suportar a execução eficiente de programas que necessitam de tradução dinâmica de binários. Devido ao fato que programas distintos apresentam características diferentes, tais estratégias foram concebidas de maneira que seu comportamento possa ser alterado dinamicamente durante a execução do programa. Portanto, a parametrização do sistema

é, em grande parte, dinâmica e sensível ao contexto da execução. Esta característica está presente, por exemplo, no monitoramento das transições para a determinação do LCN.

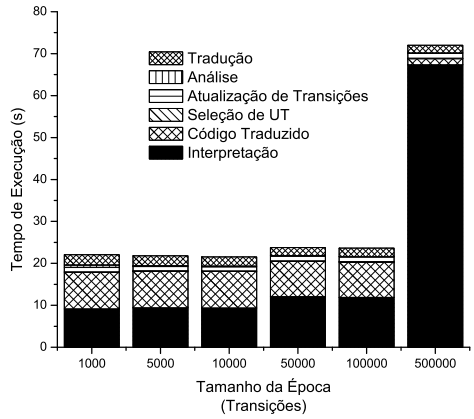
No entanto, a proposta apresentada ainda apresenta um parâmetro obrigatório que deve ser estabelecido antes da execução: o tamanho da época. Como descrito anteriormente, este parâmetro determina a quantidade de transições entre unidades de tradução entre épocas. Figura 5.1 - , Figura 5.2 - e Figura 5.3 - apresentam o perfil do tempo de execução dos programas avaliados, para diferentes quantidades de transições entre unidades de tradução e limiar de tradução em 10%.

Na maioria dos programas não há grande diferença no tempo de execução com a variação do tamanho das épocas. A Tabela 5.2 - resume os tempos de execução dos programas, juntamente com o desvio padrão entre as execuções com épocas de diversos tamanhos. Entre as aplicações com pouca variação, o desvio padrão médio é de apenas 2,27 segundos. Somente quatro aplicações apresentam grande variação: AA, DK, ICE e SMB. Dentre elas, somente SMB possui mais que uma execução que foge à média. As aplicações restantes mantém pequena variação entre uma execução e as demais.

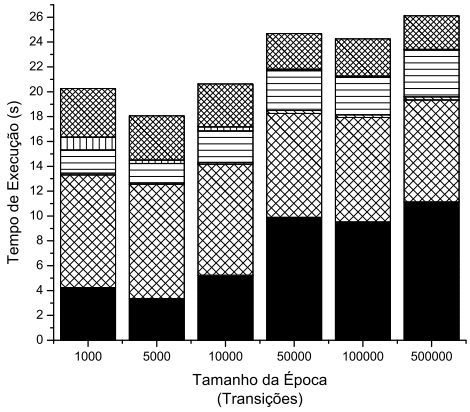
Tabela 5.2: Resumo dos Tempos de Execução

Programa	Tempo de Execução Médio (s)	Desvio Padrão (s)
1942	22,33	3,13
AA	30,80	20,21
AKN	21,19	0,91
BF	20,18	1,32
BLT	29,61	1,16
BTC	23,74	1,40
BMB	25,90	1,06
DDG	47,47	1,80
DK	23,90	7,63
GLX	17,37	1,24
ICE	27,48	16,01
LRU	21,80	2,04
MB	21,10	3,05
MLP	23,81	4,05
PAC	17,15	1,86
PEY	22,04	1,11
SMB	42,90	14,10

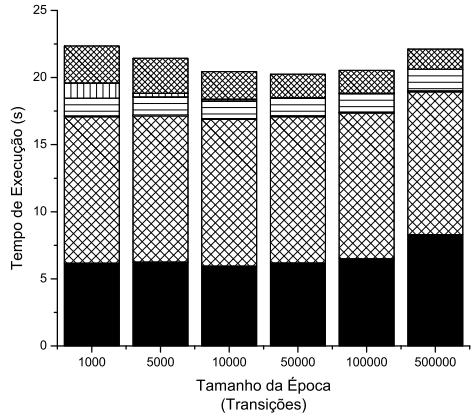
De maneira geral, quanto maior a época, menos tempo é gasto com tradução de fluxos. A Tabela 5.3 - apresenta a quantidade de ciclos encontrados e traduções realizadas em cada época. Este fenômeno está relacionado a dois fatores principais. Primeiramente, o algoritmo utilizado para detecção de ciclos HotDFS não é ótimo, ou seja, não detecta o



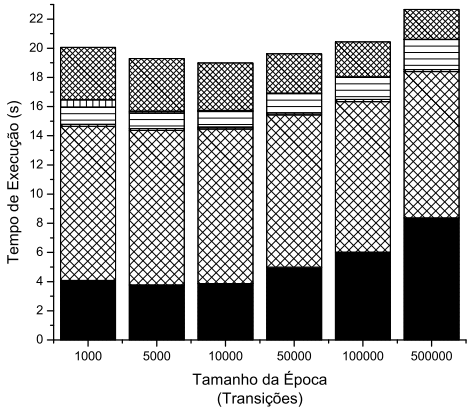
(a) AA



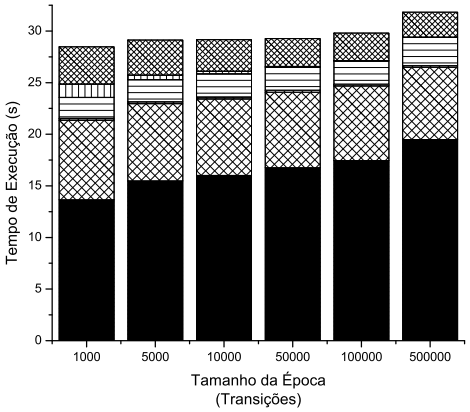
(b) 1942



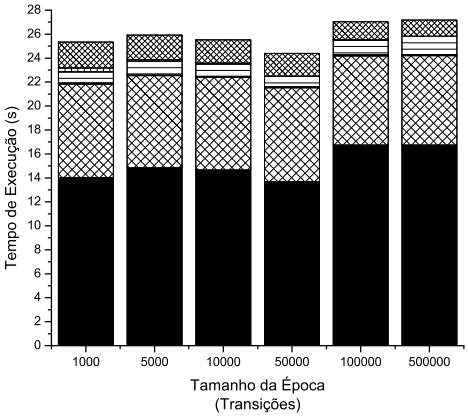
(c) AKN



(d) BF

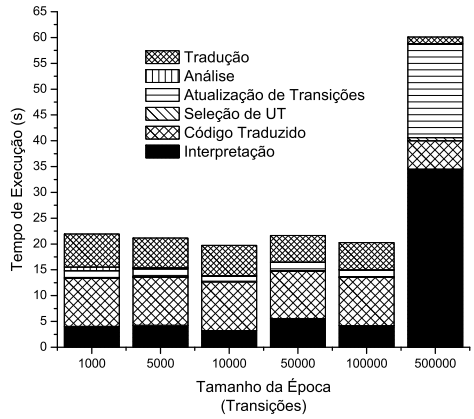


(e) BLT

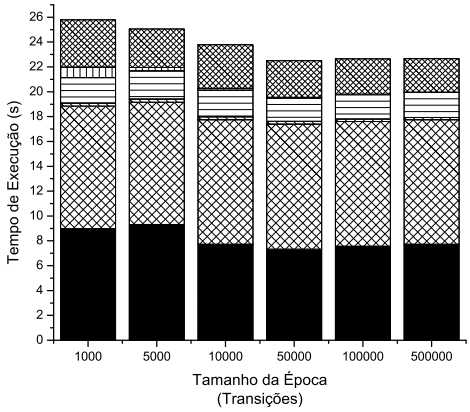


(f) BMB

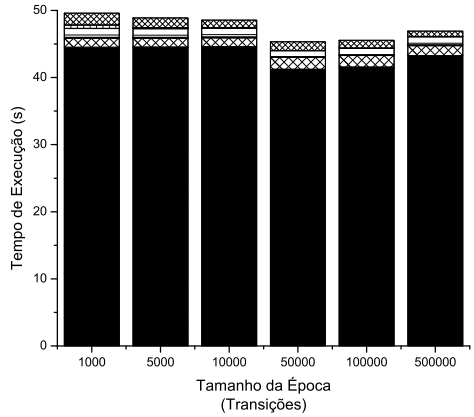
Figura 5.1: Tamanho da Época X Tempo de Execução (1)



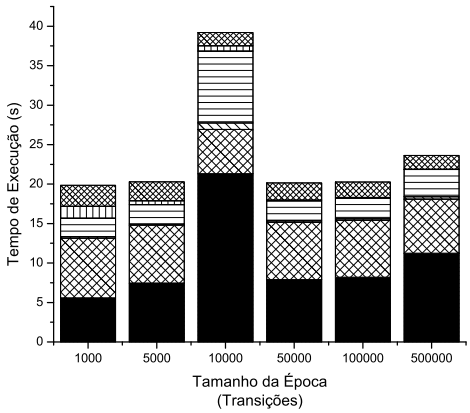
(a) ICE



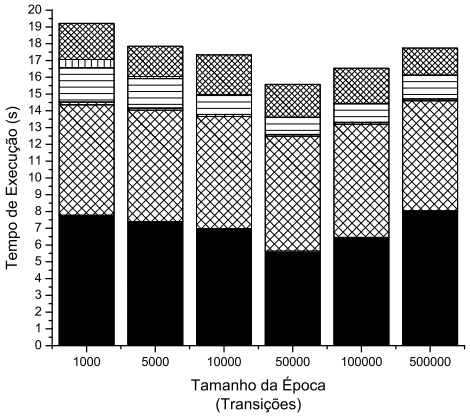
(b) BTC



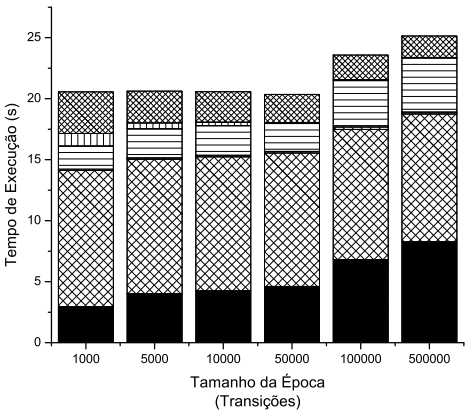
(c) DDG



(d) DK

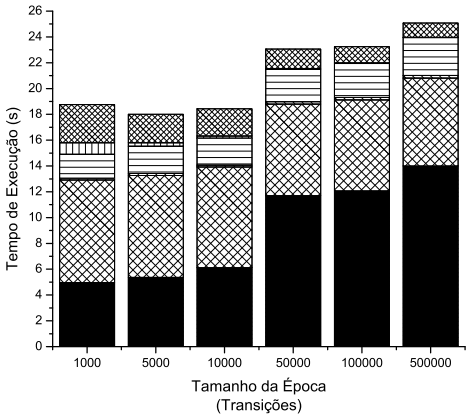


(e) GLX

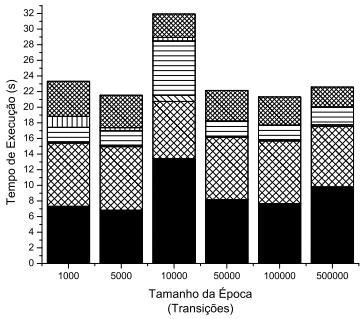


(f) LRU

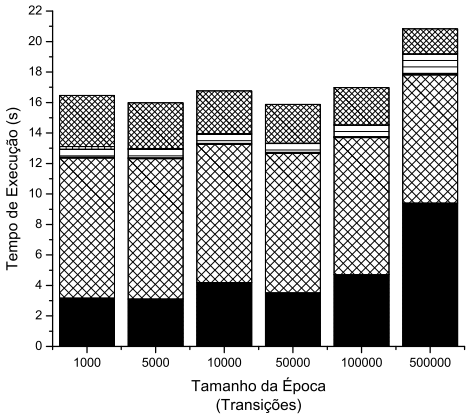
Figura 5.2: Tamanho da Época X Tempo de Execução (2)



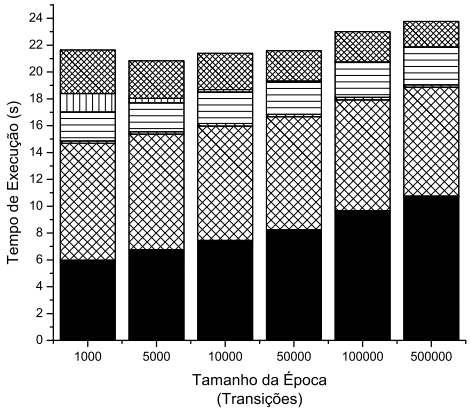
(a) MB



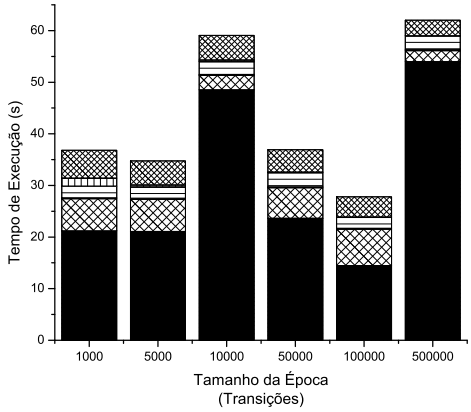
(b) MLP



(c) PAC



(d) PEY



(e) SMB

Figura 5.3: Tamanho da Época X Tempo de Execução (3)

maior ciclo possível que esteja dentro do limiar de calor do nó. A situação se agrava com épocas maiores pois o monitoramento é realizado por mais tempo, e consequentemente, uma parte maior do programa é analisada de cada vez.

Tabela 5.3: Ciclos (C) e Traduções (T) por Programa em Função do Tamanho da Época

Programa	Tamanho da Época											
	1000		5000		10000		50000		100000		500000	
	C	T	C	T	C	T	C	T	C	T	C	T
1942	10933	78	2137	71	2505	69	988	57	440	60	128	54
AA	8774	49	2692	47	1294	44	280	38	146	41	56	37
AKN	5462	55	2718	52	1086	41	125	35	78	34	40	30
BF	7433	72	1505	72	863	65	207	54	124	48	55	41
BLT	18076	72	4515	68	2413	61	877	54	230	53	72	48
BTC	17681	76	5096	62	2994	70	610	59	317	57	81	54
BMB	5835	43	1516	41	709	39	125	38	77	30	30	27
DDG	13207	35	2270	29	1211	22	223	26	132	23	42	16
DK	13767	53	3324	48	5892	33	303	43	159	39	65	35
GLX	7312	43	1380	36	911	47	159	39	96	42	52	32
ICE	12864	128	2505	115	768	117	422	103	181	105	188	26
LRU	9386	68	3118	52	1804	50	376	46	349	40	105	35
MB	8113	59	1103	44	649	42	159	30	92	25	31	22
MLP	37345	90	8965	83	5972	60	474	78	283	72	91	51
PAC	1837	67	584	60	358	56	104	51	83	49	44	33
PEY	20139	65	3451	56	1621	54	397	45	285	45	66	38
SMB	13994	108	2110	93	2008	95	389	87	191	78	134	60

Como o algoritmo HotDFS é baseado em uma heurística gulosa é possível que feche o ciclo mais rapidamente, ou seja contendo menos nós, do que um algoritmo que considere outros caminhos entre as unidades de tradução. Desta forma, a busca termina antes de encontrar ciclos que abrangem mais unidades de tradução, sem detectar todos os fluxos quentes da época. Como este fenômeno acontece em todas as épocas, a fusão de fluxos acontece de maneira mais lenta, fazendo com que haja maior tempo de execução dispendido em interpretação. Os programas AA e ICE sofrem deste fenômeno de maneira mais expressiva: seu desempenho para a época com 500000 transições é drasticamente afetado em função da redução acentuada da quantidade de ciclos detectados, conforme mostram as Figuras Figura 5.1 - e Figura 5.2 - .

Em épocas mais curtas este problema é amenizado pois a fusão entre ciclos adjacentes acontece mais rapidamente. Desta forma, o efeito do algoritmo ótimo de detecção de ciclos acontece gradativamente, poupando o custo de sua execução. De fato, um dos objetivos da proposta é apresentar algoritmos e estratégias que sejam eficazes, ao mesmo tempo

que minimizem ao máximo o processamento de tarefas auxiliares, deixando mais tempo da CPU livre para o processamento dos programas.

Além de menor tempo gasto em tarefas de tradução, épocas mais longas também reduzem o tempo de análise dos grafos de fluxo de controle. O principal fator que contribui para isto é que a análise do grafo é realizada ao final de cada época. Portanto, quanto mais transições são necessárias para determinar o final de uma época, menos épocas são finalizadas. Desta forma, os algoritmos de análise são executados menos vezes, gastando assim, menos tempo de execução. A Tabela 5.4 - apresenta a quantidade de épocas executadas para cada programa.

Tabela 5.4: Épocas Executadas por Programa

Programa	Tamanho da Época					
	1000	5000	10000	50000	100000	500000
1942	3883	696	637	184	87	22
AA	3167	648	331	81	38	9
AKN	4607	934	452	92	49	12
BF	3351	580	302	79	49	14
BLT	6017	1379	734	150	75	20
BTC	6406	1337	574	110	57	13
BMB	2586	682	318	51	41	11
DDG	4472	889	452	54	30	9
DK	6300	1427	2985	151	79	22
GLX	5786	1056	330	59	35	10
ICE	3819	794	283	102	38	130
LRU	4291	1166	617	125	110	27
MB	4384	1009	539	135	73	17
MLP	6236	1214	2374	126	62	16
PAC	1582	344	196	36	25	10
PEY	6511	1352	691	153	83	19
SMB	6112	1165	686	108	45	17

A relação entre a quantidade de épocas e o tempo de análise está presente em todos os casos analisados. Embora o grafo de fluxo de controle de épocas mais longas tenda a ser maior, por potencialmente monitorar uma região maior do código do programa, isto não influencia tanto o tempo de execução dos algoritmos de análise. Isto se deve ao fato que foram desenvolvidos algoritmos que possuem baixa complexidade assintótica, permanecendo eficientes mesmo com entradas maiores.

Por outro lado, por definição, quanto maior a época, mais transições acontecem. Como é necessário manter o registro das transições, o tempo gasto com a atualização da contagem das transições entre unidades de tradução aumenta conforme o tamanho da

época. Nota-se também que este é o maior custo relacionado ao *profiling* contínuo que é consequencia direta do tamanho da época, sendo apenas ultrapassado pelo custo de *profiling* das instruções interpretadas, que é discutido mais adiante.

A maioria dos programas avaliados apresenta o efeito da escada ascendente. Este efeito pode ser observado nas Figuras Figura 5.1 - , Figura 5.2 - e Figura 5.3 - , onde o tempo utilizado na execução de código interpretado é diretamente proporcional ao tamanho da época. Este efeito é consequência direta do fato que, para épocas maiores, menos traduções são realizadas. Além disto, como comentado anteriormente, tais traduções representam fluxos sub-ótimos em função do algoritmo de descoberta de ciclos empregado.

Outro efeito que contribui para o efeito da escada ascendente é que as fusões de fluxo que são realizadas mais comumente quando a época é pequena beneficiam a localidade temporal, aumentando a quantidade de código traduzido sendo executado. Mesmo quando o fluxo de controle é alterado, o sistema com épocas menores é capaz de reagir mais rapidamente, gerando código para as novas regiões sendo executadas mais brevemente. Assim, para valores de épocas menores, a execução tende a concentrar-se mais em código traduzido.

Embora a maior parte dos casos analisados apontarem que a variação de desempenho em função do tamanho da época é pequena, existem alguns casos onde o desempenho do sistema pode ser prejudicado. De maneira geral, nota-se que em épocas menores o tempo de execução dispendido em interpretação é menor que em épocas maiores. Embora o tempo de análise seja maior em épocas menores, é compensado pelo favorecimento à formação de fluxos de execução que levam à acentuação na execução de código traduzido, que é mais eficiente do que a interpretação.

5.4 DESEMPENHO

O objetivo principal da proposta apresentada é oferecer um sistema TDB que permita a execução eficiente de programas utilizando *profiling* contínuo para determinação de unidades de tradução quentes. Em contraste à estratégias de *profiling* parcial, como apresentadas em trabalhos anteriores (Bala et al., 2011; Jones, 2010), a proposta apresentada sugere o monitoramento integral da execução, não somente durante partes da execução. A Figura 5.4 - mostra os tempos de execução dos programas com *profiling* contínuo e *profiling* interpretativo, com tamanho de época 5000 e limiar de tradução em 1,10.

O tempo de execução dos programas executados com *profiling* contínuo foi, em média, 2,33 vezes mais rápido que os programas executados com *profiling* interpretativo. Nota-se que a quantidade de tempo gasto na interpretação de instruções e seleção de unidades de

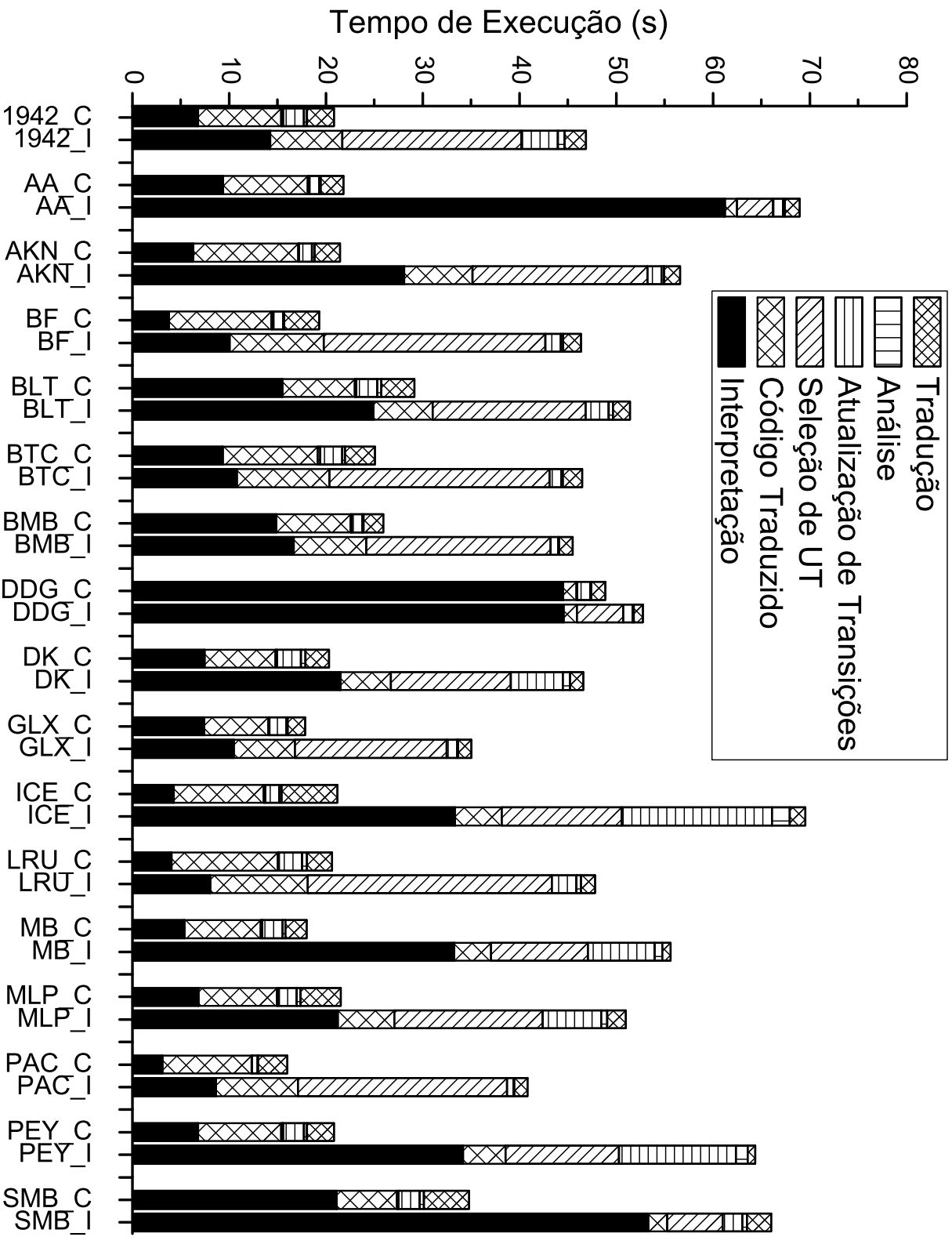


Figura 5.4: Tempos de Execução com *Profiling* Contínuo (_C) e Interpretativo(_I)

tradução são as tarefas que ocupam a maior parte do tempo de execução com *profiling* interpretativo, sendo estes, em média, 48,71% e 28,68% do tempo total de execução, respectivamente.

Um dos principais objetivos de TDB é oferecer execução eficiente de programas em relação à execução puramente interpretada. Desta forma, o maior ganho no tempo de execução se dá devido à tradução de partes do código de máquina da arquitetura alvo em código de máquina da arquitetura hospedeira, de maneira que o sistema de emulação tenha que ser invocado apenas quando há necessidade. A partir da Figura 5.4 - e da Tabela 5.5 - nota-se que a proposta de *profiling* contínuo obtém vantagem em relação à abordagem com *profiling* interpretativo, executando uma proporção maior das instruções por meio de código traduzido.

Tabela 5.5: Proporção de código executado por meio de tradução

Programa	Execução em Código Traduzido (em %)	
	P. Interpretativo	P. Contínuo
1942	80,63	95,62
AA	14,28	88,17
AKN	66,79	93,30
BF	88,55	95,75
BLT	66,21	79,45
BTC	87,62	89,40
BMB	77,95	80,55
DDG	19,65	19,59
DK	65,88	88,76
GLX	82,88	87,78
ICE	53,72	94,56
LRU	90,96	95,63
MB	47,65	92,17
MLP	68,75	90,53
PAC	88,74	95,94
PEY	50,66	91,05
SMB	22,63	70,34

Nota-se que em todas as aplicações houve ganho significativo, exceto em DDG, cuja proporção foi praticamente a mesma. Em média, com exceção de DDG, houve aumento de 67,92%, com desvio padrão médio de 27,37%. É importante lembrar que, como a execução de instruções traduzidas é mais rápida que a execução de instruções interpretadas, qualquer ganho na proporção de instruções executadas por meio de tradução ocasiona uma redução no tempo de execução total.

O fator que contribui para que o *profiling* contínuo obtenha melhor desempenho em relação ao *profiling* interpretativo é a fusão de fluxos, que acontece apenas no *profiling* contínuo. Como descrito anteriormente, a fusão de fluxos proporciona a detecção de unidades de tradução maiores e relacionadas por localidade temporal. Desta forma, uma vez que a execução seja iniciada em qualquer ponto de um fluxo traduzido, ela tende a permanecer lá até que um caminho não-traduzido seja tomado.

O diferencial entre as duas abordagens está no fato que, com *profiling* contínuo, caminhos relacionados a determinado fluxo que não eram previamente tomados com frequência podem ser fundidos aos caminhos frequentes já determinados. Desta forma, o código pode continuar sendo executado por mais tempo sem que seja necessário retornar ao laço do emulador para que outro fluxo seja selecionado. De fato, uma das principais contribuições deste trabalho é apresentar mecanismos de análise e *profiling* contínuo que permitam o reconhecimento de fluxos de execução quente de maneira incremental. Embora a abordagem incremental já tenha sido abordada anteriormente em outros trabalhos (Ebcioglu et al., 2001), os mecanismos apresentados neste trabalho utilizam algoritmos originais, que permitem múltiplas entradas e saídas a partir dos fluxos traduzidos resultantes. Além disto, a estratégia de realizar *profiling* apenas no laço do emulador também é uma contribuição que permite que a abordagem incremental tenha custo reduzido, sem a necessidade de instrumentação no código gerado.

Além disto, a fusão de blocos também contribui para que a execução possa se recuperar mais rapidamente de mudanças bruscas no fluxo de controle do programa. Tais mudanças, comumente encontradas em aplicações interativas, devem ser acompanhadas para que o desempenho da aplicação não sofra grandes alterações. Outra possibilidade que a fusão de blocos permite é a implementação de otimizações, principalmente relacionadas ao controle de fluxo da execução. Por exemplo, é possível implementar saltos incondicionais com endereçamento direto e indireto que saltem para dentro do próprio fluxo, sem ter que retornar o controle da execução ao laço do emulador.

O efeito colateral da fusão de blocos é a necessidade de retraduzir os fluxos toda vez que fazem parte de uma nova fusão. Isto é necessário pois o código gerado deve ser atualizado para que possa contemplar os novos caminhos determinados durante a análise do fluxo de controle. No entanto, este problema pode ser amenizado utilizando o mecanismo de limiar de tradução proposto no Tópico 4. A Tabela 5.6 - apresenta a proporção de traduções em relação à quantidade de ciclos detectados para os programas investigados, assim como o tempo gasto em compilação. A quantidade total de traduções e ciclos é apresentada na Tabela 5.3 - .

Tabela 5.6: Perfil da Invocação do Tradutor

Programa	*Profiling* Contínuo		*Profiling* Interpretativo	
	% Traduzidos	Tempo (s)	% Traduzidos	Tempo (s)
1942	3,32	3,62	100,00	2,18
AA	1,75	2,42	100,00	1,57
AKN	1,91	2,64	100,00	1,62
BF	4,78	3,72	100,00	1,79
BLT	1,51	4,43	100,00	1,73
BTC	1,22	3,25	100,00	1,96
BMB	2,70	2,13	100,00	1,52
DDG	1,28	1,49	100,00	1,05
DK	1,44	2,39	100,00	1,45
GLX	2,61	1,78	100,00	1,39
ICE	4,59	5,89	100,00	1,57
LRU	1,67	2,71	100,00	1,49
MB	3,99	2,45	100,00	0,93
MLP	0,93	4,27	100,00	1,98
PAC	10,27	3,07	100,00	1,52
PEY	1,62	2,89	100,00	0,92
SMB	4,41	4,71	100,00	2,63

Portanto, nota-se que, embora muitos fluxos tenham sido descartados, as execuções com *profiling* contínuo obtiveram proporções satisfatórias em termos de código executado por meio de tradução. Isto se deve ao fato que o limiar de tradução, conforme descrito na Seção 4.2.2, impede que fluxos de execução menos quentes incluam fluxos quentes. Desta forma, somente traduções que levam ao aumento da execução de instruções traduzidas recorrentes são executadas.

Embora a Tabela 5.6 - mostre que a abordagem com *profiling* contínuo tenha gasto, em média, 2,01 vezes mais tempo que a abordagem com *profiling* interpretativo, os fluxos traduzidos resultantes compensaram diminuindo significantemente o tempo de execução dos programas, principalmente reduzindo o tempo gasto com interpretação, como mostrado anteriormente na Figura 5.4 - .

Outro ponto interessante a ressaltar é que grande parte das traduções ocorre em função das fusões com pelo menos um fluxo traduzido. A Tabela 5.7 - mostra a quantidade de fusões realizadas durante a execução dos experimentos. Lembrando que o termo fusão remete à fusão entre quaisquer unidades de tradução, como por exemplo, entre somente blocos básicos, entre blocos básicos e fluxos traduzidos e apenas entre fluxos traduzidos. A terceira coluna da tabela representa a fusão entre unidades de tradução com pelo menos um fluxo traduzido.

Tabela 5.7: Perfil das Fusões

Programa	Fusões	Fusões entre Fluxos
1942	79	27
AA	47	21
AKN	52	19
BF	72	37
BLT	68	33
BTC	62	24
BMB	41	16
DDG	29	12
DK	48	16
GLX	36	10
ICE	115	44
LRU	52	25
MB	44	25
MLP	83	39
PAC	60	35
PEY	56	32
SMB	93	51

Devido ao fato do código fonte em alto nível não estar disponível para os programas avaliados, não é possível mostrar a detecção de estruturas de fluxo de controle complexas a partir da fusão de fluxos. No entanto, é possível inferir que esta detecção é possível. Por exemplo, em um programa com estruturas de repetição aninhadas, a estrutura interna é reconhecida primeiro, formando um fluxo traduzido, que substitui os nós constituintes no grafo de fluxo de controle. Subsequentemente, as transições entre o fluxo traduzido e o laço que o contém são determinadas, fazendo com que a estrutura de repetição interna seja um dos nós constituintes do laço externo. Desta forma, quando a época termina e a quantidade de transições entre os laços torna-se satisfatória, o laço externo, que contém o laço interno como um de seus nós constituintes, é fundido em um único fluxo. Assim, a estrutura toda é traduzida em um único fluxo, permitindo que a execução permaneça por mais tempo dentro de uma região de código traduzida.

Outro objetivo da proposta apresentada é diminuir a fragmentação das unidades de tradução. Com efeito, a fragmentação das unidades de tradução é consequência direta da abordagem de *profiling* interpretativo: como o grafo de fluxo de controle não contempla transições entre fluxos traduzidos e as demais unidades de tradução, este torna-se fragmentado. Como esperado, o uso de *profiling* contínuo permite manter a fragmentação das unidades de tradução em níveis mínimos. A Tabela 5.8 - mostra a

quantidade média de fragmentos gerados por época em ambas abordagens de *profiling* utilizadas.

Tabela 5.8: Fragmentação

Programa	Fragmentos por Época	
	Profiling Contínuo	*Profiling* Interpretativo
1942	1,04	102,04
AA	3,99	81,75
AKN	2,01	51,21
BF	1,20	63,15
BLT	1,02	79,80
BTC	4,73	101,67
BMB	1,07	63,97
DDG	1,00	43,66
DK	1,64	41,43
GLX	1,00	63,06
ICE	2,21	31,14
LRU	1,03	74,39
MB	1,01	41,06
MLP	4,22	58,13
PAC	2,15	58,17
PEY	1,01	46,59
SMB	1,17	96,33

Grafos de fluxo de controle fragmentados são grafos desconexos onde cada componente corresponde a um subgrafo conexo entre unidades de tradução. Estes componentes são os fragmentos. Utilizando *profiling* contínuo, a fragmentação é minimizada pois as transições entre os fluxos traduzidos e as demais unidades de tradução são mantidas e monitoradas. Desta forma, o programa só apresenta fragmentos na ocorrência de interrupções. Por outro lado, *profiling* interpretativo apresenta alto grau de fragmentação, permitindo apenas fusões entre blocos básicos. Além de prejudicar o potencial de fusão entre fluxos traduzidos, a fragmentação também leva ao aumento do tempo de execução dispendido nas tarefas de seleção de unidades de tradução e atualização das transições, como observado na Figura 5.4 - .

Portanto, a fragmentação não somente aumenta significativamente a proporção de instruções executadas por interpretação, mas também aumenta o tempo de tarefas auxiliares ao processo de TDB, como as tarefas executadas no laço de emulação, já que este é invocado com maior frequência. Tais efeitos prejudicam o desempenho das aplicações de maneira significativa, como já demonstrado anteriormente.

Desta forma, o aumento da proporção de código executado por tradução não somente impacta na proporção de tempo gasto na execução por interpretação. A detecção de fluxos relacionados por localidade temporal também contribui com a execução eficiente, pois reduz outros custos relacionados mostrados na Figura 5.4 - , como por exemplo, a atualização de transições, seleção de unidades de tradução a serem executadas e análise excessiva do fluxo de controle.

A taxa significativa de fusões entre fluxos traduzidos mostra que a estratégia de análise adotada, juntamente com *profiling* contínuo e os resultados obtidos é adequada para manter o programa executando a grande maioria de seu tempo em código traduzido quando comparado à abordagem com *profiling* interpretativo. Além disso, a taxa de fusão também demonstra a capacidade de adaptação das unidades de tradução em função das alterações no fluxo de controle do programa sem causar compilação excessiva: um dos principais custos a serem evitados no contexto de tradução dinâmica.

Embora o foco deste trabalho seja a descoberta fluxos de execução quentes em tempo de execução, também houve esforço para desenvolver um tradutor eficiente. Embora foram tenham sido utilizadas apenas técnicas tradicionais na geração de código, como descrito na Seção 4.2.3, o ganho de desempenho em relação à interpretação pura foi relevante. A Tabela 5.9 - mostra os tempos de execução utilizando a proposta apresentada e de execuções puramente interpretadas.

Tabela 5.9: Tempo de Execução com TDB e Interpretação Pura

Programa	TDB Com P. Contínuo (s)	Interpretação Pura (s)	Ganho
1942	18.06	134.93	7.47
AA	21.79	144.52	6.63
AKN	21.43	167.14	7.80
BF	19.29	157.39	8.16
BLT	29.13	133.52	4.58
BTC	25.05	157.91	6.30
BMB	25.92	142.40	5.49
DDG	48.88	100.44	2.05
DK	20.30	118.43	5.83
GLX	17.85	110.58	6.19
ICE	21.16	138.76	6.56
LRU	20.62	163.18	7.91
MB	18.01	123.80	6.87
MLP	21.54	130.14	6.04
PAC	15.99	138.27	8.65
PEY	20.84	137.00	6.57
SMB	34.77	131.29	3.78

Nota-se que, em média, a execução com TDB foi 6,29 vezes mais rápida que a interpretação pura. Dentre diversos fatores, este ganho significativo é possível principalmente pela utilização de unidades de tradução longas como fluxos traduzidos. Como apontam trabalhos anteriores (Hong et al., 2012; Jones, 2010; Ottoni et al., 2011; Smith e Nair, 2005), o uso de unidades de tradução longas permite que mais otimizações em nível de geração de código final sejam aplicadas pelo tradutor dinâmico. Além disto, tais unidades permitem que o código seja executado por mais tempo sem ter a intervenção do emulador constantemente. Outro motivo apontado por estes trabalhos está relacionado ao fato que o código traduzido necessita de menos instruções para emular as instruções da arquitetura original do que a interpretação.

Com objetivo de reduzir os custos do *profiling* contínuo, foram propostas estratégias complementares que permitem a detecção de código quente a partir da análise das transições entre unidades de tradução, sendo estas blocos básicos interpretados ou fluxos de execução traduzidos. Para este fim, o monitoramento da execução é feito exclusivamente no laço de execução do emulador e durante a interpretação de instruções, deixando que os fluxos traduzidos sejam executados sem a necessidade de instrumentação. Embora esta abordagem permita a diminuição do custo necessário para utilizar *profiling* contínuo, existe a necessidade de executar o código do *profiling* em alguns pontos do programa.

Em particular, a instrumentação inserida na interpretação das instruções é a responsável pelo maior custo relacionado à *profiling* no sistema. A Tabela 5.10 - mostra o custo do *profiling* diante do tempo total de interpretação para todos os programas. Nota-se que, em média, 45,39% do tempo gasto com interpretação é utilizado em *profiling*. No entanto, este custo é necessário, pois está intimamente ligado ao processo de detecção de blocos básicos. O custo do *profiling* para *profiling* interpretativo é praticamente o mesmo, já que a detecção dos blocos básicos é feita exatamente da mesma forma. Portanto, é desnecessário apresentar estes dados na tabela.

Conforme a execução acontece, o custo da interpretação diminui. Isto se deve ao fato que a maior parte do custo associado ao *profiling* da interpretação está ligado à descoberta de blocos básicos. Conforme a execução do programa progride, somente as atividades menos onerosas passam a fazer parte da interpretação, como, por exemplo, a contagem das execuções dos blocos.

No entanto, execução dos trechos instrumentados só acontece durante a interpretação. Desta forma, conforme o programa vai sendo executado, na maior parte do tempo por meio de traduções, o custo do *profiling* contínuo tende a estar relacionado principalmente à outras tarefas. Estas tarefas incluem principalmente: análise de fluxo de controle, seleção de unidades de tradução e atualização de transições.

Tabela 5.10: Custo do *Profiling* na Interpretação

Programa	% da Interpretação em *Profiling*
1942	46,25
AA	44,77
AKN	45,75
BF	46,18
BLT	46,30
BTC	45,60
BMB	43,51
DDG	45,04
DK	45,79
GLX	44,63
ICE	46,53
LRU	46,32
MB	45,27
MLP	45,17
PAC	45,40
PEY	45,13
SMB	44,05

As estratégias desenvolvidas neste trabalho, em especial a fusão de fluxo, foram desenvolvidas com o objetivo de, além de proporcionar execução adaptativa, diminuir ao máximo o custo do *profiling* no sistema. O principal recurso utilizado para este fim foi garantir que a atualização das estruturas ligadas à *profiling*, exceto a detecção de blocos básicos, só tenham que ser atualizadas no laço do emulador. Como a formação de unidades de tradução que contemplem fluxos de execução relacionados por localidade temporal fazem com que mais instruções sejam executadas antes da invocação do laço do emulador, menos transições entre unidades de tradução são realizadas em dado período de tempo, diminuindo assim, os custos relacionados à atualização de transições e seleção de unidades de tradução. A Tabela 5.11 - mosta o tempo utilizado na atualização de transições e na seleção de unidades de tradução em ambas abordagens de *profiling* utilizadas.

Nota-se que o tempo gasto com a atualização das transições varia significativamente para alguns programas, principalmente para 1942, DK, GLX, ICE, MB, MLP e PEY. Nestes programas foi gasto, em média, 4,42 vezes mais tempo com atualização de transições na abordagem interpretativa. Para os demais programas, o custo da atualização é praticamente o mesmo. Este panorama demonstra que, embora a proposta propicie a execução de mais instruções traduzidas, nem sempre é possivel reduzir o custo de atualizar

Tabela 5.11: Custos de Atualização de Transições e Seleção de UT

Programa	*Profiling* Contínuo		*Profiling* Interpretativo	
	Atualização (s)	Seleção (s)	Atualização (s)	Seleção(s)
1942	1,56	0,12	3,74	18,52
AA	1,08	0,08	1,02	3,73
AKN	1,35	0,06	1,45	18,06
BF	1,07	0,12	1,63	22,91
BLT	2,15	0,18	2,42	15,81
BTC	2,28	0,26	1,23	22,74
BMB	1,09	0,11	0,80	19,05
DDG	1,32	0,09	0,99	4,79
DK	2,45	0,17	5,38	12,42
GLX	1,76	0,13	6,33	15,68
ICE	1,46	0,16	15,55	12,36
LRU	2,38	0,12	2,57	25,21
MB	2,12	0,16	6,93	10,01
MLP	1,86	0,19	6,02	15,34
PAC	0,57	0,05	0,67	21,59
PEY	2,16	0,17	12,06	11,74
SMB	2,20	0,16	2,06	5,72

as transições em todas as situações possíveis. No entanto, o custo de atualização não é tão alto a ponto que se torne proibitivo.

Por outro lado, há grande diferença no custo da seleção de unidades de tradução a serem executadas. Neste quesito, praticamente em todos os programas houve grande diferença no custo entre a abordagem interpretativa e a contínua. Em média, a abordagem interpretativa foi 131,79 vezes mais onerosa que a abordagem apresentada. Este fator, inicialmente inesperado é resultado direto da quantidade de transições realizadas diretamente entre fluxos traduzidos. Tais transições são, em sua maioria, evitadas na abordagem proposta, uma vez que transições frequentes entre dois fluxos traduzidos são eliminados por meio de fusão de fluxos.

Como as épocas são delineadas por quantidade de transições, e não por contagem de execuções de unidades de tradução como proposto em trabalhos anteriores (Jones, 2010), o sistema recupera-se rapidamente quando o emulador passa a executar somente instruções interpretadas, ou seja, quando o fluxo de execução é alterado para regiões não executadas previamente. Desta forma, a quantidade de invocações aos algoritmos de análise de fluxo de controle é regulada automaticamente pelo fluxo de controle do programa. O tempo gasto na análise de fluxo de controle é mostrado na Tabela 5.12 - .

Tabela 5.12: Custo da Análise de Fluxo de Controle

Programa	Tempo de Análise (s)	
	Profiling Contínuo	*Profiling* Interpretativo
1942	0,29	0,70
AA	0,17	0,17
AKN	0,27	0,28
BF	0,13	0,22
BLT	0,45	0,43
BTC	0,26	0,14
BMB	0,12	0,09
DDG	0,13	0,09
DK	0,50	0,71
GLX	0,12	0,08
ICE	0,22	1,83
LRU	0,48	0,49
MB	0,27	0,78
MLP	0,39	0,66
PAC	0,06	0,06
PEY	0,33	1,23
SMB	0,42	0,45

O mesmo algoritmo de análise foi utilizado em ambas abordagens, portanto seu custo é praticamente o mesmo em ambas. No entanto, existem dois fatores que contribuem para que haja diferença no tempo de execução dos algoritmos de análise. Primeiramente, o tempo gasto com análise é diretamente proporcional à quantidade de épocas executadas. A Tabela 5.13 - mostra a quantidade de épocas executadas em cada abordagem.

É seguro afirmar que o tempo gasto com análise é proporcional à quantidade de épocas executadas, devido ao fato que a análise é executada exatamente uma vez no final de cada época. Logo, quanto mais épocas, haverá mais chamadas aos algoritmos de análise de fluxo de dados. Outro fator que pode afetar o tempo de execução dos algoritmos apresentados é a complexidade do grafo de fluxo de controle. Embora a complexidade assintótica dos algoritmos apresentados seja linear sobre o tamanho do grafo, existem casos onde os grafos gerados utilizando a abordagem interpretativa sejam pequenos demais, devido à fragmentação. Nestes casos, a execução dos algoritmos de análise se torna trivial, minimizando seu tempo de execução.

Embora os algoritmos escolhidos para análise de fluxo de dados sejam sub-ótimos, são capazes de encontrar, na maioria dos programas, os fluxos mais comumente utilizados sem causar tradução excessiva. Por fim, o tempo gasto na análise de fluxo de dados é muito abaixo do esperado. Portanto, é interessante investigar outros algoritmos, potencialmente

Tabela 5.13: Épocas Executadas

Programa	P. Contínuo	P. Interpretativo
1942	696	2222
AA	648	718
AKN	934	1073
BF	580	1230
BLT	1379	1836
BTC	1337	884
BMB	682	593
DDG	889	707
DK	1427	3747
GLX	1056	808
ICE	794	11379
LRU	1166	1566
MB	1009	5020
MLP	1214	4367
PAC	344	495
PEY	1352	8766
SMB	1165	1499

mais onerosos, que possam encontrar fluxos ótimos com o objetivo de solucionar problemas nas execuções de programas como DDG, que obteve o pior desempenho entre os programas avaliados.

5.5 CONSIDERAÇÕES FINAIS

A proposta deste trabalho apresenta melhorias significativas em relação à abordagem de *profiling* interpretativo. Os resultados mostram que as estratégias elaboradas fazem com que o sistema seja capaz de determinar os fluxos de execução quentes do programa, fazendo com que a execução das instruções seja realizada a maior parte por meio de tradução. Além disto, mecanismos são empregados para impedir a tradução excessiva, mantendo a taxa de execução traduzida em níveis aceitáveis. Além de manter a quantidade de instruções interpretadas em níveis aceitáveis minimizando a fragmentação, a abordagem de *profiling* contínuo utilizada também ajuda a minimizar o custo de outras tarefas de emulação. Por fim, os resultados também apresentam evidências que as técnicas empregadas na proposta podem ser implementadas eficientemente, inclusive os trechos de instrumentação necessários durante a interpretação das instruções. Portanto, a proposta apresentada no Tópico 4 é uma alternativa viável ao uso de *profiling* interpretativo, cumprindo os

objetivos propostos para este trabalho. O próximo Tópico contempla as conclusões finais, juntamente com as propostas de trabalhos futuros.

6

CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou um sistema TDB que utiliza *profiling* contínuo para detectar unidades de tradução quente de maneira eficiente. Os objetivos específicos deste trabalho foram cumpridos em forma de uma proposta detalhada de cada mecanismo e algoritmo empregado, além de uma implementação de prova de conceito juntamente com análise experimental das contribuições esperadas.

O principal diferencial da abordagem de *profiling* contínuo em relação a outros trabalhos está no fato que nenhuma instrumentação é necessária no código traduzido. Desta forma, todas as informações sobre o fluxo de execução do programa emulado são coletadas no laço de emulação e nas instruções interpretadas. Desta forma, o custo de *profiling* não é aumentado em relação à abordagem interpretativa, ao mesmo tempo que possibilita a determinação das transições entre todas as unidades de tradução do programa, sejam elas blocos básicos interpretados ou fluxos quentes traduzidos.

Com os dados colhidos durante o processo de *profiling* contínuo é possível atingir o objetivo de diminuir a fragmentação das unidades de tradução, detectando fluxos de execução mais complexos, inclusive entre fluxos previamente traduzidos e outras unidades de tradução, em um processo denominado fusão de fluxo. Esta característica permite reconhecer alterações no fluxo de execução do programa, permitindo que fluxos previamente traduzidos sejam atualizados para transferência do fluxo de controle a novos fluxos quentes, sem a necessidade de voltar ao laço do interpretador. Com isto, grande parte da execução é mantida por meio de traduções, diminuindo a quantidade de interpretação e *profiling* necessário conforme a execução acontece.

O principal algoritmo desenvolvido para o reconhecimento de fluxos quentes, HotDFS, utiliza uma heurística gulosa para determinar ciclos quentes na execução dos programas. Embora o algoritmo seja simples e baseado em algoritmos conhecidos, ele contém ajustes que propiciam a detecção de fluxos quentes que, em média, executam 85,21% das instruções das aplicações avaliadas.

A partir do mecanismo de controle de traduções e retraduções proposto, as chamadas ao tradutor dinâmico foram mantidas sob controle. Embora a quantidade de invocações tenha sido praticamente o dobro da abordagem interpretativa, o ganho de desempenho promovido pela tradução de unidades cada vez maiores compensa o custo das traduções adicionais.

Os resultados mostram que, embora haja custo de execução, *profiling* contínuo também pode ter oportunidade para reduzir os custos relacionados à TDB. Em geral, a fusão de fluxos proporcionada diminui a quantidade de transições entre fluxos traduzidos, aumentando a duração das épocas de execução. Desta forma, menos épocas são processadas, o que alivia o processamento das tarefas de análise de fluxo de controle.

Os resultados obtidos utilizando o emulador de NES mostram que a abordagem sugerida permite a emulação eficiente de programas, com 85,21% das instruções executadas por meio de traduções, sendo até 6,29 vezes mais rápido que interpretação tradicional e 2,34 vezes mais rápido que a abordagem interpretativa proposta por Jones (Jones, 2010).

Os resultados também mostram que, embora a execução seja suficientemente eficiente, há espaço para trabalhos futuros. A próxima seção apresenta propostas de trabalhos futuros que tem como principal objetivo melhorar o desempenho da proposta apresentada neste trabalho.

Trabalhos Futuros

Como visto nos resultados, o custo do *profiling* na interpretação para determinação de blocos básicos é alto, em média 49,39% do tempo total de interpretação. É possível diminuir este custo substituindo a instrumentação por amostragem (Buytaert et al., 2007; Cuthbertson et al., 2009; Schneider et al., 2007). Em técnicas baseadas em amostragem, o programa é interrompido periodicamente e métricas sobre a execução são recolhidas. Desta forma, o custo da obtenção dos dados da execução do programa é reduzido drasticamente. Trabalhos como HQEMU (Hong et al., 2012) utilizam amostragem para realização de *profiling*, que podem reduzir o custo para apenas 1,4% do tempo total de execução.

Embora a utilização de amostragem para aquisição de perfis de execução seja interessante, principalmente pelo custo reduzido, existem desafios relacionados à utilização de amostragem. O principal desafio é a necessidade de utilizar análise estatística para determinar os fluxos de execução a partir dos dados amostrados. Portanto, é necessário desenvolver algoritmos eficientes para este fim. Além disto, o perfil encontrado por meio de amostragem não é determinístico, portanto é necessário que os algoritmos de análise de fluxo de controle sejam adequados a estes requisitos.

Outro ponto que pode ser melhorado é a determinação dinâmica dos limiares que servem como parâmetros para a abordagem proposta. Em específico, o limiar de calor do nó, o limiar de tradução e o tamanho da época são parâmetros importantes para o desempenho da execução e devem ser investigados com mais profundidade para otimizar a execução do programa.

Outra melhoria possível é determinar algoritmos de detecção de ciclos que obtenham ciclos ótimos durante a fase de análise de fluxo de controle, diferentemente do HotDFS. Desta forma, é possível detectar ciclos mais rapidamente, potencialmente diminuindo a necessidade posterior de análise para fusão de ciclos temporalmente adjacentes.

A principal motivação para a investigação de algoritmos melhores de detecção de fluxo quente está no fato que os resultados mostram que o tempo atualmente dispendido em análise de fluxo de controle é baixo, enquanto existem programas cuja execução sofre da não-otimalidade do algoritmo HotDFS, como o DDG, apresentado no Capítulo 5. Desta forma, é possível a investigação de algoritmos ótimos, mesmo que potencialmente mais onerosos, para determinação de fluxos quentes.

Outro possível direcionamento para esta pesquisa é a implementação da abordagem sugerida como suporte para emulação de sistemas modernos. Em específico, é interessante verificar o desempenho desta implementação em programas com características diferentes de jogos, tais como aplicações científicas de alto desempenho e outras aplicações *desktop*. Desta forma, é possível a comparação direta com outras abordagens (Hong et al., 2012; Jones, 2010; Ottoni et al., 2011) que são avaliadas utilizando estes tipos de aplicação.

É possível diminuir o custo da tradução ainda mais por meio da realização das traduções em fluxos paralelos de execução. Trabalhos recentes (Böhm et al., 2011a) mostram que esta abordagem pode levar a ganhos de desempenho ainda maiores, pois não é necessário terminar a tradução para que a execução do programa continue. Desta forma, é interessante investigar estas técnicas e determinar como as fases de análise, síntese e tradução podem se beneficiar de processamento paralelo.

REFERÊNCIAS

Altman, E.; Kaeli, D.; Sheffer, Y. Welcome to the opportunities of binary translation. *Computer*, v. 33, n. 3, p. 40 –45, 2000.

Bala, V.; Duesterwald, E.; Banerjia, S. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, v. 46, n. 4, p. 41–52, 2011.

Banerjia, S.; Havanki, W. A.; Conte, T. M. Treegion scheduling for highly parallel processors. In: *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, London, UK, UK: Springer-Verlag, 1997, p. 1074–1078 (*Euro-Par '97*, v.1).

Bellard, F. QEMU, a fast and portable dynamic translator. In: *Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA: USENIX Association, 2005, p. 41–46 (*ATEC '05*, v.1).

Böhm, I.; Franke, B.; Topham, N. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In: *Embedded Computer Systems (SAMOS), 2010 International Conference on*, 2010, p. 1 –10.

Böhm, I.; Edler von Koch, T. J.; Kyle, S. C.; Franke, B.; Topham, N. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA: ACM, 2011a, p. 74–85 (*PLDI '11*, v.1).

Böhm, I.; Edler von Koch, T. J.; Kyle, S. C.; Franke, B.; Topham, N. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. *SIGPLAN Not.*, v. 46, n. 6, p. 74–85, 2011b.

Buytaert, D.; Georges, A.; Hind, M.; Arnold, M.; Eeckhout, L.; De Bosschere, K. Using hpm-sampling to drive dynamic compilation. *SIGPLAN Not.*, v. 42, n. 10, p. 553–568, 2007.

Chernoff, A.; Herdeg, M.; Hookway, R.; Reeve, C.; Rubin, N.; Tye, T.; Bharadwaj Yadavalli, S.; Yates, J. FX!32 a profile-directed binary translator. *Micro, IEEE*, v. 18, n. 2, p. 56 –64, 1998.

Cmelik, B.; Keppel, D. Shade: a fast instruction-set simulator for execution profiling. *SIGMETRICS Perform. Eval. Rev.*, v. 22, n. 1, p. 128–137, 1994.

Cormen, T. H.; Stein, C.; Rivest, R. L.; Leiserson, C. E. *Introduction to algorithms*. 2nd ed. McGraw-Hill Higher Education, 2001.

Cuthbertson, J.; Viswanathan, S.; Bobrovsky, K.; Astapchuk, A.; Srinivasan, E. K. U. A practical approach to hardware performance monitoring based dynamic optimizations in a production jvm. In: *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, Washington, DC, USA: IEEE Computer Society, 2009, p. 190–199 (*CGO '09*, v.1).

Dewar, R. B. K. Indirect threaded code. *Commun. ACM*, v. 18, n. 6, p. 330–331, 1975.

Diskin, P. Nintendo entertainment system documentation. 2004.
Disponível em `http://nesdev.com/NESDoc.pdf`

Duesterwald, E.; Bala, V. Software profiling for hot path prediction: less is more. *SIGOPS Oper. Syst. Rev.*, v. 34, n. 5, p. 202–211, 2000.

Ebcioglu, K.; Altman, E.; Gschwind, M.; Sathaye, S. Dynamic binary translation and optimization. *Computers, IEEE Transactions on*, v. 50, n. 6, p. 529 –548, 2001.

Fayzullin, M. NES system architecture. 2005.
Disponível em `http://fms.komkon.org/EMUL8/NES.html`

Foleiss, J. H.; D'amato, A. L. T.; da Silva, A. F. Dynamic binary translation: A model-driven approach. In: *Proceedings of the XXXI International Conference of the Chilean Computer Science Society*, Valparaiso, Chile, 2012.

Hecht, M. S. *Flow analysis of computer programs*. New York, NY, USA: Elsevier Science Inc., 1977.

Hong, D.-Y.; Hsu, C.-C.; Yew, P.-C.; Wu, J.-J.; Hsu, W.-C.; Liu, P.; Wang, C.-M.; Chung, Y.-C. HQEMU: a multi-threaded and retargetable dynamic binary

translator on multicores. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, New York, NY, USA: ACM, 2012, p. 104–113 (*CGO '12*, v.1).

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

Hu, W.; Wang, J.; Gao, X.; Chen, Y.; Liu, Q.; Li, G. Godson-3: A scalable multicore risc processor with x86 emulation. *Micro, IEEE*, v. 29, n. 2, p. 17 –29, 2009.

Jones, D. *High speed simulation of microprocessor systems using ltu dynamic binary translation*. Tese de Doutoramento, University of Edinburgh, 2010.

Jones, D.; Topham, N. High speed cpu simulation using ltu dynamic binary translation. In: *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, Berlin, Heidelberg: Springer-Verlag, 2009, p. 50–64 (*HiPEAC '09*, v.).

Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, Washington, DC, USA: IEEE Computer Society, 2004, p. 75– (*CGO '04*, v.1).

Levine, J. R. *Linkers and loaders*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

Li, S.; Qiao, L.; Tang, Z.; Cheng, B.; Gao, X. Performance characterization of spec cpu2006 benchmarks on intel and amd platform. In: *Education Technology and Computer Science, 2009. ETCS '09. First International Workshop on*, 2009, p. 116 –121.

Muchnick, S. S. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 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. In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*, New York, NY, USA: ACM, 2011, p. 26:1–26:10 (*CF '11*, v.1).

Powell, D. C.; Franke, B. Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set

simulators. In: *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, New York, NY, USA: ACM, 2009, p. 315–324 (*CODES+ISSS '09*, v.1).

Probst, M.; Krall, A.; Scholz, B. Register liveness analysis for optimizing dynamic binary translation. In: *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002, p. 35 – 44.

Schneider, F. T.; Payer, M.; Gross, T. R. Online optimizations driven by hardware performance monitoring. *SIGPLAN Not.*, v. 42, n. 6, p. 373–382, 2007.

Smith, J.; Nair, R. *Virtual machines: Versatile platforms for systems and processes (the morgan kaufmann series in computer architecture and design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

Tarjan, R. Depth-first search and linear graph algorithms. In: *Switching and Automata Theory, 1971., 12th Annual Symposium on*, 1971, p. 114 –121.

Taylor, B. NTSC 2C02 technical reference. 2004.
Disponível em `http://nesdev.com/2A03%20technical%20reference.txt`

Thulasiraman, K.; Swamy, M. N. S. *Graphs: theory and algorithms*. New York, NY, USA: John Wiley & Sons, Inc., 1992.

Ung, D.; Cifuentes, C. Machine-adaptable dynamic binary translation. *SIGPLAN Not.*, v. 35, n. 7, p. 41–51, 2000.