Compilação Dinâmica com Seleção Inteligente de Otimizações

Thais Aparecida Silva Camacho[1], **Anderson Faustino da Silva**[1], **Vanderson Martins do Rosario**[2], **Edson Borin**[2]

[1]DIN – Universidade Estadual de Maringá (UEM) – Maringá – PR – Brazil

[2]IC – Universidade de Campinas (UNICAMP) – Campinas – SP – Brasil

thaiscamachoo@gmail.com, anderson@din.uem.br,{vanderson.rosario,edson}@ic.unicamp.br

Abstract. *Systems based on dynamic compilation generate target code at run time. Thus, the compilation time is included in the system runtime and it is crucial to build a low-cost system, which can at the same time generate good quality code and have low compilation cost. In this paper, we present a novel approach to use machine learning to perform an intelligent selection of optimizations to apply for each region dynamic compiled. Such a system was tested in a dynamic binary translator,* `OI-DBT`*, whose performance was increased by 26.32% when using this smart optimization approach.*

Resumo. *Sistemas que utilizam compilação dinâmica geram código alvo em tempo de execução, fazendo com que o tempo de compilação seja incluído no tempo total do sistema. Portanto, é crucial que o sistema de compilação dinâmico tenham ao mesmo tempo um baixo custo e seja capaz de gerar código de boa qualidade. Neste artigo apresentamos um sistema de aprendizado de máquina para seleção inteligente de otimizações que aprende quais as melhores sequências de otimizações para cada região de código compilado por um compilador dinâmico. O sistema foi implementado e testado em um tradutor dinâmico de binários, o* `OI-DBT`*, trazendo um ganho médio de desempenho de 26,32%.*

1. Introdução

Sistemas de compilação dinâmica, ou *Just-In-Time* (`JIT`), compilam a maior parte ou todo o código de uma aplicação concomitante à sua execução, ao contrário dos compiladores *Ahead-of-time* que compilam todo código antes da sua execução. Tal mecanismo gera código alvo em tempo de execução; portanto, o tempo de execução do sistema engloba o tempo de compilação e execução do programa. Desta forma, é importante ser seletivo no que compilar e como compilar. Sistemas de compilação dinâmica, como os Tradutores Dinâmicos de Binários (`TDB`), que traduzem/emulam um binário de uma arquitetura em outra, utilizam estratégias de seleção de regiões fixas e previamente estabelecidas (por especialistas [Ishizaki et al. 2003] ou por *auto-tuning* [Hoste et al. 2010]). Independente das características da região sendo compilada ou da arquitetura alvo, utiliza-se sempre o mesmo plano de otimização, ou seja, as mesmas sequências de transformações, também conhecidas como otimizações [Muchnick 1997].

Agradecimentos: ao CNPq (313012/2017-2) e CAPES (PROCAD 2966/2014), e também a Fapesp (2013/08293-7) e Petrobras pelo apoio financeiro.

No entanto, sabe-se que para cada par trecho de código e arquitetura alvo existe uma sequência ótima de otimizações [Muchnick 1997] e, portanto, o ideal seria encontrar um plano de compilação específico para cada região (parte do código fonte selecionado por um compilador JIT dinamicamente) sendo compilado. Pois cada região tem suas especificidades, fazendo com que cada uma possa ser beneficiada por um plano de compilação.

Neste artigo mostramos como compilação dinâmica com seleção inteligente de otimizações (planos de compilação específicos por região) pode aumentar o desempenho de sistemas de compilação dinâmica. Para isto, apresentamos a nova geração do sistema OI-DBT[1] que possui além do modo de execução padrão, o qual aplica o mesmo plano de compilação a todas as regiões, quatro novos modos de execução: (1) um modo capaz de gerar bons planos de compilação e formar uma base de conhecimento; (2) um modo baseado na estratégia de raciocínio baseado em casos (RBC); (3) um modo capaz de modificar um determinado plano de execução; e por fim (4) um modo que gera diferentes planos a cada compilação da mesma região. O objetivo principal dos novos modos de execução é fornecer um sistema de compilação dinâmica de alto desempenho, que considere as especificidades de cada região e desta forma seja capaz de criar um plano de compilação específico para cada região e que possa ser treinado de forma automática para cada arquitetura alvo (auto-tuning) sem a necessidade de intervenção de um especialista. Sendo, este, o primeiro trabalho em nosso conhecimento a propor um compilador dinâmico que é capaz de aprender bons planos de compilação, aplicar diferentes otimizações para cada região de código compilado por um JIT (não se limitando a métodos), e considerando também a ordem da aplicação das otimizações e o custo de aplicação das otimizações.

Os nossos resultados indicam que criar planos de compilação específicos para cada região melhorou o desempenho do OI-DBT. De fato, é possível obter na média um ganho de desempenho de 26,32% frente a versão original do OI-DBT. Em resumo, o presente trabalho trás as seguintes contribuições:

- um sistema de aprendizado e construção de bons planos de compilação para diferentes regiões de um TDB/JIT, baseado em um algoritmo genético que busca encontrar planos de compilação que minimizem o custo de compilação e maximizem o ganho de desempenho;
- o uso de raciocínio baseado em casos para escolher planos de compilação em um TDB/JIT;
- o uso de um algoritmo de mutação para ajustar planos de compilação de forma contínua, durante as execuções de um TDB/JIT; e,
- o teste de ambas as técnicas em um TDB/JIT real, o OI-DBT, que utiliza a infraestrutura LLVM como otimizador e gerador de código.

2. Trabalhos Relacionados

Compiladores dinâmicos são ferramentas bastante presentes na computação e muito utilizados em cenários reais. Como exemplos, podemos achar a *Hip-Hop Virtual Machine* (HHVM) [Ottoni 2018], um tradutor dinâmico de PHP desenvolvido pela Facebook e utilizada em seus web-sites; e as diversas implementações da *Java Virtual Machine* [Venners 1998], um tradutor dinâmico de *Java byte code*, incluindo a Dalvik/ART

[1]Código fonte: https://github.com/thaisacs/oi-dbt

utilizada pelos dispositivos Android [Yadav and Bhadoria 2015]. Apesar disso, pouco foi explorado no uso de compilação adaptativa em compiladores dinâmicos. Hoste, Georges e Eeckhout [Hoste et al. 2010] propuseram o uso de algoritmos genéticos para encontrar uma boa sequência de otimização para um compilador JIT (JVM) procurando sequências Pareto-ótimas em aumento do desempenho e custo de compilação gerando um aumento de 26% de desempenho nos benchmarks testados. Contudo, a abordagem não leva em consideração a possibilidade de se aplicar diferentes sequências de otimizações para cada região. Cavazos e O'Boyle [Cavazos and O'boyle 2006] apresenta um mecanismo para escolher qual nível de otimização aplicar em cada método na JVM utilizando-se de informações extraídas do método e uma heurística previamente treinada gerada com regressão logística. Apesar de ser uma abordagem que aplica uma otimização específica por método, o trabalho apenas considera quatro possíveis otimizações padrões (O0, O1, O2 e O3). Os autores apontam um ganho de desempenho de 12% com o uso da técnica. Sanchez *et al.* [Sanchez et al. 2011] apresentaram um mecanismo de aprendizado de máquina para aprender e depois decidir quais otimizações são vantajosas para cada método na JVM, mas eles apenas consideram desligar otimizações nas sequências já usadas, não considerando novas ordens. Os autores mostram que a técnica diminuiu o custo de compilação, aumentando o desempenho em programas rápidos e pequenos, mas não conseguiu melhorar o desempenho em programas maiores. Dessa forma, em nosso conhecimento, as características do sistema apresentado nesse trabalho são inéditos.

3. OI-DBT: Um Tradutor Dinâmico de OpenISA

O `OI-DBT` é um `TDB` que permite executar binários OpenISA [Auler and Borin 2015], uma arquitetura livre em desenvolvimento na Unicamp, em arquiteturas `x86`, `ARM`, entre outras. Um `TDB` é um emulador que utiliza de compilação dinâmica para executar binários de uma arquitetura em outra com alto desempenho. O `OI-DBT` é um `TDB` com suporte a múltiplos alvos, ou seja, o mesmo `TDB` suporta tradução para diversas arquiteturas. Para isso, o `OI-DBT` inicia a emulação com um interpretador escrito em `C` que pode ser compilado para diversas arquiteturas e, em seguida, utiliza de técnicas como `NET` [Duesterwald and Bala 2000], `NETPlus` [Hiniker et al. 2005] ou `MRET2` [Wang et al. 2007] para selecionar regiões de código que são traduzidas para a representação intermediaria da infraestrutura `LLVM`[2] e, em seguida otimizadas e compiladas utilizando a infraestrutura da `LLVM`.

O `OI-DBT` emprega duas `threads` para emulação, uma que é responsável por executar o código e outra que é responsável pelas compilações. Escondendo assim parte da sobrecarga de compilação. Apesar de estar sob desenvolvimento, o `OI-DBT` é um `TDB` moderno e com uma infraestrutura de otimizações completa (oriundo da infraestrutura `LLVM`), sendo ideal para avaliar o uso de diferentes planos de compilação nas regiões durante a emulação.

Anterior a esse trabalho, também como em outros compiladores dinâmicos, o `OI-DBT` apenas tinha suporte a aplicar um mesmo plano de compilação a todas as regiões. Nesse trabalho, propomos adicionar o suporte a otimizações inteligentes ao `OI-DBT` permitindo que ele aplique planos de compilação diferentes para cada região na tentativa de obter o melhor ganho de desempenho com o menor custo de otimização para cada região.

[2]www.llvm.org

Isso é feito por meio do módulo Sistema de Otimização Adaptativo SOA, adicionado em sua arquitetura. Tornando, portanto, o OI−DBT o primeiro TDB/JIT, em nosso conhecimento, com seleção inteligente de otimizações por região que leva em consideração ambos o tempo de execução e o tempo de compilação.

4. Otimizações Inteligentes em Compiladores Dinâmicos

Prover um TDB de alto desempenho que crie dinamicamente bons planos de compilação, a um custo baixo, trás diversas vantagens por diminuir o tempo de compilação e aumentar o desempenho final do JIT sem intervenção de especialistas. Para alcançar tal objetivo, propomos projetar um gerador de planos de compilação eficiente, o qual deve considerar as particularidades de cada região como também minimizar o custo do sistema como um todo.

A nova geração do OI−DBT é capaz de criar dinamicamente bons planos de compilação, os quais consideram o problema de geração de código como um problema dependente do código fonte (da região) ao mesmo tempo que minimiza o custo total do sistema. A ideia é extrair conhecimento de cada nova região e então usar tal conhecimento para montar um bom plano de compilação. O novo OI−DBT possui quatro novos modos de execução, como descrito a seguir.

Gerador de conhecimento: Este modo de execução cria uma base de conhecimento, composta por planos de compilação para cada região encontrada pelo sistema.

Gerador de planos de compilação baseado em RBC: Este modo de execução inspeciona uma base de conhecimento com o objetivo de encontrar bons planos de compilação para cada região.

Gerador estático de planos de compilação: Este modo de execução tem por objetivo gerar um novo plano de compilação para cada região de forma aleatória, ou seja, sem considerar planos anteriores.

Gerador contínuo de planos de compilação: Este modo de execução também não utiliza conhecimento prévio, mas é capaz de aprender durante a execução do sistema. Isto indica que tal modo parte de um plano que pode não ocasionar um bom desempenho ao sistema, contudo tenta aperfeiçoá-lo nas próximas execuções do sistema.

4.1. Modo: Gerador de Conhecimento

Este modo de execução utiliza um algoritmo genético (AG) para reduzir o espaço de busca e encontrar bons planos de compilação que irão compor a base de conhecimento. O AG inicia com uma população, a qual sofre um processo iterativo que a modifica durante diversas gerações. A primeira geração é composta por indivíduos (planos de compilação) gerados aleatoriamente. A partir deste ponto, a cada geração, dois operadores genéticos são aplicados, (1) *crossover* e (2) mutação.

O AG proposto é parametrizável e permite alterar: o tamanho da população, a quantidade de gerações, a taxa de mutação, o peso da compilação, o peso da execução, o tamanho mínimo do indivíduo e o tamanho máximo do indivíduo. Cada indivíduo é representado por um vetor, onde cada elemento representa uma transformação.

O *fitness* de cada indivíduo é calculado baseado nos custos do tempo de compilação somado ao tempo de execução da região com as otimizações representadas

pelo indivíduo aplicadas. Esta funcionalidade provê ao SOA o atrativo de encontrar planos de compilação capazes de balancear entre tempo de compilação e tempo de execução, de forma a prover um sistema de alto desempenho a um custo baixo.

Após a execução de todas gerações, o AG grava na base de conhecimento a melhor relação $< DNA, plano\ de\ compilação >$, para cada região processada. O DNA descreve as características de uma região de acordo com suas instruções, ou seja, cada gene codifica uma determinada instrução da região. É importante destacar que o OI−DBT é baseado na infraestrutura LLVM, desta forma cada gene representa uma instrução de tal infraestrutura. Por sua vez, um plano de compilação representa uma boa sequência de transformações para tal DNA.

4.2. Modo: Gerador de Planos de Compilação Baseada em RBC

Para gerar planos de compilação baseados em conhecimento prévio, o SOA utiliza uma estratégia de raciocínio baseado em casos [Richter and Weber 2013], que é uma abordagem de aprendizagem de máquina que busca resolver novos problemas adaptando soluções utilizadas para resolver problemas anteriores. Este modo executa basicamente dois passos: (1) recupera, na base de conhecimento, a relação cujo DNA mais se aproxima do DNA da região em questão (nova região a ser compilada pelo sistema); e (2) aplica o plano de compilação recuperado à nova região.

O processo de recuperar uma relação é baseado em uma medida de similaridade, a qual mede o nível de similaridade entre dois DNAs. A medida de similaridade é o *score* encontrado pelo algoritmo proposto por Needleman e Wunsch [Needleman and Wunsch 1970], o qual é um algoritmo de programação dinâmica de alinhamento global de genes.

4.3. Modo: Gerador Estático de Planos de Compilação

Este modo de execução não gera conhecimento como também não utiliza conhecimento prévio. Para cada região, o SOA gera um plano de compilação sem considerar as características da região. Este modo de execução é atrativo para avaliar o ganho de desempenho alcançado por outros modos de execução.

4.4. Modo: Gerador Contínuo de Planos de Compilação

O modo de geração contínua utiliza o algoritmo *Random Mutation Hill Climbing* (RMHC) [Zhao 2004]. Este modo parte de um plano de compilação gerado aleatoriamente e o modifica em novas execuções do sistema. O algoritmo inicia com um indivíduo (plano de compilação) gerado aleatoriamente (para cada região) e então aplica mutações neste indivíduo quando a mesma região é vista pelo SOA. Isto indica que tal modo de execução é indicado para reexecuções de uma mesma região, ou seja, quando um determinado programa é reexecutado pelo OI−DBT.

A mutação do GA, empregado pelo modo de gerador de conhecimento, apenas troca um gene por outro em uma determinada posição do DNA. Por sua vez, a mutação empregada pelo RMHC possui quatro operações: (1) troca de um gene, (2) inserção de um gene, (3) remoção de um gene, e (4) troca de posição entre dois genes (*swap*). Contudo apenas uma determinada mutação é aplicada a um DNA, a qual é escolhida aleatoriamente.

4.5. Utilização e Custo dos Modos de Execução

O modo de execução do gerador de conhecimento foi proposto para ser utilizado por um modo de execução de baixo impacto ao sistema (modo gerador de planos de compilação baseado em RBC). Embora um AG seja capaz de encontrar bons planos de compilação para ambientes baseados em JIT, este acarreta um alto custo ao sistema; ou seja um alto tempo de execução. Portanto, embora possa ser utilizado como modo principal de execução não é uma boa estratégia. Por outro lado, como um sistema de treinamento tal modo se torna uma boa estratégia. Pois uma vez que o treino foi realizado não existe a necessidade de realizá-lo novamente, acarretando que seu custo seja nulo em execuções do sistema que apenas use conhecimento gerado previamente.

O custo que incorre o modo de execução do gerador de planos de compilação baseado em RBC está associado à análise da base de conhecimento, por busca de um bom plano de compilação para uma região. É importante perceber que cada região possui seu próprio plano de compilação, portanto, quanto maior a quantidade de regiões maior será o custo do SOA. Outro detalhe a ser observado é o fato deste modo de execução não analisar o desempenho de um plano de compilação (como o calculo do *fitness* feito pelo AG). Isto tem por objetivo reduzir o custo do SOA. Dessa forma, o ideal é que diferentes bases de conhecimento sejam geradas, permitindo que tal modo de execução seja guiado por determinado objetivo.

O modo gerador contínuo além de reduzir o custo do SOA, por não avaliar o desempenho de um plano de compilação e não necessitar de uma busca em um base de dados, tem como premissa ser um modo de execução que acumula e aprimora conhecimento com o passar do tempo. Portanto, este modo possui um custo menor do que o modo baseado em RBC, além do potencial de melhorar a qualidade de um plano de otimização. Como mencionado anteriormente, o uso deste modo é indicado para reexecuções de um mesmo programa.

Por fim, embora o modo gerador estático possua um custo baixo de gerenciamento, não possui o potencial de acumular conhecimento nem considera as características de uma determinada região. Portanto, tal modo de execução é indicado como parâmetro de avaliação do ganho de desempenho obtido por outros modos.

5. Resultados e Discussão

Os experimentos foram realizados em uma máquina com processador Intel i7-3770 3,40GHz, com 4 GB de RAM. O OI-DBT foi implementado utilizando-se a infraestrutura de compilação LLVM versão 7.0 e é compilado com o GCC versão 8.3.0. Sendo executado em um Debian 10, kernel 4.19.16-1. Os parâmetros utilizados na parametrização do AG foram baseados nos parâmetros do zhao, são eles: população = 20; gerações = 20; taxa de mutação = 0.016; tamanho mínino do DNA = 10; e tamanho máximo do DNA = 25. Já no RMHC: tamanho mínino do DNA = 10; e tamanho máximo do DNA = 25. No caso do gerador estático, idem parametrização do RMHC.

Os experimentos utilizam 142 programas do *benchmark* de teste da infraestrutura LLVM. Os programas são divididos em dois grupos: (1) grupo de treino e (2) grupo de teste. O primeiro é utilizado somente durante a geração de conhecimento. O segundo é utilizado para avaliar o desempenho do sistema. As próximas seções detalham a composição de cada grupo.

Cada programa é executado 5 vezes, desta forma os dados apresentados indicam a média aritmética entre as execuções. Para avaliar o modo de execução baseado em RBC geramos diversas bases, alterando o peso entre tempo de compilação e tempo de execução (C/E). Os pesos variam entre [0.0, 0.5, 0.9, 1.0]. Portanto a configuração 0.5/0.5 indica que cada componente (compilação e execução) irá contribuir com 50% do *fitness*.

5.1. A Base de Conhecimento

A base de conhecimento é criada habilitando-se o modo gerador de conhecimento do OI-DBT. A Tabela 1 apresenta a quantidade de relações entre regiões e planos de compilação gerados neste experimento.

Tabela 1. Síntese das bases de conhecimento

Benchmark	Programas	Regiões	Benchmark	Programas	Regiões
ASC_Sequoia	1	1	McGill	3	22
BenchmarkGame	7	11	Misc	19	25
BitBench	4	8	Olden	7	43
CoyoteBench	1	1	Prolangs-C	3	1
Fhourstones	1	1	Ptrdist	3	2
Fhourstones$_3$1	1	1	Shootout	15	15
FreeBench	4	15	Stanford	10	99
mafft	9	9	Versabench	5	16
McCat	8	49			

Os resultados apresentados na Tabela 1 refletem a estratégia utilizada para geração de conhecimento, a qual se baseou na execução parcial de cada programa. Como pode ser observado, existem programas cuja quantidade de regiões é menor do que a quantidade de programas. Isto indica que não foram gerados planos de compilação para todas as regiões.

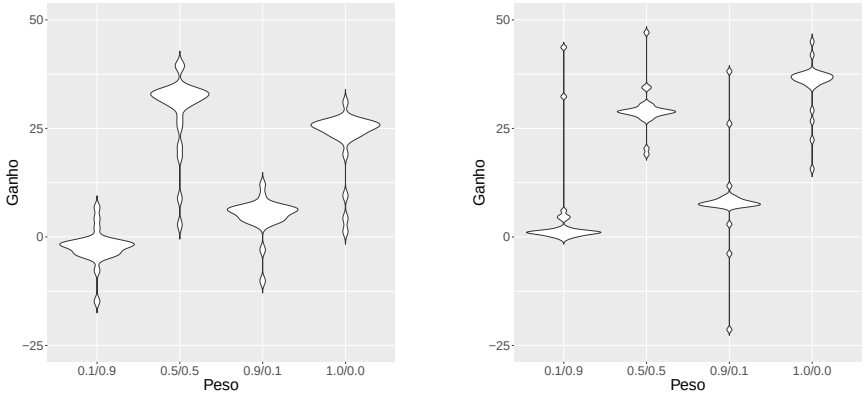
Naturalmente é esperado que, quanto maior a quantidade de conhecimento disponível, maior a probabilidade de ganho de desempenho. Contudo, é importante observar que tal hipótese pode ser inválida, pois pode ocorrer que a base não contenha uma relação cuja sua região tenha alta similaridade com a região para a qual o SOA deve encontrar um plano de compilação.

Para analisar tal hipótese criamos diversas bases, baseadas nos dados apresentados na Tabela 1. Cinco bases (Base1.x[3]) não contém informações sobre os *benchmarks* BENCHMARKGAME, FREEBENCH, OLDEN, e STANFORD. Portanto, cada Base1.x contém relações sobre 151 regiões. Outras cinco bases (Base2.x) contém, cada uma, as 319 relações apresentadas na Tabela 1.

5.2. O Desempenho dos Modos de Execução

Inicialmente analisaremos o modo de execução baseado em RBC. Enquanto as bases foram treinadas com os *benchmarks* na Tabela 1, o RBC foi avaliado utilizando 17 programas do *benchmark* MIBENCH, 5 do MEDIABENCH, e 5 do TRIMARAN. O objetivo é comparar o ganho de desempenho de diferentes relações entre tempo de compilação e tempo de execução. A Figura 1 apresenta o ganho de desempenho (em percentual), considerando a configuração 0.0/1.0 como *baseline*.

[3]O x indica a configuração, por exemplo, 0.5/0.5.



(a) Base1 (b) Base2

Figura 1. Ganho ao utilizar diferentes relações entre custo de compilação e melhora de desempenho (0.1/0.9, 0.5/0.5, 0.9/0.1, 1.0/0.0) em relação a utilizar a relação 0.0/1.0.

O ganho médio para a Base1 é de -2,33%, 35,75%, 4,98% e 25,98%, respectivamente para as configurações 0.1/0.9, 0.5/0.5, 0.9/0.1 e 1.0/0.0. Para a Base2 o ganho é 2.25%, 29,35%, 8,10% e 35,14%, respectivamente para as mesmas configurações. Tais resultados indicam três situações: (1) o ganho de desempenho está relacionado a redução do tempo de compilação; (2) aumentar o tamanho da base ocasiona um aumento no ganho de desempenho para algumas relações; e (3) utilizar uma base concisa que tente equilibrar o custo entre tempo de compilação e execução (0.5/0.5) parece ser uma boa estratégia.

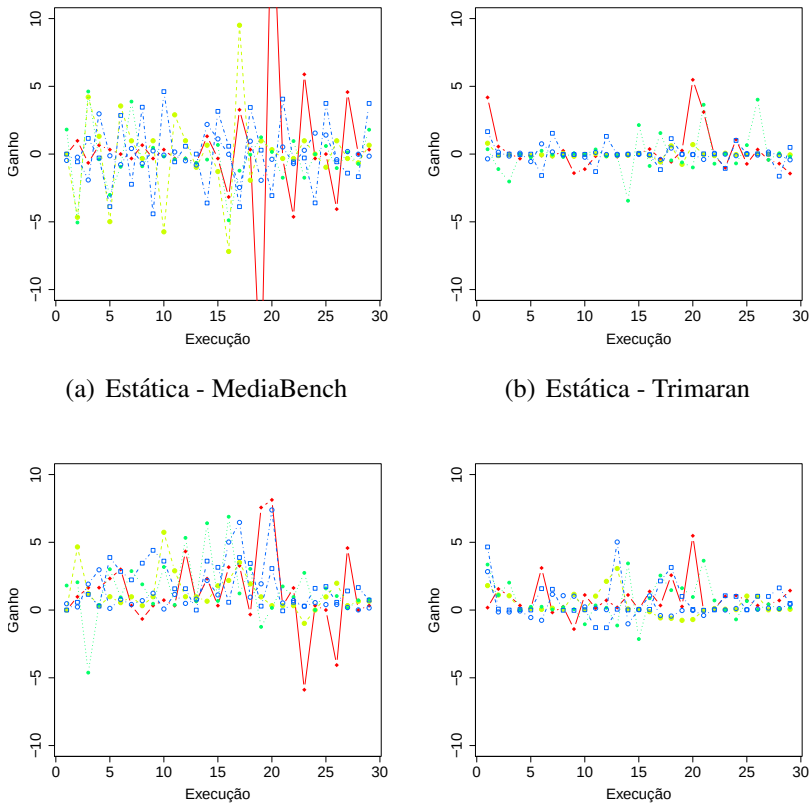
O uso de diferentes bases gera um comportamento similar ao desempenho, considerando ganho e perdas. A exceção é para a configuração 0.1/0.9, para a qual o uso de uma base maior obteve um melhor resultado. Um fato interessante é a perda de desempenho para a configuração 0.9/0.1. Para a Base1, isto é ocasionado pelo fato de tal configuração ocasionar um ganho de até 12%, contudo com perdas de desempenho que chegam a 10%. Para a Base 2, ocorre uma situação similar. Embora neste caso o ganho chegue a 38%, na maioria dos casos o desempenho não ultrapassa 8% o que indica uma média geral baixa. Portanto, embora reduzir o custo do tempo de compilação seja uma boa estratégia é crucial criar uma base de dados de boa qualidade.

No tocante ao tamanho da base de dados, é importante perceber que aumentar a quantidade de relações aumenta do custo do sistema. Como a estratégia `RBC` resolve um novo problema baseado na solução de um problema anterior, existe a necessidade de buscar na base de conhecimento um caso similar. Tal busca tem o potencial de aumentar o custo do sistema. Embora o ganho de desempenho para algumas configurações seja melhor utilizando a Base2, uma análise mais pontual indica que é melhor ter um percentual de ganho menor com um tempo total do sistema menor, do que um maior ganho que acarrete um aumento no tempo total do sistema. De fato, o uso da Base2 acarretou um aumento de até 50% no tempo total do sistema. Fato decorrente do uso de uma base 2 vezes maior.

A melhor estratégia é gerar uma base de dados concisa que considere tanto o tempo de compilação quanto o de execução, na mesma proporção. Uma base concisa reduz o custo do `SOA`, o custo total do sistema e ainda tem o potencial de proporcionar

estabilidade ao sistema. Tal fato é demonstrado pelos resultados obtidos. Independente da base de dados utilizada, o ganho obtido pela maioria dos programas se concentra próximo à média geral. Portanto, é possível concluir que RBC é uma boa estratégia para um sistema TDB/JIT de alto desempenho.

O próximo passo é analisar o desempenho do OI−DBT para os modos de geração estática e contínua. Para avaliar tais modos de execução, cada programa é executado continuamente 30 vezes. A Figura 2 apresenta o ganho de desempenho considerando a primeira execução como *baseline*.



(a) Estática - MediaBench (b) Estática - Trimaran

(c) Contínua - MediaBench (d) Contínua - Trimaran

Figura 2. Ganho com modo de geração estática e contínua em 30 execuções.

Como mencionado anteriormente, a diferença entre estes dois modos de execução está no fato do modo gerador contínuo partir de uma plano de compilação inicial e modificá-lo durante novas execuções. Embora o modo gerador estático alcance picos maiores, é possível perceber que a melhor estratégia é ajustar o plano de compilação as características do programa, especificamente, as características da região a ser compilada. Ambos modos não fornecem estabilidade ao sistema, o que indica que existe uma constante perda e ganho de desempenho. De fato, os resultados para ambos modos indicam que a melhor estratégia é utilizar o modo baseado em RBC.

O uso de conhecimento prévio além de ocasionar uma maior estabilidade ao sistema, proporciona ganhos maiores. Contudo, tal modo possui um custo superior de até 10% (para o tempo total do sistema) quando comparado com os modos estático e

contínuo. Isto indica que a busca por um plano de compilação em uma base, tem um custo superior ao de gerar um plano aleatoriamente ou ainda alterar um determinado plano. Porém é importante perceber que tal custo é amortizado pelo ganho de desempenho final, que pode chegar a 26% (como apresentado a frente).

Se considerássemos o ganho de desempenho total do modo contínuo, tal ganho chegaria a 10% em alguns casos. Tal ganho é considerável levando em consideração que tal modo possui um baixo custo. O modo contínuo tem o atrativo de gerar novos conhecimentos, o que não acontece necessariamente com o modo baseado em RBC. Embora este último seja capaz de adicionar à base de conhecimento novas relações, não são criados novos planos de compilação. Situação que ocorre no modo contínuo; contudo sem gerar uma base de conhecimento.

5.3. Plano de Compilação Fixo vs Plano de Compilação por Região

Por fim é necessário analisar o desempenho do uso de um único plano de compilação, para todas regiões, frente um modo de execução que seleciona um diferente plano de compilação para cada região.

O plano de otimização padrão do OI−DBT é baseado no LLVM O1, retirando algumas otimizações mais custosas, resultado na seguinte sequência: (”instcombine”, ”simplifycfg”, ”reassociate”, ”gvn”, ”die”, ”dce”, ”instcombine”, ”licm”, ”memcpyopt”, ”loop-unswitch”, ”instcombine”, ”indvars”, ”loop-deletion”, ”loop-predication”, ”loop-unroll”, ”simplifycfg”, ”instcombine”, ”licm”, ”gvn”).

Os resultados anteriores indicam que o uso de RBC, para uma base 0.5/0.5, é uma boa estratégia. Por isso, comparamos os resultados da versão prévia do OI−DBT, com a nova versão no modo baseado em RBC utilizando a Base1. A Figura 3 apresenta os resultados. Utilizar diferentes planos de compilação demonstrou ser uma boa estratégia para obter um sistema TDB/JIT de alto desempenho. O ganho de desempenho chega a uma proporção de 26,32% na média, se desconsiderarmos o pico (119,43%) o ganho médio seria de 22,74%, o que é um ganho considerável. O uso de diferentes planos de compilação não foi capaz de superar o desempenho do uso de um único plano de compilação apenas para um programa (para o qual a perda de desempenho foi de -0,036%, portanto insignificante). Isto indica que a nova versão do OI−DBT é promissora como um sistema TDB/JIT de alto desempenho.

Fica claro que existem campos para novas pesquisas em tradutores dinâmicos de binários de alto desempenho. Primeiro, uma estratégia promissora é utilizar o modo contínuo (estratégia de baixo custo) juntamente com um mecanismo de retrocesso. A ideia é manter o melhor plano de compilação encontrado até o momento, e somente descartá-lo caso seja encontrado um plano melhor. Desta forma, é possível retornar a um plano anterior caso ocorra perda de desempenho. Segundo, é possível implementar uma estratégia que agrupe diversos modos de execução. Desta forma, o sistema seria capaz de alterar o seu modo de execução dinamicamente de forma a se adequar as características do programa em execução. Terceiro, outro campo de pesquisa é o uso de aprendizagem contínua para gerar novos conhecimentos para a base de dados.

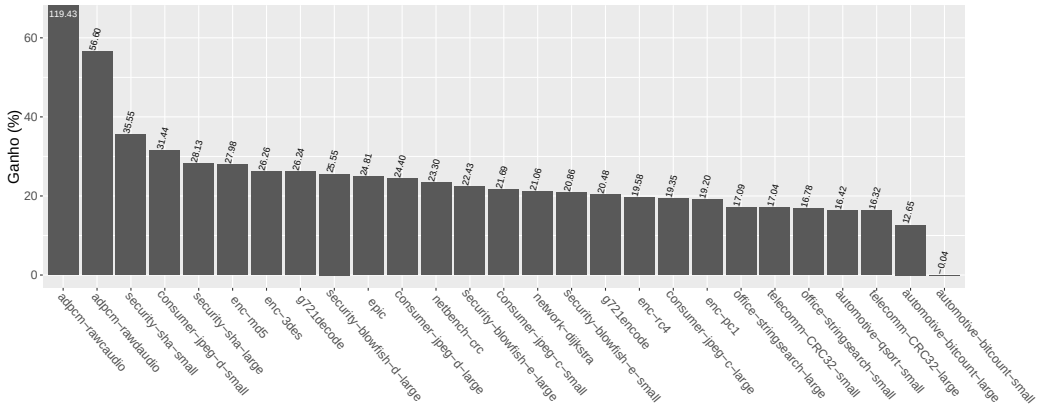


Figura 3. Ganho em relação a utilizar um plano de compilação com seleção inteligente de otimizações em comparação a sequência fixa no OI-DBT.

5.4. Impacto dos Modos de Execução no Código

O conjunto de instruções `OpenISA` é um conjunto conciso baseado em arquiteturas `RISC`. Tal característica acarreta um aumento de código, quanto este é traduzido para código `LLVM`. A Tabela 2 apresenta os dados médios, referentes ao tamanho de código. No geral, em nossos experimentos o tamanho do código duplica; com exceção para o modo contínuo. Porém independente do crescimento de código, o `SOA` irá reduzí-lo a medida que aplica um plano de compilação.

Aplicar um plano de compilação reduz consideravelmente o tamanho do código, chegando a ser 20 vezes menor do que o código original. Tal fato reduz o tamanho da *cache* de regiões, e melhora o desempenho do sistema à medida que gera código de boa qualidade. Portanto, embora a taxa de redução de código por região não ultrapasse 0,60 e o código `LLVM` seja duas vezes maior quando comparado ao código `OpenISA`, a aplicação de planos de compilação é fundamental para que o sistema `TDB/JIT` tenha alto desempenho na tradução/execução de programas diversos.

Parâmetros	Modos			
	RBC.Base1	RBC.Base2	Estático	Contínuo
LLVM/OI	2,26	2,47	2,91	4,12
Redução de código	20,70	21,23	22,01	27,65
Redução de código/Regiões	0,61	0,64	0,66	0,64

Tabela 2. Taxa de Redução de código

6. Conclusões e Trabalhos Futuros

Compiladores `JIT`, em geral, aplicam um/ou vários conjuntos fixos de otimizações para todas as regiões de código que compilam. Neste artigo mostramos como é possível a um sistema `JIT` utilizar otimizações de forma inteligente. Em outras palavras, mostramos como um sistema `JIT` que cria dinamicamente planos de compilação utilizando-se de aprendizado de máquina e baseado em um banco de conhecimentos é capaz de ser um sistema superior àquele baseado em um único plano de compilação. Especificamente, adicionamos um módulo de seleção de otimizações adaptativa à arquitetura do `OI-DBT` resultando em um ganho médio de 26,32%.

Como trabalhos futuros projetamos implementar dois novos modos de execução: (1) um que agrupe diferentes níveis de otimizações (marchas) que utilize diferentes bases de conhecimento, permitindo a re-otimização de regiões que se tornem quentes com bancos de conhecimentos mais agressivos; e (3) um novo modo para geração de conhecimento, que não seja baseado em algoritmo genético e portanto tenha um baixo custo de execução e possa ser utilizado não só em uma fase de treinamento.

Referências

Auler, R. and Borin, E. (2015). OpenISA, freedom powered by efficient binary translation. In *Architectural and Microarchitectural Support for Binary Translation*.

Cavazos, J. and O'boyle, M. F. (2006). Method-specific dynamic compilation using logistic regression. In *ACM SIGPLAN Notices*, volume 41, pages 229–240. ACM.

Duesterwald, E. and Bala, V. (2000). Software profiling for hot path prediction: Less is more. *ACM SIGOPS*, 34(5):202–211.

Hiniker, D., Hazelwood, K., and Smith, M. D. (2005). Improving region selection in dynamic optimization systems. MICRO 38, pages 141–154, Washington, DC, USA. IEEE Computer Society.

Hoste, K., Georges, A., and Eeckhout, L. (2010). Automated just-in-time compiler tuning. In *CGO'10*, pages 62–72. ACM.

Ishizaki, K., Takeuchi, M., Kawachiya, K., Suganuma, T., Gohda, O., Inagaki, T., Koseki, A., Ogata, K., Kawahito, M., Yasue, T., et al. (2003). Effectiveness of cross-platform optimizations for a java just-in-time compiler. In *ACM SIGPLAN Notices*, volume 38, pages 187–204. ACM.

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453.

Ottoni, G. (2018). HHVM JIT: A profile-guided, region-based compiler for PHP and Hack. In *ACM SIGPLAN Notices*, volume 53, pages 151–165. ACM.

Richter, M. M. and Weber, R. O. (2013). *Case-Based Reasoning: A Textbook*. Springer Publishing Company, Incorporated.

Sanchez, R. N., Amaral, J. N., Szafron, D., Pirvu, M., and Stoodley, M. (2011). Using machines to learn method-specific compilation strategies. In *CGO*, pages 257–266. IEEE.

Venners, B. (1998). *The Java Virtual Machine*. McGraw-Hill, New York.

Wang, C., Zheng, B., Kim, H., Jr., M. B., and Wu, Y. (2007). Two-pass MRET trace selection for dynamic optimization. Patent number 20070079293.

Yadav, R. and Bhadoria, R. S. (2015). Performance analysis for Android runtime environment. In *CSNT'15*, pages 1076–1079. IEEE.

Zhao, J. (2004). *Jikes RVM Adaptive Optimization System with Intelligent Algorithms*. PhD thesis, PhD thesis, School of Computer Science, The University of Manchester.