Um Algoritmo para Emparelhamento de Chamadas de Função

Francisco Demontiê, Filipe de Lima Arcanjo e Mariza A. S. Bigonha

Universidade Federal de Minas Gerais
{demontie,filipe,mariza}@dcc.ufmg.br

Resumo A maior parte dos compiladores atuais realiza uma série de otimizações no programa fonte sem garantir a preservação da semântica desse programa. Verificar a corretude de otimizações é uma tarefa difícil, pois erros de otimização podem aparecer em pontos muito diferentes daqueles onde eles tiveram origem. A fim de facilitar essa tarefa de depuração, este artigo apresenta uma técnica para o emparelhamento de programas antes e depois de otimizações. A grande inovação deste trabalho é usar funções externas como pontos de correspondência entre programas. Essa técnica é útil por várias razões. Em particular, ela pode encontrar problemas facilmente e pode servir como âncora para análises mais poderosas. Além disso, o emparelhamento proposto é rápido e confiável, pois compiladores não podem otimizar chamadas a funções que não possuem corpo, logo, tais funções precisam ser mantidas pelo otimizador. Esse trabalho apresenta um algoritmo que consiste em verificar isomorfismo de árvores de chamadas a funções externas construídas a partir da árvore de dominância de cada função do programa. Essa técnica foi implementada como um passo do compilador LLVM e executada sobre milhares de funções retiradas de *benchmarks* conhecidos. Essa abordagem conseguiu casar, na maior parte dos programas, pelo menos 70% das funções otimizadas via LLVM -O1. Esse número é ainda maior, 90%, para otimizações que não modificam o fluxo de controle do programa. Concluímos, então, que essa técnica é efetiva e útil em programas reais.

Abstract Most compilers apply a series of optimizations in source programs without ensuring that they preserve program semantics. Verifying optimization correctness is difficult, since optimization errors may appear in different places from where they originated. In order to facilitate this verification task, this paper presents a technique for matching programs before and after an optimization pipeline. The major innovation of this work is the use of library functions as matching points between programs. This technique is useful for many reasons. Particularly, it may catch problems easily and give anchor for stronger analyses. Additionally, the proposed approach is fast and reliable, as compilers cannot optimize function calls without having access to the function body. In this work we present an algorithm that verifies isomorphism between external function call trees built from the dominator tree of each function in the program. This technique was implemented as an LLVM pass and executed on thousands of functions of known benchmarks. Using the -O1 level of optimization, our approach matched, in most of the programs, at

least 70% of the functions. This number is even greater - 90% - for optimizations which do not change the program's control flow. We conclude that this technique is effective and useful for real-world programs.

1 Introdução

Garantir que um compilador sempre gera código correto é uma tarefa difícil. Alternativo a isso, A. Pnueli *et al.* [1] propuseram uma técnica denominada *validação de tradução*, que consiste em adicionar uma fase de verificação a cada tradução de um programa para verificar se o código gerado é equivalente ao código fonte. Nesse contexto, validadores de tradução começaram a ser empregados para testar otimizações realizadas por compiladores. Tendo em vista que verificar se dois programas são semanticamente equivalentes é um problema indecidível, diferentes heurísticas e técnicas vem sendo propostas para implementar validadores de tradução para otimizações de compiladores.

Barrett *et al.* [2] propuseram a ferramenta TVOC, que gera condições de verificação e utiliza um provador automático de teoremas para verificar a equivalência entre os programas. Apesar de prover bons resultados, a técnica acarreta em um acréscimo substancial do tempo de execução de uma compilação. Em contrapartida, uma série de trabalhos [3,4] propuseram técnicas que realizam transformações em grafos de valores construídos a partir do programa fonte e após otimizações. A ideia central das técnicas que utilizam esse conceito é gerar um grafo de valores, semelhante a uma árvore de expressões, para cada representação do programa maximizando o compartilhamento de nós. A partir daí, transformações são realizadas no grafo do programa original. Se ao finalizar as transformações os dois grafos forem isomorfos, diz-se que os programas são equivalentes. Embora mais eficientes, essas técnicas são, em geral, complexas para serem implementadas e lidam com um conjunto predefinido de otimizações.

Este artigo propõe uma técnica para emparelhamento de programas baseado em chamadas a funções externas, ou funções de biblioteca. Chamamos de emparelhamento de programas o ato de mapear instruções de um programa em instruções de um outro programa. Utilizar funções externas para realizar esse emparelhamento pode ser útil. A técnica é confiável, pois, uma vez que compiladores não podem realizar otimizações em funções externas, por não possuírem acesso ao corpo dessas funções, elas precisam ser mantidas pelo otimizador. A inserção dessa técnica no processo de otimização não acrescenta grande *overhead*. Além disso, essa abordagem permite encontrar problemas facilmente e pode servir como âncora para análises mais poderosas, como, por exemplo, propagação simbólica de intervalos [9], que computa faixas de valores para cada variável no programa. Vale salientar que o artigo não propõe uma técnica para validação de tradução, e sim um algoritmo que pode servir como base para esse propósito.

O algoritmo proposto consiste em verificar o isomorfismo entre árvores de chamadas a funções externas construídas antes e depois de um conjunto de otimizações. Essas árvores são construídas a partir das árvores de dominância de

cada função do programa. Na árvore gerada pelo algoritmo, uma chamada $G()$ é filha de uma chamada $F()$ se $G()$ está em um bloco básico que é filho de um bloco que contenha a chamada $F()$, ou se as duas chamadas estão no mesmo bloco básico e a chamada $G()$ é uma instrução posterior à chamada $F()$. Modificações na ordem em que as chamadas aparecem nas duas árvores podem indicar um erro durante a fase de otimização.

O algoritmo para emparelhamento de programas foi implementado como um passo do compilador LLVM e executado sobre 649.505 linhas de código de 1.629 programas dos *benchmarks* encontrados no conjunto de testes do compilador. Para cada uma das funções do programa, o passo foi executado usando as otimizações dos níveis O1, O2 e O3 de forma incremental. Para averiguar a aplicabilidade da técnica proposta, foram conduzidos alguns experimentos preliminares. Os resultados obtidos mostram que o algoritmo conseguiu emparelhar, na maior parte dos programas, pelo menos 70% das funções otimizadas no nível O1. Esse número aumenta para 90% se forem consideradas apenas otimizações que não modificam o fluxo de controle do programa. Estes resultados também evidenciam que a abordagem proposta pode ser aplicada para emparelhar programas reais.

O restante deste artigo está organizado da seguinte forma: Seção 2 apresenta os conceitos de validação de tradução e árvore de dominância, importantes para o entendimento da solução; Seção 3 descreve a solução proposta e apresenta o algoritmo; Seção 4 apresenta os experimentos realizados nos *benchmarks* do conjunto de testes do compilador LLVM; Seção 5 apresenta uma revisão da literatura destacando trabalhos relacionados; Seção 6 conclui esse artigo destacando as contribuições oriundas da técnica proposta, bem como relacionando os trabalhos futuros.

2 Fundamentação Teórica

2.1 Validação de Tradução

Validação de tradução [1] consiste em adicionar, a cada tradução realizada pelo compilador, uma fase de validação que verifica se o código produzido pela tradução preserva a semântica do programa fonte. Contudo, provar que dois programas são semanticamente equivalentes é um problema indecidível. É fácil perceber que se fosse possível estabelecer tal equivalência, poder-se-ia determinar se um programa é ou não equivalente a um laço infinito. Sendo assim, determinar se dois programas são equivalentes resolveria o problema da parada [13]. Portanto, validadores de tradução utilizam diferentes técnicas e heurísticas para fornecer evidências de que uma tradução está ou não correta.

A abordagem introduzida por Pnueli *et al.* propõe que o processo de validação de tradução deve conter alguns ingredientes, como:

1. Um arcabouço semântico comum para representação do programa fonte e do programa alvo gerado pelo tradutor.
2. Uma formalização da noção de "implementação correta", baseada no arcabouço semântico.

3. Um método automático de prova, baseado em análise estática ou simulação, que verifique se um modelo do arcabouço semântico, representando o programa alvo, implementa corretamente um outro modelo, representando o programa fonte.

Diferentes arcabouços semânticos e métodos de prova são encontrados na literatura [2,3,4,5,6,7,8], sempre buscando minimizar a quantidade de falsos negativos, ou seja, minimizar a quantidade de falsos erros de tradução encontrados.

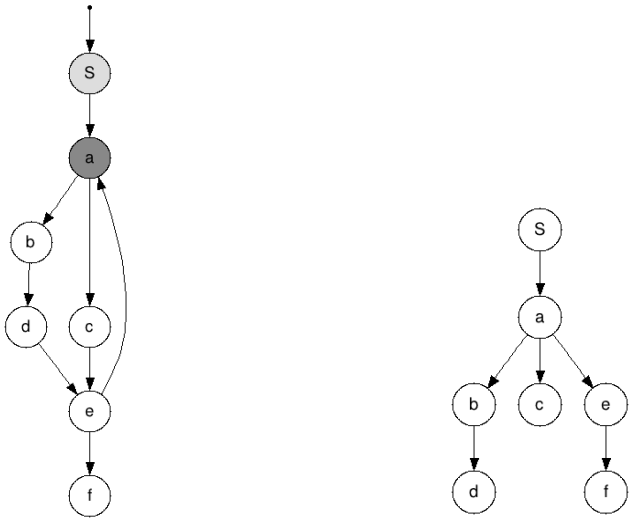
2.2 Árvore de Dominância

Conforme será descrito na Seção 3, o algoritmo proposto nesse artigo utiliza a *árvore de dominância* de cada função de um programa para construir árvores de chamadas de funções externas. O conceito de *dominância* é originário da Teoria dos Grafos e foi primeiramente introduzido por Prosser [10]. Atualmente, esse conceito é utilizado em diferentes áreas da computação com diferentes aplicações, como análise de uso de memória, computação de dependências de controle e computação do formato de Atribuição Estática Única (SSA, do inglês *Static Single Assignment*) [11]. Em particular, o conceito de dominância é fundamental para a construção dos compiladores atuais, visto que é aplicado na computação de SSA e em várias otimizações.

No contexto de grafos de fluxo de controle, diz-se que um nó d domina um outro nó n se todo caminho no grafo partindo do nó inicial S, ou nó raiz, até n passa por d. Um nó nesse grafo é um *bloco básico*, que consiste em um conjunto de instruções com apenas um ponto de entrada e um ponto de saída. Formalmente, podemos definir o conjunto de dominadores de um nó conforme na Equação 1. O nó d é dito *dominador imediato* do nó n (denotamos $idom(n)$), diferente de d, se d domina n mas não domina nenhum outro dominador de n. Ou seja, o dominador imediato de um nó é o seu dominador mais próximo. Figura 1a mostra os dominadores do nó e em um grafo de fluxo de controle, destacando o seu dominador imediato.

$$\begin{aligned} D(S) &= \{S\} \\ D(n) &= \{n\} \cup \bigcap_{p \in pred(n)}^{D(p)} \end{aligned} \tag{1}$$

A partir desses conceitos, é possível construir o que se chama de *árvore de dominância*. Uma árvore de dominância contém uma aresta (d, n) se o nó d é o dominador imediato do nó n. É possível formalizar as arestas da árvore de dominância pela fórmula $\{ (idom(n), n) \mid n \in V - \{S\} \}$, sendo V o conjunto de vértices do grafo. Dessa forma, é possível perceber que todos os nós pertencentes à subárvore de um nó d, isto é, seus nós descendentes, são dominados por d. A Figura 1b ilustra a árvore de dominância do grafo de fluxo de controle na Figura 1a.



(a) Grafo de Fluxo de Controle (b) Árvore de Dominância

Figura 1: Um exemplo de grafo de fluxo de controle, com os dominadores do nó *e*, e sua árvore de dominância.

3 Emparelhamento de Chamadas de Função

A abordagem para emparelhamento de chamadas de funções proposta nesse artigo se baseia em uma observação simples: compiladores otimizadores corretos não podem alterar a ordem em que as funções de bibliotecas externas são chamadas. Uma alteração desse tipo poderia modificar a semântica do programa, uma vez que existe a possibilidade de que uma chamada altere o comportamento de chamadas subsequentes, por um mecanismo ao qual o compilador não tenha acesso. Para tornar essa noção mais clara, a Figura 2 apresenta um exemplo elementar na linguagem C.

O exemplo contém duas funções: *set_needs_shutdown()* e *needs_shutdown()*. Como a implementação dessas duas funções se encontra em uma unidade de compilação diferente, um compilador otimizador é obrigado a assumir que a chamada a uma dessas funções pode afetar o comportamento da outra – como de fato ocorre. Em função disso, o compilador não pode alterar a ordem relativa das chamadas nem remover chamadas que não façam parte de código comprovadamente morto. Essa observação pode ser utilizada para encontrar um emparelhamento que associe pontos de programa antes e depois da execução de uma sequência de otimizações.

Infelizmente, não existe uma correspondência direta entre o número de chamadas a funções externas no código do programa e o número de vezes em que essas funções serão chamadas durante a execução. Além disso, otimizações como *desenlaço* (*loop unrolling*) [12] podem fazer com que uma única chamada no pro-

```
//powerplant.a
int flag = 0;

void set_needs_shutdown() {
    flag = 1;
}

int needs_shutdown() {
    return flag;
}

//main program
int main()
{
    //...
    if (may_explode) {
        set_needs_shutdown();
    }
    //...
    if (needs_shutdown()) {
        shutdown_powerplant();
    }
}
```

Figura 2: A função *set_needs_shutdown()* modifica o estado interno da biblioteca e, com isso, o resultado da chamada *needs_shutdown()*. Um compilador correto deve manter a ordem relativa dessas duas chamadas.

grama fonte seja substituida por várias chamadas, sem que haja modificação na semântica. Isso torna a formulação de uma estratégia de emparelhamento uma tarefa desafiadora: para emparelhar pontos de programa em geral, é necessário lidar com transformações como desenlaço e eliminação de código morto. Uma vez que o presente trabalho representa nossos esforços iniciais em direção a uma técnica de emparelhamento geral, assumiremos que o compilador não realiza otimizações que modificam o fluxo de controle. Como será mostrado na seção 4, mesmo sob essa hipótese ainda é possível parear a maior parte das chamadas que aparecem em programas reais.

Sob essa nova hipótese, a quantidade de chamadas a funções de bibliotecas presentes no código não pode mudar em consequência de uma transformação. Isso permite que o problema de pareamento seja formulado em termos de árvores de dominância de chamadas. Uma árvore de dominância de chamadas é uma representação compacta da ordem relativa em que chamadas a procedimentos acontecem. Se não ocorrerem erros nem transformações de fluxo de controle durante uma série de transformações, um programa P e sua versão modificada P' devem induzir árvores de dominância de chamadas $T(P)$ e $T(P')$ que sejam isomorficas.

Essa última observação é a base de nosso algoritmo de emparelhamento de programas. Ele consiste em duas etapas: (1) a construção de árvores de dominância de chamadas de função a partir de um programa e (2) a aplicação de um algoritmo linear de isomorfismo de árvores enraizadas para parear essas árvores. Seção 3.1 apresenta uma descrição mais formal dessas árvores e de como elas podem ser construidas. Seção 3.2 apresenta o algoritmo para isomorfismo de árvores

3.1 Árvores de Dominância de Chamadas a Funções Externas

A fim de construir uma estrutura de dados propícia para verificar a correspondência entre pontos de um programa, o algoritmo utiliza a árvore de dominância de cada função para determinar uma relação entre chamadas a funções externas. Considere uma função do programa, seu grafo de fluxo de controle, sua árvore de dominância DT correspondente e um conjunto S de funções externas. Considere ainda duas instruções I_1 e I_2 de chamadas às funções $f' \in S$ e $f'' \in S$, respectivamente, de maneira que não existe outra chamada a uma função externa entre I_1 e I_2. Na árvore gerada pelo algoritmo, o nó correspondente à instrução I_2 será filho do nó correspondente à instrução I_1 se:

- Ambas as instruções, I_1 e I_2 estão em um mesmo bloco básico e I_2 é uma instrução posterior a I_1.
- As instruções estão em blocos básicos diferentes B_1 e B_2, respectivamente, mas B_2 é filho de B_1 na árvore de dominância DT.

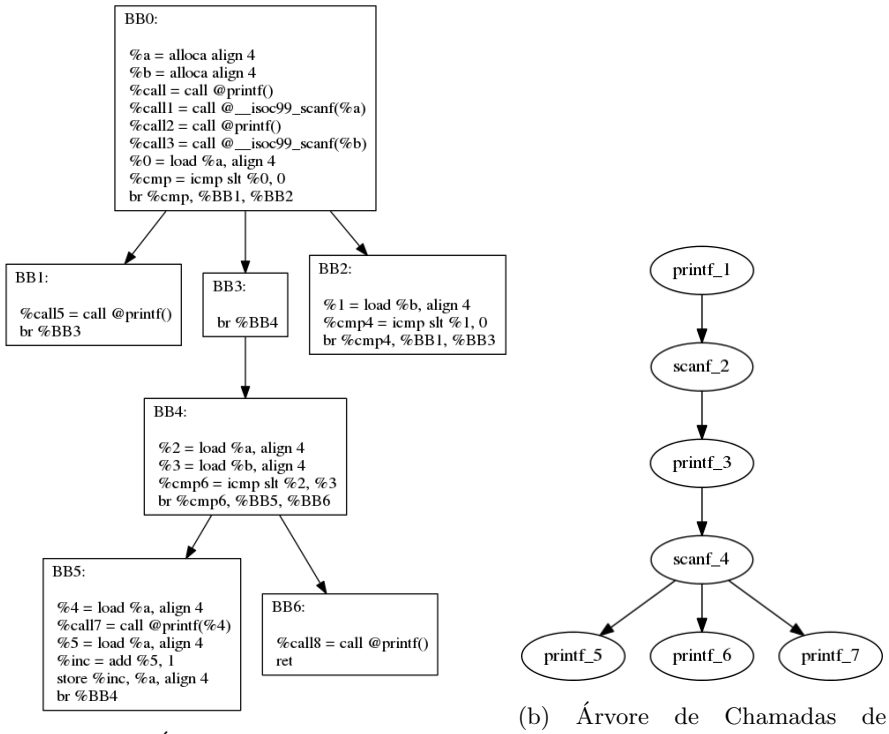
A Figura 3 ilustra a árvore de dominância de uma função escrita em C e a árvore de chamadas de função gerada pelo algoritmo.

Utilizar a árvore de dominância de uma função para guiar a construção da árvore de chamadas de funções é uma abordagem segura. Visto que um compilador correto não pode mudar a ordem relativa entre duas chamadas a funções externas, a árvore construída tem que permanecer a mesma se o fluxo de controle do programa não tiver sofrido modificações. O Algoritmo 1 constrói uma árvore com as propriedades citadas realizando uma busca em profundidade na árvore de dominância DT.

O custo assintótico do algoritmo é linear no número de instruções do programa. A construção da árvore consiste em uma busca em profundidade na árvore de dominância, o que é linear no número de blocos básicos. Além disso, cada instrução do bloco básico é visitada apenas uma vez. Como a inserção de um filho em um nó é realizada em $O(1)$, o custo do algoritmo pode ser dado como $O(n)$, sendo n o número de instruções do programa.

3.2 Isomorfismo de Árvores

Sejam $T_1 = (V_1, E_1, r_1)$ e $T_2 = (V_2, E_2, r_2)$ duas árvores de dominância de chamadas de funções com raízes r_1 e r_2, onde V_n e E_n são, respectivamente, os conjuntos de vértices e arestas da árvore T_n. Denotaremos por $f(v, T_i)$ a função



(a) Árvore de Dominância

(b) Árvore de Chamadas de Função

Figura 3: Um exemplo de árvore de dominância de programa escrito em C e a árvore de chamadas de função gerada.

associada a um nó $v \in T_i$ qualquer de uma das duas árvores T_i. Dizemos que essas duas árvores são *isomórficas* se houver uma bijeção $M \subseteq V_1 \times V_2$ que satisfaça:

- $f(u, T_1) = f(v, T_2)$ para todo par $(u, v) \in M$;
- Para todo arco $(u, u') \in E_1$ existe um arco $(v, v') \in E_2$ tal que $(u, v) \in M$ e $(u', v') \in M$.
- $(r_1, r_2) \in M$

O Algoritmo 2, baseado em [15], verifica se duas árvores enraizadas são isomórficas. Isso é feito atribuindo-se a cada subárvore de raiz v das duas árvores um identificador numérico I_v de tal modo que subárvores isomorficas recebam identificadores idênticos. A atribuição de identificadores utiliza um dicionário global I que mapeia listas da forma $[f(r, T_i), I_1, ..., I_k]$ contendo o procedimento associado a um nó e os identificadores de seus filhos em números inteiros.

Assumindo que o custo de acessar uma posição do dicionário seja $O(1)$, o algoritmo tem complexidade de tempo $O(|V_1| + |V_2|)$.

```
def GeraÁrvore(DT, S):
    Criar nova árvore T < V, E > com nó raiz R;
    T.V ← {R};
    T.E ← {};
    DFS(DT.raiz, R, T);
    return T
end
def DFS(noDT, folha, T):
    novaFolha = VisitaBlocoBásico(noDT.bloco, folha, T); for
    filho ∈ noDt.filho do
        DFS(filho, novaFolha);
    end
end
def VisitaBlocoBásico(blocoBasico, folha, T):
    for I ∈ blocoBasico.instrucoes do
        if I é uma chamada à função f e f ∈ S then
            Criar nó n com label f;
            T.E ← T.E ∪ (folha, n);
            folha ← n;
        end
    end
    return folha
end
```

Algoritmo 1: Geração da árvore de chamadas de funções externas.

```
Seja I um dicionário que mapeia listas em inteiros;
def NumerarNós(T, r):
    L ← [];
    for v ∈ T | (r, v) ∈ E(T) do
        L ← NumerarNós(T, v) :: L;
    end
    L = f(r, T) ::sorted(L);
    if L ∉ I then
        I(L) ← I.size();
    end
    return I(L)
end
def VerificarIsomorfismo(T1, r1, T2, r2):
    m ← NumerarNós(T1, r1);
    n ← NumerarNós(T2, r2);
    return m = n
end
```

Algoritmo 2: Isomorfismo de árvores rotuladas.

4 Experimentos

A fim de avaliar a aplicabilidade da técnica proposta nesse trabalho em programas reais, o algoritmo de emparelhamento programas foi implementado como um passo do compilador LLVM. Tendo em vista o objetivo dos experimentos, foram escolhidos *benchmarks* presentes no conjunto de testes do próprio compilador. O passo implementado foi executado sobre 649.505 linhas de código de 1.629 programas C e C++ desses *benchmarks* utilizando as otimizações dos níveis O1, O2 e O3 do LLVM. Para obter a variação da quantidade de funções emparelhadas em função das otimizações, cada execução do algoritmo adicionou uma nova otimização até que fosse executado todo o *pipeline* de um nível. Dessa forma, foi possível perceber que, à medida que as otimizações foram encadeadas, principalmente as que modificam o fluxo de controle do programa, o algoritmo emparelhou menos funções.

A Figura 4 apresenta um diagrama de caixa com os resultados obtidos da execução do algoritmo nos programas selecionados com o nível de otimização O1. É possível perceber que para maior parte das otimizações foi possível emparelhar mais de 60% das funções. Além disso, nota-se que à medida que são adicionadas otimizações que modificam o fluxo de controle do programa, o desvio padrão aumenta. Isso ocorre pois o algoritmo não lida de maneira especial com essas otimizações e por isso não é possível encontrar um isomorfismo entre as árvores construídas.

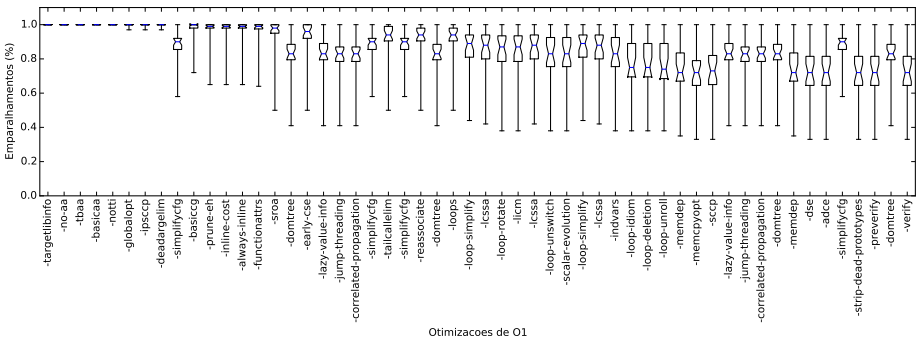


Figura 4: Diagrama de caixa: taxa de emparelhamentos por otimizações do nível O1.

A Figura 5 apresenta um gráfico de linha gerado apenas com as porcentagens médias de emparelhamento para cada otimização do nível O1. Já a Figura 6 apresenta a média de emparelhamentos para as otimizações do nível O2. Dessa forma, é possível constatar, conforme observado anteriormente, que a média de emparelhamentos para o nível O1 é superior a 70% para todas as otimizações,

porém há queda significativa utilizando o nível O2. Os dados obtidos nesse experimento permitiram concluir que a técnica proposta pode ser utilizada para emparelhar programas reais, resultando em uma boa taxa de emparelhamento mesmo em compiladores otimizantes.

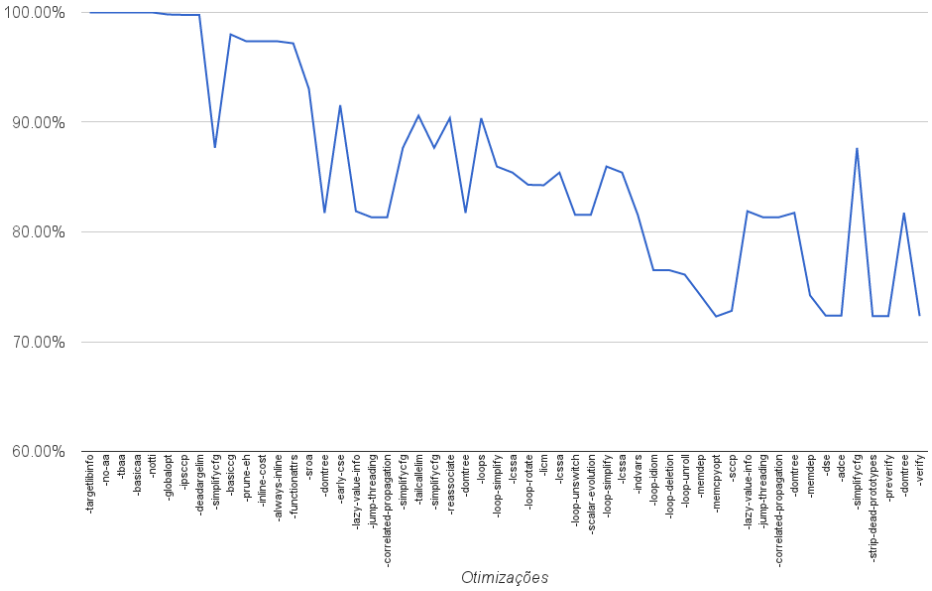


Figura 5: Gráfico de porcentagem média para otimizações do nível O1.

Também foram realizados experimentos para mensurar o tempo de execução do passo implementado. A Figura 7 exibe os resultados obtidos da execução dos experimentos como porcentagem em relação ao tempo total de execução. Todos os outros apresentam tempos de execução semelhantes e abaixo de 1%. Com isso, é possível perceber que, de fato, a técnica proposta não acrescenta grande sobrecarga ao processo de compilação.

5 Trabalhos Relacionados

Barrett *et al.* [2] propuseram uma ferramenta de Validação de Tradução denominada TVOC. A ferramenta recebe como entrada as representações fonte e alvo do programa na representação intermediária WHIRL (*Winning Hierarchical Intermediate Representation Language*). Então, é gerado um conjunto de condições de verificação que é submetido ao provador automático de teoremas CVC Lite [14]. Apesar de resolverem um conjunto de restrições de forma exata, provadores de teorema são computacionalmente caros e podem aumentar substancialmente o tempo de compilação se incorporados ao processo.

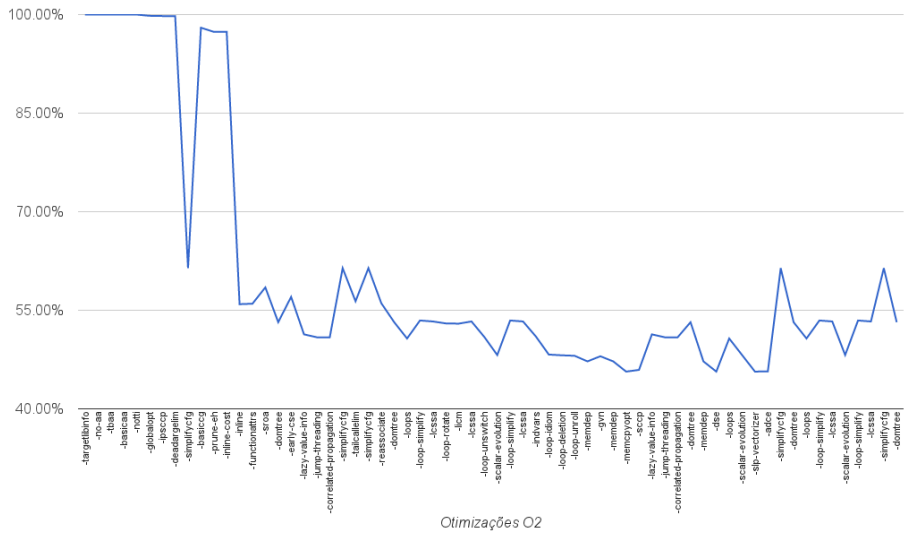


Figura 6: Gráfico de porcentagem média para otimizações do nível O2.

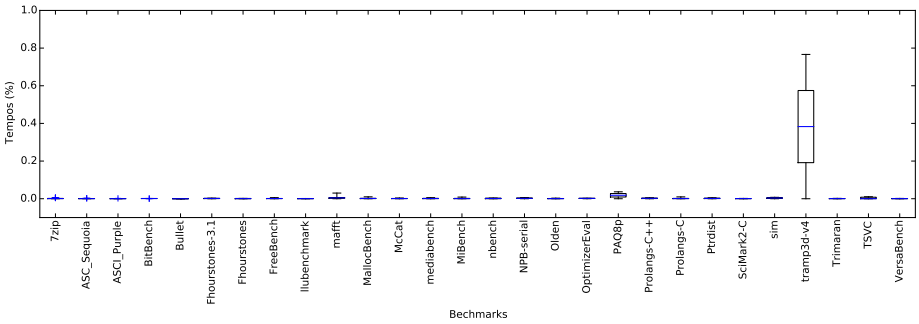


Figura 7: Gráfico de porcentagem de tempo de execução do passo implementado por *benchmark*.

Necula [5] implementou um validador de traduções para o compilador GCC 2.7 considerando um conjunto fixo de otimizações. O autor argumenta que uma das constribuições de seu trabalho está na evidência de que é possível implementar um validador de tradução, capaz de verificar a corretude de várias transformações realizadas por um compilador real, com o esforço tipicamente necessário para se implementar um passo de um compilador. A ferramenta proposta realiza avaliação simbólica. O objetivo é inferir uma relação entre as simulações de dois programas distintos. Para isso, são extraídas restrições da simulação de cada um dos programas a fim de verificar se eles são equivalentes. Embora a

técnica apresente bons resultados, ao adicionar algumas otimizações que modificam o fluxo de controle do programa, pode-se quebrar o conjunto de restrições.

Tate *et al.* [3] propuseram uma técnica para validação de tradução na qual um grafo de valores é gerado para cada função do programa original e sua versão otimizada. Então, é realizado um processo chamado de saturação de igualdade, adicionando termos equivalentes ao grafo. Após concluir a saturação, é utilizada uma heurística para obter o grafo que representa o programa final. Se o grafo obtido e o grafo gerado para a função após um conjunto de otimizações são iguais, conclui-se que as funções são equivalentes. Todavia, Tristan *et al.* [4], em seu trabalho, mostram que para a validação de tradução não se faz necessario o processo de saturação. Foi proposta uma técnica que consiste em normalizar o grafo de valores aplicando apenas as transformações que sabe-se seguramente que um otimizador aplicaria. Eles evidenciam que essa técnica pode ser mais eficiente em termos de tempo de execução.

Ainda, encontra-se na literatura várias ferramentas para verificar a corretude de traduções que instrumentam o compilador afim de obter as transformações realizadas [6,7,8]. Em geral, instrumentar o compilador traz resultados significativos, com baixo número de falsos negativos. Todavia, tal instrumentação geralmente requer um grande esforço na implementação e aumenta o tempo de uma compilação. Portanto, não é objetivo desse trabalho instrumentar o otimizador afim de verificar a preservação da semântica após uma sequência de otimizações.

6 Conclusão

Esse artigo apresentou uma nova técnica para emparelhamento de programas utilizando como âncora chamadas a funções de biblioteca, que pode ser utilizada como base para implementação de validadores de tradução em compiladores. O algoritmo proposto consiste, basicamente, em construir árvores de chamadas a funções de biblioteca visitando as instruções na ordem de uma busca em profundidade na árvore de dominância de cada função do programa. Uma vez que se possui as árvores para uma função antes e depois de um conjunto de otimizações, verifica-se o isomorfismo entre elas. Essa técnica foi implementada como um passo do compilador LLVM e executada sobre *benchmarks* do conjunto de testes do compilador.

Constatou-se que ao utilizar otimizações que não modificam o fluxo de controle do programa, foi possível, em geral, emparelhar mais de 90% das funções. Porém, a porcentagem de funções emparelhadas ao utilizar otimizações mais agressivas, principalmente as dos níveis O2 e O3, ainda é baixo para alguns programas. Apesar disso, conclui-se que a técnica proposta pode ser utilizada de maneira efetiva para emparelhar programas reais. Nesse contexto, acredita-se ser possível utilizar esse emparelhamento como âncora para executar análises mais poderosas, como propagação simbólica de intervalos, nos parâmetros das chamadas de função para implementar validadores de tradução mais robustos.

Como trabalhos futuros, pretendemos encontrar maneiras de lidar com algumas otimizações que modificam o fluxo de controle do programa sem adicionar um comportamento exponencial ao algoritmo. Eliminação de código morto, por exemplo, pode ser bastante problemática, visto que uma chamada de função pode ser removida e as árvores não serão mais isomórficas. Neste caso, seria preciso verificar para cada nó, a partir do qual se perdeu a correspondência, qual deles resulta no melhor emparelhamento da subárvore (explosão exponencial). Ainda, poder-se-ia utilizar uma técnica semelhante à de Tristan *et al.* [4] e realizar transformações no grafo original de acordo com o que o otimizador provavelmente faria.

Referências

[1] Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998).

[2] Barrett, C. W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.: TVOC: A translation validator for optimizing compilers. In Computer Aided Verification. Lecture Notes in Computer Science, vol. 3576, pp. 291–295. Springer (2005).

[3] Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In 36th Principles of Programming Languages (POPL), pp. 264–276. ACM (2009).

[4] Tristan, J.-B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In PLDI '11: Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (2011).

[5] Necula, G. C.: Translation validation for an optimizing compiler. In Programming Language Design and Implementation, pp. 83–95. ACM Press (2000).

[6] Kanade, A., Sanyal, A., Khedker, U.: A PVS based framework for validating compiler optimizations. In 4th Software Engineering and Formal Methods, pp. 108–117. IEEE Computer Society (2006).

[7] Rinard, M., Marinov, D.: Credible compilation with pointers. In Proceedings of the Run-Time Result Verification Workshop, Trento (July 2000).

[8] Zuck, L. D., Pnueli, A., Fang, Y., Goldberg, B.: VOC: A translation validator for optimizing compilers. Electr. Notes Theor. Comput. Sci., 65(2) (2002).

[9] Blume, W.,Eigenmann, R.: Symbolic range propagation. Proceedings of the 9th International Parallel Processing Symposrum (April 1995).

[10] Prosser, R. T.: Applications of Boolean matrices to the analysis of flow diagrams. AFIPS Joint Computer Conferences: Papers presented at the December 1–3, eastern joint IRE-AIEE-ACM computer conference (Boston, MA: ACM): pp. 133–138 (1959).

[11] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N.,Zadeck, F. K.: An efficient method for computing static single assignment form. In Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 25-35, Austin, Texas, January 11-13. ACM SIGACT and SIGPLAN (1989).

[12] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA (2006).

[13] Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, Series 2, 42, pp. 230–265 (1936).

[14] Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker. In Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004).
[15] Aho, A. V., Hopcroft, J. E., Ullman, J. D.: The design and analysis of computer algorithms, Addison-Wesley Longman Publishing Co., Inc., Boston, MA (1974).