Um Algoritmo para Emparelhamento de Chamadas de Função

Francisco Demontiê dos Santos Junior, 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. Depurar 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. Esta 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 passe 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 of the compilers apply a series of optimizations in the source program without ensure that they preserves semantics. Debugging optimizations is difficult. In this context, matching library function calls may be useful. This approach is fast, reliable (compilers cannot optimize function calls without having access to the function body), may catch problems easily and give anchor for stronger analysis (such as symbolic range analysis). In this work we present an algorithm that verifies isomorphism between external function call trees built from the dominators tree of each function in the program. This algorithm was implemented as a LLVM pass and executed on the LLVM test-suite's benchmarks. Using the -O1 level of optimization, our approach matched at least 75% of the functions (and more then 90% for optimizations which do not change the programs' control flow). We conclude that this technique can be widely applicable to real programs.

1 Introdução

Testar se um compilador sempre gera código correto é uma tarefa difícil. Alternativo a isso, A. Pnueli et al[1] propuseram uma técnica chamada de 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 propuseram a ferramenta TVOC, que gera condições de verificação e utiliza um provador automático de teoremas para verificar a equivalência entre os programas. Em contrapartida, uma série de trabalhos 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 final das transformações os dois grafos são iguais, diz-se que os programas são equivalentes.

Neste artigo é proposta 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, no sentido de que compiladores não podem realizar otimizações em funções externas (por não possuírem acesso ao corpo dessas funções) e, portanto, elas precisam ser mantidas pelo otimizador. A inserção dessa técnica no processo de otimização não acrescenta grande *overhead*. Além disso, é possível encontrar problemas facilmente e pode servir como âncora para análises mais poderosas, como, por exemplo, propagação simbólica de intervalos[6].

O algoritmo apresentado 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(). Dessa forma, modificações na ordem em que as chamadas aparecem nas duas árvores podem sugerir um problema durante a fase de otimização.

O algoritmo para emparelhamento de programas foi implementado como um passe do compilador LLVM e executado sobre XX linhas de código de XX programas dos *benchmarks* encontrados no conjunto de testes do compilador. Para cada uma das funções do programa, o passe foi executado usando as otimizações dos níveis O1, O2 e O3 de forma incremental. O objetivo dos experimentos

realizados foi averiguar a aplicabilidade da técnica proposta para encontrar a correspondência entre chamadas a funções externas. Nossos resultados mostraram que o algoritmo conseguiu emparelhar, maior parte dos programas, pelo menos 70% das funções otimizadas no nível O1. Esse número aumenta para 90% se considerarmos apenas otimizações que não modificam o fluxo de controle do programa. Com isso, foi possível concluir que a técnica pode ser aplicada para emparelhar programas reais.

2 Fundamentação Teórica

- Translation validation
- Dominator tree

3 Solução

- Tree isomorphism
- Dealing with tricky optimizations

Primeiramente, são construídas árvores de chamadas de funções para o programa antes e depois das otimizações. Cada árvore é construída a partir da árvore de dominância de uma função do programa, sendo a análise, portanto, intraprocedural. A árvore de dominância é percorrida na ordem de uma Busca em Profundidade e cada instrução do bloco básico é visitada para obter-se as chamadas de funções. Quando uma nova chamada de função é encontrada, um novo nó é adicionado à árvore como filho do último nó visitado no ramo da DFS.

Construídas as árvores de chamadas de função para um programa, antes e depois de um conjunto de otimizações, verifica-se isomorfismo entre elas. O algoritmo utilizado para tal consiste em ... (pedir a Filipe para descrever).

Inserir pseudo-código.

4 Experimentos

5 Trabalhos Relacionados

A fim de facilitar a tarefa de testar o código gerado por compiladores, foi proposta a técnica de Validação de Tradução[1]. Essa técnica consiste em verificar a equivalência entre o programa fonte e o programa gerado pelo compilador a cada tradução, em detrimento de provar que o compilador sempre gera código corretamente. A partir de então, validadores de traduções utilizando diferentes técnicas e algoritmos foram propostos. Barrett et al propuseram a ferramenta TVOC[2]. 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.

Apesar de resolverem o 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.

Necula[5] implementou um validador de traduções para o compilador GCC 2.7 para um conjunto fixo de otimizações. Uma contribuição do trabalho foi o argumento 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 passe 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 modifiquem 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 eles realizam 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 otimizações são iguais, conclui-se que as funções são equivalentes. Todavia, Tristan et al[4] mostraram que para 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 mostraram 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 [pegar referências do artigo de Tristan]. 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

Nesse artigo é apresentada 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 largura 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. Implementamos essa técnica como um passe do

compilador LLVM e executamos sobre *benchmarks* do conjunto de testes do compilador.

Verificamos que ao utilizar otimizações que não modificam o fluxo de controle do programa, conseguimos, em geral, emparelhar mais de 90% das funções. Porém, a porcentagem de funções emparelhadas quando utilizamos otimizações mais agressivas (principalmente as dos níveis O2 e O3) ainda é baixo para alguns programas. Apesar disso, concluímos que a técnica proposta pode ser utilizada de maneira efetiva para emparelhar programas reais. Nesse contexto, acreditamos 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 modifiquem 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. Assim, 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, poderia-se 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] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In Computer Aided Verification, volume 3576 of Lecture Notes in Computer Science, pages 291–295. Springer, 2005.

[3] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In 36th Principles of Programming Languages (POPL), pages 264–276. ACM, 2009.

[4] J.-B. Tristan, P. Govereau, and G. Morrisett. 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] George C. Necula. Translation validation for an optimizing compiler. In Programming Language Design and Implementation 2000, pages 83–95. ACM Press, 2000.

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