Um Algoritmo Inter-Procedural para Análise de Largura de Variáveis

Douglas do Couto Teixeira[1], Fernando Magno Quintão Pereira[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Av. Antônio Carlos, 6627 – 31270-010 – Belo Horizonte – Minas Gerais – MG – Brasil

{douglas, fpereira}@dcc.ufmg.br

Abstract. *During this project we have developed an inter-procedural range analysis algorithm that scales up to programs with millions of assembly instructions. Contrary to many previous techniques, we handle comparisons between variables without resorting to expensive algorithms. We achieve path sensitiveness by using Bodik's Extended Static Single Assignment form as the intermediate representation. We show that by processing the strongly connected components that constitute the graph of dependences between variables in topological order we not only gain in time, but also in precision. We have implemented our technique in LLVM, an industrial strength compiler, and have been able to process over 4 million assembly instructions in a few seconds.*

Resumo. *Durante este projeto foi desenvolvido um algoritmo inter-procedural que é capaz de processar programas com milhões de instruções* assembly. *Ao contrário de muitos trabalhos anteriores, nosso algoritmo trata comparações entre variaveis sem recorrer a algoritmos custosos. Nós obtemos sensibilidade à fluxo de execução usando como representação intermediária o formato e-SSA (Extended Static Single Assignment) descrito por Bodik. Nós também mostramos que processar os componentes fortemente conexos do grafo em ordem topológica não só reduz o tempo de execução do programa, mas também aumenta sua precisão. Nós implementamos nossa técnica em LLVM, um compilador industrial, e pudemos processar cerca de quatro milhões de instruções* assembly *em poucos segundos.*

1. Introdução

Compiladores usam análise de largura de variáveis para tentar inferir o intervalo de valores que as variáveis discretas podem assumir durante a execução do programa. Essa análise tem sido estudada desde a década de 70, quando Cousot [Cousot and Cousot 1977] usou-a como exemplo de interpretação abstrata de programas. Desde então, ela tem sido usada para vários fins. Por exemplo, essa análise possibilita compiladores otimizantes removerem do texto do programa testes de overflow desnecessários [Souza et al. 2011] e testes de limite de arranjo desnecessários [Bodik et al. 2000]. Além disso, análise de largura de variáveis pode ser usada em predição estática de desvios condicionais [Patterson 1995], para detectar possíveis *overflows* [Simon 2008, Wagner et al. 2000], e até mesmo em síntese de *hardware* [Cong et al. 2005, Mahlke et al. 2001].

Dada a importância de análise de largura de variáveis e o tempo que ela tem sido estudada, não é surpresa que a literatura seja repleta de trabalhos descrevendo variações algorítmicas dessa análise [Gawlitza et al. 2009, Mahlke et al. 2001, Stephenson et al. 2000, Su and Wagner 2005]. Entretanto, estes trabalhos ou não provêem evidências experimentais de que o algoritmo usado por eles é capaz de analisar grandes programas [Stephenson et al. 2000] ou implementam somente uma forma muito simples de análise de largura de variáveis [Oh et al. 2012].

Este trabalho conclui um esforço de dois anos para produzir uma implementação sólida e escalável de análise de largura de variáveis. Nosso primeiro esforço nessa direção foi o trabalho de Teixeira e Pereira [Teixeira and Pereira 2011], que é baseado em um sistema de restrições proposto por Su e Wagner [Su and Wagner 2005]. Entretanto, o algoritmo descrito por Su e Wagner não é capaz de lidar com comparações entre variáveis e possui complexidade cúbica no tamanho do programa. Por causa disso, decidimos projetar e implementar um algoritmo de análise de largura de variáveis que não tenha essas limitações.

Este documento fornece uma breve descrição do algoritmo que nós desenvolvemos e provê também dados experimentais que justificam nossas decisões na elaboração do algoritmo. Nós implementamos esse algoritmo em LLVM [Lattner and Adve 2004] e usamos nossa implementação do algoritmo para analisar todo o arcabouço de testes do LLVM e os programas escritos em C presentes do SPEC CPU2006 [Henning 2006]. Nós mostramos evidências empíricas de que a nossa implementacão é rápida: ela é capaz de analisar todo o código fonte do GCC em cerca de 15 segundos, por exemplo. Além disso, nossa implementação é precisa: nós obtemos resultados comparáveis aos mostrados por Stephenson *et al.* [Stephenson et al. 2000] e, ao contrário do trabalho de Stephenson, nós não assumimos que o programa está correto.

1.1. Contribuições

Nesta seção nós listamos as contribuições do nosso trabalho. São elas:

- Nós implementamos nosso algoritmo em um compilador usado na indústria, o LLVM. Acreditamos que a nossa implementação seja uma das mais eficientes e precisas implementações de análise de largura de variáveis presente em um compilador industrial;
- Nós provemos extensa documentação sobre o nosso algoritmo, incluindo uma galeria de exemplos e casos de uso da nossa análise. Nossa implementação, bem como toda a documentação, é acessivel ao público através do site: *http://code.google.com/p/range-analysis/*;
- Atualmente, existem pessoas de vários países do mundo usando nossa implementação de análise de largura de variáveis para os mais diversos fins;
- Nós temos feito contribuições científicas com esse projeto: publicamos um artigo no *Simpósio Brasileiro de Linguagens de Programação*, em 2011 [Teixeira and Pereira 2011].
- Nós conseguimos incentivo do Google para esse projeto, através de um programa chamado *Google Summer of Code* [1].

[1] `http://code.google.com/soc/`

Contribuições específicas do aluno: o aluno de iniciação científica, primeiro autor deste documento, trabalhou neste projeto desde o começo e contribuiu em todas as etapas do seu desenvolvimento. O aluno é o autor do artigo sobre análise de largura de variáveis no Simpósio Brasileiro De Linguagens de Programação em 2011 [Teixeira and Pereira 2011]. Em uma fase posterior do projeto, o aluno foi fundamental para a especificação e implementação do algoritmo interprocedural de análise de largura de variáveis.

2. Algoritmo

Nesta seção descrevemos as etapas do nosso algoritmo de análise de largura de variáveis. Este algoritmo consiste, basicamente, de quatro passos. Primeiro, nós convertemos o programa para o formato e-SSA; isso nos permite extrair restrições (constraints) do código-fonte mais facilmente. Em seguida, lemos o código-fonte do programa, instrução por instrução, e extraímos as restrições relacionadas às variáveis inteiras. Após isso, construímos um grafo de dependência a partir do conjunto de restrições obtido anteriormente. E finalmente, resolvemos o sistema de restrições aplicando diferente operadores ao grafo de restrições até encontrarmos um ponto fixo.

2.1. Representação Intermediária

Atualmente, a maioria dos compiladores utiliza como representação intermediária o formato SSA (*Static Single Assignment Information*) [Cytron et al. 1991]. A Figura 1 (a) mostra um programa em uma linguagem de alto-nível, o mesmo programa em SSA (b), as restrições extraídas do programa (c) e o resultado do nosso algoritmo de análise de largura de variáveis aplicado a esse programa (d). Em (d), cada $I[V] = [...]$, onde V é uma variável do programa, nos diz o intervalo associado a V obtido pela nossa análise. Por exemplo, $I[I_1] = [0, +\infty]$ nos diz que todos os possíveis valores da variável i_1 estão contidos no intervalo $[0, +\infty]$. Entretanto, o resultado do algoritmo não é muito preciso nesse caso porque ele não é capaz de extrair informações dos desvios condicionais do programa. Mostraremos posteriormente como uma representação intermediária mais adequada melhora significativamente a precisão do nosso algoritmo, sem adicionar um tempo considerável em sua execução.

A informação contida nos desvios condicionais pode ser usada para fazer com que o nosso algoritmo seja mais preciso. Com essas informações, nós podemos dizer, por exemplo, que $J_0 \subseteq [-\infty, 100]$. Para obter a informação dos desvios condicionais, nós precisamos de outra representação intermediária para os programas.

2.2. Escolhendo Outra Representação Intermediária

A solução que encontramos para o programa na Figura 1 é imprecisa porque nós não levamos em consideração os desvios condicionais. Esses desvios nos fornecem informações úteis sobre o intervalos de algumas variáveis, mas somente em pontos específicos do programa. Por exemplo, dado o teste $(k_1 < 100)$?, na Figura 1(b), nós sabemos que $[k_1] \sqsubseteq [-\infty, 99]$ onde quer que a condição seja verdadeira. Para obter este tipo de informação, nós precisamos dividir a linha de vida da variável k_1 logo após o desvio condicional. Com isso, criamos duas novas variáveis, uma no caminho do programa onde a condição é verdadeira, e outra onde ela é falsa. Existe uma representação intermediária proposta por Bodik *et al.* [Bodik et al. 2000] que realiza essa divisão na linha de vida das

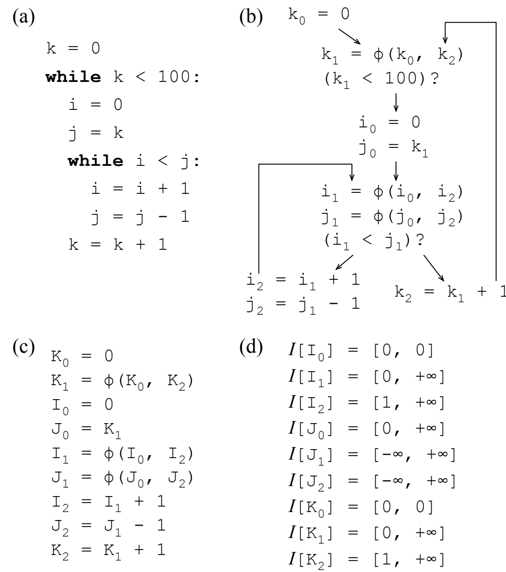


Figura 1. Programa no formato SSA

variáveis: o e-SSA *(Extended Static Single Assignment form)*. A Figura 2 mostra o programa da Figura 1 no formato e-SSA. Nas seções seguintes mostraremos que essa nova representação intermediária aumenta consideravelmente a precisão do nosso algoritmo.

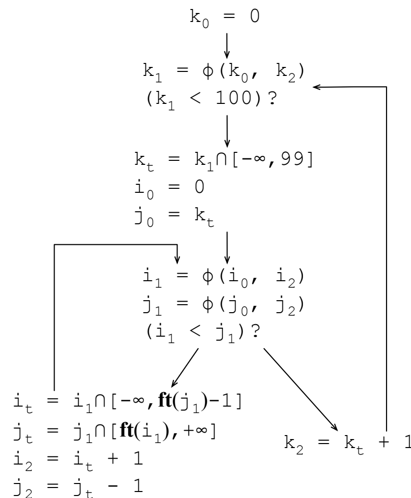


Figura 2. (a) Programa no formato e-SSA.

2.3. Construindo o Grafo de Dependências

Dado um programa no formato e-SSA, nós construímos um grafo para representar as restrições extraídas do códifo-fonte do programa. Esse grafo é uma variação do *Grafo de Dependência do Programa*, descrito por Ferrante [Ferrante et al. 1987]. Para cada variável V, criamos um vértice variável N_v, e para cada restrição C, criamos um vértice restrição N_c. Adicionamos uma aresta de N_v para N_c se V é usada em C e adicionamos

uma aresta de N_c para V se a restrição C define V. A complexidade da construção do grafo de dependências é linear no número de instruções do programa.

A Figura 3 mostra o grafo de dependência do programa mostrado na Figura 2. Note que, se existe uma comparação entre duas variáveis, nós colocamos o rótulo (**ft**), indicando que o valor de uma variável depende do valor de outra, mas esse valor será obtido posteriormente.

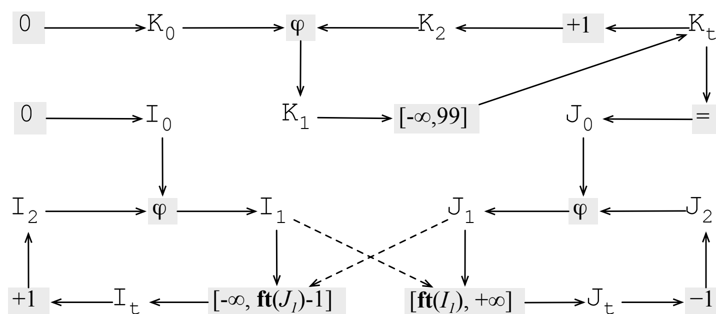


Figura 3. O grafo de dependênicas em formato e-SSA.

2.4. Resolvendo o Grafo de Dependências

Nesta seção descrevemos a etapa principal do nosso algoritmo: como resolvemos o grafo de dependências e encontramos os intervalos das variáveis inteiras do programa. Essa etapa do algoritmo é dividida em quatro partes: (i) encontrar e reduzir os componentes fortemente conexos do grafo; (ii) alargar o intervalo das variáveis; (iii) colocar os valores corretos nas restrições obtidas nos desvios condicionais, e (iv) estreitar o intervalo das variáveis tanto quanto possível. A seguir descrevemos detalhadamente cada uma dessas etapas.

2.4.1. Encontrando os Componentes Fortemente Conexos

Com o objetivo de reduzir o tempo de execução da nossa análise, usamos uma técnica de divisão e conquista para dividir o problema original em sub-problemas. Para isso, encontramos todos os componentes fortemente conexos (SCCs) do grafo e reduzimos cada SCC a um único vértice. Nesse ponto, o grafo é um DAG (*Grafo Acíclico Direcionado*). Após isso, nós ordenamos esse grafo topologicamente. As seguintes etapas do algoritmo são aplicadas a cada SCC do grafo, em ordem topológica.

2.4.2. Alargando o Intervalo das Variáveis

O primeiro passo do nosso algoritmo é determinar como os valores das variáveis do programa mudam ao longo do tempo. Para isso, usamos o operador de alargamento proposto por Cousot e Cousot [Cousot and Cousot 1977, p.247]. Os estados possíveis para o intervalo de uma variável são: (i) não muda; (ii) cresce em direção a $+\infty$; (iii) cresce em direção a $-\infty$; (iv) cresce em ambas as direções.

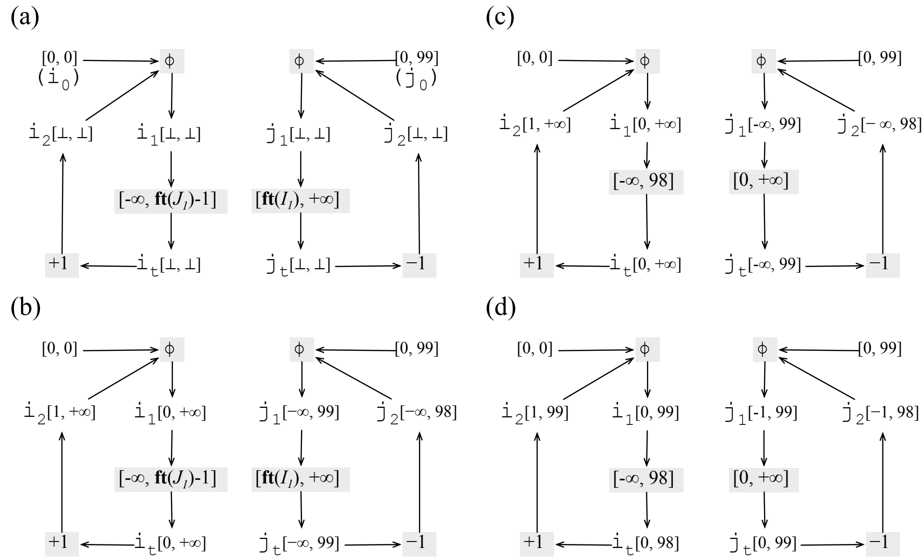


Figura 4. Quatro estágios do último SCC da Figura 4. (a) Imediatamente antes de aplicar os operadores de Cousot. (b) Depois de alargar o intervalo das variáveis. (c) Depois de corrigir as interseções. (d) Depois de estreitar o intervalo das variáveis.

2.4.3. Corrigindo as Interseções

Os intervalos encontrados quando alargamos os intervalos das variáveis nos dizem quais variáveis têm limites fixos. Podemos usar esses limites para corrigir as interseções que dependem de valores futuros, $\mathbf{ft}(V)$. Mas para isso, precisamos antes alargar os intervalos dessas variáveis.

As interseções que dependem de valores futuros ocorrem por causa de desvios condicionais envolvendo duas variáveis. Convém dizer que nós não recorremos a algoritmos custosos para resolver essas interseções. A complexidade desta etapa do algoritmo é linear no número de vértices do grafo de dependências.

2.4.4. Estreitando o Intervalo das Variáveis

A fase de alargamento do intervalo das variáveis associa valores conservativos ao intervalo de cada variável. A última etapa do nosso algoritmo consiste em estreitar esses intervalos tanto quanto possível. Nós realizamos tal processo usando o operador de estreitamento proposto por Cousot e Cousot [Cousot and Cousot 1977, 248]. A complexidade desta etapa do algoritmo é quadrática no número de vértices do grafo de dependências.

A Figura 4 mostra as etapas do nosso algoritmo aplicado ao programa da Figura 1. E a Figura 5 mostra o resultado do nosso algoritmo aplicado ao mesmo programa convertido para o formato e-SSA. Como podemos ver na Figura 5, o resultado que obtemos é muito mais preciso do que o mostrado anteriormente, quando o programa estava em SSA.

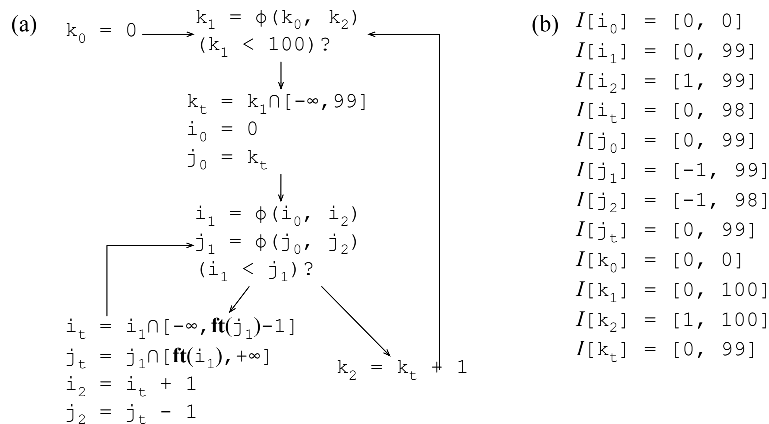


Figura 5. (a) Programa da Figura 2. (b) Resultado obtido pelo nosso algoritmo.

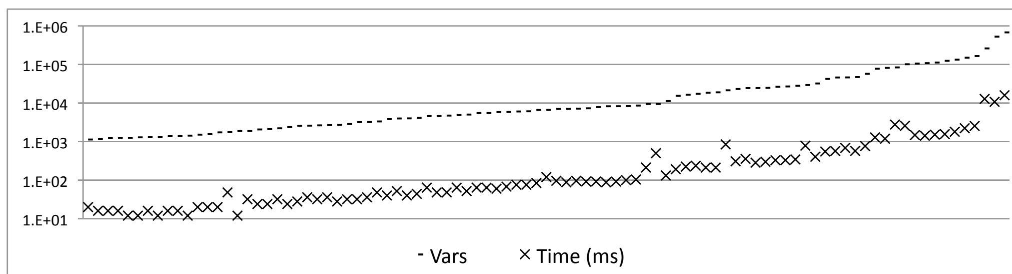


Figura 6. Correlação entre o tamanho dos programas (medidos em número de vértices do grafo de dependências) e o tempo de execução da análise (ms). Coeficiente de determinação = 0.967.

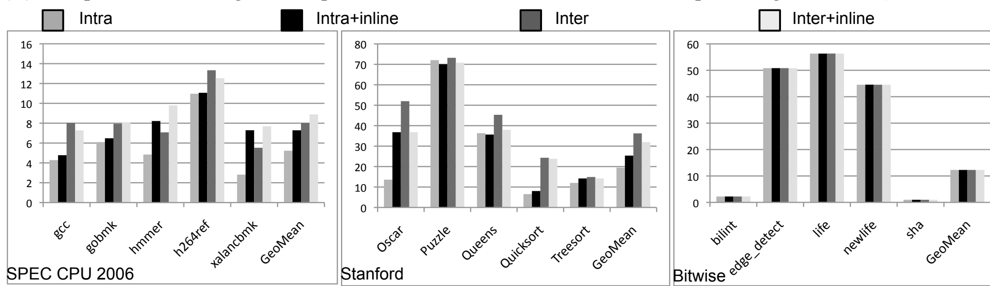
3. Experimentos

Nesta seção nós mostramos os resultados obtidos pela nossa análise ao executá-la nos programas presentes no arcabouço de testes do LLVM e nos programas presentes no SPEC CPU2006.

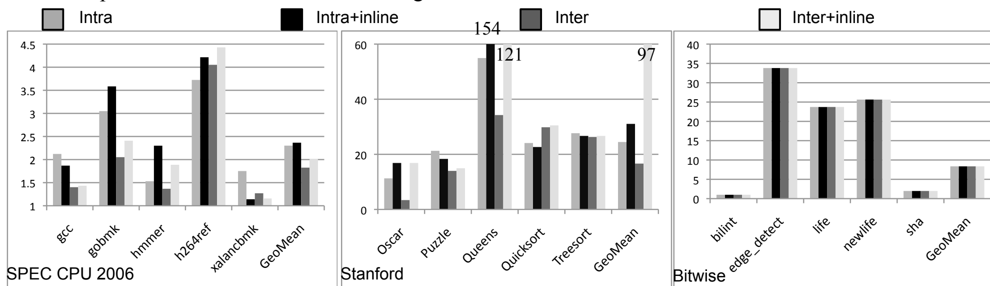
Complexidade Assimptótica A Figura 6 provê uma comparação visual entre o tempo de execução do nosso algoritmo e o tamanho dos programas usados como entrada. Os dados exibidos correspondem aos 100 maiores *benchmarks* em nosso arcabouço de testes. O tamanho dos benchmarks foi medido de acordo com o númedo de vértices no grafo de dependência. Cada ponto x no gráfico corresponde a um benchmark. Nós fomos capazes de analisar o menor dos benchmarks nesse conjunto de testes `Prolangs-C`, cujo grafo de dependências possui 1.131 vértices, em 20ms. E demoramos 15,91 segundos para analisar nosso maior benchmark, `403.gcc`, que possui 1.266.273 instruções *assembly*, e cujo grafo de dependências possui 679.652 vértices. Para esse conjunto de dados, o coeficiente de determinação é de 0.967, o que provê um forte indício que a complexidade assimptótica de nosso algoritmo é linear, embora em teoria essa complexidade seja quadrática.

Impacto da análise inter-procedural na precisão do algoritmo. A Figura 7(Parte de cima) mostra que a análise inter-procedural aumenta moderadamente a precisão do algoritmo. Nessa figura, nós mostramos resultados para os cinco maiores benchmarks em três

(A) O impacto da análise global na precisão de nossos resultados. Barras são percentagem de redução em bits.



(B) O Impacto do formato e-SSA na precisão de nossos resultados. Barras são a razão entre a precisão usando e-SSA e a precisão usando o formato SSA original.



(C) O impacto do número de iterações antes de aplicar alargamento na precisão de nossos resultados.

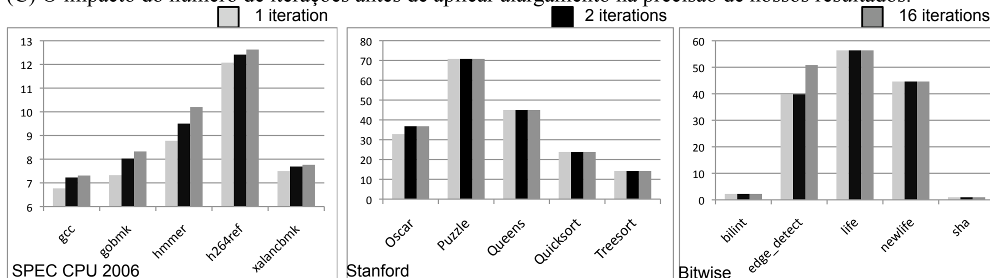


Figura 7. Impacto de diferentes decisões de implementação na precisão da análise.

diferentes categorias: SPEC CPU2006, os benchmarks de *Stanford* [2] e os benchmarks usados por Stephenson [Stephenson et al. 2000], chamados *Bitwise*. Nós medimos a precisão da nossa análise pelo número médio de bits das variáveis do programa que ela nos permite reduzir. Nossos resultados para os programas do SPEC CPU2006 são desapontadores: em média, a versão intra-procedural da nossa análise reduz 5.23% do número de bits necessários para armazenar as variáveis dos programas. Nós também implementamos uma versão inter-procedural do algoritmo. Usando a versão inter-procedural e fazendo *inline* das funções para simular sensitividade ao contexto, o número de bits reduzidos aumenta para 8.89%. Uma inspeção manual dos programas do SPEC mostra que este resultado é esperado: aqueles programas manipulam arquivos todo tempo e o código-fonte deles não contém muitas constantes, o que o que faria com que nossa análise pudesse inferir melhor os intervalos.

[2] `http://classes.engineering.wustl.edu/cse465/docs/BCCExamples/stanford.c`

Entretanto, nossa análise possui um desempenho muito melhor nos benchmarks numéricos. Em média, a versão inter-procedural do algoritmo reduz em 36.24% o número de bits necessários para armazenar as variáveis. Esse número é melhor do que o que nós obtemos quando realizamos *inlining* (31.97%), porque, em geral, as funções desses programas são chamadas uma única vez.

Finalmente, para os programas do benchmark Bitwise, nós obtemos uma redução média de 12.27%. Entretanto, essa média é baixa por causa de dois programas do benchmark: `edge_detect` e `sha`, que não possibilitam redução no número de bits necessários para armazenar suas variáveis. Os programas presentes no Bitwise foram implementados por Stephenson em *et al.* [Stephenson et al. 2000] com o objetivo de validar sua implementação de análise de largura de variáveis. Nossos resultados são comparáveis com os obtidos por eles. Os programas no Bitwise contêm somente a função `main`, por isso, diferente versões do algoritmo encontram os mesmos resultados.

O impacto da representação e-SSA na precisão. A Figura 7(Meio) mostra que converter os programas para e-SSA aumenta dramaticamente a precisão da nossa análise. Se nós convertermos os programas de SSA, que é a representação intermediária usada em LLVM para o formato e-SSA, em alguns casos – `Stanford.queens`, por exemplo – a precisão do nosso algoritmo aumenta 154 vezes. Essa diferença impressionante se deve ao fato de a representação SSA não prover informação necessária para que nossa análise tire proveito dos desvios condicionais.

4. Conclusão

Com este trabalho nós concluímos um esforço de dois anos para prover uma implementação escalável e precisa de análise de largura de variáveis. Acreditamos que alcançamos esses objetivos: nós usamos nossa implementação do algoritmo para analisar todo o arcabouço de testes do LLVM e os programas escritos em C presentes do SPEC CPU2006 – ao todo mais de dois milhões de linhas de código – e conseguimos fazer isso em cerca de três minutos. Além disso, este trabalho tem aberto novas fronteiras em termos de contribuição científica. Atualmente nós temos vários parceiros de pesquisa usando nossa implementação de análise de largura de variáveis em vários países do mundo. Nossa implementação está disponível em `http://code.google.com/p/range-analysis/`.

Referências

Bodik, R., Gupta, R., and Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM.

Cong, J., Fan, Y., Han, G., Lin, Y., Xu, J., Zhang, Z., and Cheng, X. (18-21 Jan. 2005). Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856–861.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349.

Gawlitza, T., Leroux, J., Reineke, J., Seidl, H., Sutre, G., and Wilhelm, R. (2009). Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. (2001). Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371.

Oh, H., Heo, K., Lee, W., Lee, W., and Yi, K. (2012). Design and implementation of sparse global analyses for c-like languages. In *PLDI*, page to appear. ACM.

Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM.

Simon, A. (2008). *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 1th edition.

Souza, M. R. S., Guillon, C., Pereira, F. M. Q., and da Silva Bigonha, M. A. (2011). Dynamic elimination of overflow tests in a trace compiler. In *CC*, pages 2–21.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

Su, Z. and Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138.

Teixeira, D. and Pereira, F. M. Q. (2011). The design and implementation of a non-iterative range analysis algorithm on a production compiler. In *SBLP*, pages 45–59. SBC.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 3–17. ACM.