

**ELIMINAÇÃO DE TESTES DE OVERFLOW PARA
COMPILADORES DE TRILHAS**

RODRIGO SOL

: MARIZA ANDRADE DA SILVA BIGONHA
: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

© 2010, Rodrigo Sol.
Todos os direitos reservados.

S684e Sol, Rodrigo
Eliminação de Testes de Overflow para Compiladores
de Trilhas / Rodrigo Sol. — Belo Horizonte, 2010
xxii, 62 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais

Orientador: Mariza Andrade da Silva Bigonha

Co-Orientador: Fernando Magno Quintão Pereira

1. Computação - Tese 2. Compiladores
(Computadores) - Tese. I. Orientador II. Título.

CDU 519.6*33



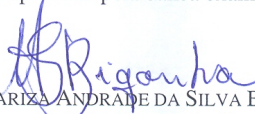
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

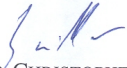
Eliminação de testes de overflow para compiladores de trilhas

MARCOS RODRIGO SOL SOUZA


Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Co-orientador
Departamento de Ciência da Computação - UFMG


ENG. CHRISTOPHE GUILLON
STMicroelectronics


PROF. VLADIMIR OLIVEIRA DI IORIO
Departamento de Informática - UFV


PROF. ROBERTO DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 25 de janeiro de 2011.

Para os meus pais

Quando decidi "suspender" minha vida por dois anos para investir nessa empreitada, ainda não tinha noção do que estava pela frente. Não imaginava que iria conviver com pessoas tão inteligentes e companheiras e menos ainda, que ficaria com o coração apertado por saber que esta convivência diária está chegando ao fim.

Inocentemente, achava que o mestrado me traria resposta para questões técnicas complicadas. Ledo engano! O mestrado só me trouxe mais dúvidas sobre questões mais complicadas ainda. Certeza mesmo é que eu estava no lugar certo, na hora certa. Um desses acasos da evolução, que alguns ainda chamam de destino. Como eu poderia saber que logo aos 30 segundos do primeiro tempo eu iria conhecer as três pessoas mais importantes dessa jornada: André, Andrei e Fernando.

São para eles que dedico especial agradecimento. Não seria possível suportar a pressão sem o companheirismo do André e do Andrei, também não seria possível completar essa jornada sem a sabedoria e a gentileza do Fernando.

Em todo agradecimento sempre existe o risco de se esquecer pessoas importantes. A vocês peço desculpas antecipadamente pela memória fraca. Mas, eu não poderia deixar de citar: os professores do LLP Roberto e Mariza Bigonha e também os amigos do LLP Terra, César e Leo (não aposte nada com esse cara). As contribuições do David Mendelin e do Christopher Guiolin também foram fundamentais para o sucesso deste trabalho.

Também não poderia deixar de agradecer pessoas que contribuiriam indiretamente, mas de forma muito importante para esta jornada: Meus pais, que como sempre me apoiaram e a Carol, que teve paciência infinita com minha falta de disponibilidade nesses anos.

A todos vocês: muito obrigado!

*“The First Rule of Program Optimization: Don’t do it. The Second Rule of Program
Optimization (for experts only!): Don’t do it yet.”*
(Michael A. Jackson)

Resumo

Compilação de trilhas é uma nova técnica utilizada por compiladores *just-in-time* (JIT) como o *TraceMonkey*, o compilador de JavaScript do navegador *Mozilla Firefox*. Diferente dos compiladores *just-in-time* tradicionais, um compilador de trilhas trabalha somente com uma parte do programa fonte, geralmente um caminho linear de instruções que são frequentemente executadas dentro de um laço.

Como uma trilha é compilada durante a interpretação de um programa, o compilador *just-in-time* tem acesso aos valores manipulados em tempo de execução. A capacidade de acessar esses valores permite ao compilador a possibilidade de produzir código de máquina mais otimizado.

Nesta dissertação é explorada a oportunidade de prover uma análise que remove testes de *overflow* desnecessários de programas JavaScript. Para mostrar que algumas operações não podem causar *overflows* é utilizada uma técnica denominada análise de largura de variáveis.

A otimização proposta é linear em tamanho e espaço com o número de instruções presentes na trilha de entrada, e é mais efetiva que as análises de largura de variável tradicionais porque utiliza valores conhecidos em tempo de execução.

A otimização proposta foi implementada no navegador *Mozilla Firefox*, e testada em mais de 1.000 programas JavaScript de diversas coleções, incluindo os 100 sítios mais visitados da Internet segundo o índice Alexa.

Foram produzidos códigos binários para as arquiteturas x86 e ST40-300. Na média, a otimização proposta foi capaz de remover 91.82% dos testes de *overflow* nos programas presentes na coleção de programas de teste do *TraceMonkey*. A otimização proposta prove uma redução do tamanho do código binário de 8.83% na plataforma ST40 e de 6.63% na plataforma x86. A otimização aumenta o tempo de execução do compilador *TraceMonkey* em 2.53%.

Palavras-chave: compilação de trilhas, otimização de código, javascript, testes de overflow.

Abstract

Trace compilation is a new technique used by just-in-time (JIT) compilers such as TraceMonkey, the JavaScript engine in the Mozilla Firefox browser. Contrary to traditional JIT machines, a trace compiler works on only part of the source program, normally a linear path inside a heavily executed loop. Because the trace is compiled during the interpretation of the source program the JIT compiler has access to the values manipulated at runtime. This observation gives to the compiler the possibility of producing binary code specialized to these values. In this thesis we explore this opportunity to provide an analysis that removes unnecessary overflow tests from JavaScript programs. Our optimization uses range analysis to show that some operations cannot produce overflows. The analysis is linear in size and space on the number of instructions present in the input trace, and it is more effective than traditional range analyses, because we have access to values known only at execution time. We have implemented our analysis on top of Firefox's TraceMonkey, and have tested it on over 1000 scripts from several industrial strength benchmarks, including the scripts present in the top 100 most visited webpages in the Alexa index. We generate binaries to either x86 or the embedded microprocessor ST40-300. On the average, we eliminate 91.82% of the overflows in the programs present in the TraceMonkey test suite. This optimization provides an average code size reduction of 8.83% on ST40 and 6.63% on x86. Our optimization increases TraceMonkey's runtime by 2.53%.

Keywords: trace compilation, code optimization, javascript, overflow tests

Lista de Figuras

2.1	Java. Do código fonte ao código de máquina	11
2.2	Um exemplo de programa (direita) com o respectivo grafo de fluxo de controle (esquerda).	21
2.3	Exemplo de código de máquina que poderia ser produzido pelo compilador de trilhas.	23
2.4	O compilador JIT Mozilla TraceMonkey.	24
2.5	Um pequeno exemplo de um programa <i>JavaScript</i> e sua representação em bytecodes.	25
2.6	Esta figura ilustra o casamento entre a trilha gravada e o segmento LIR que foi produzida pelo sistema de trilhas.	27
2.7	O código LIR, logo após a inserção do prólogo e do epílogo, e a semântica do código <i>x86</i> produzido pelo Nanajit.	28
2.8	Avaliação parcial	29
2.9	Um exemplo em que o conhecimento de valores de tempo de execução propicia a geração de código mais eficiente.	31
3.1	Grafo de restrições para a trilha mostrada na Figura 2.6 .Os números dentro das caixas mostram quando os vértices foram criados	36
3.2	Propagação de faixas para vértices aritméticos.	37
3.3	Propagação de restrições para vértices relacionais.	38
3.4	Segmento LIR antes de ser passado ao Nanajit	40
3.5	Componentes	42
3.6	Classes	44
3.7	Efetividade do algoritmo em termos de percentual de testes de <i>overflow</i> removidos por <i>script</i> (Cima) <i>Alexa</i> ; média geométrica 53.50%. (Baixo) <i>Trace-Test</i> ; média geométrica: 91.82%; <i>Hardware</i> : mesmos resultados para ST4 e x86. Os 224 <i>scripts</i> estão ordenados pela média de efetividade. . . .	47

3.8	Redução de tamanho por <i>script</i> em duas diferentes arquiteturas. (Cima) <i>ST4</i> ; média geométrica 8.83%. (Baixo) <i>x86</i> ; média geométrica: 6.63% <i>Benchmark: Trace-Test</i> . Os 224 <i>scripts</i> estão ordenados pela média de redução	48
3.9	Tempo de Execução. Os 224 <i>scripts</i> estão ordenados pela média de incremento no tempo de execução	49
3.10	Um gráfico do tipo pizza mostrando as razões que impediram o algoritmo de remover testes de <i>overflow</i>	50
3.11	Um histograma para as instruções presentes nos <i>benchmarks Trace-Teste</i> e <i>PeaceKeeper</i> , comparados aos <i>bytecodes</i> encontrados na coleção <i>Alexa</i> . Eixo X: As instruções <i>i</i> estão ordenadas por suas frequências na coleção <i>Alexa</i> . Quanto mais suave a onda, mais similar ao <i>Alexa</i> é a coleção.	51

Lista de Listagens

2.1	Tipo dinâmico em <i>Ruby</i>	12
2.2	Criação de um objeto em <i>JavaScript</i>	13
2.3	Encapsulamento de dados em <i>JavaScript</i>	14
2.4	Criação de um objeto em <i>JavaScript</i>	14
2.5	Exemplo de sub-classe em <i>JavaScript</i>	15
2.6	Exemplo de compilação <i>JIT</i>	17

Conteúdo

	ix
Resumo	xiii
Abstract	xv
Lista de Figuras	xvii
1 Introdução	1
1.1 Definição do Problema	5
1.2 Solução Proposta	5
1.3 Contribuições	6
1.4 Organização do Texto	7
2 Revisão de Literatura	9
2.1 Linguagens Interpretadas	9
2.1.1 Portabilidade	10
2.1.2 Tipo Dinâmico	11
2.1.3 Reflexão	12
2.2 <i>JavaScript</i>	13
2.2.1 Características de <i>JavaScript</i>	13
2.2.2 Interpretador SpiderMonkey	15
2.3 Compiladores <i>Just-In-Time</i>	16
2.3.1 Classificação dos compiladores <i>just-in-time</i>	18
2.3.2 Implementações de Compiladores <i>Just-In-Time</i>	19
2.4 Compilação de Trilhas	20
2.4.1 TraceMonkey	22
2.5 Avaliação Parcial de Programas	27
2.5.1 Avaliadores Parciais: <i>On-line</i> e <i>off-Line</i>	28

2.6	Avaliadores Parciais e Geração de Código	29
2.6.1	Desmembramento de Laços	30
2.6.2	Eliminação de Testes de <i>Overflow</i>	31
2.7	Análise de Largura de Variáveis	31
2.8	Conclusão	32
3	Remoção de Testes de <i>Overflow</i> Via Análise Sensível ao Fluxo	33
3.1	Construção do Grafo de Restrições	34
3.2	Propagação de Faixas de Restrições	37
3.3	Análise de Complexidade do Algoritmo Proposto	39
3.4	Implementação do Algoritmo	41
3.5	Experimentos	43
3.5.1	<i>Benchmarks</i>	44
3.5.2	O Hardware	45
3.5.3	Eficácia do Algoritmo Proposto	46
3.5.4	Avaliação da Redução do Tamanho do Código em Função da Eliminação dos Testes de Overflow	46
3.5.5	Avaliação do Efeito do Algoritmo em Relação ao Tempo de Exe- cução do TraceMonkey	47
3.5.6	Análise dos Testes de <i>Overflow</i> não Removidos do Programa . .	49
3.5.7	Análise do Ganho Obtido Devido ao Conhecimento dos Valores das Variáveis em Tempo de Execução	50
3.5.8	Análise dos Resultados de Efetividade entre as Coleções <i>Trace-</i> <i>Test</i> e <i>Alexa</i>	50
3.6	Conclusão	52
4	Conclusão	53
4.1	Trabalhos Futuros	54
	Bibliografia	57

Capítulo 1

Introdução

Compilação de código é o problema de transformar um programa escrito em uma linguagem de programação de alto nível em uma cadeia de bits – zeros e uns – que serão lidos pelo processador de um computador [Aho et al., 2006]. Esta sequência de bits é denominada um programa em linguagem de máquina. As linguagens de programação são normalmente classificadas em duas categorias: *linguagens estaticamente compiladas* e *linguagens dinamicamente compiladas* [Webber, 2005]. Programas escritos em linguagens estaticamente compiladas são completamente lidos pelo compilador antes de serem transformados em linguagens de máquina. Exemplos de linguagens de programação deste tipo incluem *C*, *C++*, *SML*, *Haskell*, além de uma vasta gama de outras linguagens de grande importância acadêmica e industrial. As linguagens dinamicamente compiladas também são comuns. Neste caso programas são *interpretados*, isto é, em vez de eles serem diretamente traduzidos para linguagem de máquina, estes programas são lidos por um outro programa, o interpretador. O interpretador se encarrega de executar todas as ações previstas pelo programa interpretado no *hardware* alvo.

Um programa é composto por diversas *funções* ou subprogramas, que recebem parâmetros de entrada e os utilizam para gerar um valor de saída. Durante a interpretação de um programa, o interpretador se encarrega de inferir quais são as funções mais utilizadas, e a partir desta informação, ele compila tais funções para código de máquina. Este modo de compilação, baseado em funções denomina-se *compilação dinâmica tradicional* [Aho et al., 2006]. Este processo de decidir durante a interpretação, compilar parte de um programa para linguagem de máquina, Aycock, [Aycock, 2003], também o denomina *compilação dinâmica*, ou compilação *just-in-time*. Dentre os exemplos de linguagens dinamicamente compiladas, citam-se *Ruby* [Thomas Chad Fowler, 2005], *Python* [Bird et al., 2009a], *Prolog* [Sterling Shapiro, 1986], *Lua* [Ierusalimsky

et al., 2007] e *Java* [Gosling et al., 2005a]. É importante notar que esta distinção entre linguagens de programação não é rígida. *Java*, por exemplo, é uma linguagem normalmente compilada dinamicamente. Embora existam também sistemas que compilem *Java* estaticamente.

Linguagens como PHP, *Java*, Perl, Ruby, JavaScript e Lua estão entre as mais populares do mundo. *Java*, por exemplo, é a segunda colocada no website *Tiobe*, que mede o índice de popularidade de linguagens de programação [Jansen, 2009]. Em outro exemplo, PHP é a primeira colocada em buscas de emprego de programação no website *Craigslist* [Authors, 2009]. JavaScript, uma linguagem dinamicamente compilada, é utilizada por programadores em todo o mundo para o desenvolvimento de aplicações para internet [Flanagan, 2001]. Linguagens tais como PHP, Perl e Bash são populares porque suas curvas de aprendizado tendem a ser mais suaves que as curvas de aprendizado de linguagens como C, C++ e Fortran. A popularidade de *Java* deve-se, em primeiro lugar, à sua portabilidade, e em segundo lugar, às poderosas abstrações para a programação orientada por objetos que a linguagem provê.

A despeito da grande popularidade, programas em linguagens dinamicamente tipadas tendem a ser menos eficientes que programas em linguagens estaticamente tipadas. Tal fato não é um defeito da linguagem, propriamente dita, mas do ambiente de execução onde tal linguagem é utilizada. Programas escritos nestas linguagens são geralmente interpretados. A interpretação é normalmente um processo mais lento que a execução de programas escritos em código de máquina. Tal lentidão deve-se ao fato da interpretação de cada instrução do programa alvo demandar a execução de dezenas, quando não centenas, de instruções da máquina real.

Segundo Richards [Richards et al., 2010], dentre as linguagens dinamicamente compiladas, *JavaScript* é a linguagem de programação usada para dar suporte as rotinas *client-side* de aplicações *web* mais popular. De acordo com o relatório Alexa 2010¹, *JavaScript* é usada em 97 dos 100 mais populares sites da *web*. Assim, é fundamental que *JavaScript* se beneficie de ambientes de execução mais eficientes.

Web Browsers normalmente interpretam programas escritos em JavaScript. Mas, para obter eficiência na execução, programas nessa linguagem podem ser compilados durante sua interpretação usando um compilador do tipo *just-in-time (JIT)*. Existem muitas formas de se realizar compilações *JIT* [Aycock, 2003]. Em 2006, Gal [Gal, 2006], e depois, Chang [Chang et al., 2009] apresentaram uma nova técnica de compilação dinâmica e a denominaram compilação de trilhas ou *trace compilation*. Uma trilha de programa é uma sequência linear de código que representa um caminho no

¹<http://www.alexa.com>

grafo de fluxo de controle de um programa. Compiladores de trilhas são compiladores que, diferentemente dos compiladores convencionais para linguagens dinâmicas, operam sobre laços individuais em vez de funções. Essa escolha está baseada na expectativa de que os programas gastam mais tempo executando comandos dentro de *loops* [Chang et al., 2009]. Uma definição formal de compiladores de trilhas pode ser vista na Seção 2.4.

Otimizações de código consistem no processo de modificar um determinado código de forma a fazê-lo trabalhar de forma mais eficiente, ou seja, otimizações buscam aumentar a velocidade de execução, diminuir a quantidade de memória utilizada ou consumir menos recursos da máquina [Sarkar, 2008]. Otimizações de código podem ser independentes ou dependentes de arquitetura [Aho et al., 2006]. Otimizações independentes de arquitetura geralmente são aplicadas em representações intermediárias geradas pelos *front-ends* dos compiladores. Otimizações dependentes de arquitetura são otimizações realizadas na linguagem de representação da máquina alvo. Várias técnicas de otimização de código foram desenvolvidas nas últimas décadas, entre elas estão as melhorias na alocação de registradores, paralelismo de instruções, eliminação de código morto, redução de força, propagação de constantes e otimização para multiprocessadores, entre outras [Sarkar, 2008].

Otimizações de código costumam demandar alto processamento para serem realizadas. Por isso, normalmente são aplicadas em uma fase de compilação das linguagens estáticas. Uma vez gerado o código binário otimizado, cada execução se beneficia das otimizações sem arcar com o ônus do seu tempo de processamento. Esse cenário é diferente quando otimizações são aplicadas em compiladores de linguagens dinâmicas. Neste caso, o custo do processamento de cada otimização realizada na compilação é somado ao tempo global de execução. Consequentemente, muitas otimizações que são largamente aplicadas à compilação de linguagens estáticas têm custo proibitivo para aplicação na compilação de linguagens dinâmicas.

A compilação de trilhas trouxe uma nova proposta de geração de código, diferente da compilação dinâmica tradicional, e abriu diversas oportunidades de pesquisa na área de otimização de código.

Por exemplo, com o surgimento dos compiladores de trilhas para linguagens dinâmicas, algumas das otimizações que somente eram apropriadas na compilação de linguagens estáticas passaram a ser importantes também para a melhora do desempenho das linguagens dinâmicas. No entanto, constitui um desafio computacional determinar quais otimizações de código podem ser utilizadas durante a compilação de trilhas de forma a aumentar a velocidade de execução e/ou reduzir o espaço utilizado em memória, comprometendo o mínimo possível o desempenho do compilador.

Dada a natureza dos compiladores de trilha, novas otimizações também podem ser criadas especificamente para esse tipo de compilador. Diferente dos compiladores para linguagens estáticas, compiladores de trilhas podem utilizar informações de tempo de execução para influenciar as escolhas durante a etapa de otimização. Esta é a principal motivação para o desenvolvimento deste projeto de dissertação.

Um exemplo de compilador *just-in-time* é o compilador *TraceMonkey* do *browser* Mozilla Firefox 3.x [Gal et al., 2009], que traduz as trilhas de programas mais executadas para código de máquina. O compilador *TraceMonkey*, entre outras otimizações, tenta realizar algumas especializações simples de tipo em programas JavaScript. Isto significa que, por exemplo, embora Javascript veja números como valores de ponto-flutuante, o *TraceMonkey* tenta manipulá-los como inteiros toda vez que é possível, partindo do pressuposto de que trabalhar com inteiros é muito mais rápido que tratar aritmética de ponto flutuante. Contudo, para aplicar essa otimização é necessário assegurar que a semântica do programa permaneça inalterada. Por exemplo, JavaScript assume aritmética de precisão arbitrária, uma propriedade que não pode ser garantida com inteiros de 32 bits. Assim, em JavaScript, caso uma operação entre dois números inteiros produza um valor acima de 32 bits, esse valor é automaticamente convertido para um número de ponto flutuante. Isso implica que cada operação aritmética de soma ou multiplicação deve ser protegida por um teste de *overflow*, que verifica o tamanho do resultado produzido. Uma otimização possível neste caso seria a eliminação desses testes de *overflow* caso os valores sejam conhecidos em tempo de execução. *Overflow* é o fenômeno que ocorre quando o resultado de uma operação aritmética é maior que o espaço alocado pelo computador para armazenar esse resultado.

Testes de *overflow* são pervasivos no código de máquina produzido pelo *TraceMonkey*, um problema de desempenho já reconhecido pela comunidade Mozilla². Testes de *overflow* dão origem a dois problemas principais:

- cada teste pode forçar uma saída antecipada do módulo de trilhas, complicando otimizações que demandam o reposicionamento de instruções, como *eliminação de redundância parcial* [Briggs et al., 1994] e *escalonamento de instruções* [Bernstein Rodeh, 1991],
- instruções que tratam *overflows* aumentam o tamanho do código. Cerca de 12.3% do código x86 produzido por cada script compilado pelo *TraceMonkey* é referente ao tratamento de ocorrências de testes de *overflow*. Este dado se refere ao conjunto de scripts de teste presente no *TraceMonkey*. Esse código extra é

²https://bugzilla.mozilla.org/show_bug.cgi?id=536641

uma complicação em pequenos dispositivos que executam JavaScript, como, por exemplo, os da família ST de microcontroladores³.

Dados esses dois problemas principais, o objetivo deste trabalho é desenvolver uma nova otimização de código para compiladores de trilhas. Essa otimização deve ser capaz de diminuir os efeitos colaterais causados pela especialização de tipos de ponto-flutuante para inteiros. Ou seja, a otimização deve ser capaz de eliminar o maior número possível de testes de *overflow* no código binário final. O algoritmo a ser desenvolvido e implementado deve ser executado em tempo e espaço lineares para não comprometer o desempenho do compilador.

1.1 Definição do Problema

Seja P um programa formado por uma trilha de código. Assuma que algumas das operações de P sejam sucedidas por um teste que verifica a ocorrência de um *overflow*. O problema de interesse consiste em determinar quais, dentre essas operações, não podem causar um *overflow*.

1.2 Solução Proposta

O problema será detalhado por meio de uma análise sensível ao fluxo, cujo objetivo será demonstrar a redundância de alguns testes de *overflows*, e os resultados alcançados serão apresentados nesta dissertação. A análise proposta deverá ser executada em tempo linear no número de instruções da trilha, e deverá ser implementada em um compilador de porte industrial. Resalte-se que essa análise não deverá depender de um compilador específico, podendo ser implementada em qualquer compilador do tipo *JIT* que utilize o paradigma de compilação por trilhas para a geração de código.

O algoritmo proposto realizará análise de largura de variáveis de acordo com as propostas de [Harrison, 1977] e [Patterson, 1995], na tentativa de estimar os maiores e menores valores que podem ser atribuídos para qualquer variável. Entretanto, nossa abordagem diferirá dos trabalhos de Harrison e Patterson porque usaremos valores conhecidos em tempo de execução de forma a colocar limites nas faixas de valores que uma variável inteira pode assumir durante a execução do programa. Esta estratégia corresponde a um tipo de *avaliação parcial* [Jones et al., 1993], feita em tempo de execução, uma vez que a análise é invocada por um compilador just-in-time enquanto a aplicação alvo está sendo executada. De posse desses valores, a otimização poderá

³<http://www.st.com/mcu/familiesdocs-51.html>

realizar uma análise de largura de variáveis muito mais agressiva que nos termos da implementação tradicional.

Em relação à implementação, a otimização proposta deverá ser similar aquela do algoritmo ABCD que atua na eliminação de testes para verificação de limites de arranjo [Bodik et al., 2000]. Entretanto, existem diferenças importantes:

- o algoritmo proposto é mais simples, justamente por ser executado em trilhas, ou seja, em uma sequência linear de instruções. Também por esse motivo, o algoritmo deve ser linear no número de variáveis de programa e não quadrático como em outras análises que mantêm cadeias de definição e uso de variáveis.
- o algoritmo realiza toda a análise de uma só vez na trilha, bastando uma leitura completa do fluxo de instruções da trilha enquanto está sendo gerada. O ABCD, por exemplo, executa sobre demanda. Além dessas diferenças, existe o fato que são usados valores de tempo de execução das variáveis, o que permite a otimização ser menos conservativa[Burke et al., 1999].

1.3 Contribuições

Como resultado do trabalho realizado, destacamos as seguintes contribuições:

- A formalização de um algoritmo que realiza a eliminação de testes de overflow em trilhas de código, gerado durante a compilação *just-in-time* de uma linguagem alvo.
- A implementação deste algoritmo no compilador *TraceMonkey*. *TraceMonkey* é usado pelo navegador Mozilla Firefox para compilar dinamicamente programas escritos em JavaScript.
- A instrumentação do navegador Mozilla Firefox a fim de obter dados estatísticos acerca dos programas produzidos durante a compilação de trilhas de JavaScript.
- Adaptação do *Nanojit*[Gal et al., 2009] a fim de permitir que o mesmo removesse os testes de overflow apontados como desnecessários pela análise realizada.
- Disponibilização de todas as técnicas e algoritmos de compilação desenvolvidos durante este projeto à comunidade de *software* livre. Incluindo também outros *patches* que não fazem parte diretamente do escopo deste trabalho, mas que foram desenvolvidos para dar suporte à implementação da otimização.

- Publicação de 02 artigos em conferência e periódico reconhecidos pela comunidade de Linguagens de Programação.

1.4 Organização do Texto

Esta dissertação está organizada em quatro capítulos. No Capítulo 1 foi fornecida uma visão básica dos fundamentos que norteiam este trabalho. No Capítulo 2, oferecemos uma revisão da literatura disponível dos principais tópicos deste projeto. No Capítulo 3 concentramos a nossa contribuição. A descrição de cada capítulo dessa dissertação, pode ser vista a seguir:

Capítulo 1 - Introdução Neste capítulo é apresentada uma introdução aos fundamentos da otimização de códigos, dos compiladores *just-in-time* e das possibilidades de se criar novas otimizações que sejam especializadas para esse tipo de compilador.

Capítulo 2 - Revisão de Literatura

Neste capítulo, é apresentada a literatura disponível sobre os alicerces deste trabalho, tais como compiladores *just-in-time*, otimizações de código e técnicas de avaliação parcial de programas.

Capítulo 3 - Remoção de Testes de *Overflow* Via Análise Sensível ao Fluxo

Neste capítulo são apresentadas nossas contribuições. O capítulo inicia-se apresentando a nova otimização proposta por esta dissertação. Em seguida, tratamos da implementação dessa nova análise em um compilador de porte industrial. Finalmente, apresentamos os experimentos utilizados para validar o trabalho.

Capítulo 4 - Conclusão

Neste capítulo são apresentadas as conclusões finais do trabalho e algumas sugestões de trabalhos futuros.

Capítulo 2

Revisão de Literatura

Neste capítulo, são apresentados os fundamentos que suportam a criação da nova otimização de código proposta nesta dissertação. Seção 2.1 apresenta as principais características das linguagens interpretadas, a saber, portabilidade, tipagem dinâmica e reflexão. Seção 2.2 apresenta um estudo de caso da linguagem *JavaScript*, e de um interpretador para essa linguagem, o *SpiderMonkey*. Em seguida, nas Seções 2.3 e 2.4 são descritos os aspectos fundamentais dos compiladores *just-in-time* e dos compiladores de trilhas. Seção 2.5 introduz uma técnica de otimização de código conhecida como Avaliação Parcial de Programas. Na Seção 2.6, é discutido como a utilização de técnicas de avaliação parcial de programas pode contribuir para a construção de otimizações de código mais eficientes. Finalmente, na Seção 2.7, é apresentada uma técnica chamada Análise de Largura de Variável, técnica esta que serve de base para a otimização proposta nesta dissertação.

2.1 Linguagens Interpretadas

Linguagens interpretadas são aquelas linguagens que são diretamente executadas por um interpretador de programas sem a necessidade de terem sido previamente compiladas. Nas linguagens compiladas, o programa é convertido de uma vez para uma representação binária que pode ser executada uma ou mais vezes diretamente pelo processador. Nas linguagens interpretadas, cada instrução é executada individualmente pelo interpretador sem a necessidade de se gerar código de máquina [Parr, 2009].

Teoricamente, qualquer linguagem pode ser interpretada ou compilada, fator que faz dessa designação uma questão prática, e ligada à forma de implementação da linguagem em si, não sendo uma propriedade das linguagens de programação [Aycock, 2003].

No passado, o projeto de linguagens de programação era bastante influenciado pela decisão de usar um compilador ou um interpretador. Caso, por exemplo, da linguagem *Smalltalk* que foi projetada para ser interpretada, e permitia que objetos genéricos interagissem uns com os outros de forma dinâmica [Deutsch Schiffman, 1984].

Atualmente, interpretadores costumam receber como entrada uma representação intermediária. No passado, era comum que os interpretadores compilassem cada instrução individualmente, o que causava, por exemplo, a recompilação da mesma instrução várias vezes, por exemplo, se ela fosse executada dentro de uma estrutura de repetição [Parr, 2009].

A maioria das linguagens interpretadas que têm maior penetração nos dias de hoje usa uma representação intermediária. Esse é o caso da linguagem *Java* e também das linguagens *Python* e *Ruby*. Nesses casos, o interpretador pode:

- gerar uma representação em forma de *bytecodes* como nas linguagens *Java* [Gosling et al., 2005b] e *Python* [Bird et al., 2009b].
- A representação pode ser dada via uma árvore de sintaxe abstrata, como é o caso da linguagem *Ruby* [Thomas Chad Fowler, 2005].

Dentre as principais vantagens das linguagens interpretadas estão os sistemas de tipo e escopo dinâmicos, a facilidade de realizar reflexão, o que enseja o uso de recursos de meta-programação e principalmente o fato de serem altamente portáteis.

A principal desvantagem da interpretação está relacionada ao desempenho. A interpretação tende a ser mais lenta que a execução de código de máquina diretamente na unidade de processamento hospedeiro. Uma técnica para minimizar esse efeito negativo é chamada de compilação *just-in-time*, que converte os trechos mais frequentemente executados em código de máquina. Compiladores *just-in-time* serão abordados na Seção 2.3.

2.1.1 Portabilidade

Um dos motivos do grande sucesso e aceitação da linguagem *Java*, nos meios acadêmicos e principalmente na indústria, está na facilidade de portar *Java* para diversos ambientes. A frase “*Write once, run anywhere*” [Gosling et al., 2005b] foi repetida à exaustão pela *Sun*, então proprietária da linguagem, para destacar a portabilidade de *Java*.

Um programa escrito em *Java* é transformado em *bytecodes* que pode ser interpretado e também compilado em qualquer plataforma que tenha uma máquina virtual *Java* disponível, sem precisar novamente do código fonte original [Gosling et al., 2005b]. A Figura 2.1 descreve o funcionamento da plataforma *Java*.

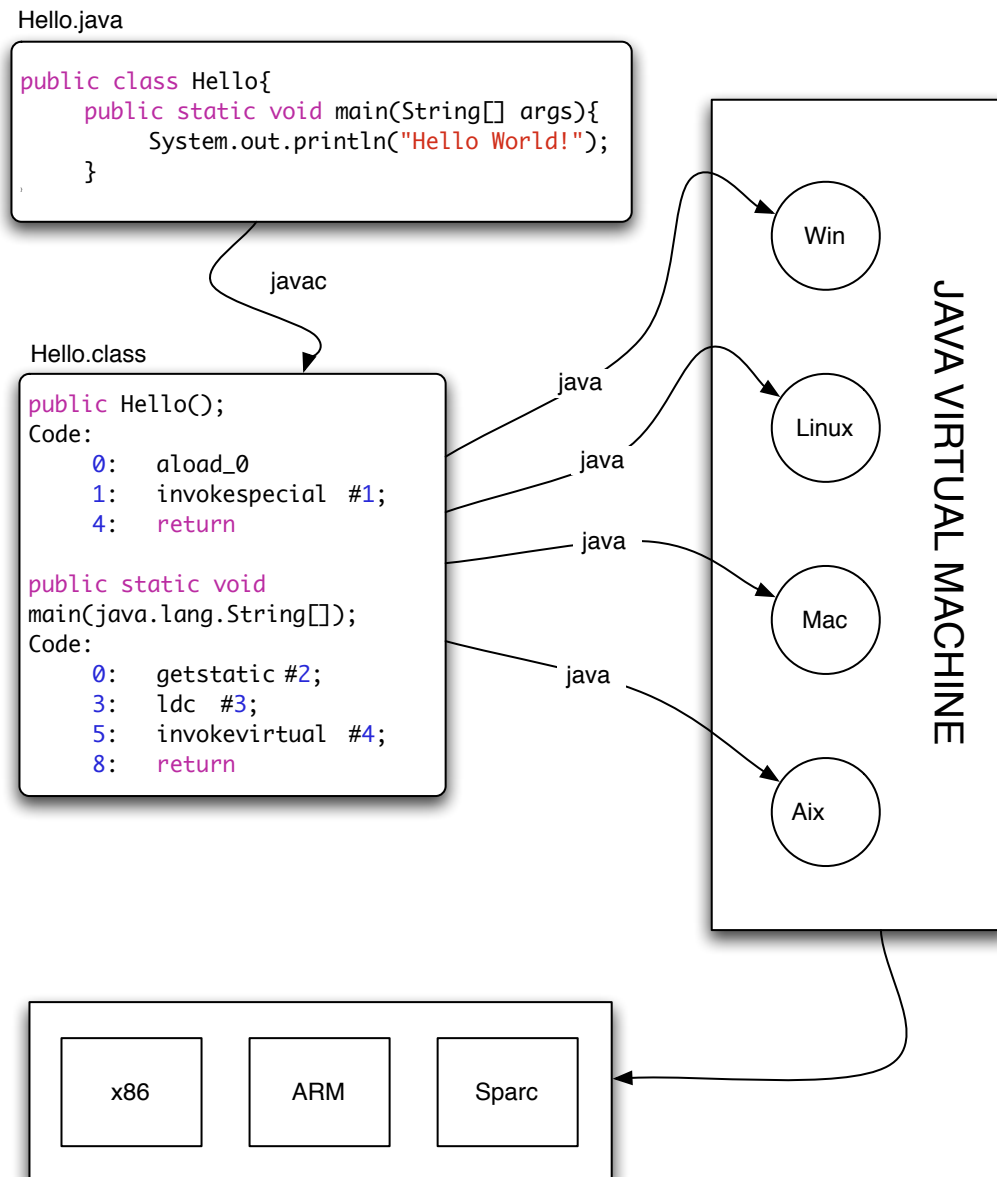


Figura 2.1: Java. Do código fonte ao código de máquina

2.1.2 Tipo Dinâmico

Para uma linguagem de programação ser considerada de tipagem dinâmica, é necessário que algumas das verificações de tipo aconteçam em tempo de execução. Sempre que possível, linguagens compiladas fazem a verificação de tipos em tempo de compilação. Nas linguagem que utilizam tipagem dinâmica, uma variável pode apontar para objetos de qualquer tipo, ou seja, dados têm tipos claramente definidos e fixo, mas variáveis

não [Williams et al., 2010].

Entre as linguagens que utilizam sistemas de tipo dinâmicos estão: *JavaScript*, *Lua* e *Erlang*. Linguagens que utilizam esse sistema de tipos tendem a ser mais flexíveis, permitindo, por exemplo, que sejam adicionadas funcionalidades e modificações nos tipos em tempo de execução. Esses sistemas de tipo apresentam uma dificuldade adicional na depuração de programas, uma vez que podem gerar exceções em tempo de execução, quando, por exemplo, uma variável tem um tipo inesperado de dado. Esse tipo de erro pode ser difícil de ser localizado [Williams et al., 2010].

Uma das vantagens dessa abordagem está na flexibilidade. A Listagem 2.1 apresenta um trecho de código em *Ruby* onde o interpretador automaticamente muda o tipo da variável para permitir um número maior sem precisar de intervenções sintáticas do programador. Nesse caso, o interpretador se torna eficiente por trabalhar com números pequenos, sempre que possível, e caso o número exceda um certo limite, o tipo da variável é automaticamente modificado para alcançar a corretude necessária. Uma das desvantagens da tipagem dinâmica está relacionada com os encargos acrescentados às verificações de tipo em tempo de execução. Esse aspecto contribui para que os programas interpretadas sejam menos eficientes.

Listagem 2.1: Tipo dinâmico em *Ruby*

```
1 num = 1000000
2 num.class
3 #Imprime: FixNum
4 num = num * 9999999999999999
5 num.class
6 #Imprime: BigNum
```

2.1.3 Reflexão

Reflexão é a capacidade de um programa de computador de observar e modificar a si mesmo. Linguagens de programação que implementam reflexão oferecem aos programas a capacidade de modificar suas estruturas e seu comportamento em tempo de execução. A implementação desse tipo de recurso em uma linguagem de programação interpretada costuma ser mais fácil de ser realizada do que nas linguagens compiladas [Thomas Chad Fowler, 2005].

2.2 *JavaScript*

JavaScript é uma linguagem de programação orientada por objetos baseada no modelo *object-prototype* [Ungar Smith, 1987]. *JavaScript* se tornou a principal linguagem de programação utilizada na construção de interfaces gráficas de aplicações web. *JavaScript* foi criada em 1995 por Breandon Eich, quando ele trabalhava no desenvolvimento do navegador *Netscape*. A implementação de uma linguagem de script no navegador *Netscape* surgiu da necessidade de adicionar maior interatividade aos documentos no formato *HTML* (*HyperText Markup Language*). Inicialmente, os programadores utilizavam *JavaScript* para fazer validações de formulários e para construir animações.

A sintaxe de *JavaScript* hoje é definida pela especificação *ECMA-262*, tendo evoluído bastante desde a primeira versão. Atualmente *JavaScript* é parte fundamental do ecossistema de aplicações da internet. A junção de *JavaScript*, à possibilidade de fazer requisições assíncronas e o padrão de organização de informações *XML* deram origem ao assíncrono chamado *AJAX* (*Assincronous JavaScript and XML*) que permitiu o desenvolvimento de aplicações para internet mais próximas em termos de usabilidade daquelas desenvolvidas para *desktop*.

2.2.1 Características de *JavaScript*

JavaScript contém um pequeno conjunto de tipo de dados, sendo primitivos os tipos booleano, numérico e string e possui alguns valores especiais como *null* e *undefined*. Todas as outras estruturas de dados do *JavaScript* são do tipo *object*.

Em *JavaScript*, objetos são implementados como uma coleção de propriedades nomeadas. Como *JavaScript* é uma linguagem interpretada, novas propriedades podem ser facilmente adicionadas a um dado objeto em tempo de execução. A Listagem 2.2 mostra a criação de um objeto *JavaScript*, e em seguida a adição de duas propriedades em tempo de execução.

Listagem 2.2: Criação de um objeto em *JavaScript*

```
1  carro = new Object ;  
2  carro.cor = 'azul' ;  
3  carro.ano = 1983;
```

JavaScript não pode ser considerada uma linguagem estritamente orientada por objetos, mas fornece aos programadores os seguintes recursos fundamentais da programação orientada por objetos:

Encapsulamento

JavaScript suporta métodos como membros de uma classe e atributos privados. A Listagem 2.3 mostra como os dados são encapsulados em *JavaScript*.

Listagem 2.3: Encapsulamento de dados em *JavaScript*

```
1 function Carro(){
2     var cor = 'azul';
3     var ano = 1983;
4     this.getCor = function(){
5         return this.cor;
6     };
7 }
```

Polimorfismo

Duas classes em *JavaScript* são capazes de responder pela mesma coleção de métodos. A Listagem 2.4 apresenta um exemplo de uso de polimorfismo em *JavaScript*.

Listagem 2.4: Criação de um objeto em *JavaScript*

```
1 function acelerar(carro){
2     carro.acelerar();
3     print(carro.velocidade);
4 }
5 function Carro(){ this.velocidade = 0; }
6 Carro.prototype.acelerar = function() {
7     this.velocidade += 1;
8 }
9 function CarroDeCorrida(){
10     this.velocidade = 0;
11 }
12 CarroDeCorrida.prototype.acelerar = function() {
13     this.velocidade += 50;
14 }
15 carro = new Carro;
16 carroDeCorrida = new CarroDeCorrida;
17 acelerar(carro);
18 acelerar(carroDeCorrida);
19 \\ Produz 1 e 50
```

Herança

JavaScript suporta a criação de subtipos que herdam o comportamento do objeto pai. A Listagem 2.5 apresenta um exemplo de utilização de subtipo em *JavaScript*.

Listagem 2.5: Exemplo de sub-classe em *JavaScript*

```
1 function Carro(){
2     this.velocidade = 0;
3 }
4 Carro.prototype.acelerar = function(){
5     this.x += 1;
6 }
7 CarroDeCorrida.prototype = new Carro;
8 CarroDeCorrida.prototype.constructor = CarroDeCorrida;
9 function CarroDeCorrida(){
10     Carro.call(this);
11     this.turbo = false;
12 }
13 CarroDeCorrida.prototype.acelerar = function(){
14     Carro.prototype.acelerar.call(this);
15     this.turbo = true;
16 }
17 f1 = new CarroDeCorrida;
18 print((f1 instanceof Carro) + ', '
19       + (f1 instanceof CarroDeCorrida));
20 f1.DoIt();
21 print(f1.velocidade + ', ' + f1.turbo);
22 \\ Imprime true,true, em se seguida: 1,true
```

JavaScript simula a herança de linguagens orientadas por objetos como *Java* ou *C++* via uma técnica conhecida por Delegação. O mecanismo de delegação foi usado por David Ungar na implementação de *Self* [Ungar Smith, 2007], e aparece em outras linguagens também baseadas em prototipagem de objetos, como *Lua*.

2.2.2 Interpretador SpiderMonkey

O *SpiderMonkey* é um interpretador concebido para ser rápido. Ele executa *bytecodes* não tipados. Sua unidade básica de operação é um tipo chamado *jsval*, um apontador para um dos possíveis tipos de valores em *JavaScript*. Além do interpretador, o

SpiderMonkey contém um compilador de *JavaScript*, o *TraceMonkey*, e um sistema de coleta de lixo.

Da mesma forma que muitos outros interpretadores, o *SpiderMonkey* tem uma estrutura monolítica de processamento, onde uma grande função implementa uma estrutura de controle do tipo *switch*. Para cada um dos tipos de *bytecodes* existentes, um bloco de código capaz de realizar o processamento necessário é associado a esta estrutura de repetição. Essa escolha de projeto certamente contribui para que o *SpiderMonkey* alcance altos índices de desempenho, porém torna o seu entendimento uma tarefa complexa, sendo um grande desafio estendê-lo com novos recursos.

2.3 Compiladores *Just-In-Time*

Plezbert, em [Plezbert Cytron, 1997], define os compiladores *just-in-time* como sistemas que executam código de forma interpretativa, mas compilam o código para código nativo sobre demanda. Segundo Plezbert, esses sistemas utilizam um único fluxo de execução para alternar o controle entre a compilação do código nativo para a execução do código nativo. Entre as vantagens dos compiladores *just-in-time* destacadas por Plezbert estão:

- o fato de os compiladores *just-in-time* serem capazes de evitar a perda de desempenho por ter que compilar todas as unidades de código anteriormente à sua execução. Por esse motivo, ele destaca ainda dois outros benefícios correlacionados:
 - a melhora de desempenho de programas que tem seu fluxo de execução orientados por eventos causados pelo usuário.
- Este é o caso de programas em *JavaScript* que executam em um browser para validar um formulário *HTML*. A execução do código só se faz necessária se e quando o usuário submeter o formulário para o servidor. A utilização de um compilador *just-in-time* evitaria, por exemplo, que um formulário que nunca tenha sido submetido para validação fosse previamente compilado desperdiçando recursos computacionais.
- A diminuição do tempo de espera entre a compilação integral e o início de interpretação.
- A diminuição do tempo de espera para iniciar a execução do programa.

Os sistemas de compilação tradicionais ao realizarem toda a compilação anterior a execução, podem criar uma espera grande entre a compilação e o começo da

execução. No caso dos compiladores *just-in-time* esse custo é amortizado ao longo da execução. Apesar de ser verdade, em termos práticos, esse último argumento levantado por Plezbert se mostra frágil. Normalmente, uma vez que um programa foi totalmente compilado, ele pode ser executado diversas vezes sem a necessidade de ser compilado novamente. Isto faz com que a espera apontada por Plezbert aconteça apenas uma vez. Nas outras execuções, o usuário já tem o ganho de ter o programa sendo executado diretamente pelo processador, sem o ônus do tempo da compilação.

Compiladores *just-in-time*, são velhos aliados daqueles que defendem as linguagens interpretadas. Desde o trabalho pioneiro de John McCarthy [McCarthy, 1960], o criador da linguagem *Lisp*, vários compiladores *just-in-time* têm sido projetados e implementados, como a máquina virtual *Java* [Gosling et al., 2005a] e o compilador da linguagem *Smalltalk* [Deutsch Schiffman, 1984].

Compiladores *just-in-time*, representam uma abordagem mista, mesclando os princípios tradicionais que pautam os interpretadores e compiladores. Um compilador *just-in-time*, inicia-se interpretando instruções de uma dada linguagem intermediária, porém, para alcançar um desempenho mais efetivo, ele realiza compilações de segmentos de código ou instruções individuais para código de máquina. Uma vez compilados, estes segmentos são mantidos em uma estrutura de *cache* para que possam ser reaproveitados no futuro se o fluxo de execução do programa retornar o controle para aquele determinado trecho já compilado.

A Listagem 2.6 apresenta um programa na linguagem *C* que realiza compilação *just-in-time* de uma instrução na arquitetura *x86*. Esse programa carrega um pequeno trecho de código em um arranjo, e passa esse arranjo como o endereço de uma função a ser executada. O trecho de código coloca o valor 1234 no registrador EAX, que é, normalmente, o lugar onde armazenam-se valores de retorno de funções. Este programa foi obtido na USENET¹.

Listagem 2.6: Exemplo de compilação *JIT*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void) {
4     char* program;
5     int (*fnptr)(void);
6     int a;
7     program = malloc(1000);          /* Space for the code */
```

¹<http://newsgroups.derkeiler.com/Archive/Comp/comp.compilers/2010-03/msg00063.html>

```

8  program[0] = 0xB8;           /* mov eax,1234h */
9  program[1] = 0x34;
10 program[2] = 0x12;
11 program[3] = 0;
12 program[4] = 0;
13 program[5] = 0xC3;           /* ret */
14 fnptr = (int (*)(void)) program;
15 a = fnptr();                 /* call the code */
16 printf("Result = %X\n",a);    /* show result */
17 }

```

John Aycock, [Aycock, 2003], fornece uma visão abrangente sobre compilação *just-in-time*, ele trata da questão central da utilidade de compiladores *just-in-time*: a melhoria de desempenho das linguagens interpretadas. Aycock afirma que programas interpretados tendem a ser mais portáteis por assumir uma representação independente de máquina, porém esse mesmo fator faz com que o desempenho desse tipo de programa seja inferior a de programas que são compilados.

2.3.1 Classificação dos compiladores *just-in-time*

Aycock, [Aycock, 2003], classifica os compiladores *just-in-time* considerando quatro propriedades fundamentais: invocação, invocação, executabilidade, concorrência e tempo real.

Uma explicação para cada uma dessas definições é apresentada a seguir.

Invocação

Os compiladores *just-in-time* podem ser classificados pela forma que são invocados. Um compilador pode ser explicitamente chamado pelo usuário, que indica quais segmentos do código devem ser compilados. Os compiladores *just-in-time* podem também ter sua invocação implícita. Nesse caso, o compilador decide de forma autônoma quando e o que será compilado. Uma invocação implícita é transparente para o usuário.

Executabilidade

Os compiladores *just-in-time* também podem ser classificados de acordo com sua capacidade de executar uma ou mais representações. Normalmente, um compilador *just-in-time* trabalha com duas representações para o programa:

- a representação original do programa fonte que é traduzida para uma determinada representação alvo, e
- a própria representação alvo da tradução.

Caso essas duas representações sejam a mesma, o compilador é considerado como não-executável. Isto acontece, por exemplo, quando o compilador *just-in-time* somente realiza otimizações *on-the-fly*, ou seja, quando o propósito do compilador é apenas otimizar uma dada representação, sem a responsabilidade de executar essa representação alvo, e sem se responsabilizar pela troca de contexto entre a execução do código de máquina e o interpretador.

Um compilador *just-in-time* é considerado poliexecutável se ele for capaz de executar mais de uma representação. Nesse caso, o compilador é responsável pela execução da transformação realizada, bem como por restaurar o estado do interpretador após a execução do segmento de código de máquina.

Concorrência

Esta propriedade é caracterizada pela forma com que o compilador *just-in-time* realiza a execução dos programas. Ele pode ser dependente da execução do programa, ou seja, enquanto a execução do programa está sendo realizada o compilador fica em estado de espera e aguarda a conclusão da execução. Nesse caso, o compilador não é considerado concorrente. Se o compilador for capaz de realizar outras atividades enquanto um programa por ele traduzido está sendo executado, então ele é considerado concorrente.

Tempo Real

Um compilador *just-in-time* é considerado de tempo real se ele puder garantir os requisitos necessários dos sistemas de tempo de real.

Para Timo [Timo, 2010], a classificação de compiladores *JIT* está desatualizada, uma classificação mais adequada aos dias de hoje deveria incluir se o compilador é ou não baseado em trilhas.

2.3.2 Implementações de Compiladores *Just-In-Time*

Segundo Timo [Timo, 2010], compilação *just-in-time* é uma técnica que compila uma representação intermediária para código nativo durante o tempo de execução. De acordo com Timo essa técnica não é estritamente necessária, uma vez que seu único objetivo é melhorar o desempenho da execução dos programas no compilador. Timo

lista vários benefícios da utilização de compiladores *just-in-time*, incluindo uma melhor compilação para arquiteturas específicas e possibilidade de se utilizar informações de tempo de execução para produzir código nativo mais eficiente. Os compiladores *just-in-time* tentam encontrar trechos de código denominados *hot spot* durante a interpretação dos programas.

O termo *hot spot* se refere a uma técnica que identifica os trechos de códigos que são frequentemente executados. Uma vez identificados, esses segmentos são compilados. O compilador para a linguagem Fortran foi um dos primeiros a adotar esse tipo de otimização. Já em 1974, o sistema utilizava um contador para identificar a frequência que determinados blocos básicos eram executados. Quanto maior fosse o contador associado a um bloco, maior eram as otimizações a ele aplicadas. A máquina virtual original do Smalltalk foi uma das primeiras a receber esse tipo de otimização, os blocos básicos mais executados eram compilados para código nativo. De acordo com Timo, a linguagem de programação *Self* foi a primeira a introduzir significativas contribuições em relação à evolução dos compiladores *just-in-time*. *Self* permite que os programadores definam previamente quais objetos devem ser mandatoriamente compilados.

2.4 Compilação de Trilhas

Embora o conceito de compilação *just-in-time* seja antigo e bem sólido na Ciência da Computação, a compilação de trilhas para programas escritos em linguagem interpretadas é uma ideia nova. O primeiro compilador de trilhas, influenciado pelo trabalho de Bala *et al.* [Bala, 2000], foi descrito por Andreas Gal em sua tese de doutorado [Gal, 2006; Gal et al., 2006]. Por ser uma ideia nova, a literatura contém apenas a descrição de duas implementações: Uma é o Projeto *Tamarim-Trace* [Chang et al., 2009]. Um compilador *JIT* implementado a partir do *Tamarim-central*, o motor do sistema *flash* da Adobe. O outro compilador é o *TraceMonkey*. *TraceMonkey* foi construído como um componente do *SpiderMonkey*, o interpretador original de *JavaScript* utilizado pelo browser *Mozilla Firefox*. A Seção 2.4.1 apresenta em detalhes o funcionamento do compilador *TraceMonkey*, uma vez que ele é o compilador escolhido para a implementação da otimização proposta nesta dissertação.

Uma descrição detalhada do compilador *Tamarim-Trace* é dado por Chang *et al.* [Chang et al., 2009], e uma introdução do uso de especialização de tipos durante a compilação de trilhas é fornecida por Gal *et al.* [Gal et al., 2009].

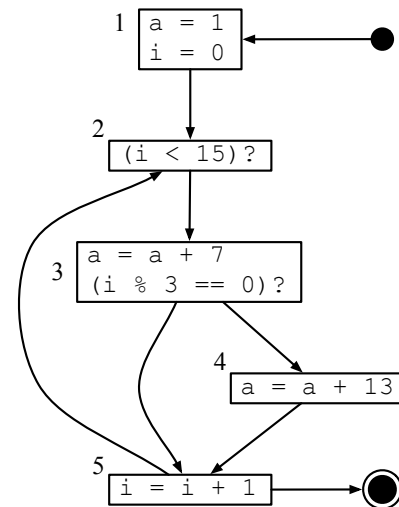
A essência da compilação de trilhas foi recentemente formalizada por Guo e Palsberg [Shuyu Guo, 2011].

```

a = 1;
for (i = 0; i < 15; i++) {
    a += 7;
    if (i % 3 == 0) {
        a += 13;
    }
}

```

(a)



(b)

Figura 2.2: Um exemplo de programa (direita) com o respectivo grafo de fluxo de controle (esquerda).

Um programa é composto de seqüências de instruções. Algumas dessas instruções podem ser avaliadas várias vezes durante a execução do programa. Por exemplo, a Figura 2.2(a) mostra um programa que contém uma iteração de 0 a 14. Em cada iteração, este programa adiciona o número 7 à variável a . Nas iterações que são múltiplos de 3, isto é, $\{0, 3, 6, 9, 12\}$, o programa também adiciona à variável a o número 13. No fim da execução desse programa o valor de a é 171. Esse programa contém diferentes *caminhos*. Esses caminhos determinam uma estrutura que, no jargão de compiladores, é conhecida como *grafo de fluxo de controle* (GFC). O GFC é mostrado na Figura 2.2(b). Um GFC é constituído por *blocos básicos* e por *arestas de fluxo*. Diferentes blocos básicos são conectados por arestas de fluxo, as quais determinam a ordem de execução entre estes blocos básicos. Segundo Allen [Allen, 1970] um bloco básico é uma porção de código do programa que possui algumas propriedades desejadas que o torna altamente passível de ser analisado. O exemplo dado possui cinco blocos básicos.

Alguns caminhos do GFC são mais executados que outros. Por exemplo, no GFC da Figura 2.2(b), a aresta entre os Blocos 1 e 2 é visitada somente uma vez. Por outro lado, o ciclo formado pelos Blocos 2, 3 e 5 é executado 10 vezes, enquanto o ciclo formado pelos blocos 2, 3, 4 e 5 é executado 5 vezes. Idealmente um compilador dinâmico deveria traduzir para código de máquina somente aqueles caminhos mais

executados. Esta é a contribuição do novo modelo de geração de código conhecido como compilação de trilhas.

A compilação dinâmica de trilhas consiste em compilar somente os caminhos mais executados de um programa, interpretando os caminhos menos executados.

A principal diferença entre um compilador dinâmico tradicional e um compilador de trilhas é que, enquanto o primeiro produz código para todo o programa, o segundo compila somente os caminhos mais executados, no caso da Figura 2.2(b) seriam compilados os ciclos $\{2, 3, 5\}$ e $\{2, 3, 4, 5\}$. Dado que um compilador de trilhas não precisa ler todo o código a ser compilado, ele gasta menos tempo com o processo de compilação. A desvantagem dessa abordagem é que ela pode gerar código redundante, como mostra o exemplo da Figura 2.3. Além disto, uma vez que o compilador tradicional enxerga todo o programa, existem situações onde este conhecimento global o ajuda a produzir melhores códigos. Ainda assim, a compilação de trilhas mostra-se uma alternativa promissora à compilação tradicional. Diversos experimentos foram realizados e mostram que o compilador de trilhas pode ser até 10 vezes mais eficiente que seu concorrente[Gal, 2006].

O programa produzido pelo compilador de trilhas para o ciclo $\{2, 3, 5\}$ é mostrado na Figura 2.3. A memória do computador é utilizada para que o interpretador passe informações para o código de máquina produzido pelo compilador dinâmico, e vice-versa. Uma parte da trilha gerada, conhecida como *epílogo* é usada para enviar dados ao programa em código de máquina. Uma outra parte da trilha, conhecida como *prólogo* é usada pelo programa em código de máquina para enviar resultados computados de volta para o interpretador. Além disto, existem blocos especiais, denominados *saídas laterais*, que lidam com situações excepcionais que podem ocorrer durante a execução da trilha. Por exemplo, caso somente o ciclo $\{2, 3, 5\}$ seja compilado, então código de tratamento de exceções deve ser gerado para lidar com os casos em que a aresta entre os Blocos 3 e 4 é tomada.

2.4.1 TraceMonkey

Para produzir código de máquina para programas *JavaScript*, o *TraceMonkey* utiliza o compilador *Nanojit*². O processo completo de compilação utiliza três representações intermediárias, um caminho que é reproduzido pela Figura 2.4.

²<https://developer.mozilla.org/en/Nanojit>

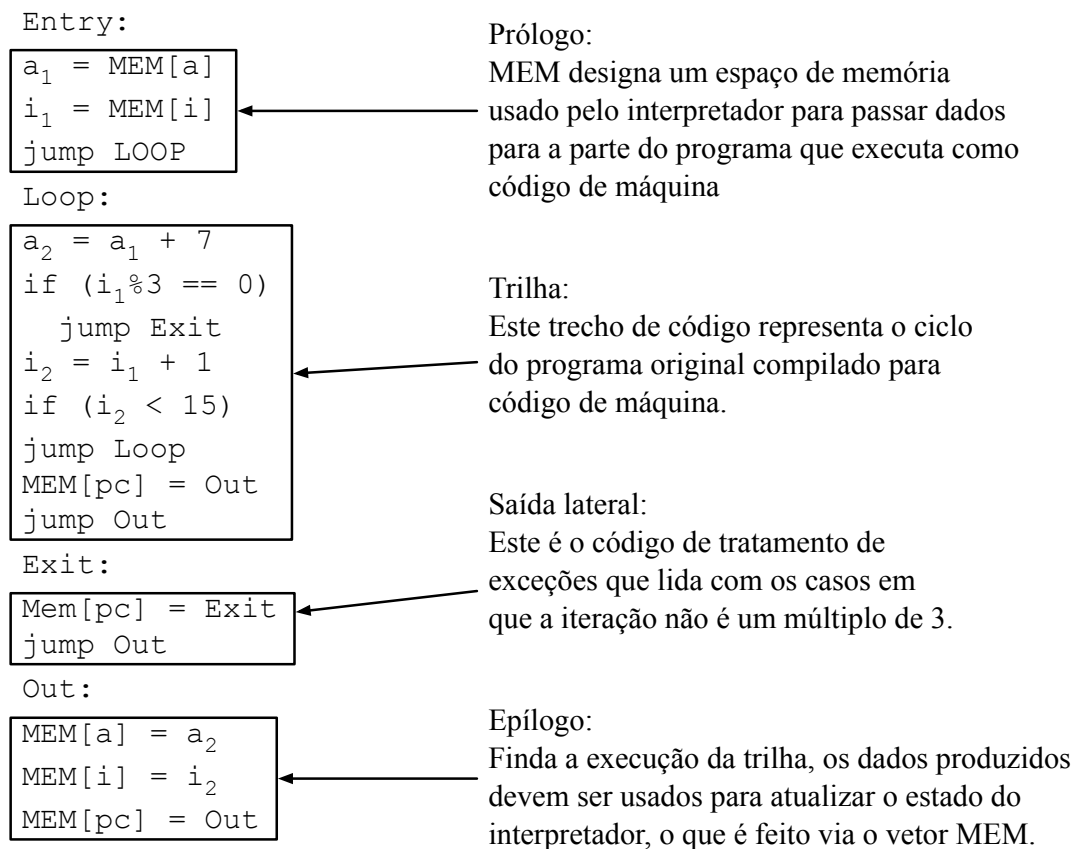


Figura 2.3: Exemplo de código de máquina que poderia ser produzido pelo compilador de trilhas.

1. **AST:** árvore de sintaxe abstrata produzida por um programa em *JavaScript*
2. **Bytecodes:** um conjunto de instruções de pilha que é diretamente interpretado pelo *SpiderMonkey*
3. **LIR:** representação de baixo nível em formato de código de três endereços que o *Nanojit* recebe como entrada.

SpiderMonkey não foi originalmente concebido como um compilador *just-in-time*, fato que explica o porquê do número excessivo de passos intermediários entre o programa fonte e o código binário. Segmentos de instruções LIR – uma *trilha* no jargão utilizado pelo *TraceMonkey* – são produzidas de acordo com um algoritmo muito simples [Gal et al., 2009]:

1. Cada desvio condicional é associado a um contador inicializado com o valor zero.

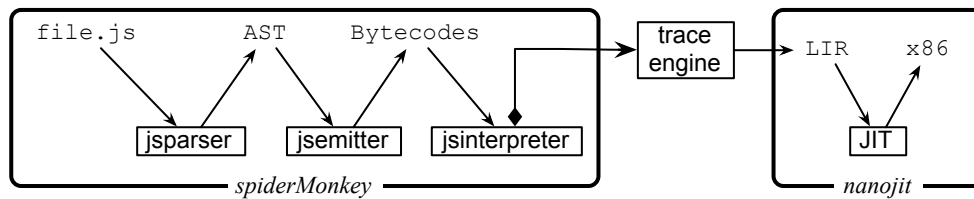


Figura 2.4: O compilador JIT Mozilla TraceMonkey.

2. Se o interpretador encontra um desvio condicional durante a interpretação do programa, então ele incrementa seu contador. O processo de verificar e incrementar contadores é chamado, de acordo com o jargão do *TraceMonkey*, de *fase de monitoramento*.
3. Se um contador contiver um valor maior que 1, e não existir ainda nenhuma trilha para este contador, então o motor de gravação de trilhas começa a traduzir os *bytecodes* em segmentos de LIR enquanto eles são interpretados. Este processo é chamado de *fase de gravação*.
4. Uma vez que o motor de trilhas encontra o desvio condicional original, aquele que iniciou o processo de gravação, o segmento LIR gerado é passado para o *Nanajit*.
5. O compilador *Nanajit* traduz o segmento LIR, incluindo os testes de *overflow* em código de máquina que é colocado diretamente na memória principal. O fluxo do programa então é alterado e começa a execução do código de máquina.
6. Após o começo da execução do código de máquina ou em caso de uma exceção acontecer, por exemplo, uma falha em um teste de *overflow* ou a saída precoce da trilha, a execução retorna ao interpretador.

2.4.1.1 Dos Bytecodes para LIR

O programa mostrado na Figura 2.5 é usado para ilustrar o processo de compilação de trilhas, e também para mostrar como a nova otimização funciona. Este é um programa artificial, claramente muito ingênuo se comparado aqueles encontrados no mundo real. Entretanto, ele contém os elementos necessários para elucidar didaticamente o funcionamento de um compilador de trilhas e da otimização. Este programa produz os *bytecodes* mostrados na Figura 2.5 (b). Observe que foi tomada a liberdade de simplificar a linguagem intermediária de *bytecodes* utilizadas pelo *TraceMonkey*.

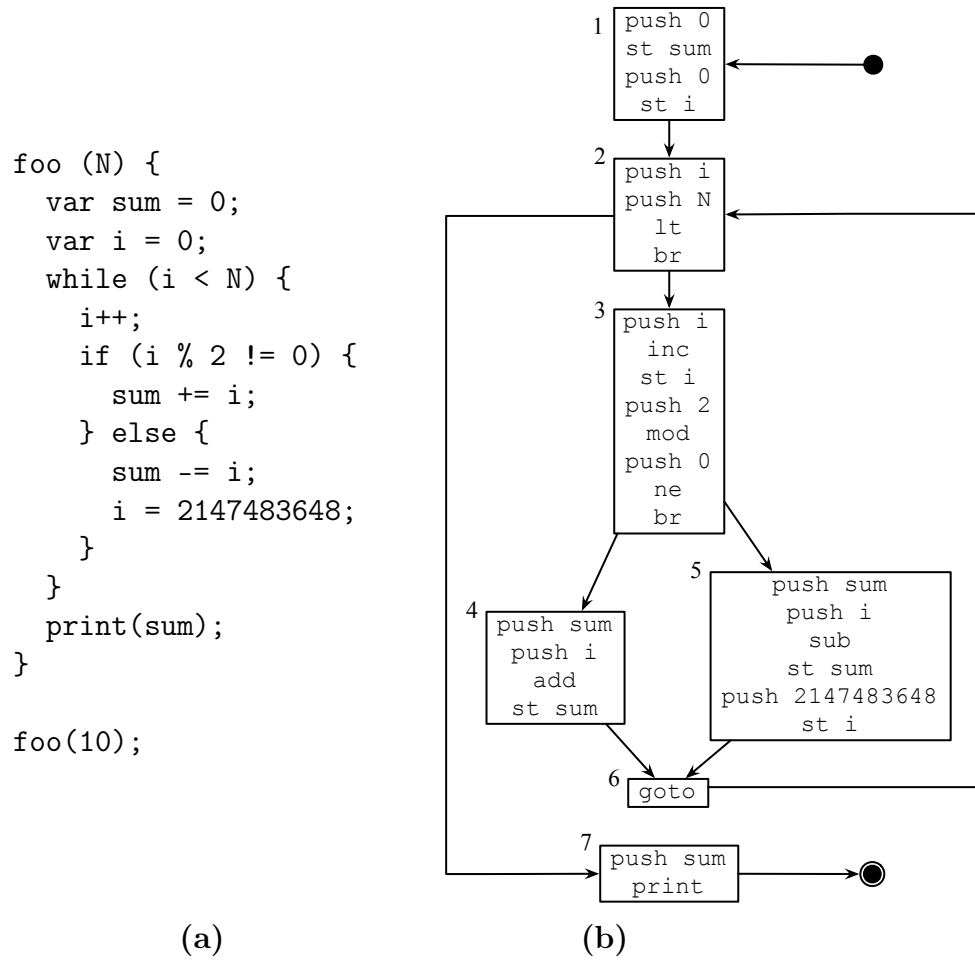


Figura 2.5: Um pequeno exemplo de um programa *JavaScript* e sua representação em bytecodes.

Uma das motivações mais importantes do projeto da otimização, apresentada em detalhes no Capítulo 3, está relacionada com o fato de o *TraceMonkey* ser capaz de produzir trilhas para caminhos de programa enquanto esses caminhos são visitados pela primeira vez. Este fato é consequência do algoritmo que o *TraceMonkey* usa para identificar trilhas. No começo do processo de interpretação, o *TraceMonkey* encontra um desvio no Bloco Básico 2, Figura 2.5 (b), e o sistema de trilhas incrementa o contador associado com aquele desvio. O próximo desvio a ser encontrado é no fim do Bloco Básico 3, e o contador desse desvio também é incrementado. O interpretador então encontra uma instrução do tipo `goto` no fim do Bloco Básico 6, que leva o fluxo de execução do programa de volta ao Bloco Básico 2. Neste momento o sistema de trilhas incrementa novamente o contador do desvio dentro desse bloco básico. A partir do momento que o sistema de trilhas encontra o valor 2 armazenado nesse contador,

ele sabe que está dentro de uma repetição, que irá começar a fase de gravação que produzirá segmentos LIR. Entretanto, este segmento não corresponde à primeira parte visitada do programa. Na segunda iteração da repetição, o Bloco Básico 5 é visitado em vez do Bloco Básico 4. Neste caso, o segmento gravado é formado pelos Blocos Básicos 2, 3, 5 e 6.

Isto acontece porque uma trilha que é monitorada pelo sistema de trilhas não é necessariamente a trilha que é gravada em formato de segmento LIR. É difícil remover testes de *overflow* durante a fase de gravação. Um teste condicional, tal como $a < N$, onde N é constante auxilia a colocar limites nas faixas de valores que a pode assumir. Entretanto, para saber que N é constante, é necessário garantir que o segmento completo que foi gravado não contém comandos que possam modificar o valor de N . Embora não seja possível remover testes de *overflow* diretamente durante a fase de gravação, é possível coletar restrições para as faixas de valores das variáveis nesse passo. Com base nesses dados, testes de *overflow* são eliminados do segmento LIR tão logo ele seja enviado para execução no *Nanojit*.

Continuando o exemplo, a Figura 2.6 mostra o casamento da trilha gravada e o segmento LIR produzido pelo sistema de trilhas.

2.4.1.2 Do LIR para o Código de Máquina

Antes de passar o segmento LIR para o *Nanojit*, o interpretador argumenta esse segmento com duas sequências de instruções. O prólogo e o epílogo. O prólogo contém código que mapeia valores aferidos pelo interpretador para o ambiente de execução onde o código binário é produzido e executado. O epílogo contém código para realizar o processo de mapeamento inverso. O *Nanojit* produz código de máquina para plataformas como *x86*, *ARM* e *ST40* a partir do segmento LIR gerado no interpretador.

Testes de *overflow* são implementados como saídas laterais (*side-exit*), uma sequência de instruções que tem duas funções:

1. recuperar o estado do programa logo depois do *overflow* acontecido, e
2. saltar a execução da trilha, voltando o controle para o interpretador.

A Figura 2.7 mostra uma versão simplificada do código *x86* que poderia ser produzido para a trilha do exemplo. Observe que a nova otimização não tem nenhum papel neste ponto do processo de compilação – ela foi incluída na discussão apenas para fornecer uma visão holística de um compilador de trilhas.

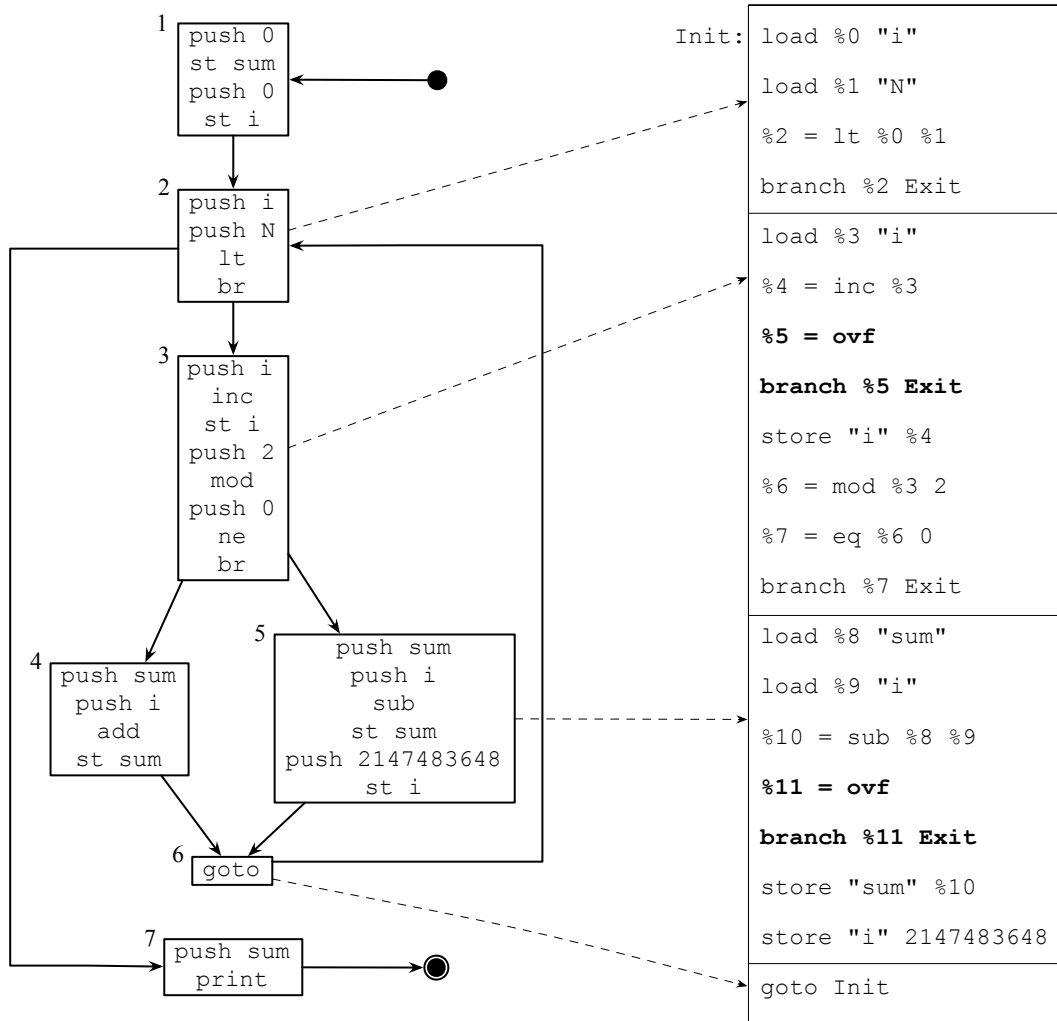


Figura 2.6: Esta figura ilustra o casamento entre a trilha gravada e o segmento LIR que foi produzida pelo sistema de trilhas.

2.5 Avaliação Parcial de Programas

A avaliação parcial de programas é uma técnica de geração automática de código cujo objetivo é aumentar a eficiência de programas em termos de tempo de execução [Jones et al., 1993]. Suponha um programa P que tenha duas entradas, identificadas como in_1 e in_2 . O resultado da avaliação parcial de P em relação à entrada in_1 é um novo programa Pin_1 , designado por residual ou especializado. O programa Pin_1 , quando executado sobre a entrada restante in_2 , produz o mesmo resultado que a execução de P sobre ambas as entradas. Avaliação parcial é um tipo de especialização de programas. A entrada in_1 é designada entrada estática e in_2 entrada dinâmica. O principal objetivo

(LIR)		(x86)	
Prologue:	Init:	Prologue:	RS1:
load %11 ITP[i]	load %0 "i"	pushl ...	movl ...
store "i" %11	load %1 "N"	movl ...	movl ...
load %12 ITP[N]	%2 = lt %0 %1	movl ...	movl ...
store "N" %12	branch %2 Exit	movl ...	movl ...
load %13 ITP[sum]	load %3 "i"	jmp Init	jmp Exit
store "sum" %13	%4 = inc %3	Init:	RS2:
goto Init	%5 = ovf	leal ...	movl ...
	branch %5 Exit	incl ...	movl ...
Exit:	store "i" %4	jvf RS1	movl ...
load %14 "i"	%6 = mod %3 2	movl ...	movl ...
store ITP[i] %11	%7 = eq %6 0	andl ...	movl ...
load %15 "N"	branch %7 Exit	testb ...	jmp Exit
store ITP[N] %12	load %8 "sum"	je Exit	
load %16 "sum"	load %9 "i"	movl ...	
store ITP[sum] %13	%10 = sub %8 %9	leal ...	
return	%11 = ovf	subl ...	
	branch %11 Exit	jvf RS2	
	store "sum" %10	movl ...	
	store "i" 2147483648	movl ...	
	goto Init	movl ...	
		cmpl ...	
		jnb Init	
		Exit:	
		movl ...	
		movl ...	
		movl ...	
		popl ...	
		leave	

Figura 2.7: O código LIR, logo após a inserção do prólogo e do epílogo, e a semântica do código *x86* produzido pelo Nanojit.

da avaliação parcial é o ganho em eficiência, pois se parte dos dados de entrada de um programa é conhecida, as estruturas do programa que dependem apenas dessa parte podem ser previamente computadas e o programa especializado contém apenas o código necessário para processar os dados ainda não conhecidos. Esse processo é descrito pela Figura 2.8.

2.5.1 Avaliadores Parciais: *On-line* e *off-Line*

Sumii [Sumii Kobayashi, 1999] classifica as técnicas de avaliação parcial de programas em dois tipos, *on-line* e *off-line*.

Avaliadores parciais *on-line*

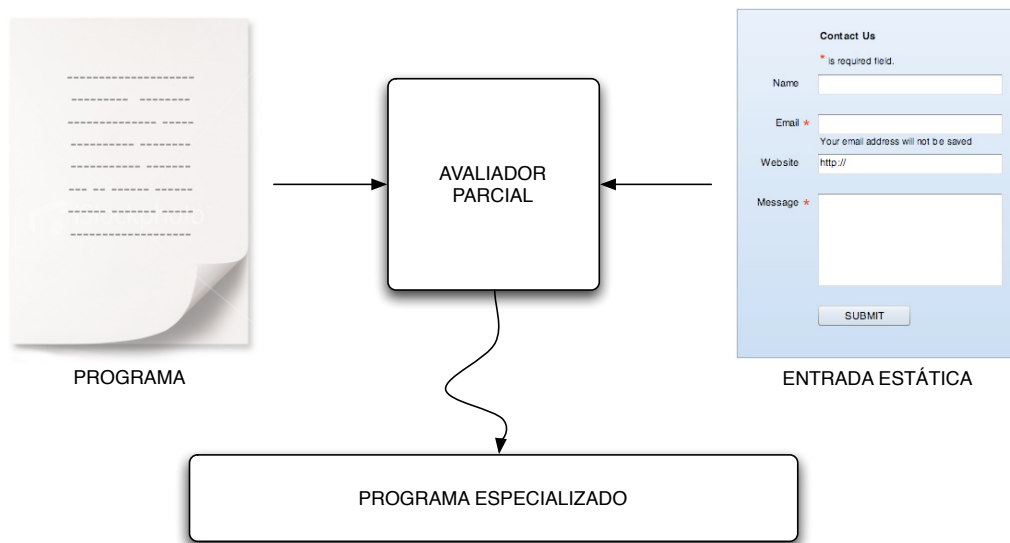


Figura 2.8: Avaliação parcial

Um avaliador parcial é denominado como sendo *on-line*, quando utiliza como argumentos um programa e um conjunto de entradas estáticas para realizar o processamento e gerar diretamente o programa especializado para o conjunto de entradas.

Avaliadores parciais *off-line*

Os avaliadores parciais denominados *off-line* trabalham em duas fases distintas. Na primeira fase é realizada uma análise para propagar informações abstratas sobre os valores estáticos correntes e sobre os valores dinâmicos contidos no código. Na segunda fase acontece a especialização.

Os avaliadores parciais *on-line* são considerados mais poderosos, uma vez que a avaliação parcial depende dos valores correntes, porém existe uma desvantagem em relação àqueles classificados como *off-line*. Estes costumam ser mais rápidos que os do tipo *on-line*.

2.6 Avaliadores Parciais e Geração de Código

Caso fosse possível para um compilador conhecer parte da entrada de um programa, ele poderia utilizar esta informação para melhorar a qualidade do código produzido, mas normalmente a entrada de um programa somente é conhecida durante a execução.

Entretanto, na compilação dinâmica, o programa é compilado durante à sua execução. Desta forma, o compilador tem acesso a alguns dos valores manipulados pelo programa. A avaliação parcial não é utilizada na compilação dinâmica tradicional, pois neste paradigma funções inteiras são traduzidas para código de máquina, e tais funções devem ser gerais o suficiente para serem invocadas diversas vezes com entradas possivelmente diferentes. A compilação de trilhas, por outro lado, abriu novas oportunidades para o emprego da avaliação parcial. Existem pelo menos duas otimizações de código que podem se beneficiar das técnicas de avaliação parcial e podem ser implementadas em um compilador de trilhas: o *desmembramento de laços* e a *eliminação de testes de overflow*. Essas otimizações são detalhadas respectivamente nas próximas duas seções.

A otimização proposta nesta dissertação baseia-se em técnicas de avaliação parcial de programas. Essa técnica tem sido usada para realizar várias otimizações de código. Shankar [Shankar et al., 2005] apresentam uma otimização que utiliza técnicas de avaliação parcial de programas para especializar programas escritos na linguagem *Java*. Shultz [Schultz et al., 2003] realizaram otimização semelhante a Shankar, também especializando programas no compilador *just-in-time* de Java. Vários ambientes de execução empregam esse tipo de técnica. Dentre eles, destacam-se: o compilador para a linguagem *Python*, *Psyco JIT*, descrito por Rigo em [Rigo, 2004] e também o ambiente de execução do Matlab [Elphick et al., 2003; Chevalier-Boisvert et al., 2010] e do Maple [Carette Kucera, 2007].

No contexto da compilação *just-in-time* a avaliação parcial tem sido usada mais como uma forma de especialização de tipo, ou seja, desde que o compilador possa provar que um valor pertence a determinado tipo de dado ele pode usar o tipo mais apropriado na geração de código de máquina [Chevalier-Boisvert et al., 2010; Chambers Ungar, 1989; Gal et al., 2009].

2.6.1 Desmembramento de Laços

Desmembramento de Laços é uma técnica originalmente concebida para maximizar a quantidade de paralelismo em um programa [Kennedy Allen, 2001]. Essa otimização é candidata a ser melhorada via utilização de informações de tempo de execução.

A Figura 2.9 ilustra essa otimização. Durante a execução do programa na Figura 2.9(a), o compilador sabe que o valor da variável n é, por exemplo, 4. Neste caso, em vez de produzir uma trilha como aquela vista na Figura 2.3, o compilador pode realizar o *desmembramento do laço* para criar a trilha vista na Figura 2.9(b).

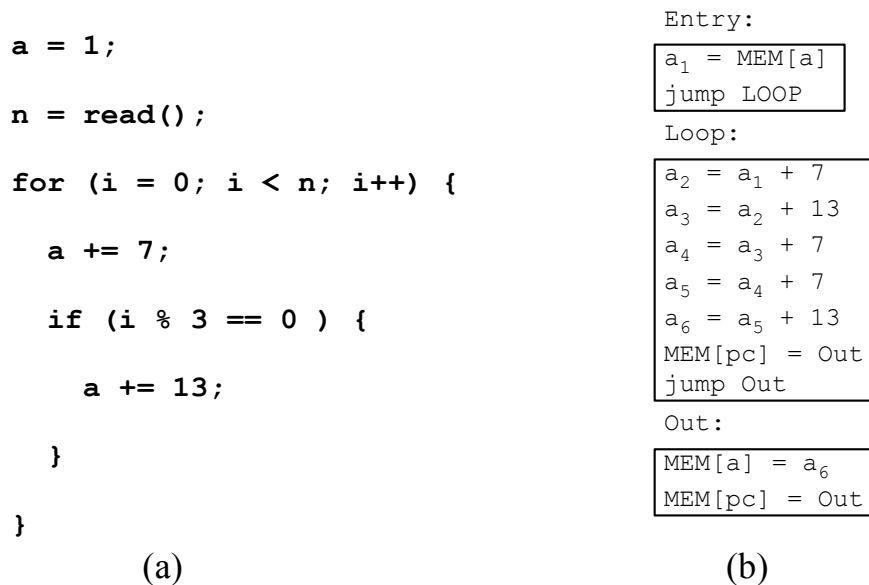


Figura 2.9: Um exemplo em que o conhecimento de valores de tempo de execução propicia a geração de código mais eficiente.

2.6.2 Eliminação de Testes de *Overflow*

Overflow é o fenômeno que ocorre quando o resultado de uma operação aritmética é maior que o espaço alocado pelo computador para armazenar este resultado. Em *JavaScript*, caso uma operação entre dois números inteiros produza um valor acima de 32 *bits*, este valor é automaticamente convertido para um número de ponto-flutuante. Isto implica que cada operação aritmética de soma ou multiplicação deve ser guardada por um teste que verifica o tamanho do resultado produzido. Contudo, caso valores em tempo de execução sejam conhecidos, podemos eliminar tais testes usando dos recursos da avaliação parcial. Esse é o caso do laço da Figura 2.3(a), caso o valor de n seja no máximo 4, como no exemplo anterior. Neste caso, a variável i será sempre menor que 4, e *overflows* não podem ocorrer devido a operações que escrevem nesta variável.

2.7 Análise de Largura de Variáveis

A otimização aqui proposta para remoção de testes de *overflow* é um tipo de análise de largura de variáveis (*range analysis*) [Harrison, 1977; Patterson, 1995]. Análise de largura de variáveis tenta descobrir os valores para os limites superiores e inferiores de um conjunto variáveis durante a execução do programa em determinado contexto.

Normalmente, esses algoritmos confiam em provadores de teoremas para realizar tal tarefa. Esta abordagem parece ser muito lenta para ser aplicada em compiladores *just-in-time*. Bodik *et al.* [Bodik et al., 2000] descreveram uma especialização de análise de largura de variáveis para a remoção de testes de limites de arranjos. Este algoritmo ficou conhecido como ABCD, e ele foi projetado para ser utilizado em compiladores *just-in-time*. Zhendong and Wagner [Su Wagner, 2005] descreveram um algoritmo de análise de largura de variáveis que pode resolver o problema em tempo polinomial. Stephenson *et al.* [Stephenson et al., 2000] usou uma análise em tempo polinomial para inferir a largura de cada variável inteira usada em um dado programa fonte. A otimização proposta nesta dissertação difere-se de todas estas abordagens anteriores, já que estas trabalham com representações estáticas do programa fonte. Este fato restringe severamente a quantidade de informações que as análises de largura de variáveis podem confiar. Conhecendo os valores das variáveis em tempo de execução, pode-se realizar uma otimização muito mais agressiva e rápida.

2.8 Conclusão

Neste capítulo foram abordados todos os fundamentos necessários para o desenvolvimento e implementação da nova otimização de código, apresentada no Capítulo 3, capaz de minimizar o número de testes de *overflow* em trilhas. Inicialmente, foram mostradas, as principais características das linguagens de programação interpretadas. Em seguida, foram apresentadas a linguagem de programação *JavaScript* e seus recursos. *JavaScript* foi escolhida para ilustrar o funcionamento de uma linguagem interpretada porque a implementação da otimização proposta foi feita em um compilador de *JavaScript*. Na sequência, descreveram-se os fundamentos dos compiladores *just-in-time*, contendo uma classificação desses compiladores em subtipos bem definidos.

Em seguida, foi apresentada uma visão abrangente sobre compilação de trilhas, ilustrada pelos compiladores *Tamarin-Trace* e principalmente pelo *TraceMonkey*. Finalmente, foram descritos os fundamentos da técnica de otimização de código conhecida como Avaliação Parcial de Programas e mostrado que otimizações de código como desmembramento de laços podem se beneficiar de informações de tempo de execução para produzirem programas mais eficientes.

Capítulo 3

Remoção de Testes de *Overflow* Via Análise Sensível ao Fluxo

Este capítulo apresenta uma nova técnica de otimização para compiladores de trilhas, capaz de reduzir o número de testes de *overflow* nas trilhas. Para atingir este objetivo é utilizada uma análise sensível ao fluxo [Patterson, 1995]. Essa análise baseia-se na direcionalidade acíclica de um grafo para determinar faixas de valores limite para as variáveis do programa. Esse grafo é denominado *grafo de restrições (constraint graph)* [Rossi et al., 2006] e possui quatro tipos de vértices:

Nome

cada vértice contém um contador que representa uma definição particular de uma variável. Esses vértices são limitados por um intervalo inteiro que denota a faixa de valores que a definição representada pode assumir.

Atribuição

Denota a cópia de um valor para uma variável.

Relacional

uma comparação relacional, usada para estabelecer limites nas faixas das variáveis. As operações de comparação consideradas são: igual (**eq**), menor que (**lt**), maior que (**gt**), menor ou igual (**le**), e maior ou igual (**ge**) .

Aritmético

operações aritméticas que podem causar um teste de *overflow*. Destacam-se dois tipos de vértices aritméticos: de operações binárias e unárias. Os vértices binários são os de adição (**add**), subtração (**sub**) e multiplicação (**mul**). Os vértices unários

são os de incremento (**inc**). e o de decremento (**dec**). A operação de divisão não é tratada porque ela legitimamente produz resultados de ponto-flutuante.

A análise sensível ao fluxo é realizada em duas fases: construção do grafo de restrições e a propagação sensível ao fluxo. Seções 3.1 e 3.2 descrevem respectivamente essas duas fases.

3.1 Construção do Grafo de Restrições

O grafo de restrições é construído durante a fase de gravação do TraceMonkey, ou seja, enquanto as instruções na trilha estão sendo visitadas e o segmento LIR está sendo produzido. Para associar as faixas de restrições às variáveis do programa fonte supõe-se que o mesmo esteja codificado em uma representação intermediária chamada *Extended Static Single Assignment* (e-SSA) [Bodik et al., 2000], um superconjunto da forma SSA [Cytron et al., 1991]. Na representação e-SSA, uma variável é renomeada em duas situações, após ter sido atribuída a ela um valor, ou após ela ter sido usada em um comando condicional.

Converter um programa para a forma e-SSA requer uma operação global do programa, um requisito que um compilador de trilha não pode atender. Entretanto, dado que a unidade de trabalho é uma trilha, e assim, uma linha simples de *bytecodes*, a conversão é possível, e acontece ao mesmo tempo que o grafo de restrições é construído, durante a fase de gravação de uma trilha.

A conversão para e-SSA funciona como a seguir:

1. mantêm-se contadores para cada variável.
2. Se for encontrado um uso para a variável v , então ela é renomeada para v_n , onde n é o valor correntemente associado ao contador de v .
3. Quando novas definições são encontradas, esse contador é incrementado.
4. Considerando que a variável é denominada com base em seu contador, o incremento dos contador de uma variável efetivamente cria uma nova definição de nome na representação proposta.

Dessa forma, apenas pela renomeação das variáveis a cada nova definição, é possível obter a trilha em representação SSA [Cytron et al., 1991].

A propriedade e-SSA advém da forma com que o algoritmo trata operadores de relação. No momento em que comandos condicionais são encontrados, como em, $a < b$,

o algoritmo aprende novas informações sobre as faixas de a e b . Assim, as variáveis a e b são redefinidas a partir do incremento de seus contadores.

Existem dois eventos que podem alterar os limites de uma variável:

Atribuições Simples

A atribuição simples é um evento que determina um valor único para uma variável.

Testes Condicionais

Os testes condicionais colocam uma restrição em um dos limites da variáveis, podendo ser o limite superior ou o limite inferior.

Estes eventos fazem com que o algoritmo incremente os contadores associados a àquelas variáveis. Portanto, é possível atribuir uma única faixa de restrições para cada nova definição de uma variável. Essas restrições de limites levam em consideração o valor atual da variável no momento em que ela é vista pelo sistema de trilhas. Esses valores são determinados via inspeção na pilha do interpretador. O seguinte algoritmo foi desenvolvido para construir o grafo de restrições.

1. inicializa os contadores associados às variáveis na trilha. Isto é feito ao mesmo tempo em que as variáveis são encontradas.
 - a) na primeira vez que uma variável é vista, seu contador é iniciado com valor zero, nas outras ele é incrementado.
 - b) Quando uma variável é vista pela primeira vez, ela é marcada como uma variável do tipo *input*, senão o contador é incrementado, e a variável é marcada como sendo do tipo *auxiliary*.
 - c) se uma variável for marcada como *input*, então seus limites superior e inferior são definidos com o valor atual que consta no interpretador, senão esses limites são definidos como desconhecidos.
2. Para cada instrução i visitada:
 - a) se i é uma operação relacional, por exemplo, do tipo $v < u$, é construído um vértice do tipo relacional que tem dois predecessores: os vértices das variáveis v_x e u_y , onde x e y são contadores. O vértice criado tem também dois sucessores, v_{x+1} e u_{y+1} .

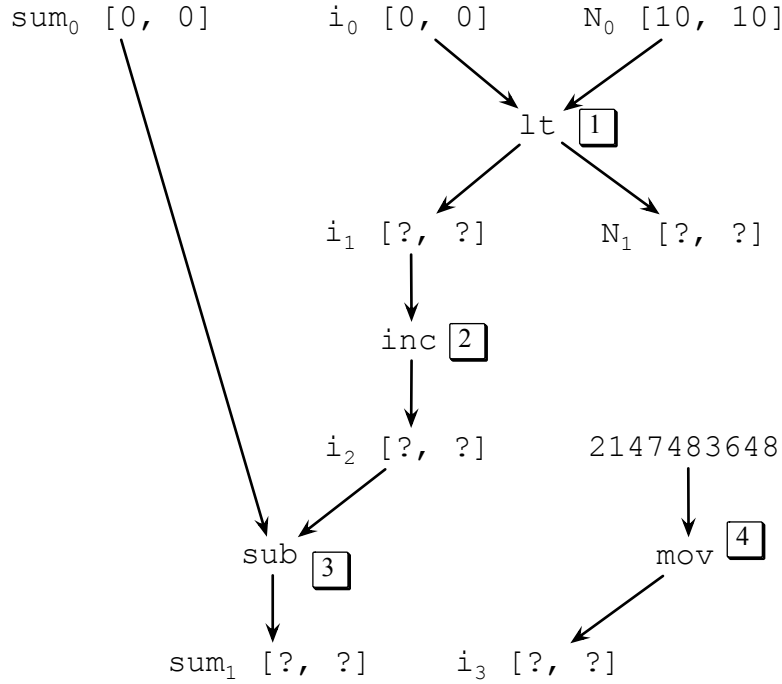


Figura 3.1: Grafo de restrições para a trilha mostrada na Figura 2.6 .Os números dentro das caixas mostram quando os vértices foram criados

- b) Para cada operação binária, por exemplo, do tipo $v = t + u$, é construído um vértice aritmético n , sendo os vértices relacionados com as variáveis t_x e u_y os predecessores de n , e seja v_z , seu sucessor.
- c) Para cada operação unária, por exemplo, do tipo $u++$, é construído um vértice n do tipo atribuição, com um predecessor u_x , e um sucessor u_{x+1} .
- d) Para cada atribuição de cópia, por exemplo, do tipo $v = u$, é construído um vértice que tem como predecessor u_x , e como sucessor v_{y+1} , assumindo y seja o contador de v .

A Figura 3.1 mostra o grafo de restrições para o exemplo da Seção 2.4.1, supondo que a função `foo` foi chamada com o parâmetro $N = 10$. Observe que foram colocados os seguintes limites nas variáveis do tipo *input*: $sum_0 \subseteq [0, 0]$, $i_0 \subseteq [0, 0]$ e $N_0 \subseteq [10, 10]$. Durante a construção do grafo de restrições, é importante manter uma lista com a ordem em que cada vértice é criado. Esta lista está representada nesta figura pelos números nas caixas, e é usada para guiar a fase de propagação de faixas.

Incremento: $x_1 = x_0 + 1$	
$\frac{x_0.l + 1 > \text{MAX_INT}}{x_1.l = +\infty}$	$\frac{x_0.u + 1 > \text{MAX_INT}}{x_1.u = +\infty}$
$\frac{x_0.l + 1 \leq \text{MAX_INT}}{x_1.l = x_0.l + 1}$	$\frac{x_0.u + 1 \leq \text{MAX_INT}}{x_1.u = x_0.u + 1}$
Adição: $x = a + b$	
$\frac{a.l + b.l < \text{MIN_INT}}{x.l = -\infty}$	$\frac{a.u + b.u < \text{MIN_INT}}{x.u = -\infty}$
$\frac{a.l + b.l > \text{MAX_INT}}{x.l = +\infty}$	$\frac{a.u + b.u > \text{MAX_INT}}{x.u = +\infty}$
$\frac{\text{MIN_INT} \leq a.l + b.l \leq \text{MAX_INT}}{x.l = a.l + b.l}$	$\frac{\text{MIN_INT} \leq a.u + b.u \leq \text{MAX_INT}}{x.l = a.u + b.u}$
Multiplicação: $x = a \times b$	
$\frac{a.l \times b.l < \text{MIN_INT}}{x.l = -\infty}$	$\frac{a.u \times b.u < \text{MIN_INT}}{x.u = -\infty}$
$\frac{a.l \times b.l > \text{MAX_INT}}{x.l = +\infty}$	$\frac{a.u \times b.u > \text{MAX_INT}}{x.u = +\infty}$
$\frac{\text{MIN_INT} \leq a.l \times b.l \leq \text{MAX_INT}}{x.l = \text{MIN}(a.\{l, u\} \times b.\{l, u\})}$	$\frac{\text{MIN_INT} \leq a.u \times b.u \leq \text{MAX_INT}}{x.l = \text{MAX}(a.\{l, u\} \times b.\{l, u\})}$

Figura 3.2: Propagação de faixas para vértices aritméticos.

3.2 Propagação de Faixas de Restrições

Durante a fase de propagação de faixas de restrições, são determinados quais testes de *overflows* são necessários e quais podem seguramente ser removidos do código alvo. A propagação de restrições é sempre precedida por uma etapa de inicialização, onde todas as variáveis do tipo *input* são substituídas pelos limites $] - \infty, +\infty[$ se elas foram atualizadas dentro da trilha. É o caso, por exemplo, das variáveis `sum0` e `i0` no exemplo da Figura 3.1. De forma a propagar as faixas de intervalos, visita-se em ordem topológica todos vértices aritméticos e relacionais do grafo de restrições. A ordem topológica é dada pela idade de um vértice. Os primeiros vértices criados

Comparação para menor que: $(a_1, b_1) \leftarrow (a < b)?$	
$a_1.u = \text{MIN}(a.u, b.l - 1)$	$b_1.l = \text{MAX}(b.l, a.u + 1)$
$a_1.l = a.l$	$b_1.u = b.u$
Comparação para maior que: $(a_1, b_1) \leftarrow (a > b)?$	
$b_1.u = \text{MIN}(b.u, a.l - 1)$	$a_1.l = \text{MAX}(a.l, b.u + 1)$
$a_1.u = a.u$	$b_1.l = b.l$
Comparação para menor ou igual: $(a_1, b_1) \leftarrow (a \leq b)?$	
$a_1.u = \text{MIN}(a.u, b.l)$	$b_1.l = \text{MAX}(b.l, a.u)$
$a_1.l = a.l$	$b_1.u = b.u$
Comparação para maior ou igual: $(a_1, b_1) \leftarrow (a \geq b)?$	
$b_1.u = \text{MIN}(b.u, a.l)$	$a_1.l = \text{MAX}(a.l, b.u)$
$a_1.u = a.u$	$b_1.l = b.l$
Comparação para igual: $(a_1, b_1) \leftarrow (a = b)?$	
$a_1.l = \text{MAX}(a.l, b.l)$	$b_1.l = \text{MAX}(a.l, b.l)$
$a_1.u = \text{MIN}(a.u, b.u)$	$b_1.u = \text{MIN}(a.u, b.u)$

Figura 3.3: Propagação de restrições para vértices relacionais.

durante a construção do grafo são os primeiros a serem visitados. Observe que essa ordem é obtida de graça, simplesmente guardando os vértices aritméticos e relacionais em uma fila quando eles são criados durante a construção do grafo.

O TraceMonkey produz trilhas a partir de programas JavaScript estritos, isto é, onde cada variável é definida antes de ser usada. Entretanto, seguindo a ordem de criação dos vértices, a propagação de faixas garante que sempre é possível alcançar um vértice aritmético, relacional ou de atribuição, e que todos os vértices do tipo *nome* que apontam para eles já foram vistos antes [Budimlic et al., 2002].

Cada um dos vértices, aritmético e relacional, causa a propagação das restrições

de uma forma particular. A Figura 3.2 mostra como a atualização é realizada para vértices aritméticos. A notação usada para exemplificar a propagação de faixas nessa figura é matemática simples, com aritmética de precisão infinita usando as regras definidas no padrão de números de ponto flutuante IEEE 754[Hough, 1981]. Os intervalos $[l, u]$ denotam a associação aos vértices a pelos vértices $a.l$ e $a.u$. Foram omitidos os casos de decremento e subtração por eles serem similares aos casos de incremento e adição respectivamente. Por simplicidade, os testes de *overflow* são verificados usando matemática de precisão infinita. A implementação atual usa *wrapping semantics*[Bozsahin, 1998], por exemplo, se $x_0 + 1 > x_1$, então $x_1 = +\infty$. Usa-se $a.\{l, u\} \times b.\{l, u\}$ como um valor padrão para $(a.l \times b.l, a.l \times b.u, a.u \times b.l, a.u \times b.u)$

Somente os vértices aritméticos, que aparecem na Figura 3.2 podem causar *overflows*, e cada um dos quatro tipos de vértices aritméticos podem causar *overflows* de uma forma diferente. Toda vez que uma das regras de inferência determina que sejam colocados limites nos intervalos, seja $+\infty$ ou $-\infty$, é mantido o teste de *overflow* associado ao vértice que está sendo visitado. A Figura 3.3 mostra como a atualização é realizada para vértices relacionais. Estes vértices não estão associados com nenhum teste de *overflow*, mas eles fornecem informações essenciais para definir restrições a serem associadas as variáveis do programa, como mostram as regras de inferência.

Após a propagação das restrições, tem-se uma estimativa conservativa dos intervalos que cada variável inteira pode assumir durante a execução do programa. Isto permite examinar o segmento LIR antes de ele ser passado para o *Nanojit*, removendo todos os testes de *overflow* desnecessários, apurados pela análise. Ou seja, uma vez que as restrições dos limites de cada variável envolvida em uma operação são conhecidas, a otimização pode remover os testes de *overflow* associados, caso sejam redundantes.

A Figura 3.4 mostra esse passo, onde foram removidos testes de *overflow* para a operação *inc*. Como *inc* recebe como entrada um vértice limitado ao intervalo $[-\infty, 9]$, então, sua saída nunca será maior que 10. Observe entretanto, que a análise não demonstra que o teste na operação *sub* também é desnecessário, embora este seja o caso.

3.3 Análise de Complexidade do Algoritmo Proposto

O propósito do algoritmo é ser executado em tempo linear no número de instruções da trilha. Para constatar esse fato, observe os seguintes pontos:

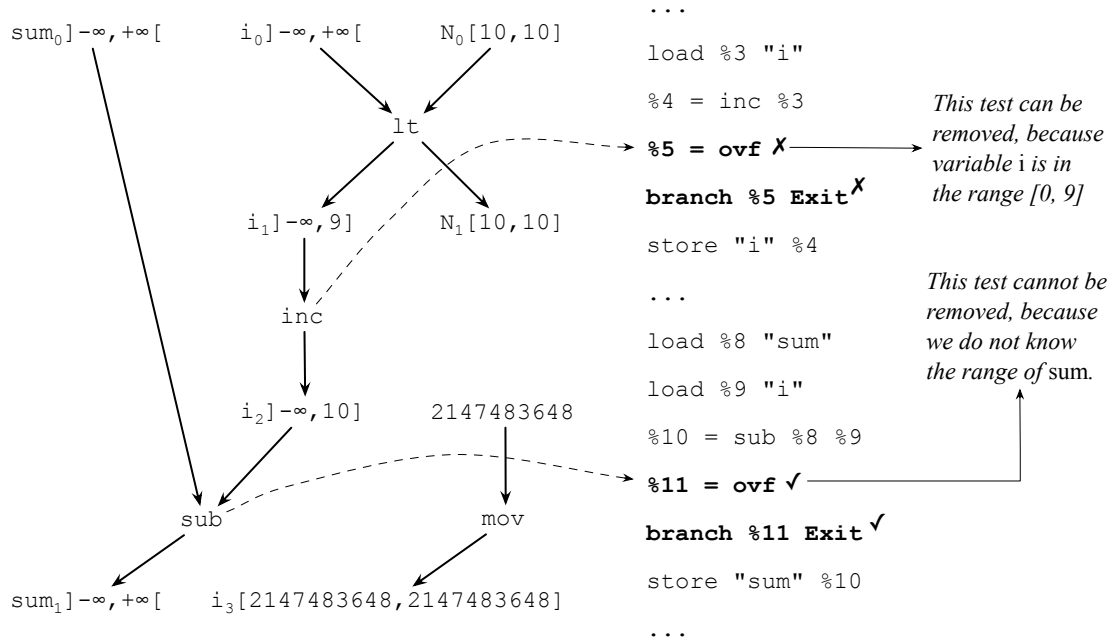


Figura 3.4: Segmento LIR antes de ser passado ao Nanojit

- o grafo de restrições possui um número de vértices do tipo relacional ou aritmético que é proporcional ao número de instruções da trilha, e o método de atualização é chamado apenas uma vez para cada um desses vértices.
- Durante a análise, o grafo de restrições é percorrido apenas uma vez durante a fase de propagação.
- Não é realizado qualquer tipo de pré-processamento no grafo de forma a ordená-lo topologicamente, uma vez que esta ordenação é obtida pela sequência em que as instruções são visitadas pelo programa fonte.
- O algoritmo é também linear em termos de espaço, porque cada tipo de vértice aritmético possui um número constante de predecessores e sucessores.

Essa baixa complexidade do algoritmo proposto contrasta com a complexidade da maioria dos algoritmos que percorrem grafos, os quais são normalmente $O(E)$, onde E é o número de arestas no grafo [Tarjan, 1971]. Estes algoritmos, têm no pior caso, complexidade quadrática no número de vértices, considerando que $E = O(V^2)$. O algoritmo proposto nessa dissertação não sofre desse problema, porque, no nosso caso, obtemos $E = O(V)$.

3.4 Implementação do Algoritmo

O algoritmo de análise sensível ao fluxo apresentado neste capítulo foi implementado no compilador *TraceMonkey*, revisão do dia 2010-10-01. Esta implementação trata os cinco tipos de operação aritmética descritas na Seção 3.1, nominalmente, adições, subtrações, incrementos, decrementos e multiplicações. A implementação atual realiza a análise de limites descrita na Seção 3.2 durante a fase de gravação do *TraceMonkey*, ou seja, enquanto um segmento de *bytecodes* de *JavaScript* está sendo traduzido para segmentos LIR. Em paralelo à implementação do algoritmo, foi realizada também uma modificação no *Nanojit* para que este removesse os testes de *overflow* apontados como desnecessários pela análise.

Para que a implementação da otimização proposta fosse possível foi necessário modificar o comportamento padrão do *TraceMonkey*. Os seguintes recursos foram modificados: o interpretador, o gravador de trilhas e o *Nanojit*. A seguir são apresentados cada um deles com a nova semântica.

Interpretador - *jsinterpret.cpp*

A otimização proposta analisa os *bytecodes* que estão sendo executados. Para realizar essa tarefa, para cada *bytecode* visto no interpretador é invocada uma chamada ao método *js_RecordBytecode* da classe *JSBytecodeOptimization*. A otimização também monitora no interpretador o momento que uma trilha começa a ser gravada. Quando isso ocorre, a otimização começa a construir o gráfico de restrições para a nova trilha.

Gravador de Trilhas - *jstracer.cpp*

A otimização proposta modifica o comportamento do mecanismo de gravação de trilhas do *TraceMonkey* em dois momentos:

1. Na geração das instruções LIR de teste de *overflow*

Toda vez que um teste de *overflow* for inserido no segmento de LIR, a otimização proposta guarda uma referência para essa instrução e realiza a associação entre o teste de *overflow* e o vértice no grafo de restrições que gerou o teste. Esta informação é usada no futuro, após a fase de propagação de restrições. Se a análise sensível ao fluxo identificar que aquela operação não causa *overflow*, então a instrução LIR associada ao vértice é colocada em um mapa que guarda quais instruções podem ser desprezadas no *Nanojit*.

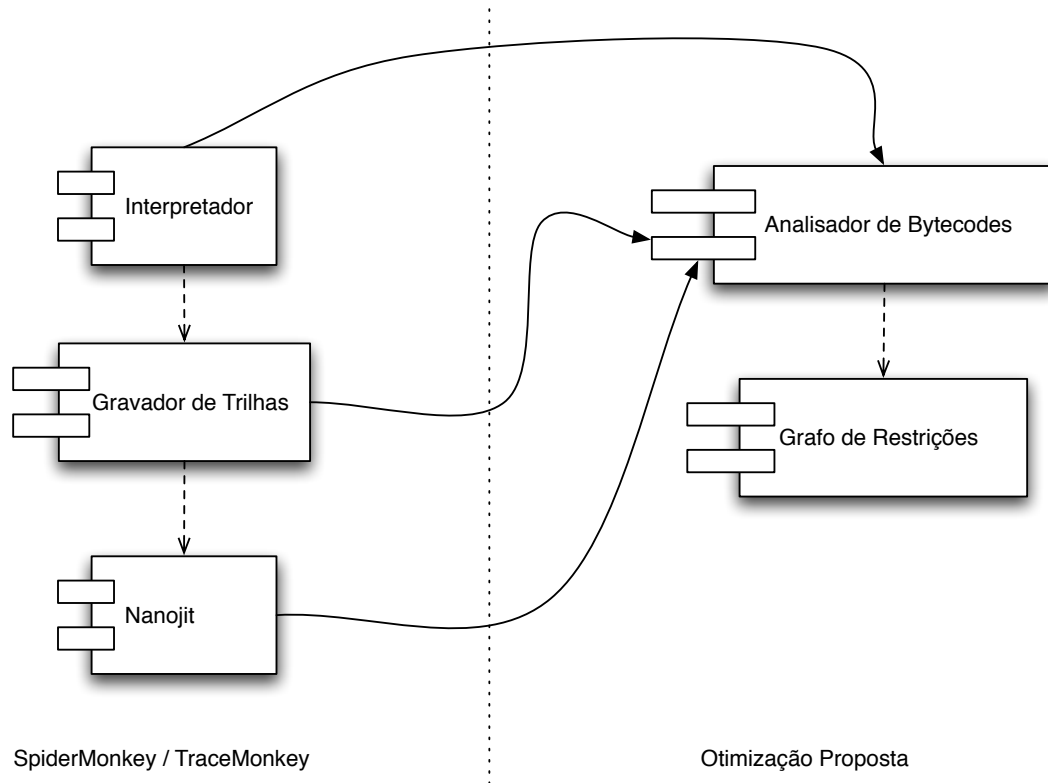


Figura 3.5: Componentes

2. Na compilação do segmento LIR

Ao chegar no fim da trilha, o gravador de trilhas envia o segmento LIR gerado para o *Nan JIT*. Nesse momento, a otimização proposta realiza a propagação das restrições, e obtém como resultado um mapa de instruções que podem ser desprezadas pelo *Nan JIT*.

Nan JIT - *nanojit assembler.cpp*

O método *gen* da classe *Assembler* do *Nan JIT* é responsável por transformar um dado segmento LIR em código nativo. Esse método foi alterado para que toda vez que for necessário traduzir uma instrução LIR de teste de *overflow* para suas respectivas instruções nativas, seja verificado no mapa de instruções gerado pela otimização proposta, se essa instrução é realmente necessária. Em caso negativo, a geração do teste de *overflow* não é realizado.

A Figura 3.5 apresenta as dependências entre os componentes do compilador *TraceMonkey* e dos componentes da otimização proposta. Internamente, a implementação

da otimização utiliza dois grandes componentes: o analisador de *bytecodes* e o grafo de restrições.

Analisador de Bytecodes - JSBytecodeOptimization.cpp

Este componente é responsável pela interface de comunicação entre o grafo de restrições e a análise sensível ao fluxo com a implementação do compilador. Ele intercepta os *bytecodes* que estão sendo executados pelo interpretador. Também é responsável por obter os valores atuais para cada variável.

Grafo de Restrições - JSConstraintGraph.cpp

Este componente contém as estruturas de dados que armazenam os vértices e arestas do grafo de restrições, bem como o algoritmo sensível a fluxo que é capaz de identificar vértices onde seus respectivos testes de *overflow* são desnecessários.

A Figura 3.6 apresenta um diagrama de classes simplificado para a implementação. Nesse diagrama foram suprimidos os métodos e atributos privados, bem como os métodos públicos auxiliares.

A implementação do algoritmo de análise sensível ao fluxo, em sua versão atual, possui algumas deficiências, isto se dá, principalmente devido a dificuldade encontrada em entender a implementação do *TraceMonkey*. Acreditamos que estas deficiências poderão ser melhoradas em um trabalho futuro, como descrito na Seção 4.1. Entre essas deficiências estão o não reconhecimento de variáveis globais e de estruturas de repetição como `foreach{...}`, `while(true){...}`.

3.5 Experimentos

Esta seção relata os resultados de uma série de experimentos que realizamos a fim de validar nosso algoritmo. Nesse sentido, organizamos esta seção de tal forma que cada um dos experimentos feitos tem o objetivo de responder cada uma das questões relacionadas a seguir. Identificamos estas questões como as mais importantes. As Subseções 3.5.3-3.5.8 representam cada uma delas. As questões são:

1. Quão eficiente é o algoritmo desenvolvido?
2. Qual é a redução de tamanho de código em função da eliminação dos testes de *overflow*?
3. Qual é o efeito do algoritmo no tempo de execução do *TraceMonkey*?

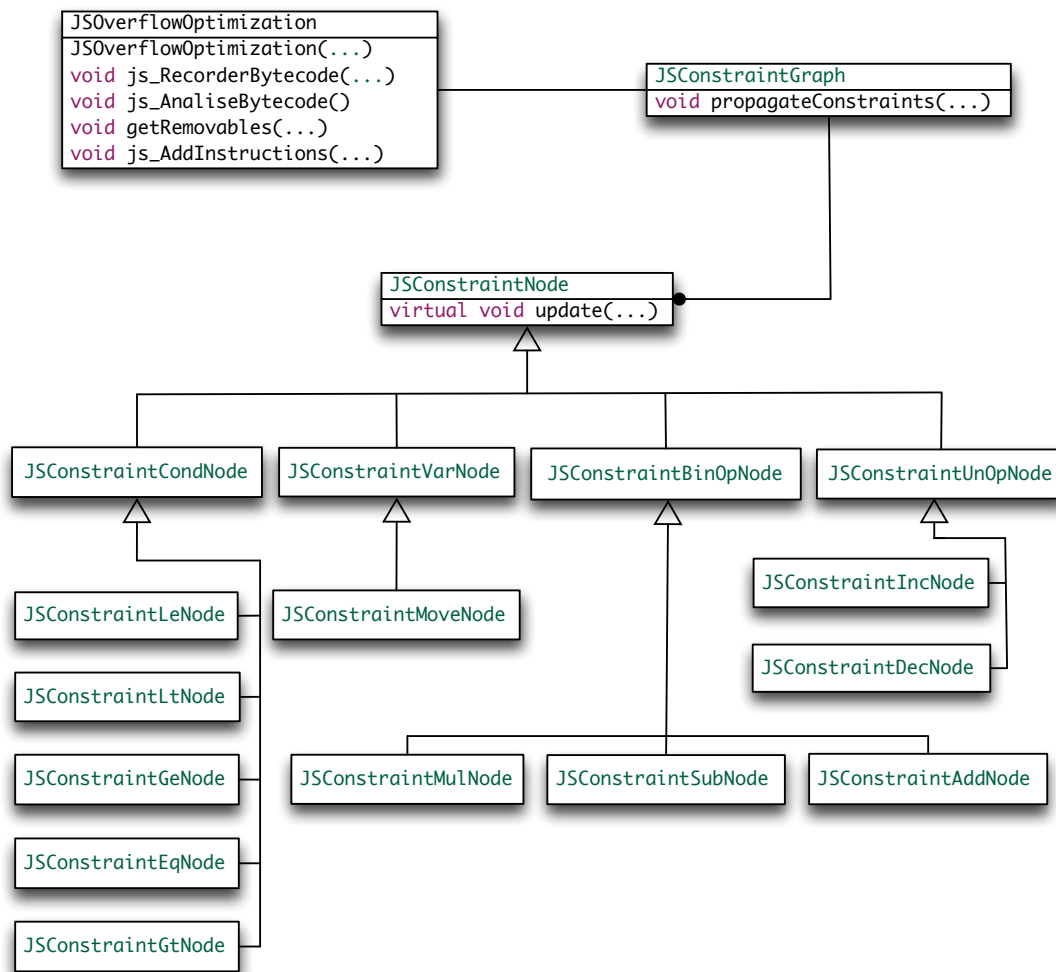


Figura 3.6: Classes

4. Por que alguns testes de *overflow* não são removidos?
5. Qual é o ganho em conhecer os valores em tempo de execução das variáveis?
6. Por que os resultados de efetividade são tão diferentes entre as coleções *Trace-Test* e *Alexa*?

3.5.1 Benchmarks

O algoritmo foi corretamente compilado e executado para cerca de um milhão de linhas de código em *JavaScript* de três diferentes *benchmarks* de programas de referência:

Alexa top 100:

Os 100 sítios mais visitados segundo o índice *Alexa*¹. Esta lista inclui sítios como Google, Facebook, Youtube, etc. Foi seguida a metodologia definida por Richards *et al.* [Richards et al., 2010], onde cada sítio é manualmente visitado com uma versão instrumentalizada do *Firefox*. Observe que são apresentados resultados para apenas 80 desses 100 sítios. Isto ocorre porque não foram encontrados testes de *overflow* em 20 desses sítios.

Trace-Test:

Conjunto de testes unitários utilizados pelo *TraceMonkey*². Esta conjunto inclui scripts de outras coleções populares como os do *Webkit Sunspider* e do *Google v8*. A maioria dos *scripts* não contém operações aritméticas, assim, os resultados mostrados são para apenas 224 *scripts* que continham no mínimo um teste de *overflow*.

PeaceKeeper:

Peacekeeper é uma ferramenta online de *benchmark* para navegadores. Ele realiza a comparação entre diversos navegadores a partir de uma coleção de *script* heterogênea, que inclui algoritmos para expressões regulares e geração de gráficos em 3D. Esse *benchmark* é largamente utilizada pela indústria³.

3.5.2 O Hardware

A versão modificada do compilador *TraceMonkey* foi utilizada em dois tipos diferentes de *hardware*:

Intel x86

2GHz Intel Core 2 Duo, com 2GB of RAM, sistema operacional Mac OS 10.5.8.

ST40-300

Um processador de tempo real fabricado pela *STMicroelectronics*. O *ST40* roda o sistema operacional *STLinux 2.3* (kernel 2.6.32), possui *clock* de 450MHz clock, provê 200MB de memória física e possui 512MB de memória virtual. Também contém dois níveis de cache com 32KB separados para instruções e 32KB para dados no Nível L1 e 256KB unificados no Nível L2.

¹<http://www.alexa.com/>

²<http://hg.mozilla.org/tracemonkey/file/c3bd2594777a/js/src/trace-test/tests>

³<http://service.futuremark.com/peacekeeper/index.action>

3.5.3 Eficácia do Algoritmo Proposto

A Figura 3.7 mostra a efetividade do algoritmo ao ser executado nas páginas reais do *Alexa* e nos *scripts* do *Trace-Test*. Os *scripts* estão ordenados no eixo x de acordo com a eficácia do algoritmo. Na média, foram removidos 91.82% dos testes de *overflow* encontrados no *benchmark Trace-Test*, e 53.50% dos testes de *overflow* do *Alexa*. Esta é uma média geométrica, ou seja, o algoritmo se mostrou efetivo na remoção de testes de *overflow* na maioria dos *scripts*.

Em particular, o algoritmo consegue remover a maioria dos testes de *overflow* usados em contadores que indexam estruturas de repetição simples. Entretanto, a média aritmética do percentual de testes de *removidos* é muito mais baixa em função de ruídos. Três *scripts* que fazem parte do *SunSpider*, incluídos no *Trace-Test*, lidam com criptografia e manipulam números grandes. Eles são responsáveis por 8.756 testes dos quais são removidos apenas 347. Em termos absolutos, foram removidos 961 dos 11.052 testes no *benchmark Trace-Test*, e 2680 de 11.126 testes no *benchmark Alexa*. A média aritmética nesse caso é de 8.7% para o *benchmark Trace-Test* e 24.08% para o *Alexa*.

A Figura 3.7 não contém um gráfico para o *PeaceKeeper* porque este *benchmark* contém apenas um *script*. Foram removidos 700 de 859 dos testes encontrados nos *scripts*.

3.5.4 Avaliação da Redução do Tamanho do Código em Função da Eliminação dos Testes de Overflow

Na plataforma x86, o *TraceMonkey* usa 8 instruções para implementar cada teste de *overflow*, incluindo instruções do tipo *branch-if-overflow* e *load*, usadas respectivamente para tomar o desvio em caso de *overflow* e para reconstruir o estado do interpretador. Além de uma instrução final de *jump* para retornar o controle ao interpretador.

No microprocessador *ST4*, o *TraceMonkey* usa 10 instruções. Todas essas instruções são eliminadas após a remoção de um teste de *overflow*. A Figura 3.8 mostra a média de redução de tamanho que o algoritmo implementado obtém em cada plataforma alvo.

Na média, o tamanho de cada *script* foi reduzido em 8.83% na plataforma *ST4*, e 6.63% no *x86*.

Na maioria dos *scripts*, a redução de tamanho foi modesta, entretanto existem casos em que o algoritmo diminuí o tamanho do binário em cerca de 60.0%. Observe que, por usar inteiros como representação de número de ponto-flutuante, o *TraceMonkey* já

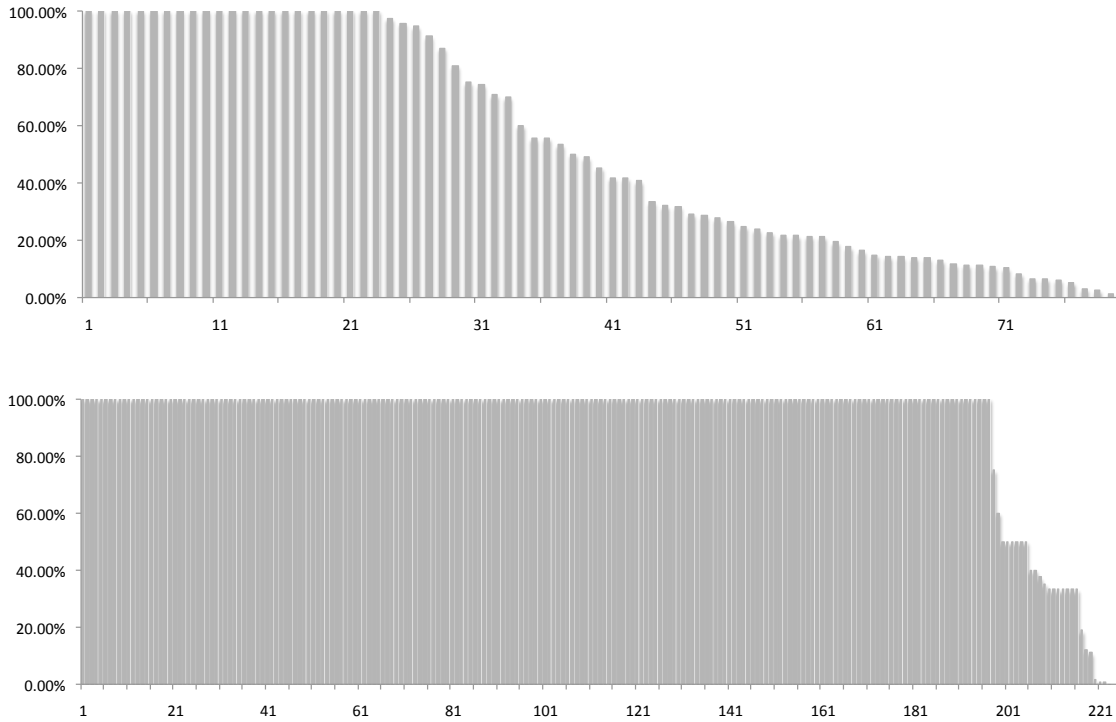


Figura 3.7: Efetividade do algoritmo em termos de percentual de testes de *overflow* removidos por *script* (Cima) *Alexa*; média geométrica 53.50%. (Baixo) *Trace-Test*; média geométrica: 91.82%; *Hardware*: mesmos resultados para ST4 e x86. Os 224 *scripts* estão ordenados pela média de efetividade.

reduz em média 31.47% do tamanho dos binários. Este dado é referente à plataforma *ST4*.

3.5.5 Avaliação do Efeito do Algoritmo em Relação ao Tempo de Execução do TraceMonkey

Na média, o algoritmo aumentou o tempo de execução do *TraceMonkey* em 2.56%, como mostrado na Figura 3.9. Este tempo de execução inclui:

- o tempo do compilador realizar a análise sintática
- o tempo de análise sintática e de interpretação do *script*
- o tempo do compilador *JIT* gerar os binários.
- o tempo de execução do binário gerado

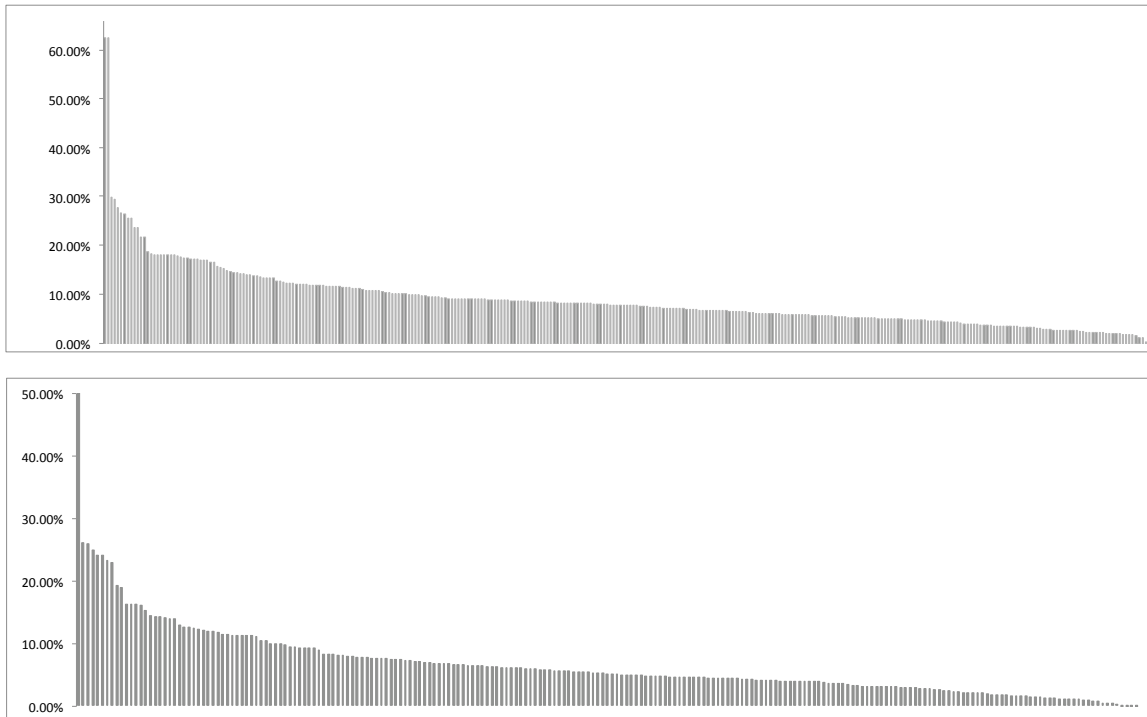
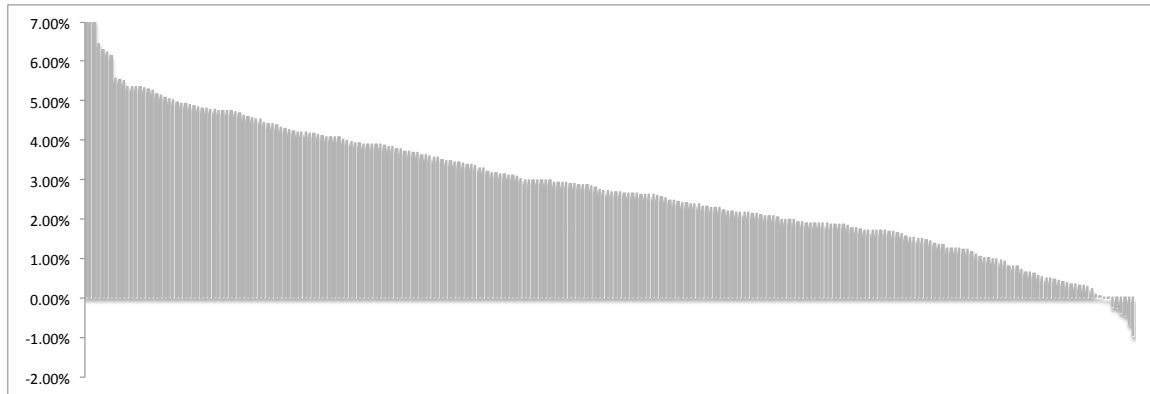


Figura 3.8: Redução de tamanho por *script* em duas diferentes arquiteturas. (Cima) *ST4*; média geométrica 8.83%. (Baixo) *x86*; média geométrica: 6.63% *Benchmark: Trace-Test*. Os 224 *scripts* estão ordenados pela média de redução

Os testes foram executados na plataforma *x86*. Embora o algoritmo aumente o tempo do compilador *just-in-time*, ele diminui o tempo de execução dos *scripts*. Mas, mesmo assim, o aspecto tempo é negativo. Isto acontece por duas razões:

- o Algoritmo foi implementado como um protótipo e não se considerou otimizações relacionadas ao desempenho.
- A maioria dos *scripts* é executada por um curto período de tempo, e por esse motivo, o efeito da remoção dos testes é negligenciado.

Em relação ao último ponto, em se tratando do *x86*, o único impacto de um teste de *overflow* em um *script* é o custo da execução da instrução *branch-if-overflow*. Entretanto, essa instrução naturalmente é predita como não tomada adicionando somente 0.5 ciclos no tempo de execução do programa.



Impacto do Algoritmo no tempo de execução do *TraceMonkey*. *Benchmark: Trace-Test. Hardware: x86*. Cada *script* foi executado 8 vezes. O tempo mais alto e o mais baixo foram desprezados. Na média, o Algoritmo aumentou o tempo de execução em 2.56%.

Figura 3.9: Tempo de Execução. Os 224 *scripts* estão ordenados pela média de incremento no tempo de execução

3.5.6 Análise dos Testes de *Overflow* não Removidos do Programa

Foi realizado um estudo manual nos 42 menores programas do conjunto Trace-Test dos quais não foram removidos ao menos um teste de *overflow*. Identificamos que cada um desses testes contém uma estrutura de repetição indexada por uma variável de indução. A Figura 3.10 explica o que foi encontrado nessa análise. Em 69% das trilhas, o Algoritmo falhou ao remover o teste de *overflow* porque a variável de indução estava limitada por uma variável global. Como explicado anteriormente, a implementação atual não é capaz de identificar valores de variáveis globais, então, não foi possível usá-las na estimação dos limites.

Em 17% dos casos restantes, a trilha foi produzida após uma estrutura de repetição do tipo *foreach*, que a implementação atual não é capaz de tratar.

Uma vez que o algoritmo trata esses casos, ele será capaz de remover ao menos mais um teste de *overflow* em cada um desses 42 *scripts* analisados.

Destacamos apenas um caso onde o algoritmo falha legitimamente ao não conseguir remover o teste de *overflow*. Neste caso, a semântica do programa pode levar a uma situação em que um teste de *overflow* legitimamente falha.

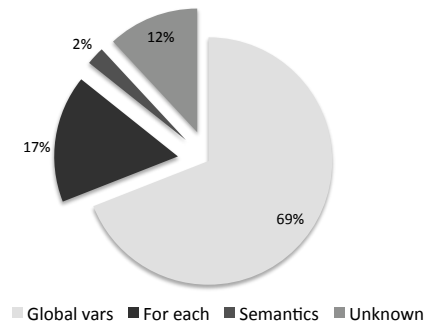


Figura 3.10: Um gráfico do tipo pizza mostrando as razões que impediram o algoritmo de remover testes de *overflow*.

3.5.7 Análise do Ganho Obtido Devido ao Conhecimento dos Valores das Variáveis em Tempo de Execução

O Algoritmo proposto é mais agressivo na estimativa da faixa de restrições para as variáveis que outras análises existentes descritas na literatura. Isto acontece porque o algoritmo é executado próximo ao compilador *just-in-time* [Chevalier-Boisvert et al., 2010], um fato que permite que os valores de tempo de execução sejam conhecidos, incluindo limites de estruturas de repetições. Estatisticamente, o algoritmo considera em valores constantes para iniciar a definição de limites para as variáveis.

Sem surpresas, uma implementação estática do mesmo Algoritmo aqui proposto, remove somente 477 testes de *overflow* do *benchmark Trace-Test*. Este número corresponde a 37% de todos os testes que são removidos quando as informações de tempo de execução são levadas em consideração.

3.5.8 Análise dos Resultados de Efetividade entre as Coleções *Trace-Test* e *Alexa*

A Figura 3.7 mostra que o Algoritmo é substancialmente mais efetivo quando aplicado nos programas do conjunto *Trace-Test* que nos programas dos 100 sítios mais visitados do *Alexa*. Isto acontece porque a maioria dos *scripts* do *Trace-Test* são pequenos e contém apenas laços simples controlados por variáveis localmente declaradas, que o Algoritmo reconhece.

Para comparar as duas coleções de *scripts*, foi realizada uma instrumentalização do *browser Firefox* que produz um histograma dos *bytecodes* que o *TraceMonkey* produz para cada *benchmark*. A Figura 3.11 mostra os resultados encontrados. Esse gráfico

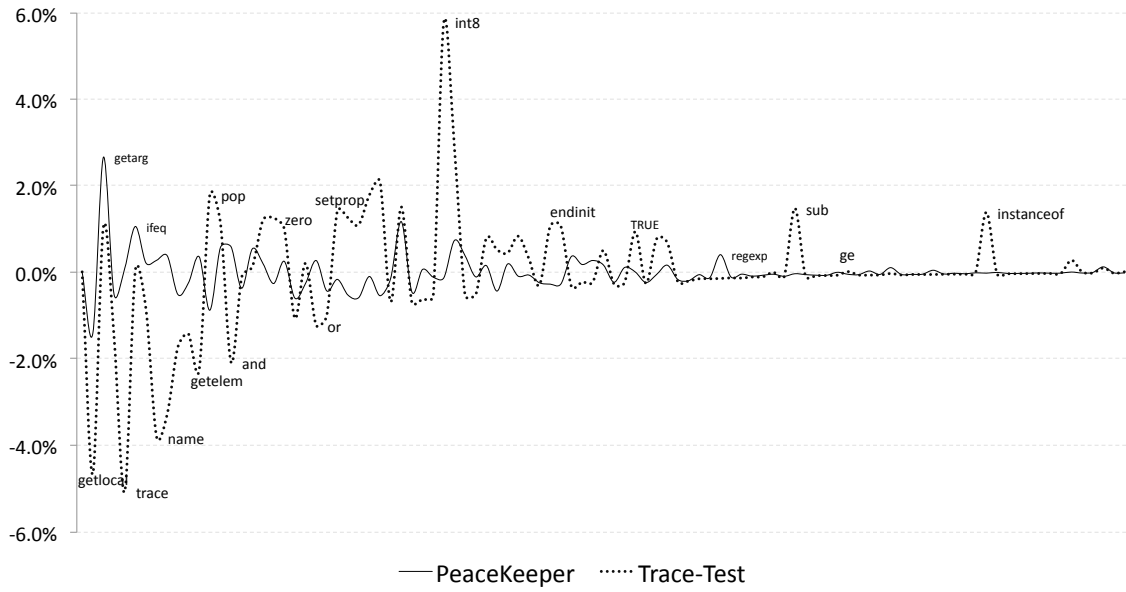


Figura 3.11: Um histograma para as instruções presentes nos *benchmarks* *Trace-Teste* e *PeaceKeeper*, comparados aos *bytecodes* encontrados na coleção *Alexa*. Eixo X: As instruções i estão ordenadas por suas frequências na coleção *Alexa*. Quanto mais suave a onda, mais similar ao *Alexa* é a coleção.

foi produzido pela plotagem de pares $(i, f(i))$, definidos como o seguinte:

1. seja I a lista formada pelos 100 *bytecodes* mais comuns encontrados no *benchmark* *Alexa*. Os *bytecodes* i foram ordenados de acordo com $a(i)$, o número de ocorrências de i no *benchmark* *Alexa*. Para cada *bytecode* i em I e coleção B em *Trace-Test*, *PeaceKeeper*, seja $b(i)$, seja i a frequência em B , e seja $f(i) = a(i) - b(i)$.

Quanto mais perto de zero os valores de $f(i)$, mais similar é a frequência de i na coleção *Alexa* e em outra coleção. Usando esse critério, pode ser visto que a coleção *PeaceKeeper* é mais próxima do *Alexa* que o *Trace-Test*. Somando-se os valores absolutos de $f(i)$, $1 \leq i \leq 100$ para cada coleção, tem-se 0.24 para a coleção *PeaceKeeper* e 0.75 para a coleção *Trace-Test*. Coincidentemente, o Algoritmo proposto é mais efetivo no *Trace-Test* que no *PeaceKeeper*.

3.6 Conclusão

Neste capítulo, foi apresentada a otimização proposta nesta dissertação que minimiza o número de testes de *overflows* em trilhas, e sua implementação baseada no compilador *TraceMonkey*. Esta otimização utiliza uma análise sensível ao fluxo para identificar os limites superior e inferior das variáveis inteiras da trilha, sendo esta análise realizada em duas etapas. Primeiro, é mostrado como é feita a construção do grafo de restrições para trilha. Em segundo lugar, é mostrado o funcionamento do algoritmo de propagação das restrições. Finalmente, foram apresentados os experimentos realizados para validar a otimização proposta.

Os experimentos mostram que a otimização atinge alto nível de efetividade na maioria dos *benchmarks* executados. Os experimentos também mostram que a implementação atual causa uma pequena sobrecarga no compilador. Os experimentos apontaram uma redução no tamanho do código de máquina produzido para alguns programas, principalmente na arquitetura *ST4*.

A instrumentalização do navegador *Firefox* com a otimização proposta permitiu a coleta de dados de páginas reais. Esses dados enriqueceram o entendimento da viabilidade e aplicação da otimização proposta. Os experimentos também revelaram que os *benchmarks* normalmente utilizados para testes de desempenho em programas *JavaScript* são bastante diferentes entre si, e são também diferentes das páginas reais da internet. Devido a essas diferenças, a utilização de apenas uma coleção para validação de uma otimização pode levar a conclusões distorcidas, já que cada coleção tem características distintas.

Capítulo 4

Conclusão

Esta dissertação apresentou uma otimização de código que utiliza uma análise sensível ao fluxo para identificar testes de *overflow* que são redundantes. A otimização é executada durante a compilação *just-in-time* de programas *JavaScript*.

O algoritmo proposto, uma variação mais agressiva do algoritmo de análise de largura de variáveis, trabalha no contexto de compilador de trilhas, confiando nos valores das variáveis que são apurados em tempo de execução. A otimização proposta é capaz de identificar faixas de restrições precisas para essas variáveis.

A otimização proposta foi implementada no compilador *TraceMonkey*, um compilador *just-in-time* usado pelo navegador *Mozilla Firefox* para melhorar o desempenho da execução dos programas em *JavaScript*.

A implementação da otimização foi submetida como um *patch* para a Fundação *Mozilla*¹.

Listamos a seguir as contribuições mais relevantes oriundas de nossa pesquisa no contexto dessa dissertação.

- A definição de um algoritmo para remoção de testes de *overflow* redundantes em trilhas.
- Implementação desse algoritmo de otimização no compilador *TraceMonkey*.
- Instrumentalização do navegador *Firefox* para obtenção de dados estatísticos sobre compilação de trilhas.
- Modificação do compilador *Nanojit* para desprezar instruções LIR de remoção de testes de *overflow* que são redundantes.

¹<https://github.com/rodrigol/Dynamic-Elimination-Overflow-Test/>

- Disponibilização de 3 *patches* para comunidade de desenvolvedores do navegador *Mozilla Firefox*.

Além da otimização aqui proposta, também foram disponibilizados dois *patches* que têm a função de auxiliar os desenvolvedores deste compilador. O primeiro *patch* gera uma representação gráfica da árvore de sintaxe abstrata de um dado programa. O segundo *patch* gera graficamente o grafo de fluxo de controle para os bytecodes do programa.

- Os seguintes artigos foram publicados em conferência e periódico:

Removing Overflow Tests via Run-Time Partial Evaluation

publicado nos anais da décima quarta edição do Simpósio Brasileiro de Linguagens de programação, pag.: 55-68, sendo agraciado com o prêmio de melhor artigo da conferência.

Dynamic Elimination of Overflow Tests in a Trace Compiler

publicado no *International Conference on Compiler Construction* 2011, pela Springer-Verlag pag.:2-21.

4.1 Trabalhos Futuros

Com o objetivo de dar continuidade ao trabalho aqui apresentado, nós sugerimos os seguintes trabalhos futuros:

Aumentar a efetividade da otimização

A implementação atual não reconhece trilhas que são iniciadas por alguns tipos de estruturas de repetição, como por exemplo, **while(true)** e **for each(...)**. A efetividade da otimização pode ser melhorada caso essas estruturas de controle sejam tratadas.

Outro aspecto da implementação atual que se melhorado, certamente traria significativos ganhos de efetividade, é a utilização de variáveis globais na análise. Por limitações do nosso entendimento do *TraceMonkey*, a implementação atual reconhece apenas as variáveis locais. Em uma observação empírica de alguns programas em *JavaScript* encontrados em páginas reais da Internet, verificou-se que os programadores costumam utilizar variáveis globais, mesmo em lugares onde elas não são necessárias, como por exemplo, na variável de indução da maioria das estruturas de repetições.

Acabar com a sobrecarga de desempenho causada pela otimização

A implementação atual não está otimizada para ser utilizada em escala industrial. Nenhum trabalho de otimização de desempenho foi realizado, fazendo com que a utilização da análise cause uma pequena sobrecarga no compilador. Conforme demonstrado por alguns experimentos, é provável que uma implementação com melhor desempenho possa inclusive reduzir o tempo global de execução de alguns programas.

Utilizar as informações em tempo de execução

As informações disponíveis em tempo de execução podem ser utilizadas para a criação de novas otimizações de código. Também é possível melhorar otimizações tradicionais como o desmembramento de laços com a utilização de informações verificadas em tempo de execução.

Implementação em outro compilador de trilhas

A otimização aqui proposta pode ser implementada em outro compilador de trilhas.

Bibliografia

- Aho, A. V. Lam, M. S. Sethi, R. Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.
- Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5:1--19.
- Authors, M. (2009). Programming language popularity. <http://langpop.com/>.
- Aycock, J. (2003). A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97-113.
- Bala, V. (2000). Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Notices*, 1--12.
- Bernstein, D. Rodeh, M. (1991). Global instruction scheduling for superscalar machines. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, 241--255, New York, NY, USA. ACM.
- Bird, S. Klein, E. Loper, E. (2009a). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, Beijing.
- Bird, S. Klein, E. Loper, E. (2009b). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, Beijing.
- Bodik, R. Gupta, R. Sarkar, V. (2000). ABCD: eliminating array bounds checks on demand. In *PLDI*, 321--333. ACM.
- Bozsahin, C. (1998). Deriving the predicate-argument structure for a free word order language.
- Briggs, P. Cooper, K. D. Torczon, L. (1994). Improvements to graph coloring register allocation. *TOPLAS*, 16(3):428--455.

- Budimlic, Z. Cooper, K. D. Harvey, T. J. Kennedy, K. Oberg, T. S. Reeves, S. W. (2002). Fast copy coalescing and live-range identification. In *PLDI*, 25–32. ACM.
- Burke, M. G. Choi, J.-D. Fink, S. Grove, D. Hind, M. Sarkar, V. Serrano, M. J. Sreedhar, V. C. Srinivasan, H. Whaley, J. (1999). The jalapeo dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande, JAVA '99*, 129–141, New York, NY, USA. ACM.
- Carette, J. Kucera, M. (2007). Partial evaluation of maple. In *PEPM*, 41–50. ACM.
- Chambers, C. Ungar, D. (1989). Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160.
- Chang, M. Smith, E. Reitmaier, R. Bebenita, M. Gal, A. Wimmer, C. Eich, B. Franz, M. (2009). Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE*, 71–80. ACM.
- Chevalier-Boisvert, M. Hendren, L. J. Verbrugge, C. (2010). Optimizing matlab through just-in-time specialization. In *CC*, 46–65. Springer.
- Cytron, R. Ferrante, J. Rosen, B. K. Wegman, M. N. Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490.
- Deutsch, L. P. Schiffman, A. M. (1984). Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 297–302, New York, NY, USA. ACM.
- Elphick, D. Leuschel, M. Cox, S. (2003). Partial evaluation of matlab. In *GPCE*, 344–363. Springer-Verlag New York, Inc.
- Flanagan, D. (2001). *JavaScript: The Definitive Guide*. O'Reilly, 4 .
- Gal, A. (2006). *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, Irvine.
- Gal, A. Eich, B. Shaver, M. Anderson, D. Kaplan, B. Hoare, G. Mandelin, D. Zbarsky, B. Orendorff, J. Ruderman, J. Smith, E. Reitmaier, R. Haghighat, M. R. Bebenita, M. Change, M. Franz, M. (2009). Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, 465 -- 478. ACM.

- Gal, A. Probst, C. W. Franz, M. (2006). Hotpathvm: an effective jit compiler for resource-constrained devices. In *VEE*, 144–153.
- Gosling, J. Joy, B. Steele, G. Bracha, G. (2005a). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.
- Gosling, J. Joy, B. Steele, G. L. Bracha, G. (2005b). *The Java Language Specification*. Addison-Wesley, Upper Saddle River, NJ, 3 .
- Harrison, W. H. (1977). Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, 3(3):243–250.
- Hough, D. (1981). Applications of the proposed IEEE-754 standard for floating point arithmetic. *Computer*, 14(3):70–74.
- Ierusalimschy, R. de Figueiredo, L. H. Celes, W. (2007). The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III*, 2–1--doformat2–26, New York, NY, USA. ACM.
- Jansen, P. (2009). Tiobe code. <http://www.tiobe.com/>.
- Jones, N. D. Gomard, C. K. Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1st .
- Kennedy, K. Allen, R. (2001). *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of ACM*, 3(4):184–195.
- Parr, T. (2009). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf.
- Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. In *PLDI*, 67–78. ACM.
- Plezbert, M. P. Cytron, R. K. (1997). Does “just in time” = “better late than never”? In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 120–131, New York, NY, USA. ACM.
- Richards, G. S., L. B., B. Vitek, J. (2010). An analysis of the dynamic behavior of javascript programs. In *PLDI*, 1–12.

- Rigo, A. (2004). Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM*, 15--26. ACM.
- Rossi, F. Beek, P. v. Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA.
- Sarkar, V. (2008). Code optimization of parallel programs: evolutionary vs. revolutionary approaches. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, 1--1, New York, NY, USA. ACM.
- Schultz, U. P. Lawall, J. L. Consel, C. (2003). Automatic program specialization for java. *TOPLAS*, 25(4):452--499.
- Shankar, A. Sastry, S. S. Bodík, R. Smith, J. E. (2005). Runtime specialization with optimistic heap analysis. *SIGPLAN Not.*, 40(10):327--343.
- Shuyu Guo, J. P. (2011). The essence of compiling with traces. In *POPL*, to appear. ACM.
- Stephenson, M. Babb, J. Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *PLDI*, 108--120. ACM.
- Sterling, L. Shapiro, E. (1986). *The Art of Prolog*. MIT Press, Cambridge (MA).
- Su, Z. Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122--138.
- Sumii, E. Kobayashi, N. (1999). Online-and-offline partial evaluation (extended abstract): a mixed approach. *SIGPLAN Not.*, 34(11):12--21.
- Tarjan, R. (1971). Depth-first search and linear graph algorithms. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:114--121.
- Thomas, D. Chad Fowler, A. H. (2005). *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, Raleigh, NC, 2. .
- Timo, L. (2010). Just-in-time compilation technique. *Report*, (1).
- Ungar, D. Smith, R. B. (1987). Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, 227--242, New York, NY, USA. ACM.
- Ungar, D. Smith, R. B. (2007). Self. In *HOPL*, 1--50.

- Webber, A. B. (2005). *Modern Programming Languages - A practical introduction*. Franklin Beedle and Associates, 1 .
- Williams, K. McCandless, J. Gregg, D. (2010). Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, 278--287, New York, NY, USA. ACM.

