PROJETO E IMPLEMENTAÇÃO DO COMPILADOR YAPc: UM COMPILADOR OTIMIZADOR PARA LINGUAGENS DE PROGRAMAÇÃO EM LÓGICA

Anderson Faustino da Silva

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Vítor Santos Costa, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Inês de Castro Dutra, Ph.D.

Prof. Roberto Ierusalimschy, Ph.D.

Prof. Rafael Dueire Lins, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

DEZEMBRO DE 2006

SILVA, ANDERSON FAUSTINO DA

Projeto e Implementação do Compilador YAPc: Um Compilador Otimizador Para Linguagens de Programação em Lógica

XVII, 185 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2006)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Compiladores

2 - Compiladores Otimizadores

3 - Compilação Dinâmica

4 - Prolog

I. COPPE/UFRJ II. Título (série)

Ao meu Deus, Jesus Cristo, O Criador dos céus e da terra.

Gostaria de agradecer primeiramente a minha esposa pelas horas em que precisei ficar distante dela. Muitas vezes, embora eu estivesse presente fisicamente, ela foi privada da minha presença. Agradeço-lhe pela compreensão e carinho dedicados. Quero também agradecer ao meu orientador, Vítor Santos Costa, pela dedicação e por ter sido um orientador exemplar. Vítor é uma pessoa brilhante.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

PROJETO E IMPLEMENTAÇÃO DO COMPILADOR YAPc: UM COMPILADOR OTIMIZADOR PARA LINGUAGENS DE PROGRAMAÇÃO EM LÓGICA

Anderson Faustino da Silva

Dezembro/2006

Orientador: Vítor Santos Costa

Programa: Engenharia de Sistemas e Computação

Prolog é uma linguagem declarativa largamente usada. Obter desempenho é um dos principais objetivos associados a qualquer implementação de Prolog. Atualmente, muitos sistemas Prolog são baseados na máquina abstrata de Warrem, a WAM. Para obter bom desempenho, trabalhos em implementação de Prolog propõem compiladores para código nativo. Estes sistemas podem ser caros, pois se baseiam em análises globais e/ou na intervenção de usuários para obter bom desempenho.

Esta tese propõe compilação dinâmica para Prolog, em um estilo de compiladores *Just-In-time*. Esta abordagem possui importantes vantagens. Primeiro, ela adapta-se às características do programa compilando apenas as partes do programa que são executadas freqüentemente. Segundo, ela não necessita de suporte da linguagem, apenas complexos padrões de execução, tais como gerência de exceções, podem ser deixadas para o interpretador. Por outro lado, o custo de compilação é agora parte do tempo total de execução.

Os resultados iniciais sugerem que YAPc alcança um desempenho substancial sobre o emulador original, e que ele é um sistema com ótimo desempenho comparado com o estado da arte em geração de código nativo.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

DESIGN AND IMPLEMENTATION OF THE YAPc COMPILER: AN OPTIMIZING COMPILER FOR LOGIC PROGRAMMING LANGUAGES

Anderson Faustino da Silva

December/2006

Advisor: Vítor Santos Costa

Department: Computing and Systems Engineering

Prolog is a widely used declarative language. Performance is an important issue in the design of Prolog systems. Currently, most Prolog systems rely on emuladors of Warren's Abstract Machine, the WAM. Towards better performance, previous research on high-performance Prolog implementations has proposed several native-code compilers. These systems can be expensive, as they rely on expensive global analysis and/or user annotations for best performance.

This thesis proposes a dynamic compilation for Prolog, in the style of Just-In-Time compilers. This approach has several important advantages. First, it adapts to the actual charateristics of the program by compiling only the parts of the program that are executed frequently.Second, it does need to fully support the language, only what is deemed important to performance. Complex execution patterns, such as exception handling, may be left to the interpreter. On the other hand, compilation cost is now part of the run-time.

The initial results suggest that the YAPc achieves very substantial performance improvements over the original emulator, and that it can approach and even out-perform state-of-the-art native code systems.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

A principal motivação para o uso de programação em lógica é permitir que os programadores descrevam *o que* eles querem separadamente de *como* alcançar este objetivo. Isto é baseado na premissa que qualquer algoritmo consiste de duas partes: uma especificação lógica, *a lógica*, e uma descrição de como executar esta especificação, *o controle*. Programas lógicos são declarações descrevendo propriedades de resultado esperado, com o controle para entender o sistema. A maior parte deste controle pode ser automaticamente provida pelo sistema, e que o mantém claramente separado da lógica.

Várias linguagens lógicas foram propostas. Destas a mais popular é Prolog, que foi originalmente criada para resolver problemas em linguagem natural. Prolog teve implementações comerciais com grande sucesso, o que gerou uma grande comunidade de usuários. A semântica de Prolog ataca um balanceamento entre eficiente implementação e completude lógica [78, 125]. Isto atenta para descrever a programação como um subconjunto de lógica de primeira ordem, que não é apenas um teorema de prova simples, mas também uma linguagem de programação usual devido a simplicidade e a implementação eficiente dos conceitos de unificação e busca.

Prolog tem sido aplicado em diversas áreas como *escrita de compiladores* [30], *prova de teoremas* [91], *base de dados dedutivas* [96] e *processamento de linguagem natural* [123, 95].

A primeira implementação de Prolog foi um interpretador desenvolvido por Roussel e Colmerauer na década de 70 [32]. Em 1977, David Warren criou o primeiro compilador de Prolog, o DEC-10 Prolog [124], que gerava código *assembly* para o DEC-10. A investigação na implementação de Prolog continuou com a *Warren Abstract Machine* (WAM) de David Warren, uma linguagem intermediária para a compilação do Prolog. A WAM oferecia várias vantagens, tais como fácil compilação, portabilidade e código compacto.

Por essas razões, a WAM revelou-se desde cedo como sendo uma forma eficiente e elegante de permitir a execução de programas Prolog numa máquina seqüencial, tornando-se rapidamente como modelo para implementações de Prolog.

A motivação original no projeto de máquinas abstratas era a construção de *hardware* para suportar Prolog eficientemente. Porém as instruções WAM realizam operações muito complexas, como a unificação, enquanto o desenvolvimento das novas arquiteturas seguiu uma direção oposta, possuir um pequeno conjunto de instruções simples. As dificuldades em obter bom desempenho em arquiteturas tradicionais e o custo de desenvolver *hardware* para suportar Prolog justificam que a WAM tenha sido principalmente usada em implementações por *software*. Estas implementações tradicionalmente compilam o código Prolog para o código de uma máquina abstrata que depois é interpretado. Como exemplo deste tipo de sistema temos o YAP [130], o SICStus Prolog [104, 86], o SWI-Prolog [128], o GNU-Prolog [44] e o XSB [100]. Contudo este tipo de sistema não aproveita ao máximo o desempenho da arquitetura utilizada. São duas as razões principais para tal fato: o *overhead* intrínseco em emular as instruções de outra arquitetura, e a complexidade das instruções.

Após o advento dos interpretadores para Prolog, os sistemas Aquarius [117] e Parma [114] provaram que em alguns casos, as linguagens lógicas podem ter desempenho comparável ao das linguagens imperativas. O bom desempenho destes sistemas deve-se aos seguintes fatores:

- ***Geração de código nativo:*** foram criadas novas máquinas abstratas constituídas por instruções simples, porém mais próximas das instruções de uma máquina seqüencial.

- ***Análise global do programa:*** permitindo especializar a unificação e explorar o determinismo em programas Prolog.

A geração de código nativo para Prolog é um problema complexo. Usualmente o problema é solucionado organizando a computação num conjunto de fases especializadas. Um exemplo é o SICStus Prolog que transforma o código Prolog em código WAM, para depois transformar em uma nova linguagem intermediária [53].

Uma alternativa a esta abordagem é a geração de código `C` a partir de Prolog [44, 82]. A filosofia desta abordagem é deixar o trabalho de geração de código, como também da otimização, para o compilador `C`.

Contudo estas abordagens diferentes possuem alguns problemas. Primeiro, a complexidade dos sistemas os torna difíceis de manter. Conseqüentemente, sistemas como Aquarius e Parma foram abandonados, não existindo mais uma atualização de tais sistemas. Segundo, o uso de análise global não consegue obter uma aceleração maior que três, além disto está técnica não funciona para qualquer tipo de progrma. Terceiro, o *overhead* dos interpretadores, como por exemplo YAP, é baixo, desta forma os sistemas que geram código nativo não conseguem um desempenho considerável. Além destas questões, outros sistemas, como por exemplo SICStus, também foram abondonados pelo fato de obterem desempenho apenas para um tipo de arquitetura.

Utilizando uma abordagem totalmente diferente, Mercury [54, 33] mudou a linguagem. O objetivo foi melhorar o desempenho de programas lógicos. Contudo, surgiu um problema: *Mercury não é Prolog*. Conseqüentemente, nem todos programas Prolog são executados por Mercury.

Nossa proposta é mostrar que o uso da compilação dinâmica torna o sistema Prolog fácil de ser mantido. Além de obter um desempenho significante sem alterar a linguagem.

O uso da compilação dinâmica é uma abordagem que tem se mostrado eficiente no contexto de linguagens orientadas a objetos, tais como SELF [23] e Java [106]. O uso da compilação dinâmica possui uma grande vantagem. Em tempo de execução, o ambiente pode inspecionar o programa com o objetivo de determinar o seu comportamento. E conseqüentemente, adaptar o sistema de compilação às características deste programa.

Da mesma maneira que ocorre em linguagens orientadas a objetos, o uso da compilação dinâmica pode reduzir o *overhead* imposto pelas características da linguagem Prolog. Porém, o longo tempo de compilação introduzido por tais ambientes pode retardar a resposta do ambiente de programação. Além disto, otimizar muitas vezes pode conflitar com depuração [93]. Desta forma, programadores têm escolhido entre abstração e eficiência.

Esta tese mostra como conciliar estes objetivos usando *compilação dinâmica* para realizar otimizações que se adaptem ao comportamento do programa e que além disto, sejam aplicadas de maneira tardia. Duas técnicas são utilizadas para este fim:

1. ***Informações de tipo*** permitem um bom desempenho por permitir em ao compilar, compilar apenas os caminhos executados baseado em informações extraídas do ambiente de execução.

2. ***Otimizações adaptativas*** atingem alta flexibilidade, sem sacrificar o desempenho através do uso de um interpretador para executar inicialmente o programa, en-

quando automaticamente compila as partes usadas freqüentemente com um compilador otimizador.

No compilador desenvolvido [107] é proposto precisamente a abordagem de *gerar código nativo para Prolog, usando compilação dinâmica*. O ponto inicial é desenvolver um compilador otimizador dinâmico, essencialmente um ambiente de execução que utiliza um modo misto de execução. Este sistema será uma evolução do sistema YAP.

O objetivo imediato é criar uma eficiente e usual implementação de Prolog. Porém, a abordagem utilizada não está comprometida com a semântica da linguagem Prolog e outras características expressivas. A proposta é preservar a ilusão do sistema diretamente executar o programa como o programador escreveu, sem a utilização de otimizações visíveis ao programador. Esta restrição tem diversas conseqüências:

- ***O programador deve ser livre para editar qualquer cláusula do sistema.***
- ***O programador deve ser capaz de entender a execução do programa e qualquer erro que ocorra no programa em termos do código fonte e construções da linguagem***. Este requesito na depuração e monitoramento do sistema desativa qualquer otimização interna que fornece a ilusão da implementação diretamente executar o programa como escrito. Programadores devem ser capazes de escolher como seus programas serão executados.
- ***O programador deve ser isolar do fato dos programas estarem sendo compilados***. Nenhum comando explícito para compilar uma cláusula ou programa deve ser fornecido. O programador apenas executa o programa. Esta ilusão de esconder o compilador será interrompida se o programador fosse distraído pelas pausas da compilação, análise ou otimização. Idealmente, qualquer pausa ocorrida pela implementação do sistema deve ser imperceptível.

Com estas restrições na semântica visível do sistema, o principal objetivo desta tese é implementar um eficiente ambiente de compilação, onde este seja adaptável às características do programa. Outro objetivo com esta restrição é obter uma rápida execução. O alvo deste objetivo é fazer Prolog mais próximo de lingugens tradicionais como `C` e Java.

Naturalmente, o objeto não é apenas implementar Prolog eficientemente, mas também uma larga classe de linguagens lógicas. Finalmente, as técnicas utilizadas não são específicas da linguagem Prolog. Acredita-se que elas podem ser aplicadas em outras linguagens, especialmente declarativas.

1.1 Contribuições

1.1.1 Desenvolvimento do Compilador Otimizador YAPc

Baseado na idéia de utilizar informações de tipo foi desenvolvido o compilador YAPc, um compilador *Just-In-Time* para Prolog. O principal objetivo do compilador é ter uma alta velocidade de compilação e gerar código compacto e otimizado.

YAPc evita as análises globais, porque estas acarretam um custo alto. Ao invés disto, YAPc utiliza algoritmos simples cuja complexidade é linear em relação à entrada do programa.

O compilador YAPc é conservativo, ele compila apenas aquelas partes do programa que são executadas freqüentemente. Além disto, ele apenas executa otimizações de maneira tardia, explorando as características do programa. Adicionalmente, muitos casos que poderiam ocorrer em princípio, mas que na prática ocorrem raramente, tais como: *overflow* de inteiros ou um tipo ilegal de argumento, nunca são compilados. Isto diminui o tempo de compilação e o tamanho do código gerado, além de permitir otimizar mais eficientemente as partes realmente executadas.

1.1.2 Desenvolvimento do Sistema YAP+

Desenvolvimento de um ambiente de compilação dinâmica para a geração de código nativo Prolog, usando como base o sistema YAP. O novo sistema YAP+ possui as seguintes características:

- ***Um modo misto de execução***. Inicialmente o programa é interpretado e automaticamente o ambiente detecta as partes do programa que deverão ser otimizadas. Assim, o ambiente gera código nativo apenas para as regiões executadas com grande freqüência.

- ***Uso de informações sobre o tipo dos dados***. Utilizando tais informações o ambiente de compilação apenas otimiza as porções de código (de uma cláusula) que realmente serão executadas.

- ***Uso de compilação dinâmica***. O sistema é capaz de descobrir oportunidades para a compilação sem a intervenção do programador. Em particular, ele deve decidir: (1) *quando compilar*, (2) *o que compilar* e (3) *quais caminhos compilar*.

- ***Otimizações adaptativas***. O ambiente de compilação é capaz de se adaptar às características de cada programa. Desta maneira, uma otimização somente é aplicada quando o ganho real for maior do que o tempo gasto em aplicá-la e o comportamento do programa favorecer a sua aplicação.

- ***Desotimização dinâmica***. O sistema é capaz de desotimizar uma parte do programa quando necessário.

1.1.3 Desenvolvimento de um Ambiente de Compilação Adaptável

YAP+ utiliza a compilação adaptável para adaptar o sistema de compilação às características do programa em execução. Durante a execução de um programa, o ambiente de execução o inspeciona com o objetivo de adaptar o sistema de compilação às suas características.

O uso da compilação adaptável determina o comportamento do compilador YAPc. O objetivo é obter um ambiente de compilação de alto desempenho, e conseqüentemente minimizar as pausas de compilação.

1.2 Organização da Tese

Esta tese contém mais sete capítulos. No próximo, capítulo 2, são apresentados os conceitos essenciais da linguagem Prolog, a máquina abstrada desenvolvida por David Warren, as técnicas utilizadas para melhorar o desempenho de programas Prolog e ambientes de execução de programas Prolog.

O capítulo 3 apresenta os princípios básicos sobre compilação *Just-In-Time*. Neste capítulo também são discutidos os compiladores das linguagens SELF e Java.

O capítulo 4 apresenta um estudo sobre o impacto dos compiladores Java *Just-In-Time* da Sun sobre a máquina virtual Java. Este descreve a máquina virtual Java, para em seguida apresentar uma análise do seu desempenho, ao utilizar um compilador *Just-In-Time*.

O capítulo 5 descreve o compilador otimizador YAPc desenvolvido nesta tese. Inicialmente é descrito o ambiente de execução, para em seguida ser descrita a compilação dinâmica e a arquitetura do compilador.

O capítulo 6 apresenta a análise detalhada do compilador desenvolvido. O objetivo é o mesmo do capítulo 4, contudo em um ambiente Prolog.

Finalmente, o último capítulo apresenta uma síntese da tese proposta e enumera alguns melhoramentos que serão realizados em trabalhos futuros.

Capítulo 2

Prolog

Prolog (PROgramming in LOGic) [111, 21] é uma linguagem de programação em lógica baseada em um subconjunto de lógica de primeira ordem, as cláusulas de Horn. Cláusulas de Horn têm um algoritmo de busca eficiente que permite uma performance competitiva. Algumas características não lógicas foram adicionadas à Prolog, tais como: *controle de fluxo*, *entrada/saída* e *meta-programação*. Pode-se dizer que um programa Prolog é uma base de dados composta de *regras* e *fatos*. Como tal, a linguagem é naturalmente declarativa: o desenvolvedor informa ao sistema como especificar o problema e o sistema, por sua vez, usa um algoritmo de busca para responder questões.

A principal utilização da linguagem Prolog reside no domínio da programação simbólica, sendo especialmente adequada à solução de problemas envolvendo relações entre objetos, por exemplo *processamento de linguagem natural* [95] e *sistemas dedutivos* [96]. Algumas das principais características da linguagem Prolog são:

- *É uma linguagem orientada ao processamento de símbolos, os chamado termos*.
- *Implementação de uma lógica como linguagem de programação*.
- *Apresenta uma semântica declarativa inerente a uma lógica*.
- *Permite a definição de programas reversíveis, isto é, o mesmo pode ser usado com diferentes entradas e saídas para o mesmo programa*.
- *Permite a obtenção de respostas alternativas*.
- *Suporta naturalmente código recursivo e iterativo para a descrição de processos e problemas, dispensando mecanismos tradicionais de controle, tais como comandos de repetição*.

- *Permite associar o processo de especificação ao processo de codificação de programas.*

- *Representa programas e dados através do mesmo formalismo.*

Diversas técnicas para implementar Prolog têm sido propostas desde que o primeiro interpretador foi desenvolvido em Marseille [31]. Existem duas principais abordagens para implementar Prolog eficientemente:

1. *compilar para bytecodes e interpretá-los* [104], ou

2. *compilar para código nativo.*

A segunda abordagem pode ser dividida em duas categorias. Uma solução é o compilador gerar código de máquina [44]. Outra alternativa é gerar código para outra linguagem, como `C`, e em seguinda gerar código nativo usando o compilador desta linguagem [82].

Cada solução tem suas vantagens e desvantagens. Gerar código de baixo nível provê programas rápidos. Interpretadores possuem tempo de compilação pequeno e são uma boa solução pela sua simplicidade quando a velocidade não é uma prioridade. Compiladores são mais complexos e difíceis de manter do que interpretadores, e a diferença é ainda mais acentuada se análise global é realizada como parte da compilação.

Este capítulo apresenta os conceitos essenciais da linguagem Prolog, a máquina abstrada desenvolvida por David Warren [125], técnicas utilizadas para melhorar o desempenho e, finaliza descrevendo alguns ambiente de execução de programas Prolog.

2.1 A Linguagem Prolog

Um programa escrito em Prolog é a especificação de um determinado problema através da utilização de sentenças em cláusula de Horn. Em `Pascal` ou em `C`, um programa é uma seqüência de instruções a serem executadas uma após a outra. Já um programa em lógica se assemelha mais a um banco de dados, onde há várias informações e relações registradas, por exemplo do tipo: *'o homem é inteligente'* ou *'X é inteligente se X é homem'*. Uma segunda diferença no paradigma de programação em lógica é o conceito de *consulta* e de *resposta* de um programa. Nas linguagens convencionais, um programa começa sempre a partir do mesmo ponto de entrada. Na programação em lógica, a chamada é feita através

de uma *pergunta*, ou consulta. E o resultado do programa são as condições necessárias para que a afirmação seja verdadeira.

Basicamente, um programa Prolog é formado por um conjunto de *cláusulas*, que podem ser constituídas por fatos ou regras. A figura 2.1 mostra o programa *append* que realiza a concatenaçã de duas listas. Este programa contém um fato e uma regra. Um fato consiste na declaração de uma cláusula que possui corpo vazio. Uma regra em Prolog consiste em duas partes:

1. o *cabeça*, que é representado pelo predicado que se pretende provar, juntamente com seus argumentos. Na sintaxe de Edimburgo [122], o cabeça é separado do corpo pelo símbolo ":-".

2. o *corpo*, que consiste num conjunto de objetivos que devem ser satisfeitos a fim de que o predicado presente no cabeçalho seja verdadeiro.

```
append([ ], L, L).
append([X|L1], L2, [X|L3]) :-
                append(L1, L2, L3).
```

Figura 2.1: Código Prolog do programa *append*.

Uma consulta em Prolog é sempre uma seqüência de um ou mais objetivos. Para obter a resposta, o sistema Prolog tenta *satisfazer* todos os objetivos que compõem a consulta, interpretando-os portanto como uma conjunção. Satisfazer um objetivo significa demonstrar que esse objetivo é verdadeiro, assumindo que as relações que o implicam são verdadeiras no contexto do programa. Se a consulta contém variáveis, o sistema Prolog deverá encontrar valores que atribuídos às variáveis, satisfazem a todos os sub-objetivos propostos na consulta. A particular instanciação das variáveis com os valores que tornam o objetivo verdadeiro é a *resposta*. Se não for possível encontrar, no contexto do programa, nenhuma instanciação comum de suas variáveis que permita derivar todos os sub-objetivos propostos o sistema informa que não há resposta. O sistema Prolog aceita os fatos e regras como um conjunto de axiomas e a consulta do usuário como um teorema a ser provado. A tarefa do sistema é demonstrar que o teorema pode ser provado com base nos axiomas representados pelo conjunto das cláusulas que constituem o programa.

2.1.1 Dados

Os dados em Prolog e suas manipulações são modelados como lógica de primeira ordem. As variáveis lógicas representam tipos dinâmicos e são instanciadas por instanciação. Para identificar o objetivo mais comum entre dois objetos Prolog utiliza unificação.

2.1.1.1 Variáveis Lógicas

Prolog é uma linguagem tipada dinamicamente: variáveis podem conter objetos de qualquer tipo em tempo de execução. Inicialmente o valor de uma variável é *desconhecido*: variáveis assumem valores através de *instanciação*. Variáveis podem ser passadas como argumentos para predicados ou como argumentos de dados compostos. Variáveis podem ser instanciadas apenas uma vez, contudo elas podem referenciar outras variáveis. Quando uma variável é instanciada para um valor, este valor é visto por todas as variáveis que a referenciam.

2.1.1.2 Tipos Dinâmicos

Tipos de dados compostos são novos tipos que podem ser criados em tempo de execução e variáveis podem conter valores de diferentes tipos. Tipos comuns são:

- *átomos*: constantes únicas, por exemplo: `1`, `xyz`;
- *inteiros*;
- *listas*: denotadas por colchetes, por exemplo `[Cabeça | Cauda]` ou `[1,2,3,4]`;
- *estruturas*: por exemplo: `f(X,L)` ou `f(X,g(K))`.

Estruturas são similares às estruturas em `C` ou registros de `Pascal`, elas possuem um nome (em Prolog chamado de *functor*) e uma quantidade fixa de parâmetros (em Prolog chamado de *aridade*).

2.1.1.3 Unificação

Unificação [73] é uma operação de casamento de padrões que encontra a instância comum mais geral de dois objetos de dados. A operação de unificação é capaz de combinar componentes de objetos de qualquer tamanho em uma simples operação primitiva. Ligação

de variáveis é feita através de unificação. Como uma parte do casamento de padrão, as variáveis no termo são instanciadas para torná-las iguais. Por exemplo, unificar `g(A,B,1)` e `g(K,2,W)` combina `A` com `K`, `B` com `2` e `1` com `W`.

2.1.2 Controle

Durante a execução, Prolog tenta satisfazer as cláusulas na ordem em que elas estão listadas. Durante este processo, Prolog utiliza uma execução para frente. Ele seleciona um literal, unifica e continua até não existirem mais literais. Quando o sistema invoca um predicado com mais de uma cláusula, Prolog tenta satisfazer a primeira e constrói *pontos de escolha* para as outras. Se o sistema não pode tornar a cláusula verdadeira ocorre um *retrocesso*. O sistema retorna para o *ponto de escolha* mais recente e tenta a próxima cláusula. Neste processo, todas as ligações realizadas durante a tentativa de tornar a cláusula verdadeira são desfeitas, pois executar a próxima cláusula pode fornecer às variáveis diferentes valores. Note que em um dado caminho de execução a variável pode conter apenas um valor, mas em diferentes caminhos a variável pode conter valores diferentes. Prolog é uma linguagem com uma única atribuição: se a unificação tentar fornecer à variável um valor diferente causará um falha e ocorrerá um *retrocesso*. Por exemplo: tentar unificar `g(1,2)` e `g(A,A)` causará uma falha porque os números `1` e `2` são diferentes.

Prolog provê ainda mecanismos auxiliares para gerenciar o fluxo de controle, a saber:

- *operação de corte*,
- *disjunção*,
- *condicional*, e
- *negação*.

2.1.2.1 Operação de Corte

A operação de corte é usada para evitar o *retrocesso*. Um corte no corpo de uma cláusula indica que esta foi a escolha correta, e que portanto o sistema não deve testar nenhuma outra cláusula para o mesmo predicado quando ocorrer retrocesso. Executar uma operação de corte tem o mesmo efeito em uma execução para frente de executar um `true`, ou seja, isto não possui nenhum efeito. Mas altera o comportamento do *retrocesso*. Por exemplo:

```
g(A) :- k(A), !, f(A).
g(A) - p(A).
```

Durante a execução de `g(A)`, se `k(A)` for verdadeiro então a operação de corte é executada. A operação de corte remove o *ponto de escolha* criado em `k(A)` e o *ponto de escolha* que o sistema criou quando invocou `g(A)`. Como resultado, se `f(A)` falha então o predicado `g(A)` falha. Se não existisse a operação de corte e `f(A)` falhasse ocorreria *retrocesso* primeiro para `k(A)`, e se este também falhase ocorreria *retrocesso* para a segunda clásula do predicado `g`. Neste caso, apenas quando `p(A)` falha o predicado `g(A)` falha.

2.1.2.2 Disjunção

A disjunção é uma maneira concisa de denotar uma escolha entre diferentes alternativas. Isto é mais conciso do que definir um predicado que tem cada alternativa como uma cláusula separada. Por exemplo:

```
g(A) :- (A = 1 ; A = 2; A = 3).
```

Este predicado retorna três diferentes soluções e é equivalente a:

```
g(1).
g(2).
g(3).
```

A disjunção é freqüentemente usada em conjunto com a operação de corte.

2.1.2.3 Condicional

Uma construção condicional, o *if-then-else*, é usada para denotar uma seleção entre duas alternativas em uma cláusula, quando é conhecida que se uma alternativa é tomada a outra não será necessária. Por exemplo: o predicado `g(A)` pode ser escrito com uma construção *if-then-else*:

```
g(A) :- (k(A) -> f(A) ; p(A)).
```

Esta sintaxe tem semântica idêntica como a definição anterior. A flecha `->` em um *if-then-else* atua como uma operação de corte que remove *pontos de escolha* até o ponto onde a operação *if-then-else* iniciou.

2.1.2.4 Negação

Prolog implementa negação como falha, denotado por `\+(objetivo)`. Isto não é realmente negação no sentido lógico, por isso o símbolo `\+` foi escolhido ao invés de `not`. Um objetivo negativo é verdadeiro se o objetivo falha, e falha se o objetivo é verdadeiro. Por exemplo:

```
d(K) :- \+ l(K).
```

O predicado `d(K)` será verdadeiro se `t(K)` falhar. Isto possui semântica idêntica a:

```
d(K) :- l(K), !, fail.
d(K).
```

Se `l(K)` é verdadeiro então `fail` causa uma falha, e o corte assegura que a segunda cláusula não será testada. Se `l(K)` falha então a segunda cláusula é testada porque a operação de corte não é executada. Negação como falha nunca faz uma ligação de qualquer variável do objetivo que é negada. Isto é diferente da negação em lógica pura, que deve retornar todos resultados que não são iguais para aquele que satisfaz o objetivo. Negação como falha fornece resultados lógicos corretos se o objetivo negado não possui variáveis não instanciadas.

2.2 A Máquina Abstrata de Warren

Na década de 80, David Warren criou a *Warren Abstract Machine* (WAM) [125, 6], um modelo para a execução de Prolog que rapidamente tornou-se padrão para as implementações de Prolog.

As principais formas de implementação de um ambiente de execução Prolog usando a WAM são três:

1. *implementações por hardware* [119];

2. *implementações usando um interpretador* [20, 85]; e

3. *implementações que traduzem o código WAM para código nativo* [118, 82, 42].

A motivação original do projeto de máquinas abstratas era a construção de *hardware* para suportar Prolog de maneira eficiente. As implementações por *hardware* não tiveram muito sucesso devido ao mercado limitado: poucas pessoas estão interessadas em comprar uma máquina capaz de executar apenas Prolog. E também, mesmo que Prolog fosse popular, o *hardware* especializado teria muita dificuldade em conseguir competir com os microprocessadores que se beneficiam de investimentos maiores [51].

O custo de desenvolver *hardware* para suportar Prolog justifica que a WAM tenha sido principalmente usada em implementações por *software*. Estas implementações tradicionalmente compilam o código Prolog para o código de uma máquina abstrata que é depois interpretado em tempo de execução. As implementações baseadas no emulador da WAM tornaram-se muito populares, pois são relativamente fáceis de implementar e têm bom desempenho. Contudo este tipos de sistemas não aproveitam ao máximo o desempenho da máquina. São duas as razões principais para tal fato:

1. *Existe um overhead intrínseco em interpretar as instruções de uma outra arquitetura* [99].

2. *A granularidade das instruções WAM não permite que muitas otimizações possam ser aplicadas*.

Implementações que geram código nativo permitem um ganho significativo de desempenho na execução de programas Prolog. No entanto, a transformação de código Prolog para código nativo é um problema complexo resultando em sistemas de difícil manutenção. Uma forma de simplificar o compilador de código nativo é gerar código `C`. O exemplo deste tipo de implementação é o sistema *wamcc* [29]. Infelizmente, o desempenho deste sistema não é muito impressionante, mesmo comparado com sistemas que usam interpretadores. Um trabalho mais recente na geração de Prolog para código `C`, com melhores resultados,foi desenvolvido por Morales, Carmo e Hermenegildo [82] e será discutido na seção 2.4.6.

2.2.1 Organização da Memória

A memória na WAM é dividida em cinco áreas, que são apresentadas na figura 2.2: *duas pilhas para os objetos de dados, uma pilha para unificação, uma para a interação entre*

unificação e retrocesso, e uma área para armazenar o código do programa.

- *Pilha global ou heap.* Esta pilha mantém listas e estruturas, os termos compostos de Prolog. Além disto, esta pilha pode manter também variáveis.
- *Pilha local.* Esta pilha mantém ambientes e pontos de escolha. Ambientes contêm endereços de retorno e variáveis criadas pela cláusula. Pontos de escolha encapsulam o estado de execução e permitem retrocesso.
- *Trail.* Esta pilha é usada durante o retrocesso. Ela contém as variáveis que precisam ser reinicializadas durante o retrocesso.
- *Push Down List.* Esta pilha é usada durante a unificação de termos compostos.
- *Área de código.* Esta área mantém o código compilado de um programa, ou seja, a base de dados.

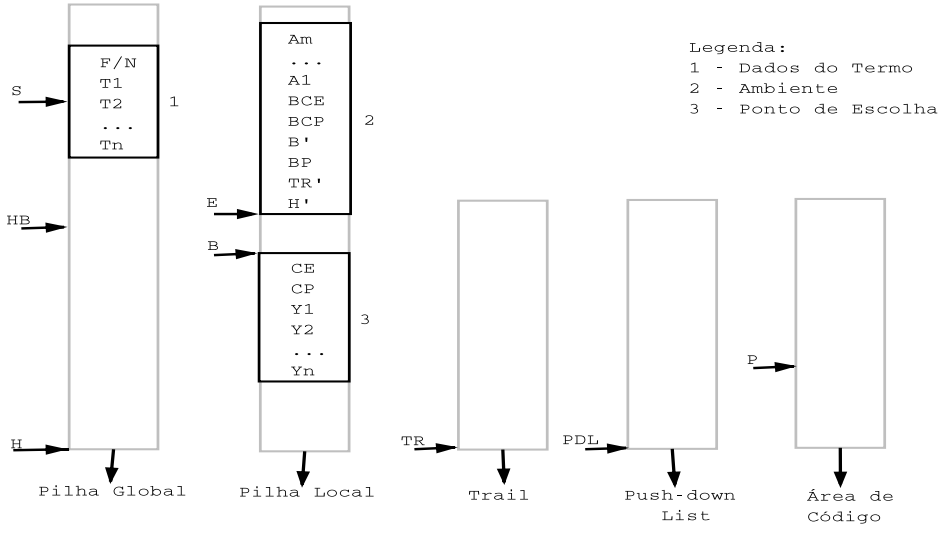


Figura 2.2: Organização da Memória na WAM.

2.2.2 O Estado de Execução

O estado corrente da execução de Prolog é definido pelos registradores da WAM. A tabela 2.1 mostra a descrição de cada registrador da WAM.

O registrador `HB` mantém o valor de `H` (endereço da *heap* onde o próximo objeto será construído) armazenado no ponto de escolha mais recente. O registrador `S` é usado durante a unificação de termos. Este registrador aponta para o argumento do termo que está

Registrador	Descrição
P	Contador de Programa
CP	Endereço de Retorno
E	Ambiente Corrente
B	Ponto mais recente de retrocesso
A	Topo da pilha local
TR	Topo da *Trail*
H	Topo da *heap*
HB	Ponto de retrocesso na *heap*
S	Ponteiro para a estrutura na *heap*
Mode	Registrador de modo
A1, A2, ...	Registradores para argumentos
X1, X2, ...	Variáveis temporárias

Tabela 2.1: Registradores da WAM.

sendo unificado: na WAM os argumentos são acessados através de sucessivos incrementos de `S`.

Na WAM existem dois modos de execução: modo *leitura* e modo *escrita*. O modo de leitura ocorre quando o objeto unificado com a cabeça da cláusula é um termo já construído. O modo de escrita ocorre quando a cabeça da cláusula é unificada com um variável livre, e o termo precisa ser construído.

Os registradores `A` e os registradores `X` são idênticos, os nomes diferentes apenas indicam o seu uso. Os registradores `A` são usados para passar argumentos. Os registradores `X` são usados para manter as variáveis temporárias.

Uma *variável temporária* é uma variável que tem sua primeira ocorrência na cabeça ou na estrutura ou no último objetivo, e que não ocorre em mais do que um objetivo no corpo, onde a cabeça da cláusula é contada como parte do primeiro objetivo. Variáveis temporárias não precisam ser armazenadas no ambiente da cláusula.

Uma *variável permanente* é aquela usada em mais de uma cláusula do objetivo. As variáveis permanentes são armazenadas no ambiente e são indexadas por deslocamentos a partir do ponteiro do ambiente. Elas são referenciadas como `Y1`, `Y2`, etc.

2.2.3 O Conjunto de Instruções

Programas Prolog são codificados como uma seqüência de instruções da WAM. Em geral, existe uma instrução para cada símbolo Prolog. As instruções da WAM consistem de um código de operação com alguns operandos. O código de operação geralmente codifica o tipo do símbolo Prolog junto com o contexto em que o símbolo ocorre. Os operan-

dos incluem *inteiros*, *deslocamentos* e *endereços* que identificam os diferentes tipos de símbolos Prolog.

O conjunto de intruções da WAM são classificadas em:

- *instruções get*: correspondem aos argumentos da cabeça da cláusula e são responsáveis por combiná-los com os argumentos passados nos registradores `A`. A tabela 2.2 descreve as instruções *get*.

- *instruções put*: correspondem aos argumentos de um objetivo no corpo da cláusula e são responsáveis por carregar os argumentos nos registradores `A`. A tabela 2.3 descreve as instruções *put*.

- *instruções unify*: correspondem aos argumentos de uma estrutura (ou lista) e são responsáveis por unificar com estruturas existentes e por construir novas estruturas. A tabela 2.4 descreve as instruções *unify*.

- *instruções de controle de fluxo*: correspondem aos predicados que formam a cabeça e o corpo da cláusula e são responsáveis por controlar o fluxo da execução e alocar ambientes associados com a chamada de predicados. A tabela 2.5 descreve as instruções de controle da execução.

- *instruções de indexação*: são responsáveis por determinar qual cláusula do predicado deverá ser executada. A tabela 2.6 descreve as instruções de indexação.

- *instruções para gerenciar retrocesso*: unem diferentes cláusulas que formam um predicado e são responsáveis por gerenciar os pontos de escolha. A tabela 2.7 descreve as instruções para gerenciar retrocessos.

As instruções *get* realizam a unificação com os argumentos da cabeça da cláusula. As instruções *get_structure* e *get_list* alteram o valor do registrador de modo que determinam se a execução continua no modo de leitura ou escrita. Por exemplo, se o registrador $A1$ em uma instrução *get_list* possui uma variável não instanciada, o registrador de modo assume o modo de escrita. Por outro lado, se este registrador contém uma referência a uma lista, *get_list* atribui a este o modo de leitura. No caso do argumento ser de outro tipo, a instrução falha e ocorre um retrocesso que restaura o estado da máquina para o ponto de escolha mais recente.

Unificação com Registradores	
get_variable Vn, Ai	Move An para Vi.
get_value Vn, Ai	Unifica Vn com Ai.
get_constant C, Ai	Unifica a constante *C* com Ai.
get_nil Ai	Unifica a constante *nil* com Ai.
get_list Ai	Unifica um ponteiro para uma lista com Ai.
get_structure F/N, Ai	Unifica o funtor *F/N* com Ai.

Tabela 2.2: Instruções *get*.

A WAM usa a notação *variável/valor*. Instruções anotadas com *"variável"* assumem que é a primeira ocorrência do argumento na cláusula. Neste caso, a operação de unificação é simplificada. Por exemplo, a instrução:

```
get_variable X2, A1
```

unifica *X*2 com *A*1. Como *X*2 não foi inicializada, a unificação se reduz a $X2 = A1$. Instruções anotadas com *"valor"* assumem que seus argumentos já foram inicializados. Neste caso, uma unificação completa é realizada, por exemplo *a(X,X)* é compilado para:

```
get_value A1,A2.
```

As instruções *put* carregam os argumentos do predicado. A instrução *put_unsafe_value* é usada no lugar da instrução *put_value* no último objetivo no qual uma variável *não segura* aparece. Uma variável *não segura* é uma variável permanente que não ocorreu primeiro na cabeça ou na estrutura, isto é, uma variável que foi inicializada pela instrução *put_variable*. A instrução *put_unsafe_value* assegura que a variável *não segura* não está armazenada no ambiente corrente, ligando a variável com um novo valor na *heap*. Isto previne possíveis referências erradas para a parte do ambiente que será descartado pelas instruções *execute* ou *call*.

As instruções *unify* possuem diferentes comportamentos dependendo do modo de execução. No modo de leitura elas realizam uma simples unificação, enquanto no modo de escrita elas criam um novo termo na *heap* e realizam uma unificação com este novo termo.

A instrução *unify_local_value* é usada no lugar da instrução *unify_value* se a variável não foi inicializada com um valor global, por exemplo através da instrução *unify_variable*. A instrução *unify_void* representa uma seqüência de *N* ocorrências de variáveis.

Carga dos Argumentos	
put_variable Vn, Ai	Cria uma nova variável, armazena em *Vn* e *Ai*.
put_value Vn, Ai	Copia *Vn* para *Ai*.
put_unsafe_value Vn, Ri	Move *Vn* para *Ri* e coloca na *heap*.
put_constant C, Ai	Copia a contante *C* para *Ai*.
put_nil Ai	Copia a contante *nil* para *Ai*.
put_structure F/N, Ai	Cria o funtor *F/N* e o armazena em *Ai*.
put_list Ai	Cria um ponteiro para uma lista e o armazena em *Ai*.

Tabela 2.3: Instruções *put*.

Uma seqüência de instruções *unify* sé precedida por uma instrução *get* ou *put* concernente a uma estrutura ou a uma lista. Estas instruções precedentes determinam qual o modo de execução da intrução *unify*. Em modo de leitura, a instrução *unify* realiza unificação com sucessivos argumentos de uma estrutura existente, endereçada pelo registrador `S`. Em modo de escrita, a instrução *unify* constrói os sucessivos argumentos da nova estrutura, endereçados através do registrador `H`.

Unificação com Argumentos que são Estruturas	
unify_variable Vn	Move o próximo argumento para *Vi*.
unify_value Vn	Unifica *Vi* com o próximo argumento.
unify_local_value Vn	Unifica *Vi* com o próximo argumento e globaliza.
unify_constant C	Unifica a constante *C* com o próximo argumento.
unify_nil	Unifica a constante *nil* com o próximo argumento.
unify_void N	Salta N argumentos.

Tabela 2.4: Instruções *unify*.

Nas instruções de controle da execução da tabela 2.5, `P` representa o predicado e `N` é o número de variáveis no ambiente. Estas instruções são usadas na tradução de cláusulas que contenham qualquer quantidade de objetivos. O tamanho do ambiente é especificado dinamicamente pela instrução *call*.

Controle da Execução	
call P, N	Chama o predicado *P/N*.
execute P	Salta para o predicado *P*.
proceed	Retorna.
allocate	Cria um novo ambiente.
deallocate	Remove o ambiente corrente.

Tabela 2.5: Instruções de controle de execução.

As instruções *switch* saltam para a cláusula correta ou conjunto de cláusulas depen-

dendo do tipo do primeiro argumento. Esta técnica é chamada de *indexação*. As instruções que gerenciam pontos de escolha (retrocessos) unem um conjunto de cláusulas.

Cada cláusula é precedida por uma instrução *try_me_else*, *retry_me_else* ou *trust_me_else*, dependendo se é a primeira, a intermediária, ou a última cláusula do predicado.

A instrução *switch_on_term* salta para um dos endereços contidos nos argumentos (*V, C, L, S*), dependendo se o primeiro argumento é uma variável, uma constante, uma lista ou uma estrutura. *V* é o endereço de uma instrução *try_me_else* (ou *trust_me_else*), que precede a primeira cláusula do predicado. *L* pode ser o endereço de uma simples cláusula cuja chave é uma lista, ou o endereço de uma seqüência de cláusulas identificadas pela seqüência de instruções *try*, *retry* e *trust*. *C* e *S* podem ser os endereços de uma simples cláusula ou uma seqüência de cláusulas (como *L*), ou mais geralmente podem ser, respectivamente, o endereço de uma instrução *switch_on_constant* ou *switch_on_structure*, que fornece acesso a tabela *hash* da cláusula ou cláusulas que combinam com a chave.

Seleção de Cláusulas	
switch_on_term V, C, L, S	Salto baseado no tipo de A1.
switch_on_constant N, T	Salto baseado em *hash* da constante armazenada em A1.
switch_on_structure N, T	Salto baseado em *hash* da estrutura armazenada em A1.

Tabela 2.6: Instruções *switch*.

Retrocesso	
try_me_else L	Cria um ponto de escolha para *L* e falha.
retry_me_else L	Altera o endereço da nova tentativa para *L* e falha.
trust_me_else fail	Remove o ponto de escolha mais recente e falha.
try L	Cria um ponto de escolha e salta para *L*.
retry L	Altera o endereço da nova tentativa e salta para *L*.
trust L	Remove o ponto de escolha mais recente e salta para *L*.

Tabela 2.7: Instruções para gerenciar retrocesso.

A WAM não aborda sobre a instrução de *corte*, que remove todos os pontos de escolha criado após entrar no predicado corrente. A implementação desta instrução foi abordada apenas em trabalho posterior [6].

A figura 2.3 mostra a compilação do programa *append* em código WAM.

```
append/3:

        switch_on_term V1, C1, C2, fail     : Salta para V1 se A1 é uma variável.
                                            : Salta para C1 se A1 é uma constante.
                                            : Salta para C2 se A1 é uma lista.
                                            : Falha se A1 é uma estrutura.
V1:   try_me_else V2                        : Cria um ponto de escolha.
C1:   get_nil A1                            : Unifica A1 com nil.
        get_value A2, A3                    : Unifica A2 com A3.
        proceed                             : Retorna.

V2:   trust_me_else fail                    : Remove o ponto de escolha.
C2:   get_list A1                           : Inicia unificação de A1 com a lista.
        unify_variable A4                   : Carrega cabeça da lista em A4.
        unify_variable A1                   : Carrega calda da lista em A1.
        get_list A3                         : Inicia unificação de A3 com a lista.
        unify_value A4                      : Unifica cabeça da lista com A4.
        unify_variable A3                   : Carrega calda da lista em A3.
        execute append/3                    : Salta para append/3.
```

Figura 2.3: Código WAM do programa *append*.

2.3 Melhorando o Desempenho de Programas Prolog

Implementações de Prolog tiveram um grande progresso no seu desempenho com o desenvolvimento da WAM. Contudo, a velocidade de execução pode ser melhorada ainda mais. Diversas técnicas foram propostas:

1. *reduzir a granularidade das instruções*;
2. *explorar determinismo*;
3. *especializar a unificação*; e
4. *utilizar a análise de fluxo de dados*.

2.3.1 Reduzir a Granularidade das Instruções

A WAM mapeia de forma elegante a linguagem Prolog para uma máquina seqüencial. Algumas instruções da WAM podem ser bastante complexas, como por exemplo *get_structure* e *unify_value*. Devido a isto muitas otimizações não são possíveis. Um exemplo simples é o predicado:

```
x(1).
```

Este predicado é compilado para:

```
get_constant 1, A1
proceed
```

A instrução *get_constant* realiza duas operações. Primeiro, o argumento `A1` é desreferenciado, segundo `A1` é unificado com o valor 1. Algumas operações podem ser desnecessárias. Por exemplo, se o predicado `x(K)` é chamado com um átomo desreferenciado, então a unificação deveria ser reduzida para um simples teste que verifica se o valor é o correto.

Granularidade grossa faz sentido num emulador, contudo para fins de otimização é importante reduzir a granularidade de cada instrução WAM.

O compilador otimizador desenvolvido por Tamura [68, 113] reduz a granularidade das instruções através da representação de cada instrução WAM como um subgrafo contendo simples operações como instruções de seleção sobre *tags* de tipos, saltos, atribuições e desreferenciação. Em seguida, este grafo pode ser otimizado através de regras de reescrita.

2.3.2 Explorar o Determinismo

A maioria dos predicados escritos por programadores são intencionalmente escritos a fim de fornecerem apenas uma solução, isto é, eles são determinísticos. Infelizmente, usar índices para escolher a correta cláusula é ineficiente. Pois, para possíveis retrocessos existe a necessidade de salvar e restaurar o estado da máquina.

Significantes melhorias sobre a WAM são obtidas evitando retrocesso sobre predicados determinísticos. Turk [115] foi o primeiro a descrever otimizações que reduzem o tempo necessário para restaurar o estado da máquina durante um retrocesso.

SICStus Prolog [104, 86] reduz o *overhead* de um retrocesso criando pontos de escolha em duas fases: primeiro, ele salva apenas uma pequena parte do estado da máquina, adiando salvar o restante do estado até o ponto na cláusula onde ela pode ser determinada através da unificação da cabeça e um simples teste.

Outros sistemas generalizam a indexação sobre o primeiro argumento. BIM_Prolog

[76] podia indexar qualquer argumento. SEPIA [79] incorporava heurísticas para decidir quais argumentos são importantes para uma seleção determinística. Este usava o primeiro argumento indexável de um predicado. Se existem várias possibilidades, o sistema utiliza o argumento que selecionará a menor quantidade de cláusulas.

Como será visto no capítulo 5, o sistema proposto utiliza *type-feedback profiling* [59]. A principal idéia deste tipo de *profiling* é extrair em tempo de execução o tipo mais comum da informação, e fornecé-la como parâmetro ao compilador. Desta maneira o compilador do sistema gera código nativo especializado para as cláusulas executadas.

2.3.3 Especializar Unificação

Significantes melhorias sobre a WAM são possíveis para a unificação. Turk [115] descreve otimizações para compilar unificação, reduzir o *overhead* de explicitamente manter um *bit* de modo e remover alguns desreferenciamentos supérfluos e checagem de *tag*. Marien [75] descreve um método de compilar unificação no modo de escrita que usa um número mínimo de operações e evita instruções supérfluas de desreferenciação e checagem de *tag*. Van Roy [116] introduz uma simples notação e a estende para uma unificação do modo de leitura, mas este esquema gera código excessivo. O sistema Aquarius [117] modifica esta técnica para limitar a expansão do código. Meier [78] desenvolveu uma técnica que generaliza a idéia de Marien para os modos de escrita e leitura e consegue tamanho de código linear.

Beer [14] sugere o uso de uma representação simplificada de variáveis onde ligações são muito rápidas. Este trabalho introduz alguns novos *tags* para esta representação, eles são chamados "variáveis não inicializadas", e mantém referências delas em tempo de execução. Ele mostra que o tempo de desreferenciação é reduzido significantemente.

Para reduzir o *overhead* da unificação é proposto utilizar duas versões para unificação, que poderá ser ainda otimizada pelo compilador otimizador desenvolvido.

2.3.4 Análise de Fluxo de Dados

Tradicionalmente, análise global de programas lógicos é usada para derivar informações que serão utilizadas para melhorar a execução do programa. Informações de tipo e controle podem ser derivadas e usadas para aumentar a velocidade de execução e reduzir o tamanho do código. Os algoritmos utilizados por Prolog são instâncias de um método geral chamado de interpretação abstrata [35]. A idéia é executar o programa sobre um domínio simples. Se um pequeno conjunto de condições for satisfeito, esta exe-

cução termina e seus resultados provêem uma correta aproximação de informações sobre o programa original.

Warren [126] foi o primeiro a estudar a praticidade da análise global em programação lógica. Ele descreve dois analisadores de fluxo de dados:

1. *MA3, um analisador e-paralelo*, e

2. *Ms, um analisador experimental*.

MA3 deriva tipos instanciados e mantém referências das estruturas e termos compostos, enquanto Ms deriva tipos instanciados que não são variáveis. Seu trabalho demostra que ambos analisadores são efetivos em derivar tipos e não aumentam o tempo de compilação de forma significativa. Marien [77], Van Roy [117], e mais recentemente Morales [82] também obtiveram resultados similares.

Type-feedback profiling é bastante usual neste contexto para guiar na seleção dos caminhos executáveis a partir de uma execução dinâmica. O compilador otimizador desenvolvido nesta tese compila apenas caminhos relevantes, e interpreta o restante do código.

2.4 Sistemas

Emuladores de Prolog compilam código Prolog para o código de uma máquina abstrata que é depois interpretado em tempo de execução. Como exemplo deste tipo de implementação temos o SICStus Prolog [104, 86]. Contudo este tipo de sistema não aproveita ao máximo a capacidade da máquina. SICStus Prolog Ã© um bom ambiente apenas para arquiteturas RISC, obtendo bons resultados para programas pequenos. Atualmente este projeto foi abandonado.

Aquarius [117] e Parma [114] foram projetados para provar que as linguagens lógicas podem ter desempenho semelhante ao das linguagens imperativas. O bom desempenho destes sistemas deve-se a:

- *Geração de código nativo*. Foram criadas novas máquinas abstratas constituídas por instruções simples, mais próximas do conjunto de instruções de um microprocessador comum.

- *Análise global do programa*. Isto permite especializar ainda mais a unificação e explorar ao máximo o determinismo do programa.

Segundo o autor de Aquarius [117], este é mais rápido do que as implementações anteriores e é competitivo com a linguagem C para os programas onde análises globais funcionam muito bem. Contudo, este sistema também foi abandonado por causa da sua complexidade. Parma através de análises globais gera um código compacto. Além disto, em alguns casos o desempenho é melhor do que o obtido por SISCtus Prolog. Contudo, assim como este último também foi abandonado.

Gerar código de máquina é um problema complexo. Uma solução é gerar código C [44, 82, 54] a partir de Prolog e deixar o trabalho de otimização para o compilador C.

GNU Prolog produz código executável muito compacto, evitando a ligação do código nativo com as bibliotecas de predicados *buit-in* não utilizados. Os resultados obtidos [82] pelo trabalho de Morales foram consideráveis para programas que utilizam predicados *built-ins* aritméticos e para programas onde análises globais funcionam.

Como o trabalho desenvolvido por Morales, os melhores resultados obtidos por Mercury mostram que o bom desempenho somente é obtido para os programas onde análises globais funcionam muito bem. Contudo, o projeto de Mercury optou por alterar a linguagem para melhorar o desempenho. Portanto, Mercury não é Prolog e conseqüentemente nem todos os programas escritos em Prolog podem ser reescritos em Mercury.

2.4.1 SICStus Prolog

SICStus Prolog [104, 86] foi desenvolvido pelo Swedish Institute of Computer Science (SICS). O objetivo principal é ter um sistema portável e compacto. O modelo implementa certas características que não são tratadas na WAM original, tais como:

- *shallow backtracking* [19],
- *predicados aritméticos*,
- *cortes* e
- *coleta de lixo* [9].

A implementação consiste de quatro módulos: *o interpretador, o ambiente de execução, o ambiente de programação e o compilador*. Sua implementação é baseada na WAM com significantes extensões. SICStus Prolog provê um ambiente que incorpora três diferentes níveis de execução:

1. *interpretação*;

2. *emulação*; e

3. *geração de código nativo*.

Interpretar predicados fornece suporte a depuração e as atualizações dinâmicas na base de dados. Cláusulas interpretadas são representadas como termos, para acelerar as atualizações na base de dados. Predicados emulados simplificam a carga do sistema, e a gerência de memória, além de permitirem compilação rápida. Execução rápida é feita compilando código WAM para código nativo.

Para interpretar predicados, SICStus Prolog utiliza *meta-interpretação*, ou seja, um intrepretador escrito em Prolog. Quando um predicado interpretado é chamado, o controle é simplesmente transferido para o meta-interpretador. Por outro lado, predicados nativos precisam de suporte especial. Este suporte é implementado através de um protocolo chamada/retorno: o controle é transferido para código nativo através de uma chamada de função e o retorno é feito através de um `return`. Existem cinco casos possíveis para o controle ser transferido para código nativo ou vice-versa.

1. *O interpretador chama um predicado em código nativo*.

2. *O interpretador realiza um retrocesso para código nativo*.

3. *O interpretador continua a execução em código nativo, então completando uma chamada nativa para um predicado emulado*.

4. *Código nativo chama código emulado*.

5. *Código nativo retorna para um código emulado*.

A figura 2.4 apresenta a estrutura do compilador SICStus, que consiste nas cinco fases seguintes:

1. *O conversor* converte o código fonte de uma cláusula para uma árvore sintática abstrata. Durante esta conversão o compilador realiza integração de alguns predicados. Nesta conversão algumas informações estáticas são coletadas e armazenadas na árvore, como, por exemplo, cortes. Informações de controle, como disjunções, negações, construções *if-then-else*, são quebradas como predicados internos, para serem compilados recursivamente. Variáveis são classificadas como temporárias e permanentes e as variáveis permanentes são alocadas.

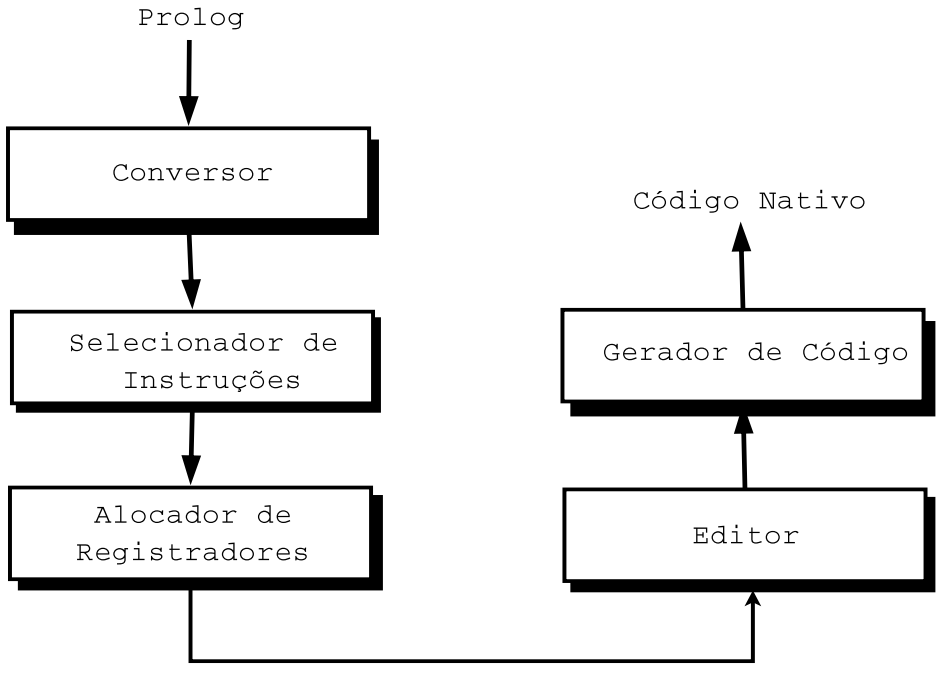


Figura 2.4: Estrutura do compilador SICStus Prolog.

2. *O selecionador de instruções* traduz a árvore sintática para uma seqüência de instruções e a partir deste ponto não é mais necessário nenhum processamento sobre a cláusula corrente.

3. *O alocador de registradores* aloca registradores para as variáveis temporárias da cláusula. A alocação de registradores é realizada por blocos, e envolve computar o conjunto de variáveis vivas para cada instrução.

4. *O editor* aplica a otimização *shallow backtracking*.

5. *O gerador de código* gera código nativo.

2.4.2 Aquarius

Aquarius Prolog [117], desenvolvido por Peter Van Roy, possui o propósito de provar que a linguagem Prolog pode ser implementada de maneira eficiente e que a velocidade de execução para algumas classes de problemas pode ser mais rápida do que linguagens imperativas como `C`.

Os principais objetivos de Aquarius Prolog são:

- *Alto desempenho*. Código compilado deve ser executado tão rápido quanto possível.

- *Portabilidade*. A saída do compilador deve ser facilmente adaptada para qualquer arquitetura seqüencial.

O compilador possui as seguintes características:

- *Reduzir a granulidade das instruções*: para gerar código eficiente Aquarius usa um modelo de execução e um conjunto de instruções que permite aplicar extensas otimizações.

- *Explorar o determinismo*: a maioria dos predicados é executado de forma determinística. Estes predicados são compilados como comando de seleção e utilizam retrocesso para simular um salto condicional. Aquarius explora o determinismo trocando retrocessos por saltos condicionais.

- *Especializar unificações*: a unificação é uma operação de casamento de padrões. A semântica desta operação corresponde a várias ações na implementação, incluindo passagem de parâmetros, atribuições de valores, alocação de memória e salto condicional. Geralmente existe apenas a necessidade de atribuir um valor a uma variável. Aquarius simplifica o mecanismo geral da unificação.

- *Análise de fluxo de dados*: fornece as informações necessárias à exploração de determinismo e à especialização de unificações quando possível.

A figura 2.5 mostra a arquitetura do compilador Aquarius. O compilador Aquarius portanto possui as quatro fases seguintes:

1. *O conversor para Kernel Prolog* converte código Prolog para a representação *Kernel Prolog*.

2. *O otimizador Kernel Prolog* aplica otimizações à representação *Kernel Prolog*.

3. *O conversor Kernel Prolog para BAM* transforma a representação *Kernel Prolog* na representação *Berkeley Abstract Machime*.

4. *O otimizador BAM* aplica diversas otimizações à representação *Berkeley Abstract Machine*.

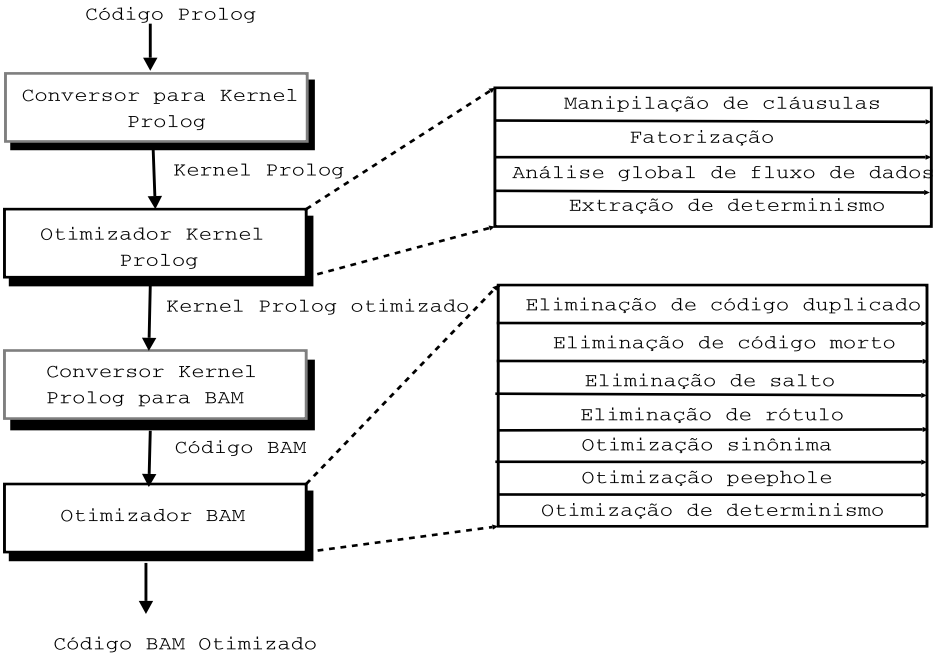


Figura 2.5: Estrutura do compilador Aquarius.

O fluxo de controle da representação *Kernel Prolog* é simples. *Kernel Prolog* consiste de um conjunto de primitivas, usadas apenas pelo compilador, e uma estrutura de seleção. *Kernel Prolog* não possui disjunções, estruturas *if-then-else*, cortes, negações ou expressões aritméticas.

Três otimizações são aplicadas à representação *Kernel Prolog*:

1. *Fatoração*: agrupa conjuntos de cláusulas em um predicado se elas possuem unificações em comum. Isto reduz o número de unificações e certos passos de retrocesso.

2. *Análise global de fluxo de dados*: analisa o programa, anota-o com tipos, e reestrutura-o. O analisador aplica interpretação abstrata para determinar os tipos dos argumentos dos predicados.

3. *Transformação determinística*: reescreve o programa para tornar seu determinismo explícito, substituindo retrocesso por saltos condicionais. A transformação determinística converte o predicado em uma série de estruturas de seleção. Algumas vezes isto é parcialmente bem sucedido, pois certos saltos na estrutura de seleção podem reter disjunções que não podem ser convertidas em código determinístico.

A compilação para *Berkeley Abstract Machine* (BAM) é feita por predicado em dois passos. No primeiro passo, as instruções que constróem o ambiente do predicado são

compiladas por um compilador de predicados. Isto inclui compilar estruturas de seleção determinísticas em saltos condicionais e as disjunções em instruções de ponto de escolha. No segundo passo, as cláusulas são compiladas pelo compilador de cláusulas. O compilador de cláusulas compila chamadas e unificações. O compilador de cláusulas também realiza alocação de registradores.

Depois de compilar o programa de *Kernel Prolog* para código *Berkeley Abstract Machine*, uma série de otimizações são aplicadas, a saber:

- *Eliminação de código duplicado*: troca os blocos contínuos duplicados, exceto a última ocorrência, por um salto para o último bloco.
- *Eliminação de código morto*: elimina todo código, de um predicado, não atingido.
- *Eliminação de salto*: reagrupa os blocos básicos para minimizar a quantidade de saltos, chamadas e instruções de retorno.
- *Eliminação de rótulo*: remove todos os rótulos não atingidos por pelo menos um salto.
- *Otimização de sinônimos*: analisa o código e troca todos os modos de endereçamentos por um modo de endereçamento mais simples que contém o mesmo valor.
- *Otimização peephole*: troca as instruções de uma janela por instruções mais simples. No compilador Aquarius, uma janela é um conjunto de três instruções.
- *Otimização de determinismo*: remove uma instrução de escolha se ela é seguida por uma seqüência de instruções que não podem falhar e uma instrução de corte.

2.4.3 Parma

Parma [114] é um compilador Prolog experimental para arquitetura MIPS, desenvolvido no departamento de Ciência da Computação da Universidade New South Wales em Sidney na Austrália. O componente mais importante no desempenho do Parma é a fase de análise global, que examina o programa como um todo para reunir informações que serão utilizadas durante a compilação do programa.

O paradigma utilizado na análise global em Parma é a interpretação abstrata [34]. A fase de análise é implementada por aproximadamente 4.500 linhas de SICStus Prolog [20]. A análise assume que o ponto de entrada para o programa é uma chamada para o predicado *main/0*.

O esforço maior foi devotado para o projeto do domínio abstrato do analisador de Parma. A observação de que a interpretação abstrata poderia reunir informações sobre características operacionais dos programas, tais como desreferenciamento, foi provavelmente o aspecto mais importante deste trabalho. Implementações anteriores de Prolog aceitaram o custo de muitas operações inerentes à linguagem. As informações reunidas pela fase de análise do Parma permitem que muitas dessas operações sejam removidas. As características do domínio abstrato são necessárias para prevenir perdas de informações durante a análise. Uma descrição do domínio abstrato em Parma pode ser encontrada em [114].

Parma utiliza um modelo de memória similar ao da WAM. A maior diferença está no armazenamento de informações em pontos de escolha, para reduzir o custo do retrocesso. Como na WAM, argumentos do objetivo são passados através de registradores.

A fase de compilação do Parma é implementada por aproximadamente 6.000 linhas de SICStus Prolog. Diferente da fase de análise, reduzir o tempo de execução não foi uma prioridade na implementação da fase de compilação. Como resultado, o tempo de execução total do Parma é bastante lento. A figura 2.6 mostra a estrutura do compilador de Parma.

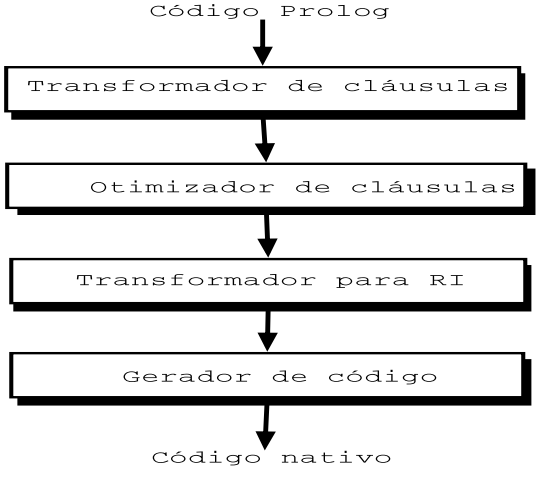


Figura 2.6: Estrutura do compilador usado por Parma.

O compilador de Parma portanto compila as cláusulas Prolog para linguagem Assembly MIPS em quatro fases:

1. *O transformador de cláusulas* transforma cláusulas para uma forma mais simples, mas que ainda é Prolog. Basicamente, unificações são movidas para fora da cabeça

da cláusula e os sub-objetivos no corpo da cláusula são divididos em componentes mais simples. Isto é conveniente para as fases subseqüentes de compilação.

2. *O otimizador de cláusulas* aplica um pequeno conjunto de transformações à esta nova forma. Apenas a transformação de reordenar algumas conjunções é descrita em [114]. Esta reordenação envolve apenas unificações e chamadas para alguns predicados *built-in*.

3. *O transformador para RI* traduz as cláusulas Prolog para instruções de uma representação intermediária de baixo nível. É nesta fase onde a maioria do trabalho de compilação acontece. As informações da fase de análise são usadas para simplificar operações de unificação. Em geral, o custo de uma operação de unificação é muito alto, podendo requerer mais de 100 instruções MIPS. A linguagem intermediária produzida por este estágio é basicamente um código de três endereços reduzido para a forma *load-store*. Esta representação é bem adequada para as otimizações que devem ser aplicadas para produzir um bom código. A linguagem intermediária utilizada por Parma possui um conjunto de registradores usados para representar o estado da execução e passar argumentos. Na última fase de compilação o alocador de registrador atribui registradores reais aos registradores simbólicos.

4. *O gerador de código* do compilador traduz código intermediário para linguagem Assembly MIPS [105]. Nesta fase o código é manipulado em blocos básicos e registradores reais são atribuídos aos registradores simbólicos. Após esta fase, o programa fonte Prolog está totalmente traduzido em linguagem Assembly MIPS.

2.4.4 GNU Prolog

GNU Prolog é um compilador Prolog desenvolvido por Diaz [44, 43]. O desenvolvimento do GNU Prolog iniciou-se em janeiro de 1996 com o nome de Calypso. Posteriormente, ele foi lançado como um produto GNU em abril de 1999 com o nome de GNU Prolog.

GNU Prolog é baseado na experiência do autor com o *wamcc* [29]. O principal objetivo do GNU Prolog foi desenvolver um sistema Prolog livre, aberto, robusto e extensível. GNU Prolog é capaz de produzir códigos executáveis eficientes.

O desempenho do *wamcc* desotimizado era bem próximo de sistemas comerciais baseados em um interpretador otimizado. Do ponto de vista do usuário a principal vantagem do *wamcc* era a sua habilidade para produzir código executável *stand alone*, enquanto a

maioria dos outros sistemas Prolog requer interpretador. Entretanto, há uma série de desvantagens quando se compila Prolog para C: o do arquivo C gerado e o tempo necessário para compilar um programa tão grande. Na verdade, um programa Prolog resulta em muitas instruções WAM e fazer integração de cada instrução geraria um arquivo C extremamente grande que não seria fácil de compilar. A *wamcc* resolve este problema traduzindo a maioria das instruções WAM para uma chamada para uma função C. A execução fica mais lenta mas a compilação é muito mais rápida, além do código executável ser menor. Infelizmente, mesmo com esta solução, o compilador C toma muito tempo para programas fontes grandes, especialmente durante a fase de otimização.

GNU Prolog permite parar o compilador em qualquer estágio. Isto pode ser útil, por exemplo, para ver e/ou analisar o código WAM produzido. A figura 2.7 apresenta a estrutura do compilador GNU Prolog, que consiste nas três fases seguintes:

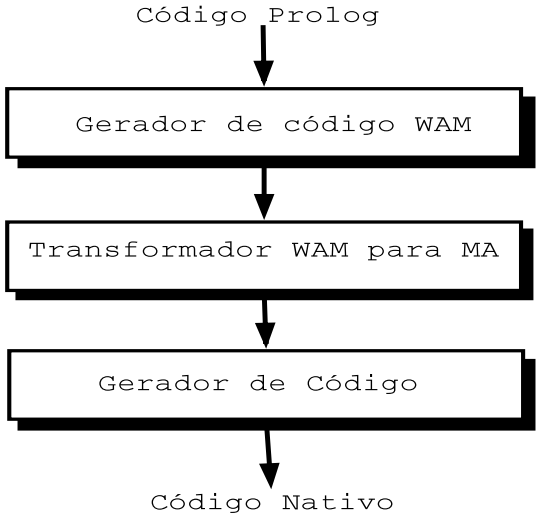


Figura 2.7: Estrutura do compilador GNU-Prolog.

1. *O gerador de código WAM* é o sub-compilador *pl2wam* que recebe um programa Prolog e produz um arquivo WAM. Este compilador é completamente escrito em GNU Prolog. Ele compila cada cláusula em várias fases. Primeiro, a cláusula é simplificada. Em seguida, a cláusula é traduzida em um formato interno mais prático e as suas variáveis são classificadas como permanentes ou temporárias. Subseqüentemente, o código WAM associado à cláusula é gerado. Finalmente, registradores (variáveis temporárias) são atribuídos e otimizados. O código das cláusulas do predicado é então agrupado e o código de indexação é gerado. O compilador *pl2wam*

aplica várias otimizações: reordenação de unificações, integração de procedimentos e otimização do último subtermo. A maioria destas otimizações pode ser desativada usando opções na linha de comando. Desativar todas as opções torna possível estudar o processo básico de compilação Prolog para WAM.

2. *O transformador WAM para MA* traduz arquivo WAM para arquivo MA. A linguagem MA é uma linguagem independente de máquina e que foi especificamente projetada para o GNU Prolog. O conjunto de instruções MA é bem simples, seguindo o padrão original utilizado pela máquina abstrata Lisp [50, 51], e fugindo do complexo conjunto de instruções utilizado pelo Aquarius Prolog [117]. A linguagem MA é baseada em apenas 11 instruções, na maioria para assegurar o controle do Prolog e para chamar uma função `C`. Segundo o autor, este novo processo de compilação é entre 5 a 10 vezes mais rápido que *wamcc+gcc*. Após a geração de código MA, o programa é ligado a uma biblioteca GNU Prolog para produzir um código executável. Foi observado que no compilador GNU Prolog a maioria das instruções WAM são implementadas através de uma chamada à uma função `C`. A única exceção são instruções para gerenciar o controle do Prolog.

3. *O gerador de código* do processo de compilação consiste em mapear o arquivo MA para Assembly da máquina alvo. Dado que a linguagem MA é baseada num conjunto de instruções reduzido, a escrita do tradutor é simplificada. Mesmo assim, produzir instruções de máquina não é uma tarefa fácil.

Após ser gerado código da máquina alvo, todos os arquivos objetos são ligados com as bibliotecas Prolog. Este estágio resolve símbolos externos, por exemplo, uma chamada a um predicado definido em outro módulo. Finalmente, obtém-se um arquivo executável.

2.4.5 Mercury

O objetivo de Mercury [54, 33] é melhorar a produtividade de grupos de programadores que desenvolvem grandes aplicações escritas em lógica. O projeto de Mercury propõe as seguintes alternativas para Prolog:

- *Um sistema de módulos*: para suportar grupos de programadores cooperando na construção de grandes aplicações.

- *Um sistema de determinismo*: as informações de determinismo permitem ao compilador limitar o retrocesso para os predicados onde o programador espera que ele aconteça.

- *Um sistema de tipos*: as informações de tipo permitem ao compilador especializar a representação de termos para cada tipo de dado.

- *Um sistema de modos*: as informações de modo permitem ao compilador emitir código especializado para cada unificação e para cada modo de uso de predicados multi-modos.

Os sistemas de tipo, modo e determinismo de Mercury facilitam o trabalho de compilação, porque provêem o compilador com informações exatas sobre os predicados. E possibilitam a detecção de erros cometidos pelos programadores.

O gerador de código de Mercury é diferente dos geradores de código para Prolog, pois o compilador é projetado para fazer o melhor uso das informações providas pelas declarações em Mercury.

O compilador de Mercury usa duas representações internas. A primeira é uma forma anotada do programa fonte, chamada de estrutura de dados de alto nível (EDAN). A fase de análise do compilador utiliza esta representação. O gerador de código traduz esta representação em código `C`, chamada de estrutura de dados de baixo nível (EDBN). A fase de otimização *peephole* utiliza esta segunda representação, que também é a saída do compilador.

O compilador de Mercury possui as fases mostradas na figura 2.8.

O compilador Mercury portanto possui as cinco fases seguintes:

1. *O tradutor para EDAN* do compilador converte o programa fonte para a representação EDAN.

2. *O analisador de tipo* realiza a análise de tipo. A análise de tipo processa cada predicado e tenta encontrar um único tipo de atribuição para cada variável. Se isto não ocorre, o predicado é *rejeitado*.

3. *O analisador de modo* realiza a análise de modo. A análise de modos é um estágio crucial para o compilador. Durante a análise de modos, cada predicado é transformado de uma forma declarativa em um procedimento. Para cada modo declarado de um predicado, o compilador cria um procedimento e o instala em uma tabela de procedimentos.

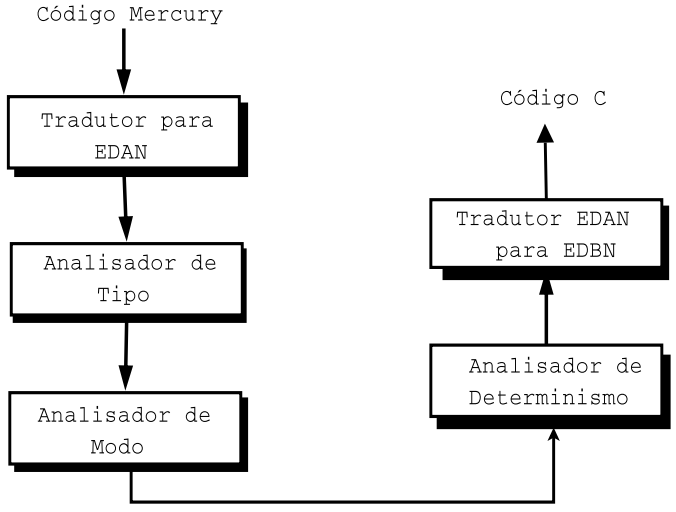


Figura 2.8: Estrutura do compilador Mercury.

4. *O analisador de determinismo* realiza a análise de determinismo. A análise de determinismo calcula o determinismo de cada objetivo. Nesta fase o compilador anota cada objetivo com o determinismo calculado. Se o determinismo calculado não combina com o determinismo declarado no predicado, o compilador rejeita o predicado e emite um erro.

5. *O tradutor EDAN para EDBN* gera código para cada objetivo. Nesta fase, o compilador utiliza informações, sobre o objetivo, coletas na fase de análise. Segundo os autores, estas informações ajudam a gerar código compacto e eficiente.

2.4.6 O Compilador CIAO

Morales, Carro e Hermenegildo [82] desenvolveram um compilador Prolog que gera código `C`. O compilador usa análise de tipos e de determinismo para melhorar o código final, removendo checagens de tipo e de modo, além de realizar chamadas para versões especializadas de alguns predicados *built-in*. A figura 2.9 mostra as fases deste compilador.

O compilador de Prolog para `C` está portanto estruturado nas três fases seguintes:

1. *O tradutor WAM* realiza a normalização das cláusulas, escrevendo as cláusulas em uma representação simplificada e homogênea do código WAM. Esta fase também realiza análise local, que fornece informações sobre o tipo e o estado das variáveis. Esta análise permite melhorar o código, mesmo no caso de não estarem disponíveis informações externas.

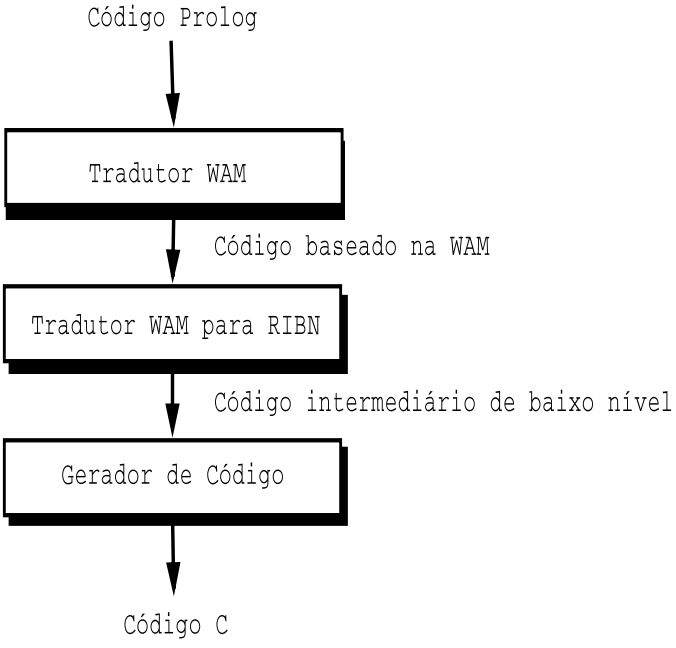


Figura 2.9: Estrutura do compilador CIAO.

2. *O tradutor WAM para RIBN* divide as instruções WAM em instruções mais simples, que são mais suscetíveis a otimizações. Isto também permite simplificar a geração final de código `C`.

3. *O gerador de código* produz código `C`. Cada bloco básico da representação intermediária de baixo nível é traduzido para uma função `C`, onde o estado da máquina abstrata é o argumento de entrada da função, e os dados necessários ao próximo bloco básico são os argumentos de saída. Isto evita a construção de funções gigantescas que poderiam criar problemas para o compilador `C`.

Este trabalho implementa as seguintes otimizações:

- *Unificação two-stream*. A unificação de um registrador com uma estrutura ou constante requer testes para determinar o modo de unificação (leitura ou escrita). Além disto, no modo leitura, um teste adicional é realizado para comparar o valor do registrador com a constante, ou com o *functor* da estrutura. Estes testes podem ser freqüentemente efetuados em tempo de compilação se informações suficientes forem conhecidas sobre a variável.

- *Geração de árvore de índice*. Informações de tipo são usadas para otimizar a geração da árvore de índice. Uma árvore de índice é gerada selecionando alguns literais

do começo da clásula, predicados *built-ins* e unificações, que fornecem informações de tipo e/ou de modo. As informações são usadas para construir uma árvore de decisão para o tipo do primeiro argumento. Quando informações de tipo estão disponíveis, a árvore de índice pode ser otimizada removendo alguns testes.

- *Argumentos de saída não inicializados*. A idéia é, quando possível, deixar a chamada do predicado preencher os conteúdos dos argumentos de saída em registradores pré-estabelecidos. Isso evita alocação e inicialização de variáveis livres, o que é mais lento.

- *Versões otimizadas de predicados*. Chamadas para predicados podem ser otimizadas na presença de informações de tipo e de modo. Versões especializadas podem existir e serem selecionadas usando padrões de chamadas deduzidos das informações de tipo. A implementação atual não suporta versões automáticas de predicados de usuário, apenas otimiza predicados *built-in* escritos em C que resulta em relevantes *speedups* em muitos casos.

2.5 Considerações Gerais

Prolog foi desenvolvido no início da década de 70 por Colmerauer [32]. Este primeiro sistema foi um emulador. O trabalho de David Warren no final desta mesma década resultou no primeiro compilador Prolog [122]. A sintaxe e a semântica deste compilador tornou-se padrão na comunidade de programação em lógica, comumente conhecido como o padrão Edimburgo. O trabalho de Warren na implementação de Prolog culminou no desenvolvimento da *Warren Abstract Machine* em 1983 [125], um modelo de execução que se tornou o padrão de implementações Prolog.

Porém, estas implementações são uma ordem de magnitude mais lenta do que linguagens imperativas. Como resultado, a aplicação prática de programação em lógica esbarra em duas questões. Por um lado, ela pode gerar apenas interesse ao mundo acadêmico, com pouco uso no mundo real. Ou ela poderia prosperar como uma ferramenta prática. A escolha entre estas duas questões depende crucialmente em melhorar a eficiência na execução.

Aquarius [117] foi desenvolvido com o objetivo de provar que a linguagem de programação Prolog pode ser implementada de uma foma eficiente e ser uma linguagem competitiva com linguagens como C. O esforço no projeto de Aquarius foi codificar cada

característica de Prolog da maneira mais simples possível.

Os sistemas descritos nas seções 2.3 e 2.4 desenvolveram basicamente quatro técnicas para melhorar a eficiência da execução de Prolog, são elas:

1. ***Reduzir a granularidade das instruções WAM.*** O objetivo é utilizar um modelo de execução simplificado [68, 113], que mantenha as características da máquina abstrata de Warren, mas que seja mais fácil de otimizar para uma arquitetura real.

2. ***Explorar o determinismo.*** Como diversos predicados escritos por programadores são determinísticos, deve-se compilar tais programas utilizando eficientes saltos condicionais [104, 76, 79].

3. ***Especializar unificação.*** Devido a unificação poder realizar diversas tarefas, o objetivo é compilar a unificação para um código simples [115, 75, 116, 117, 78, 14].

4. ***Utilizar análise de fluxo de dados.*** Outra técnica é derivar informações sobre o tipo dos dados através de análise global e usá-las para otimizar o código gerado [126, 77, 117, 82]

Além destas técnicas, os sistemas GNU Prolog e Ciao utilizam a geração de código nativo para melhorar o desempenho dos programas Prolog.

No contexto de Prolog, a proposta da presente tese é também utilizar as mesmas técnicas, porém, através de compilação dinâmica. Os objetivos são manter a portabilidade dos programas Prolog e desenvolver um sistema fácil de se manter.

A presente tese propõe que um sistema de compilação dinâmica utilizado por um emulador possa melhorar significativamente o desempenho de aplicações Prolog, sem precisar de análises globais. Além disto, a tese propõe que o sistema de compilação não seja intrusivo ao ambiente de execução a medida que este utiliza técnicas simples.

Deutsch and Schiffman foram os pioneiros em usar compilação dinâmica em sistemas orientados a objetos [41]. Um trabalho relevante nesta área foi o compilador para a linguagem Self desenvolvido por Holzle e Chambers [58, 23]. Contudo, compilação dinâmica tornou-se popular somente com o advento da liguagem Java [112, 26, 88, 69, 87, 129].

Para entender as técnicas implementadas na tese proposta é necessário primeiramente entender os conceitos fundamentais sobre compilação dinâmica. Este é o objetivo do próximo capítulo: *descrever os princípios da compilação dinâmica e o funcionamento dos compiladores dinâmicos utilizados em linguagens como Self e Java e as otimizações por eles aplicadas*.

Capítulo 3

Compilação *Just-In-Time*

Em linguagens orientadas a objetos, interpretar um método é lento porque cada instrução emulada requer a execução de um *template* consistindo de diversas instruções de máquina [99]. Para se obter bom desempenho é necessário compilar as instruções para código de máquina. Se o processo de compilação ocorre enquanto o programa está sendo executado, ele é chamado de compilação dinâmica ou ainda *Just-In-Time* (JIT) [92].

Deutsch e Schiffman foram os pioneiros em usar compilação dinâmica em sistemas orientados a objetos. Sua implementação de Smalltalk [41] dinamicamente traduz os *bytecodes* definidos pela máquina virtual Smalltalk [49] em código nativo e armazena o código compilado para uso posterior. Eles estimam que usar apenas uma simples compilação dinâmica ao invés de interpretar acelera seu sistema por um fator de 1,6 e que um compilador sofisticado possui apenas um fator de 2.

Franz [48] descreve uma variação de compilação dinâmica que gera código de máquina durante a carga para uma representação intermediária compacta. A compilação a partir de código intermediário para código de máquina é rápida o suficiente para fazer o carregador competitivo com carregadores padrões.

Compilação dinâmica é também usual para outras aplicações além de linguagens de programação. Kessler [64] usou para implementar rápidos *breakpoints* em depuradores. Em sistemas operacionais, compilação dinâmica tem sido usada para suporte eficiente a paralelismo [25], eliminar o *overhead* de protocolos de pilha [1] e ligação dinâmica [57]. Além desta áreas, pode ser citado ainda otimização de buscas em banco de dados [22] e geração de micro-código [97].

Em seguida são apresentados os princípios básicos de compilação *Just-In-Time*. Primeiramente são abordados os dois mecanismos utilizados por compiladores JIT na escolha de quais unidades compilar: (1) *compilar cada unidade da aplicação imediatamente*

antes de sua execução e (2) *compilar apenas as unidades executadas freqüentemente*. Em seguida são discutidos os compiladores das linguagens Self [23, 58] e Java [26, 80, 112].

3.1 Princípios de Compilação *Just-In-Time*

Um compilador JIT traduz código fonte em código de máquina a medida que este código é executado. Conseqüentemente, o tempo de compilação passa a estar inserido no tempo total de execução da aplicação. Desta forma, idealmente o compilador JIT deverá ser rápido, efetivo e leve, além de ser capaz de gerar código nativo de alta qualidade. O compilador JIT deve ser muito seletivo sobre quais unidades compilar: *deverá apenas traduzir as unidades cujo tempo gasto em tradução seja amortizado pelo ganho de desempenho em código nativo*. Neste caso o compilador dinâmico deve ser agressivo em identificar as otimizações que possam alcançar um alto desempenho.

Tais compiladores utilizam técnicas de otimização de código para obter código de alta qualidade. Contudo, nem sempre se justifica aplicar técnicas agressivas para todas as unidades do código fonte, pois nem todas as unidades são executadas freqüentemente. Otimizações leves podem ser efetivas para gerar códigos rápidos e de boa qualidade, sem aumentar drasticamente o tempo de compilação.

Um compilador JIT pode usar duas abordagens para traduzir código fonte em código nativo: *ele pode traduzir uma unidade imediatamente antes de executá-la*; ou, *baseado em informações coletadas durante a execução, identificar as unidades freqüentemente executadas e então otimizar somente estas*.

Um exemplo da primeira abordagem é o ambientes Java Kaffe [62]. Exemplos da segunda abordagem são ambientes SELF [58], Jikes [7, 61] e JUDO [26]. Inicialmente a aplicação é compilada por um compilador não otimizador e, também baseado em informações coletadas durante a execução inicial, partes da aplicação são compiladas por um compilador otimizador [23]. O compilador de Java JUDO utiliza a mesma abordagem. Outros ambientes Java, por exemplo Sun HotSpot Compiter [80, 88] e IBM JIT Compiler [112], também utilizam a segunda abordagem. Porém, estes inicialmente interpretam as unidades e compilam somente as unidades executadas freqüentemente.

A abordagem de interpretar (ou compilar) primeiro e depois compilar (ou recompilar) é baseada na observação de que a maioria dos programas gastam a maior parte do seu tempo em uma pequena faixa de código. Esta abordagem compila apenas estas partes de código. Para tal, as unidades de código são intrumentadas com contadores.

Cada unidade possui dois contadores: um contador de entrada e um contador de retorno. O primeiro é incrementado no ínicio da execução de cada unidade. O outro é incrementado quando um salto de retorno à unidade é executado. Se estes contadores excedem um limite pré-definido a unidade é escalonada para compilação.

Por outro lado, os contadores das unidades que não são executados freqüentemente, por exemplo apenas uma vez no início da aplicação, nunca atingirão o limite determinado e conseqüentemente nunca serão compilados. Isto reduz drasticamente o número de unidades que são compiladas. Desta maneira, o compilador gera menos código e pode gastar mais tempo otimizando o código das unidades mais importantes. É esperado que os contadores de freqüência, de todas as unidades executadas, alcançem o limite determinado e que estas unidades sejam compiladas sem gastar demasiado tempo com sua interpretação.

Uma vantagem da segunda abordagem é que garante a possibilidade de aproveitar informações obtidas durante a interpretação. Por exemplo, todas as estruturas de dados que são usadas pela unidade já estão carregadas e as unidades que são chamadas são conhecidas. Adicionalmente, o interpretador pode coletar informações como o tipo das variáveis locais. Esta informação pode ser usada para aplicar várias otimizações.

Algumas otimizações altamente efetivas são complicadas por causa da flexibilidade de determinadas linguagens de programação. Por exemplo, em Java, métodos virtuais limitam a aplicação da integração de métodos. A integração é difícil para métodos virtuais porque não se sabe estaticamente que método será chamado. Integrar métodos virtuais pode acarretar, em alguns casos, a necessidade do método compilado ser invalidado quando uma nova classe é carregada. Nestes casos, o método é compilado novamente sem esta otimização.

Remover o código compilado resulta em um retorno ao interpretador. Uma solução para este caso é substituir o código compilado pelo interpretado. Este exemplo mostra uma vantagem de ter simultaneamente código compilado e interpretado. Esta transição é chamada de *desotimização*. Para tal, o compilador deve gerar uma estrutura de dados que permita a reconstrução do estado do interpretador até o ponto de chamada do código compilado.

Desotimização é fundamental para que o compilador possa realizar agressivas otimizações que aceleram a execução normal, mas podem lidar com situações onde uma determinada otimização deve ser desfeita. Existem casos críticos onde o método compilado é desotimizado, por exemplo, quanto ocorrem exceções. Com desotimização, o código compilado não precisa gerenciar tais situações, que em geral são casos incomuns.

Como mencionado anteriormente, uma unidade é compilada quando seus contadores excedem um certo limite. Na abordagem mais simples a decisão é tomada antes de iniciar a execução da unidade e o código compilado é executado ao invés de interpretado. Infelizmente esta solução nem sempre é suficiente. Se uma unidade interpretada executa um longo laço pode ser necessário mudar para código compilado durante a execução de código [47]. Neste caso, faz sentido compilar a unidade enquanto ela está sendo interpretada e mudar o contexto de execução do emulador para o código compilado.

A última vantagem desta abordagem é que quando o compilador encontra situações que são difíceis de serem gerenciadas, a compilação da unidade é abortada e o interpretador continua a sua execução.

Outra questão importante é a escolha das otimizações que serão aplicadas ao compilar as regiões executadas com mais freqüência.

Existem ambientes Java que aplicam indiscriminadamente um conjunto de otimizações a tais regiões. Em contraste, o ambiente Self utiliza *profiling* obtido durante a execução para ajustar as otimizações aplicadas.

O uso de *profiling* no ajuste das otimizações aplicadas fornecem um melhor ganho de desempenho, pois nem sempre todas otimizações são aplicáveis a uma determinada classe de aplicações, como veremos no próximo capítulo.

3.2 Otimizações

Antes de destacar as características dos compiladores dinâmicos: Self, JUDO, os compiladores da Sun e o da IBM, esta seção apresenta uma breve descrição das otimizações por eles aplicadas. Elas podem ser agrupadas nas seguintes categorias:

- ***Otimizações para simplificar ou eliminar redundâncias***: *propagação de cópias, propagação de constantes, avaliação de expressões constantes, eliminação de expressões comuns e numeração de valores*.
- ***Otimizações para eliminar checagens***: *eliminação de checagem de exceção, eliminação de checagem de faixa de array e versões de laço*.
- ***Otimizações sobre procedimentos***: *integração de procedimentos e integração de métodos virtuais*.
- ***Otimizações de controle de fluxo***: *deslocamento de código de laço, loop unrolling e deslocamento de código*.

- ***Otimizações de baixo nível***: *eliminação de código morto, idiomas de máquina e otimização peephole*.

3.2.1 Simplificar ou Eliminar Redundâncias

Este grupo de otimizações são técnicas realizadas em tempo de compilação. O objetivo destas otimizações é eliminar computações redundantes.

3.2.1.1 Propagação de Cópias

Propagação de cópia [5] é uma transformação que, para uma atribuição `a` ← `b` troca os usos da variável `a` pelo valor da variável `b`, desde que instruções intermediárias não interfiram no valor de `a` ou `b`. Um exemplo é mostrado na figura 3.1. Propagação de cópia pode ser dividida em fase local e fase global [84]. A fase local é aplicada apenas sobre blocos básicos, sendo necessário o uso de uma fase prévia de compilação para dividir o programa fonte em blocos básicos. A fase global não necessita da divisão do programa fonte em blocos básicos, pois esta é aplicada sobre um módulo do programa (procedimento ou método). Esta otimização reduz a pressão por registradores e elimina instruções redundantes de movimentação entre registradores.

```
b = a*6
c = b
print a[c]
r = b
a[r] = a[r] + c
```

(a) código original

```
b = a*6
print a[b]
a[b] = a[b] + b
```

(b) após propagação de cópia

Figura 3.1: Propagação de cópia.

3.2.1.2 Propagação de Constantes

Propagação de constantes [18, 65, 127] é uma transformação que dada uma atribuição `a` ← `b` para uma variável `a` e uma constante `b`, troca o último uso da variável pelo valor da constante `b`, a medida que no fluxo não existam atribuições que alterem o valor de `a`. Um exemplo é mostrado na figura 3.2. Esta otimização é uma das mais importantes que um compilador pode aplicar e muitos compiladores otimizadores a aplicam de forma agressiva. Programas tipicamente contém muitas constantes, e propagando-as pelo programa o compilador pode realizar uma quantidade significante de pré-computações. Esta

otimização releva oportunidades para outras otimizações. Em adição, a óbvia possibilidade de aplicar eliminação de código morto, otimizações em laços são afetadas porque constantes geralmente aparecem nas faixas de suas iterações. Propagação de constante é particularmente importante para arquiteturas *RISC* [55] devido a dois fatores:

1. *Estas arquiteturas movem pequenas constantes inteiras para os lugares onde elas realmente são usadas*.

2. *Também possuem instruções com uma constante inteira como operando, o que permite a geração de código mais eficiente*.

Propagação de constante reduz a quantidade de registradores necessários por um procedimento e aumenta a efetividade de diversas outras otimizações, como por exemplo avaliação de expressões constantes.

```
b = 50
c = 4
for x = 1, b
   d[x] = d[x] + c
```

(a) código original

```
for x = 1, 50
   d[x] = d[x] + 4
```

(b) após propagação de constantes

Figura 3.2: Propagação de constante.

3.2.1.3 Avaliação de Expressões Constantes

Avaliação de expressões constantes [5] avalia expressões do código fonte, em tempo de compilação, com o objetivo de determinar quais deles possuem valores contantes. Por exemplo: $x = 3, 1 * *2$ é transformado em $x = 9, 61$. Esta otimização é uma transformação simples de aplicar na maioria dos casos. Em sua forma simples, avaliar expressões constantes envolve determinar quais dos operandos são valores constantes e trocar a expressão pelo seu valor. Para valores lógicos esta otimização é sempre aplicável. Para inteiros é quase sempre aplicável. Para valores de ponto flutuante a situação é mais complicada, devido a existência de possíveis exceções. A otimização avaliação de expressões constantes é normalmente estruturada como uma subrotina que pode ser invocada a qualquer momento durante o processo de otimização.

3.2.1.4 Eliminação de Expressões Comuns

Em muitos casos, um conjunto de computações podem conter expressões idênticas. Isto é verdadeiro para código de usuário e computações de endereços gerados pelo compilador. O compilador pode armazenar o valor destas expressões para usos posteriores[4, 28]. Esta transformação remove o recálculo de expressões comuns e troca-as pelo uso de valores calculados previamente.

Uma ocorrência de uma expressão em uma computação é uma expressão comum se:

- *existe outra ocorrência desta mesma expressão cuja avaliação sempre antecede sua última ocorrência*; e
- *se os operandos se mantiveram inalterados entre as duas últimas avaliações*.

Embora geralmente seja uma boa idéia eliminar expressões comuns quando possível, o compilador pode considerar a atual pressão por registradores e o custo de recomputações. Se armazenar os valores temporários força *spills* adicionais, a transformação pode não ser implementada.

```
a = b + 5
for i=1, 10
    a[i] = a[i] + b + 5
```

(a) código orignal

```
a = b + 5
for i=1, 10
    a[i] = a[i] + a
```

(b) após eliminar expressões comuns

Figura 3.3: Eliminação de expressões comuns.

3.2.1.5 Numeração de Valores

Numeração de valores foi originalmente desenvolvida por Cocke and Shwartz [28]. Esta otimização [98, 8] é um método para determinar se duas computações são equivalentes e eliminar uma delas. Um exemplo é mostrado na figura 3.4. Aplicar numeração de valores significa associar a cada nodo de uma árvore um valor simbólico único. Os valores das sub-árvores e o valor da raiz determinam o valor simbólico da árvore. Uma tabela *hash* armazena todas sub-árvores que já contenham um valor símbolo. Quando duas sub-árvores colidem é realizada uma comparação detalhada para determinar se elas são realmente idênticas.

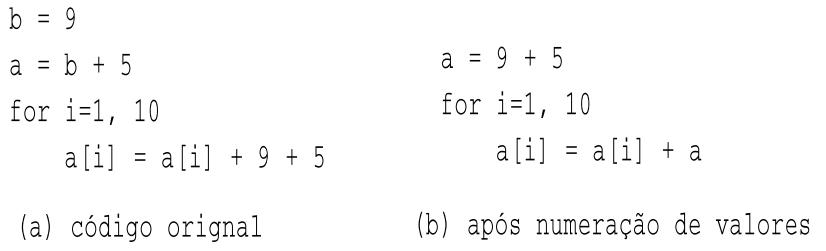


Figura 3.4: Numeração de valores.

3.2.2 Eliminar Checagens

Existem várias otimizações que melhoram o desempenho identificando computações desnecessárias e eliminando-as.

3.2.2.1 Eliminação de Checagem de Exceções

Checagem de exceção, que pode ser checagem de ponteiro nulo e/ou checagem de faixa de *array* [52], pode ser eliminada se puder ser provado que uma exceção não pode ocorrer. Checagem de ponteiro nulo pode ser resolvido com um simples fluxo de dados. Sempre que um objeto é checado, um *flag* indicando que este objeto já foi testado é simplesmente propagado ao longo do fluxo de dados. Para a checagem de faixa de *array*, descrevemos a seguir um método para eliminá-la.

3.2.2.2 Eliminação de Checagem de Faixa de *Array*

Checagem de faixa [52] determina se o valor de uma variável está dentro de uma faixa para todos os seus usos no programa. Algumas linguagens requerem que checagens deste tipo sejam realizadas para todos os acessos a *arrays*. O problema é que este teste pode ser muito caro se cada acesso for acompanhado de dois *traps* condicionais por dimensão, para determinar a validade do acesso. O *overhead* para cada checagem se torna ainda maior quando os acessos a um *array* são otimizados. Algumas implementações resolvem este problema fornecendo ao usuário mecanismos para ativar ou desativar esta checagem durante o tempo de compilação. A solução ideal, contudo, é otimizar a checagem, o que possui um custo mínimo. A aplicação desta otimização ocorre em duas fases. Primeiro, é necessário representar as restrições de checagem de faixa que devem ser satisfeitas e em seguida usa-se outras otimizações conhecidas para eliminar a checagem desnecessária, a saber: *eliminação de redundância parcial* [83] ou *eliminação de subexpressão comum* [12].

3.2.2.3 Versões de Laço

Versões de laço [81] é uma técnica para mover uma checagem de faixa de *array* para fora de um laço. O objetivo é providenciar duas cópias do laço: *o laço seguro e o laço inseguro* (original). O código para checagem de exceção é primeiro criado na entrada do laço examinando toda a faixa do índice do *array* acessado no laço. Duas versões do laço são então geradas, uma que não requer checagem de exceção e outra que é o laço original não seguro. Dependendo do resultado do teste da faixa do índice na entrada do laço, o laço seguro ou o laço não seguro é selecionado para execução.

3.2.3 Otimizar Procedimentos

As otimizações descritas nesta seção tem como objetivo reduzir o *overhead* das chamadas de procedimentos.

3.2.3.1 Integração de Procedimentos

Integração de procedimentos (ou métodos) [103, 13] é uma técnica bem conhecida que troca chamadas aos procedimentos por cópias de seus corpos. Um exemplo é mostrado na figura 3.5.

```
soma(x, y) =
  return x + y

func() =
  a = 1
  b = 6
  c = soma(a, b)
```

(a) código original

```
func() =
  a = 1
  b = 6
  c = a + b
```

(a) após integração de procedimentos

Figura 3.5: Integração de procedimentos.

Integrar procedimentos reduz o *overhead* da invocação, que pode ser significante especialmente em linguagens orientadas a objetos. Isto também aumenta o escopo de compilação, expondo novas oportunidades de otimização e eliminando o *overhead* em criar segmentos de pilha, passar argumentos e retornar valores. Excessiva aplicação desta técnica pode degradar o desempenho. Isto ocorre devido ao crescimento do tamanho do código, que pode diminuir severamente a localidade dos dados, conseqüentemente diminuindo a quantidade de instruções por ciclo [108]. Crescimento explosivo de código pode

ser evitado se concentrando apenas em métodos freqüentes. Experiências tem mostrado que escolher procedimentos pequenos é uma boa heurística [131].

3.2.3.2 Integração de Métodos Virtuais

Integração de métodos virtuais [40] é uma técnica utilizada por linguagens que possuem mecanismos para especificar métodos que podem ter comportamentos diferentes durante a execução do programa. Integrar métodos virtuais é difícil porque uma mesma chamada pode invocar diferentes métodos durante a execução do programa. Desta forma, pode ser impossível identificar qual código integrar. Porém, em muitos programas algumas chamadas virtuais executam apenas um único método, isto é, são *monomórficas* ao invés de *polimórficas*. De fato, a maioria das chamadas são provavelmente *monomórficas*. A estratégia para integrar tais métodos é selecionar um código para integrar e gerar um teste para certificar que este é o método correto. Se o teste falha, a chamada padrão de método virtual é utilizada.

3.2.4 Otimizar o Controle de Fluxo

As otimizações desta seção alteram a ordem de execução e são usadas para explorar o paralelismo e melhorar a localidade na memória.

3.2.4.1 Deslocamento de Código de Laço

Deslocamento de código laço-invariante [12] reconhece computações em laços que produzem o mesmo resultado em cada iteração e move-os para fora do laço. Um exemplo é mostrado na figura 3.6. Uma instância importante desta otimização é o cálculo de endereços que acessam elementos de *arrays*. Identificar uma computação invariante é bem simples. Primeiro, se identificam os laços através de uma análise de controle de fluxo. Em seguida, através de informações de fluxo de dados conecta-se os usos das variáveis do laço com as suas definições e neste ponto já se pode conhecer quais definições podem afetar um determinado uso de uma variável. O conjunto de instruções laço-invariante é então definido indutivamente. Uma instrução é laço-invariante se para cada um de seus operandos:

1. *este é constante*,
2. *todas definições que alcançam este uso do operando estão localizadas fora do laço*, ou

3. *existe apenas uma definição do operando que alcança a instrução, e esta definição é uma instrução dentro do laço que é em si mesma laço-invariante*.

Um caso especial de deslocamento de código laço-invariante ocorre em laço que computa redução. Uma redução pode ser uma operação de adição, multiplicação, ou valor máximo que simplifica valores acumulados. Neste caso especial, o laço pode ser trocado por uma simples operação.

```
for i = 1,n
     a[i] = a[i] + sqrt(c)
```

(a) laço original

```
if (n > 0) k = sqrt(c)
for i = 1,n
     a[i] = a[i] + k
```

(b) após deslocamento

Figura 3.6: Deslocamento de código de laço.

3.2.4.2 *Loop Unrolling*

Unrolling [12] replica o corpo do laço em um fator *u*. Desta forma a iteração do laço passa a ser *u* ao invés de *1*. Um exemplo é mostrado na figura 3.7. Esta transformação pode melhorar a qualidade do código por:

- *reduzir o overhead do laço*;
- *aumentar o paralelismo das instruções*; e
- *melhorar a localidade dos dados*.

Os benefícios desta otimização foram estudados em diversas arquiteturas [45] e é uma técnica fundamental para gerar uma seqüência longa de instruções requeridas por arquiteturas VLIW [46].

```
for i=2, n-1
     a[i] = a[i] + a[i-1] * a[i+1]
```

(a) laço original

```
for i=2, n-2, 2
     a[i] = a[i] + a[i-1] * a[i+1]
     a[i+1] = a[i+1] + a[i] * a[i+2]
```

(b) laço *unrolled* duas vezes

Figura 3.7: *Loop unrolling*.

3.2.5 Otimizações de Baixo Nível

As otimizações de baixo nível são utilizadas para explorar a arquitetura alvo.

3.2.5.1 Eliminação de Código Morto

Alguns programas podem possuir porções de código (*código morto*) que não contribuem para o resultado final. Eliminação de código morto analisa o código para detectar estas porções de código e em seguida eliminá-las [67].

Uma instrução está morta se computa apenas valores que não são mais usados em nenhum caminho executável a partir deste ponto. Uma variável está morta se não é usada em outro caminho, a partir da localização no código onde é definida até o ponto final da rotina em questão. Se o valor de uma variável morta é atribuído a uma variável local, a variável e a instrução de atribuição estão mortas se a variável não é usada em algum caminho executável da saída do procedimento. Se este valor é atribuído a uma variável de ampla visibilidade, geralmente é necessário uma análise inter-procedural para determinar se ela está morta. Um exemplo é mostrado na figura 3.8

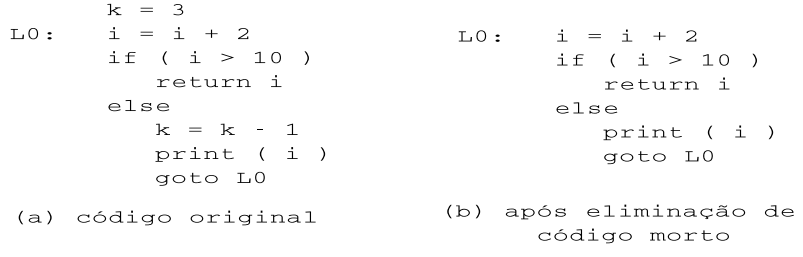


Figura 3.8: Eliminação de código morto.

3.2.5.2 Idiomas de Máquina

Idiomas de máquina [84] são instruções, ou uma seqüência de instruções, para uma arquitetura particular que provêem um mecanismo mais eficiente para executar uma determinada computação do que o mecanismo para uma arquitetura mais genérica. Um exemplo é mostrado na figura 3.9. Alguns idiomas de máquina são instâncias de *instructions combining*, que troca uma seqüência de instruções por uma simples que possui o mesmo efeito. Idiomas de máquina e *instructions combining* são otimizações apenas aplicadas após a geração de código específico ou em uma fase próxima ao final do processo de compilação. Um exemplo é o que ocorre na arquitetura SPARC [110]. Uma subtração que produz um valor é combinada com um teste que compara dois operandos.

```
sub   r1, r2, r3
...
subcc r1, r2, r0
```

```
subcc r1, r2, r3
```

(a) código original (b) após idiomas de máquina

Figura 3.9: Idiomas de máquina.

3.2.5.3 Otimização *Peephole*

Esta técnica inspeciona cada instrução ou seqüência de instruções adjacentes para determinar se ela pode ser trocada por uma instrução melhor [109]. Esta otimização é invocada por módulos dependentes de máquina no último estágio de compilação. Um exemplo é mostrado na figura 3.10.

```
movl $802356, %eax
movl (%eax), %ecx
```

```
movl (802356), %ecx
```

(a) código original (b) após *peepholing*

Figura 3.10: Peephole.

3.3 Compiladores *Just-In-Time*

Compilação dinâmica tem sido empregada no contexto de linguagens orientadas a objetos desde o trabalho pioneiro de Deutsch e Schiffman [41].

O compilador de Self [23, 58] inovou pelo fato de utilizar *type-feedback profile* para melhorar o ganho de desempenho na execução dos programa. O ambiente de execução de Self utiliza dois compiladores. Esta mesma abordagem foi utilizada pelo compilador Java desenvolvido pelo laboratório da Intel [26]. Outros ambientes de execução como os da Sun [88, 80] e da IBM [112] empregam um modo misto de execução, empregando um emulador e um compilador JIT.

3.3.1 O Compilador Self

O compilador otimizador Self [23, 58] foi projetado para um sistema que utiliza a programação exploratória e visa superar os problemas de desempenho criados pelo escalonador

dinâmico, mas sendo compatível com interatividade. Foi considerado muito importante obter alto desempenho para grandes aplicações. Isto supriu uma necessidade existente nas versões anteriores do compilador [58].

O ambiente de execução inicialmente gera código nativo com um compilador não otimizador de alta velocidade. Em um segundo passo, recompila as partes críticas do programa com um compilador otimizador. Informações dinâmicas são usadas para guiar as partes do programa que devem ser recompiladas e para guiar que otimizações aplicar.

Dado que o sistema Self usa compilação dinâmica, o compilador deve ser tão rápido quanto possível para minimizar as pausas na execução causadas pela compilação. Segundo os autores, a compilação é rápida e não intrusiva. O código do compilador é o mais compacto possível visando facilitar a análise de sua estrutura, facilitar a depuração e deixá-lo leve. Os autores evitaram otimizações e análises globais de alto custo, além de apenas usarem algoritmos cuja complexidade fosse linear no tamanho do código da aplicação, como por exemplo alocação de registradores baseada em contagem de uso [58].

A estrutura do compilador é apresentada na figura 3.11.

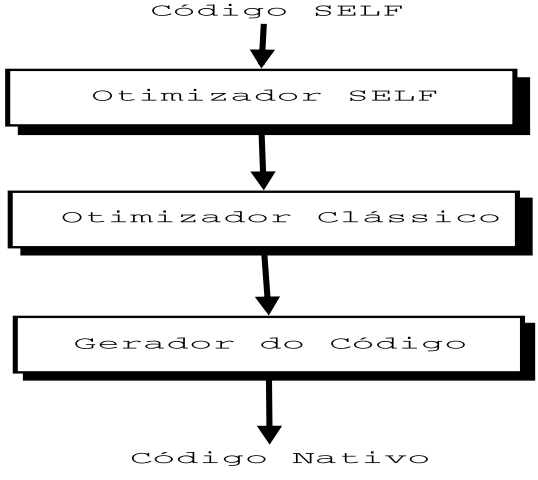


Figura 3.11: Estrutura do Compilador Self.

O compilador está portanto dividido nas seguintes três fases:

1. *O otimizador Self* realiza várias otimizações que são características de linguagens puramente orientadas a objetos, por exemplo *type feedback-based inline* [24] e *splitting* [23]. O resultado é um código intermediário na forma de um grafo.

2. *O otimizador clássico* realiza algumas otimizações clássicas, tais como: *constant folding*, eliminação de expressões comuns, propagação de cópias, eliminação de

código morto e escalonamento de instruções.

3. *O gerador de código* completa o processo de compilação. Nesta fase é realizada a alocação de registradores e a geração de código nativo. Para a alocação de registradores um alocador baseado na contagem de uso atribui registradores às variáveis gerando código nativo a partir do grafo de fluxo de controle. O código nativo é gerado em um único passo através do grafo.

O compilador Self realiza poucas otimizações no código intermediário antes de gerar código nativo. Em particular, o compilador não realiza análises de fluxo de dados ou alocação de registradores baseado em coloração de grafos, devido estas técnicas serem consideradas caras em termos de velocidade de compilação.

Segundo os autores, *type feedback* e recompilação adaptativa melhoram o desempenho e a interatividade. Em seus estudos, os programas executaram mais rápido em um fator de 1,7 quando compilados com *type feedback*. Além disto, a quantidade de chamadas diminuiu em um fator de 3,6. Seus estudos indicam que programas orientados a objetos podem executar eficientemente sem suporte especial de *hardware*, utilizando a estratégia correta de compilação.

3.3.2 *Java Under Dynamic Optimizations*

O compilador *Java Under Dynamic Optimizations* (JUDO) [26], desenvolvido pelo laboratório da Intel, implementa um mecanismo de recompilação dinâmica. O propósito de JUDO é usar a tradução de código simples para métodos não freqüentes e aplicar otimizações caras apenas para métodos freqüentes.

Recompilação dinâmica acontece durante o tempo de execução. Desta maneira, deve-se ter a certeza de que o tempo gasto em recompilação é compensado pelo desempenho obtido pela recompilação. Inicialmente, todos os métodos são compilados por um gerador rápido de código que produz razoavelmente bom código. Minimizar o tempo de compilação e obter um histórico de informações são as maiores preocupações deste compilador. Enquando o programa está em execução, o ambiente de execução identifica métodos freqüentes e realiza otimizações caras para melhorar a qualidade do código destes métodos.

A estrutura de JUDO consiste de três componentes:

1. *um gerador rápido de código* (compilador base) [2];

2. *um compilador otimizador* [26]; e

3. *um coletor de informações* (*profiler*).

A figura 3.12 mostra a estrutura do sistema. Todos os métodos são compilados para código nativo pelo gerador rápido de código à primeira chamada. O gerador rápido de código insere instrumentação no código nativo para coletar um histórico de informações. A instrumentação atualiza o histórico do método, sempre que um método é executado. Baseado nos dados coletados, métodos podem ser identificados como métodos freqüentes e recompilados pelo compilador otimizador, usando as informações do histórico para guiar as otimizações.

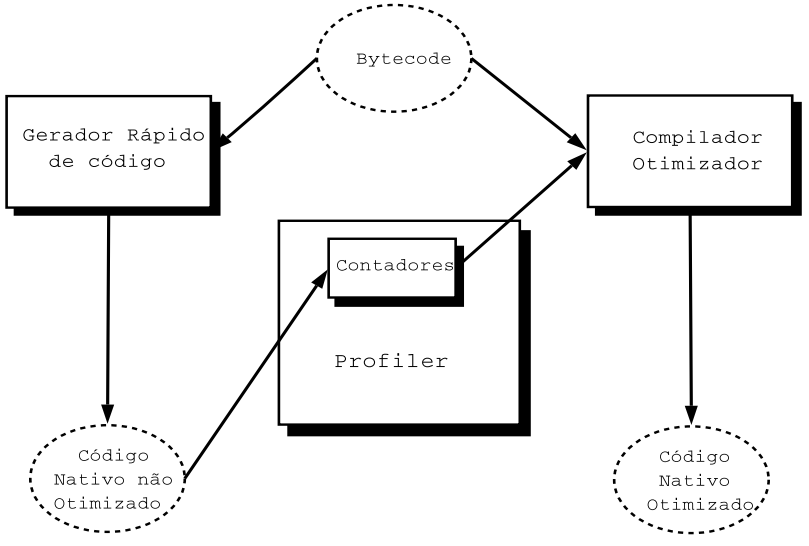


Figura 3.12: Componentes de JUDO.

O principal objetivo do gerador rápido é produzir código nativo com a qualidade do código razoável. Esta geração de código é realizada em duas passagens pelos *bytecodes* e segundo os autores possui complexidade de tempo linear:

1. *O detector de blocos básicos* coleta informações sobre as faixas de blocos básicos e a profundidade dos operandos de pilha;

2. *O otimizador e gerador de código* utiliza seleção tardia de código [2] para gerar código nativo eficiente e realiza algumas otimizações leves, por exemplo, checagem de faixa e eliminação de expressões comuns.

O gerador rápido instrumenta o código com contadores para coletar um histórico de informações e acionar a recompilação. A estrutura do gerador rápido de código pode ser vista na figura 3.13(A).

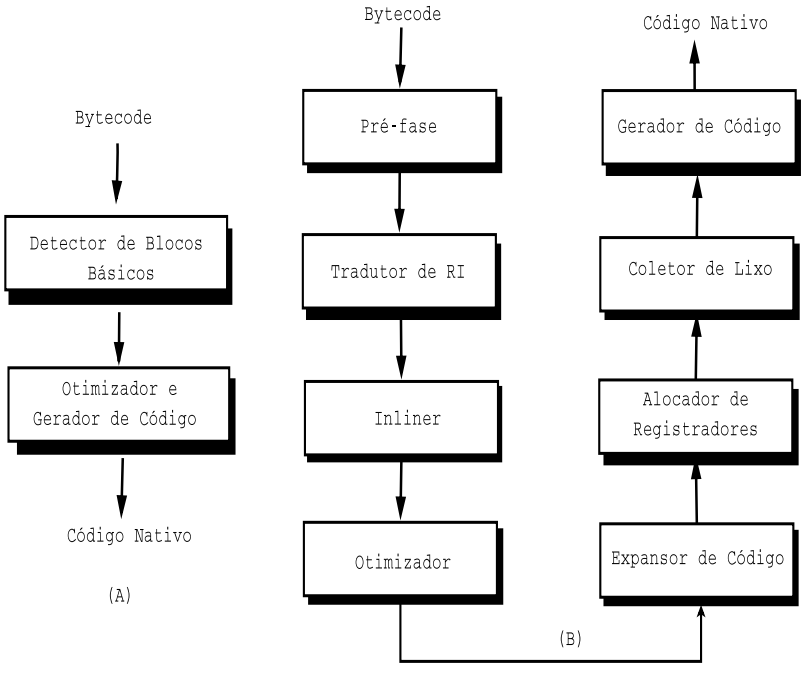


Figura 3.13: Estrutura dos compiladores utilizados por JUDO.

O compilador otimizador, figura 3.13(B), realiza a compilação dos métodos em oito fases:

1. *A pré-fase* atravessa os *bytecodes* originais e detecta os blocos básicos existentes.

2. *O tradutor de RI* constrói um grafo de controle de fluxo e decide qual seqüência de intrução de representação intermediária será utilizada para cada bloco básico. A eliminação de expressões comuns é feita durante esta construção.

3. *O inliner* identifica quais métodos são candidatos a serem integrados. Após esse passo, um grafo de controle de fluxo e uma representação intermediária são construídos para os métodos candidatos e estes são inseridos aos métodos chamadores.

4. *O otimizador* aplica as seguintes otimizações: propagação de cópia, avaliação de expressões constantes, eliminação de código morto, eliminação de checagem de faixa de *array* e deslocamento de código laço-invariante.

5. *O expansor de código* expande apenas algumas instruções da representação intermediária para código nativo.

6. *O alocador de registradores* atribui ao código registradores físicos e gera *spill code* [15].

7. *O coletor de lixo* fornece informações de suporte ao coletor de lixo construindo o mapa do coletor para cada instrução.

8. *O gerador de código* emite código nativo.

A avaliação deste sistema [26] mostra que ele gera código de boa qualidade e gerencia exceções de uma maneira eficiente.

3.3.3 Os Compiladores Java da Sun

A máquina virtual Java da Sun [80] está disponível em duas versões: a máquina virtual *cliente* e a máquina virtual *servidora*. A máquina virtual Java HotSpot *cliente* é para a execução de aplicações interativas e como tal é ajustada para compilação rápida. A máquina virtual Java HotSpot *servidora* é proposta para um máximo ganho de desempenho na velocidade de execução de aplicações longas. Ambas compartilham o mesmo ambiente de execução, porém usam diferentes compiladores JIT, chamados: compilador *cliente* e compilador *servidor*.

O compilador *cliente* [80] possui uma velocidade de compilação significativamente mais alta por não aplicar otimizações que consumam tempos longos. Por outro lado, o compilador *servidor* [88] é proposto para aplicações que possuem um longo tempo de execução onde o tempo de inicialização pode ser negligenciado e apenas o tempo de execução é relevante.

Como tal a estrutura interna do compilador *cliente* é muito mais simples do que a do compilador *servidor*. Ela é organizada como um *frontend* independente de máquina e um *backend* dependente de máquina. A estrutura do compilador *cliente* é apresentada na figura 3.14.

O compilador *cliente* utiliza um processo de compilação composto de cinco fases, são elas:

1. *O tradutor HIR* constrói uma representação intermediária de alto nível a partir dos *bytecodes*.

2. *O otimizador* aplica apenas otimizações simples como *constant folding*. Nesta fase, os laços mais internos são detectados para facilitar a alocação de registradores.

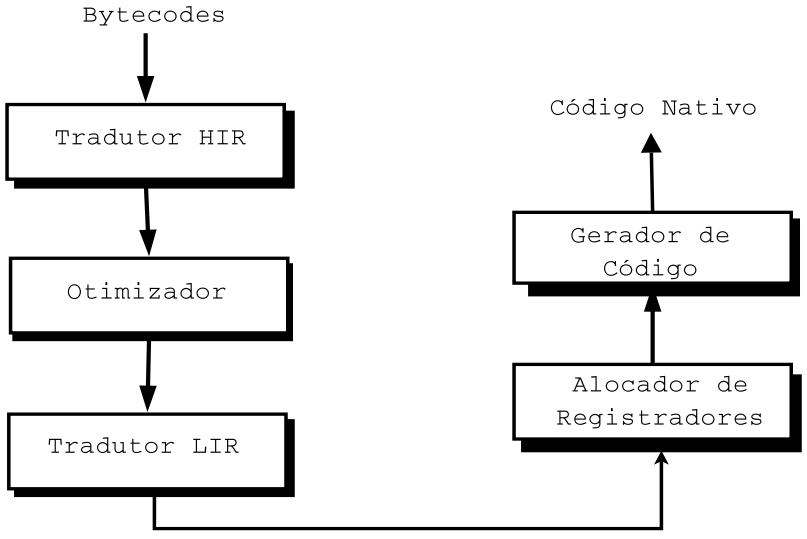


Figura 3.14: Estrutura do compilador JIT *cliente* da Sun.

3. *O tradutor LIR* converte a representação de alto nível em uma representação de baixo nível similar ao código da máquina alvo.

4. *O alocador de registradores* realiza a alocação de registradores. A heurística usada para a alocação de registradores assume que todas as variáveis locais estão armazenadas na pilha. Registradores são alocados quando necessário e liberados quando o valor é armazenado em uma variável local. Se um registrador fica completamente sem uso dentro de um laço ou na entrada de um método, então este registrador é usado para armazenar a variável local usada freqüentemente. Esta abordagem reduz o número de leituras e escritas na memória, especialmente em arquiteturas com poucos registradores.

5. *O gerador de código* gera código nativo para o método.

O compilador *servidor* [88] é um compilador otimizador completo que realiza várias otimizações clássicas tais como: eliminação de expressões comuns, *loop unrolling* e alocação de registradores baseados em coloração de grafos. Além de aplicar otimizações específicas a Java, tais como: integração de métodos virtuais [40], eliminação de checagem nula [63] e eliminação de checagem da faixa em *arrays* [52]. Estas otimizações reduzem o *overhead* necessário para garantir a semântica segura da linguagem Java.

Espera-se que estas otimizações gerem um código de alta qualidade e com baixo tempo de execução. Porém, estas otimizações consomem um elevado tempo de compilação, tendo-se uma baixa velocidade de compilação comparada com o compilador *cliente*

ou outros compiladores JIT. Segundo os autores, o compilador *servidor* é a melhor escolha para aplicações Java que possuem um longo tempo de execução pois o tempo inicial necessário para a compilação pode ser negligenciado: apenas o tempo de execução do código gerado é relevante.

O compilador *servidor* utiliza uma representação intermediária baseada em um grafo na forma *static single assignment* [8, 36, 9]. Operações são representadas através de nodos, os operandos de entrada são representados por arestas para os nodos que produzem os valores de entrada (arestas de fluxo de dados). O fluxo de controle é representado por arestas explícitas que não precisam necessariamente ser as mesmas arestas de fluxo de dados. Isto permite otimizações de fluxo de dados alterando a ordem dos nodos sem destruir o correto fluxo de controle.

A estutura do compilador é apresentada na figura 3.15.

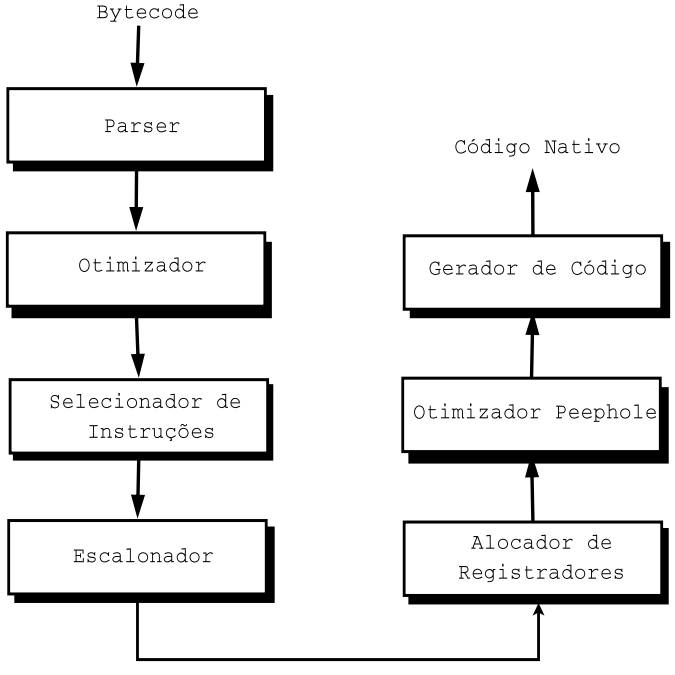


Figura 3.15: Estrutura do Compilador Server.

O compilador possui portanto as seguintes fases:

1. *O parser* necessita de duas iterações sobre os *bytecodes*. A primeira iteração identifica os blocos básicos, onde um bloco básico é uma seqüência de *bytecodes* que não possui um salto em seu corpo. A segunda iteração visita todos os blocos básicos e traduz os *bytecodes* do bloco para nodos da representação SSA. Devido aos nodos de instrução serem também conectados por arestas de fluxo de controle, a estrutura

explícita de blocos básicos é revelada. Isto permite uma posterior reordenação dos nodos de instruções.

2. *O otimizador* aplica otimizações independentes de máquina. Otimizações como avaliação de expressões constantes e numeração de valores são aplicadas durante a fase inicial. Laços não podem ser otimizados completamente durante esta fase devido ao fim do laço não ser ainda conhecido quando o seu início é processado. Como tal, as otimizações descritas anteriormente juntamente com otimizações globais incluindo: propagação de constantes, *loop unrolling* e eliminação de saltos [84] são em seguida reexecutadas até alcançar um ponto fixo, onde nenhuma otimização futura seja possível. Este processo pode requerer várias iterações sobre todos os blocos básicos, e consumir um tempo significante.

3. *O selecionador de instruções* seleciona as instruções da arquitetura alvo que representaram o código do método. A tradução de instruções independentes de máquina para instruções da arquitetura destino é realizada através de um sistema de reescrita de baixo para cima [90, 56]. Este sistema usa uma descrição da arquitetura destino para auxiliar na estimativa do custo de cada instrução. Quando o custo estimado de cada instrução de máquina é conhecido, torna-se possível selecionar a melhor instrução de máquina.

4. *O escalonador* escalona o código. A ordem final das instruções é calculada antes da fase de alocação de registradores. Instruções ligadas pelas arestas de fluxo de controle são agrupadas em blocos básicos novamente. Cada bloco tem uma freqüência de execução associada que é estimada a partir da profundidade de um laço e por predição de saltos. Dentro de um bloco básico, as instruções são ordenadas por um escalonador local.

5. *O alocador de registradores* realiza a alocação de registradores. O alocador global de registradores utiliza coloração de grafos. Primeiro, as faixas vivas são coletadas e conservativamente agrupados, após isto cores são atribuídas aos nodos. Se a coloração falha, código para manter dados na memória é inserido e o algoritmo é repetido.

6. *O otimizador peephole* otimiza seqüências de código específicas de arquitetura. Esta fase também insere dados adicionais necessários para desotimização, coleta de lixo, e gerência de exceções.

7. *O gerador de código* finalmente, gera código executável e o instalado no ambiente de execução.

O capítulo 4 apresenta um estudo sobre o impacto dos compiladores Java *Just-In-Time* da Sun.

3.3.4 O Compilador da IBM

A estrutura básica do compilador Java *Just-In-Time* da IBM [112] é similar aquelas usadas por ambientes estáticos. As fases de compilação são mostradas na figura 3.16.

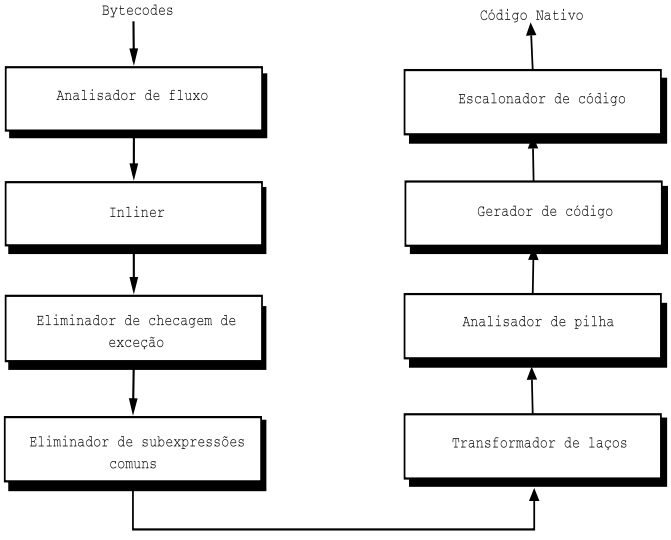


Figura 3.16: Estrutura do compilador JIT da IBM.

O compilador possui portanto as seguintes fases:

1. *O analisador de fluxo* constrói um conjunto de blocos básicos. Após os blocos básicos serem gerados, é criada uma representação intermediária chamada de *extended bytecode*. Esta representação pode possuir alguns novos *opcodes*, que são introduzidos para representar operações após otimizações. Um exemplo de novos *opcodes*, que são produzidos pela eliminação de expressões comuns, é um para obter o ponteiro de um *array* para um endereço efetivo comum. Segundo os autores, isto permite que consecutivos elementos seajam acessados através de sucessivas instruções *load-store* sem indexar cada elemento.

2. *O inliner* aplica a integração de métodos apenas aos métodos freqüentes para reduzir o *overhead* de invocação e evitar a explosão no crescimento do código fonte,

que pode acarretar uma degradação do desempenho devido a saturação da cache de instruções.

3. *O eliminador de checagem de exceção* elimina as checagens de exceções. Para eliminar a checagem de ponteiros nulos o compilador utiliza um grafo de fluxo de dados. A técnica usada é bem simples. Toda vez que um objeto é testado como ponteiro nulo, uma informação indicando que este objeto já foi testado é propagada pelo grafo. O compilador utiliza uma extensão do algoritmo de Gupta [52] para eliminar a checagem de faixa em *arrays*. O método de Gupta basicamente propaga o conjunto de índices checados para frente e para trás, utilizando uma análise de fluxo de dados.

4. *O eliminador de subexpressões comuns* elimina as subexpressões redundantes. Três técnicas são aplicadas para eliminar subexpressões comuns e reduzir o *overhead* dos acessos a *arrays* e instâncias de variáveis, a saber: *substituição de escalares, geração de endereços efetivos e eliminação de redundância parcial*. Técnicas como numeração de valores, que são comumente aplicadas sobre formas SSA, não são aplicadas neste caso por terem sido consideradas muito caras. Substituição de escalares troca as variáveis indexadas por referências de variáveis locais. Geração de endereços efetivos produz uma instrução apontando para o elemento interno do *array* para referências consecutivas. Este método reduz a pressão sobre registradores. Em ambas as técnicas, o código pode ser movido para fora do laço se ele é invariante. Para minimizar o custo desta otimização, o compilador a aplica apenas para laços. A eliminação de redundância parcial elimina todos acessos de variáveis que sejam idênticos em algum caminho de execução, e também move acessos invariantes para fora de laços.

5. *O transformador de laços* otimiza os laços contidos no programa. Para aplicar versões de laço o compilador utiliza os seguintes critérios: (1) não existe invocação de método com o laço, (2) existe uma variável *loop induction* cujos valores inicial e final são invariantes e cujo passo é constante e (3) para todos acessos de *arrays* em um laço, o *array* deve ser uma variável que é laço-invariante, e os índices do *array* devem ser constantes, variáveis laço-invariante ou variáveis *loop induction* com constante deslocamentos. Aplicação desta otimização é também limitada pelo tamanho do corpo do laço alvo, em vista desta otimização poder aumentar o tamanho do código.

6. *O analisador de pilha* analisa a pilha de execução do ambiente. O compilador provê uma tabela com seqüências de *bytecodes* freqüentes, para aplicar idiomas de máquina e desta forma atenuar a ineficiência na geração de código causada pela semântica de pilha.

7. *O gerador de código nativo* gera código nativo. Juntamente com a geração de código nativo, ocorre a alocação de registradores e a aplicação da otimização idiomas de máquina. Algoritmos de alocação de registradores como coloração de grafos são considerados caros e inapropriados para compiladores JIT. Ao invés disto, o compilador JIT IBM utiliza um algoritmo rápido para alocar registradores que não requer uma fase extra no processo de compilação. O alocador de registradores, melhor chamado de *gerente de registradores*, aloca registradores para variáveis de pilha como uma primeira prioridade e então aloca os registradores restantes para variáveis locais baseada na contagem de uso em cada instrução. Durante a geração de código, diversos modos de endereçamento de operandos providos pela arquitetura IA32 são explorados para ganho de desempenho.

8. *O escalonador de código* escalona o código nativo. O código nativo final gerado consiste de instruções que foram escalonadas pelo escalonador de código que trabalha de forma síncrona com o gerador de código em faixas de blocos básicos, explorando as características da arquitetura alvo para tentar obter um melhor desempenho durante a execução do programa.

Em uma avaliação deste sistema [112], feita pelos autores, os resultados obtidos para três pacotes de *benchmarks* mostram que a máquina virtual Java da IBM, juntamente com seu compilador JIT, é um ambiente que possui um bom desempenho para plataformas Intel.

3.4 Considerações Gerais

Para melhorar o desempenho de linguagens orientadas a objetos, como: Self e Java, duas soluções são propostas:

1. ***um modelo de compilação estática***, e

2. ***um modelo de compilação Just-In-Time***.

O modelo de compilação estática traduz código fonte em código nativo antes de iniciar a execução do programa. Neste modelo, o *overhead* de compilação pode ser ignorado. Portando, este pode utilizar otimizações agressivas. Por outro lado, este modelo não suporta carga dinâmica, e não possui vantagens em linguagens, como por exemplo Java, que suportam flexibilidade e reusabilidade de um programa. O compilador JIT traduz código fonte em código nativo quando o código é invocado durante a execução. Isto permite carga dinâmica. Contudo, o tempo total de execução do programa inclui o tempo de compilação, e conseqüentemente o compilador JIT deve ser mais eficiente do que o compilador estático.

Desde que o *overhead* de compilação de um compilador dinâmico, em contraste com um compilador estático, é incluído no tempo de execução do programa, o compilador dinâmico precisa ser muito seletivo sobre *quando*, *como* e principalmente *o que* compilar. Mais especificamente, ele deveria apenas compilar partes do programa se o tempo gasto na compilação puder ser amortizado pelo ganho de desempenho do código compilado. Uma vez que regiões freqüentes são detectadas, o compilador dinâmico pode ser agressivo em identificar boas oportunidades para otimizações que podem alcançar um ótimo desempenho. A escolha entre o *overhead* de compilação e o benefício no desempenho é uma questão crucial para compiladores dinâmicos.

Os sistemas dinâmicos desenvolvidos são classificados em duas categorias:

1. *apenas compilador*: Self e JUDO apenas compilam.

2. *modo misto de execução*: Sun HotSpot Compilers e IBM JIT Compiler provêem um interpretador para permitir um ambiente com modo misto de execução com código interpretado e compilado.

O sistema Self foi pioneiro em um sistema de compilação baseado em *on-line profiling*. O objetivo deste sistema é evitar as longas pausas de compilação e melhorar a sensibilidade para programas interativos. Ele é baseado em uma abordagem de utilizar apenas compilador, e para recompilar métodos ele utiliza contadores para indicar os candidados a compilação. O sistema de recompilação possui a vantagem de utilizar informações sobre o tipo dos dados para obter um melhor desempenho dos código recompilados.

O compilador JUDO emprega otimizações dinâmicas através de recompilação, provendo dois diferentes compiladores: um gerador rápido de código e um compilador otimizador. Para acionar a recompilação, JUDO insere contadores na entrada de cada método e no início de cada laço no primeiro nível de código compilado. Isto acarreta um

overhead ao desempenho. O código nativo precisa ser recompilado para remover este *overhead*. Devido ao código recompilado não ser instrumentado, futuras otimizações não são possíveis neste sistema.

A noção de um modo misto de execução foi considerada como um *compilador contínuo* ou *smart JIT* em [92], e o estudo de um modelo de execução com três modos usando um interpretador, um compilador rápido não otimizador e um compilador otimizador foi descrito em [3]. Estes trabalhos mostram a vantagem de se utilizar um modo misto de execução para balancear entre o custo da compilação e a qualidade do código gerado.

Os ambientes da Sun e o da IBM implementam um sistema de otimizações adaptativas. Eles executam o programa usando um interpretador, e detectam as partes freqüentes durante a execução. Os ambientes monitoram o programa continuamente de forma que o sistema pode otimizar partes do programa, com o objetivo de obter um melhor desempenho.

Após entender as principais questões sobre compilação *Just-In-Time*, o próximo passo é analisar o ganho de desempenho na utilização de um compilador dinâmico, como também na aplicação de otimizações. Este é o objetivo do próximo capítulo.

Foi escolhido analisar um ambiente com modo misto de execução, por esta abordagem parecer mais apropriada para Prolog. Além disto, foi decidido estudar a linguagem de programação Java por ser uma linguagem com mais trabalhos recentes.

Capítulo 4

Avaliação de *Just-In-Time* para Java

A linguagen Java [10] é o mais popular exemplo da tecnologia JIT em ação. A linguagem de programação Java foi desenvolvida pela Sun Microsystems [80] como uma linguagem orientada a objetos para uso de propósito geral. Java foi projetada como uma linguagem portável que roda em diferentes plataformas.

Para garantir portabilidade e independência de plataforma aplicações Java não são distribuídas em código nativo. Ao invés disso, ambientes Java possuem o conceito de uma máquina virtual Java. Código fonte Java é compilado para uma representação binária compacta chamada Java *bytecodes* que é ou interpretado por uma máquina virtual ou compilado. O programa fonte é portanto armazenado em um formato binário bem conhecido, o formato *class file*, contendo os *bytecodes* juntamente com uma tabela de símbolos e outras informações auxiliares. A máquina virtual Java é definida de forma independente da linguagem de programação Java, apenas o formato *class file* conecta os dois.

O modelo de execução padrão de Java tem problemas de desempenho. Interpretar um método Java pode ser muito lento devido a cada *bytecode* requerer executar pelo menos uma *template* consistindo de diversas intruções de máquina, limitando consideravelmente o desempenho. Para se obter um melhor desempenho é necessário compilar *bytecodes* para código de máquina. Compiladores *Java Just-In-Time* vão um passo à frente, realizando a compilação em tempo de execução e apenas para os métodos executados com mais freqüência. O preço a pagar é que o tempo total de execução agora sofre o *overhead* de compilação.

Uma questão importante é entender qual o impacto e o benefício de um compilador JIT ao sistema. Este capítulo enfoca esta questão. Primeiramente a máquina virtual Java é descrita. Em seguida é analisado o desempenho dos compiladores JIT da Sun. Além disto, foram estudadas algumas otimizações aplicadas por estes compiladores. O estudo

foi feito em um número de otimizações bem conhecidas, a saber:

- *integração de procedimentos*: é particularmente importante para linguagens orientadas a objetos: ela reduz o *overhead* da invocação de métodos, enquanto aumenta a oportunidade de aplicar outras otimizações;
- *numeração de valores*: troca computações redundantes por cópias;
- *eliminação de expressões condicionais*: simultaneamente elimina expressões e remove código morto;
- *eliminação de checagem de array*: permite ao compilador remover checagem em situações onde ele pode determinar que o índice é válido para o *array*;
- *coalescing*: remove cópias entre registradores; e
- *peepholing*: examina uma seqüência de instruções com o objetivo de trocá-las por uma seqüência mais eficiente.

Também foi estudado o impacto de usar *on-stack-replacement* [47], que permite mover do interpretador para o código compilado durante a execução do método.

4.1 A Máquina Virtual Java

A máquina virtual Java é um computador abstrato [72], capaz de carregar classes e executar os *bytecodes*, que são instruções codificadas no formato da máquina virtual Java, nelas contidos. Ela é composta por três elementos:

1. *um carregador de classe, que carrega as classes da API Java e as do programa a ser executado*;
2. *uma heap, a região de dados que armazena as classes*; e
3. *um motor de execução (emulador) responsável por interpretar os bytecodes que implementam os métodos das classes*.

A figura 4.1 apresenta os componentes da máquina virtual Java. A *heap* é uma área de dados na qual todas as instâncias de classes e *arrays* são armazenados. A *heap* é criada durante a iniciação da máquina virtual Java. O armazenamento de dados na *heap* é gerenciado por um *coletor de lixo*, pois objetos Java não são desalocados explicitamente

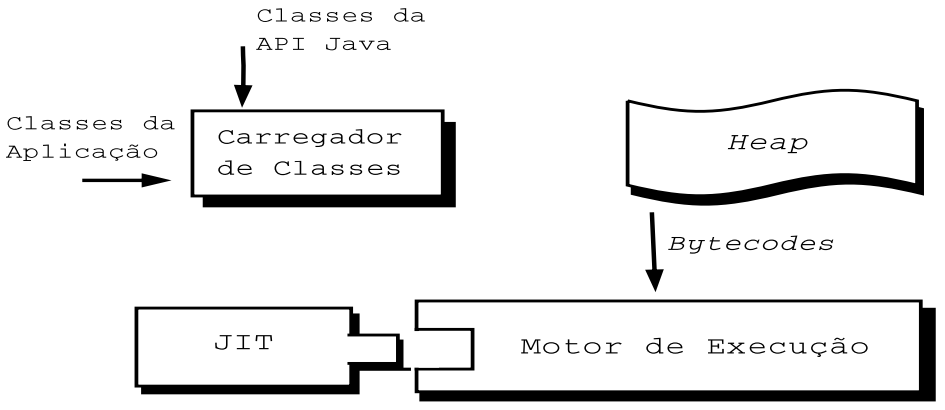


Figura 4.1: Arquitetura da Máquina Virtual Java.

[120]. A *heap* pode ter um tamanho fixo, ou pode expandir caso o programa crie vários objetos e contrair quando objetos não são mais referenciados.

O motor de execução suporta a execução de instruções da máquina virtual. Um motor de execução simples apenas interpreta os *bytecodes*, um por um. O desempenho pode ser melhorado usando um compilador JIT. Nesta solução, os *bytecodes* do método são compilados para código nativo durante a primeira invocação do método e armazenados em uma *cache* para um possível reuso, caso este método seja novamente invocado. Ou ainda, a máquina virtual inicia interpretando *bytecodes*, porém o programa em execução é monitorado para detecção das áreas de código executadas com freqüência, as chamadas *hot-spots*. Durante a execução do programa, a máquina virtual Java gera código nativo destas áreas e continua interpretando os outros *bytecodes*.

Uma máquina virtual Java possui dois tipos de métodos:

1. *métodos Java*; e

2. *métodos nativos*.

Um método Java é escrito em linguagem Java, compilado para *bytecodes* e armazenado em arquivos de classes. Um método nativo é escrito em uma outra linguagem, tal como `C`, e compilado para código de máquina nativo de um particular processador. Métodos nativos são armazenados em bibliotecas dinâmicas. Quando um programa Java invoca um método nativo, a máquina virtual carrega a biblioteca dinâmica que contém a implementação do método e o invoca.

4.1.1 O Carregador de Classes

A máquina virtual Java possui uma arquitetura flexível para carregadores de classes, permitindo a um programa Java carregar dinamicamente classes tanto do disco local, como da *Internet*.

O carregador de classes [71] é responsável não apenas por localizar e importar os dados binários das classes. Ele também tem as funções de verificar a corretude dos dados importados, alocar e inicializar memória para as variáveis das classes e resolver as referências simbólicas. Estas atividades são realizadas na seguinte ordem:

1. *Carregar*: encontrar e importar os dados binários para uma classe.
2. *Ligar*: executar a verificação, a preparação, e opcionalmente a definição.
   (a) *Verificar*: assegurar a exatidão da classe importada.
   (b) *Preparar*: alocar memória para variáveis da classe e iniciar a memória alocada com o valor padrão zero (0).
   (c) *Definir*: tranformar referências simbólicas em referências diretas.
3. *Iniciar*: invocar o código Java que inicia as variáveis da classe com seus valores apropriados.

Um programa Java pode fazer uso de diferentes carregadores de classes. Existe um carregador padrão, que é um componente da implementação da máquina Java, e podem existir carregadores definidos pelo usuário, capaz de carregar classes por meios não convencionais. Enquanto o carregador padrão é parte da implementação da máquina Java, o carregador definido pelo usuário é uma classe Java, inicialmente carregada pela máquina virtual Java e criado como qualquer outro objeto. Veja figura 4.2.

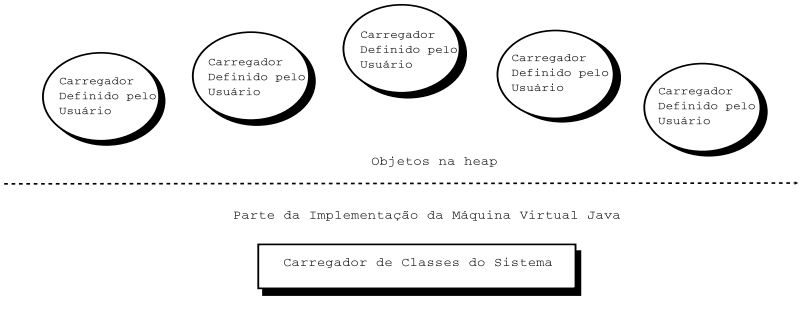


Figura 4.2: Arquitetura do Carregador de Classes.

Para cada classe carregada, a máquina virtual mantém um histórico contendo qual carregador carregou a classe. Quando uma classe faz referência a uma classe não carregada, a máquina virtual Java utiliza para a carga da classe referenciada o carregador da classe que contém a referência. O uso deste mecanismo presupõe que uma classe apenas referencie classes carregadas pelo mesmo carregador.

4.1.2 A *Heap*

Em uma instância da máquina virtual Java, informações sobre os tipos carregados são armazenadas em uma área lógica da memória denominada área de métodos [120]. A área de métodos é compartilhada por todas as *threads* do programa. Porém, quando duas *threads* tentam encontrar uma classe que ainda não foi carregada, apenas uma carrega a classe, enquanto a outra fica bloqueada em espera.

Quando um programa Java em execução cria uma nova instância de uma classe, a memória para este novo objeto é alocada em uma *heap*. A máquina Java possui apenas uma *heap*, que é compartilhada por todas as *threads* do programa. Note que *threads* podem portanto acessar objetos pertencentes a outra *thread*. Java fornece primitivas de sincronização, tais como *wait*, *notify* e *notifyall* para que o programador possa evitar condições de corrida.

A máquina virtual Java possui uma instrução que aloca memória na *heap* para um novo objeto, porém não possui uma instrução para liberar memória. A liberação das áreas de memória ocupadas por objetos que não são referenciados pelo programa é de responsabilidade do coletor de lixo [9] da máquina virtual Java.

O coletor de lixo é um componente fundamental da máquina virtual Java responsável por gerenciar a *heap* e a área de métodos. O coletor de lixo, além de liberar as áreas utilizadas pelos objetos e classes que não estão sendo referenciados, possui a capacidade de realocar classes e suas instâncias para reduzir a fragmentação da área de métodos e/ou da *heap*.

Em geral, o tamanho das áreas de métodos e da *heap* não são fixos. Durante a execução de um programa Java, estas podem ser expandidas ou contraídas pela máquina virtual Java.

4.1.3 O Motor de Execução

O núcleo da máquina virtual Java é seu motor de execução [120], cujo comportamento é definido por um conjunto de instruções.

Cada *thread* do programa é uma instância do motor de execução. Durante o período de vida da *thread*, ela está executando *bytecodes* ou métodos nativos. A *thread* pode executar *bytecodes* diretamente, interpretando, ou indiretamente, executando o código nativo resultante da compilação.

A máquina virtual Java pode usar *threads* que não são visíveis ao programa, por exemplo, a *thread* que executa a coleta de lixo. Tais *threads* não precisam ser instâncias do motor de execução. Entretanto, as *threads* que pertencem ao programa são motores de execução em ação.

O motor de execução funciona executando *bytecodes* de uma instrução por vez. Este processo ocorre para cada *thread* do programa.

Uma seqüência de *bytecodes* é uma seqüência de instruções. Cada instrução consiste de um código de operação seguido por zero ou mais operandos. O código de operação indica a operação a ser executada. Os operandos fornecem os dados necessários para executar a operação especificada. No próprio código de operação é implícito a existência ou não de operandos e dos mecanismos para acessá-los. Muitas instruções não fazem exame de nenhum operando e consistem somente em um código de operação.

O motor de execução busca um código de operação e se esse código possuir operandos, busca os operandos. O emulador executa a ação solicitada pelo código e em seguida busca um outro código. A execução dos *bytecodes* continua até que uma *thread* termine retornando de seu método inicial.

Cada tipo de código do conjunto de instruções da máquina virtual Java possui um *minemônio*, no estilo típico de linguagem *assembly*.

4.2 Ambiente Experimental

Para estudar o desempenho do compiladores JIT da Sun foram utilizadas dezoito programas Java, nove dos quais (*Crypt, FFT, LU, Heap Sort, Sparse, Euler, MolDyn, Monte Carlo e Raytracer*) fazem parte do conjunto de programas *Java Grande Forum* [60]. Os outros nove (*Antlr, Batik, Bloat, Chart, Fop, Hsqldb, Jython, Pmd e Xalan*) são programas do *DaCapo Suite* [16, 17, 38].

O *benchmark Java Grande Forum* é um conjunto de testes desenvolvido com o propósito de medir e comparar ambientes alternativos de execução Java. Os programas contidos neste *benchmark* requerem uma larga quantidade de processamento ou acesso à memória em larga escala. Este *benchmark* inclui programas científicos e simulações financeiras.

Os programas deste *benchmark* utilizados nesta seção representam tanto códigos simples, que podem ser encontrados em grandes programas, como programas completos. Estes programas não possuem componentes gráficos e não efetuam processamento de entrada e saída.

O *benchmark DaCapo* possui o mesmo propósito do *benchmark Java Grande Forum*. Contudo, o *benchmark DaCapo* consiste de um conjunto de programas reais, conhecidos por gerarem acessos irregulares à memória. Além disto, o *benchmark DaCapo* também se difere do *benchmark Java Grande Forum* por possuir componentes gráficos e processamento de entrada e saída.

Todas as medidas foram obtidas em um Pentium(R) Intel(R) 4 CPU 2.80 GHz, com 1GB de memória, rodando Linux e usando a máquina virtual da Sun 1.5.0. Cada programa foi executado cinco vezes, e a partir destes dados foi calculado a média aritmética. O desvio padrão dos dados obtidos foi inferior a 0,07%.

Para um estudo detalhado os programas foram instrumentados através do instrumentador da máquina virtual Java. Além disto foi utilizada a biblioteca *Program Counter Library* [94] para coletar as seguintes informações de hardware:

- *quantidade de instruções de hardware*;
- *acessos à memória*;
- *falhas na cache*; e
- *instruções por ciclo (IPC)*

A descrição de cada programa utilizado no estudo dos compiladores JIT é a seguinte:

- ***crypt***: criptografar 50.000.000 bytes.
- ***fft***: transformação de *fourier* de uma dimensão em 16M números complexos.
- ***lu***: fatorar um sistema 2.000×2.000.
- ***heap sort***: ordenar de um vetor de 25.000.000 elementos inteiros utilizando o método *heap sort*.
- ***sparse***: multiplicar duas matrizes esparsas com 500.000×500.000 elementos.
- ***euler***: dinâmica computacional de fluídos em um malha de tamanho 96×384.

- ***moldyn***: simulação de dinâmica molecular em 8788 partículas.
- ***monte carlo***: simulação monte carlo em uma séria de 60.000 vezes.
- ***raytracer***: renderizar uma figura com 500×500 pixels.
- ***antlr***: gerar um analisador sintático e um analisador léxico para 320 gramáticas.
- ***batik***: renderizar 6 arquivos SVG.
- ***bloat***: otimizar e analisar 82 programas Java.
- ***chart***: desenhar 52 gráficos complexos e renderizar cada um como um arquivo PDF.
- ***fop***: gerar um arquivo PDF a partir de um arquivo XSL-FO.
- ***hsqldb***: executar operações *JDBC* em memória, contendo 20 clientes com 5 transações por cliente.
- ***jython***: interpretar 4 programas Python.
- ***pmd***: analisar as classes Java contidas em 17 programas.
- ***xalan***: transformar um documento XML em um documento HTML.

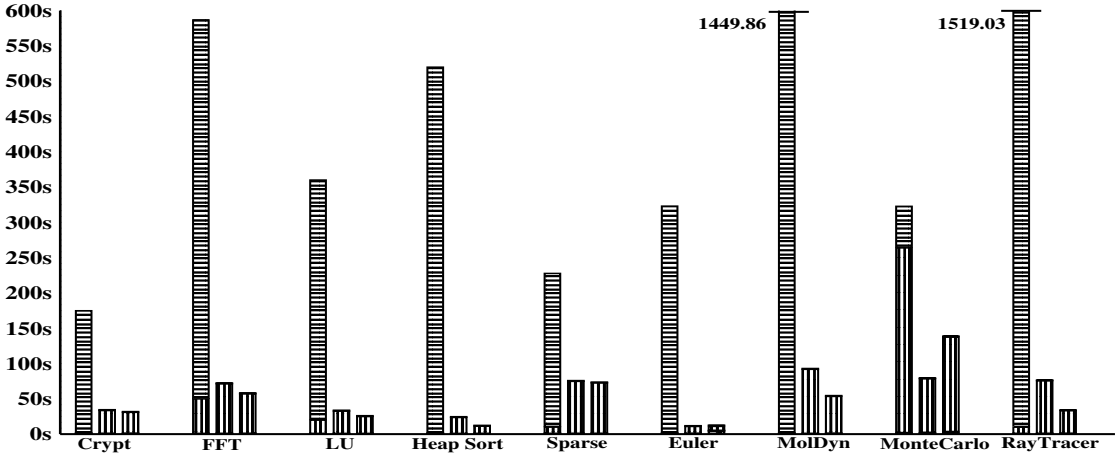
4.3 Análise da Compilação Dinâmica

A figura 4.3 apresenta o tempo de execução dos programas. Esta figura possui três colunas por programa:

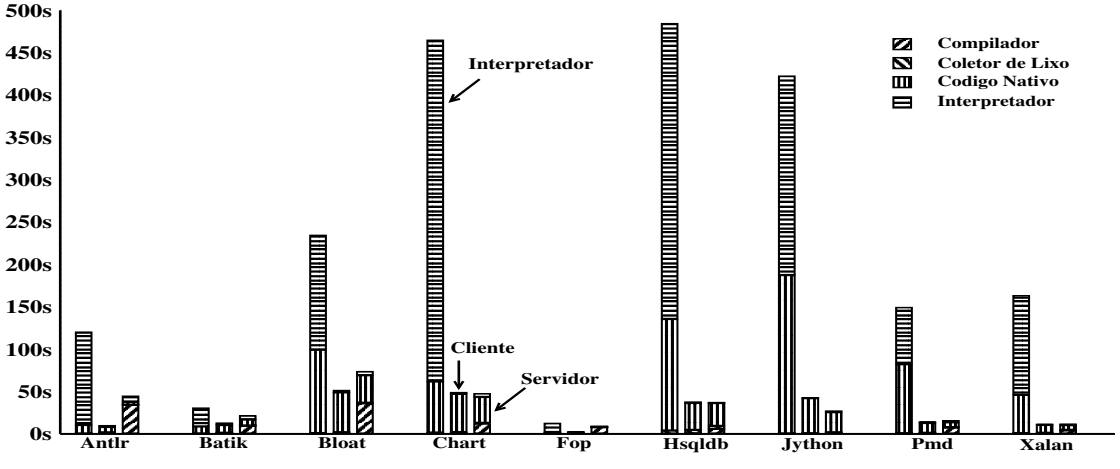
1. *a da esquerda representa a execução utilizando apenas o interpretador*;
2. *a central utiliza o compilador cliente (descrito na seção 3.3.3)*; e
3. *a da direita o compilador servidor (também descrito na seção 3.3.3)*.

Para uma análise detalhada, o tempo de execução foi decomposto em quatro componentes:

1. *tempo gasto pelo compilador*;
2. *tempo gasto pelo coletor de lixo*;



(a) Programas Científicos



(b) Programas Não-Científicos

Figura 4.3: Tempo de execução dos programas.

3. *tempo gasto executando código nativo*; e

4. *tempo gasto pelo interpretador*.

Como esperado, um ambiente de compilação dinâmica é bem mais rápido do que um ambiente apenas interpretado, veja figura 4.3. O interpretador da Sun é entre 2,23 e 48,11 vezes mais lento do que um ambiente que utiliza compilação dinâmica. A maior diferença ocorre para os programas científicos, tais como: *LU*, *Euler*, *MolDyn* e *RayTracer*.

O baixo desempenho do interpretador é devido principalmente ao *overhead* da busca e decodificação de cada *bytecode*. Neste caso, a quantidade de instruções na CPU chega a crescer em uma ordem de grandeza [106]. Além disto, o interpretador requer mais

acesso à memória, aumentando a quantidade de acessos de leitura e escrita na memória. Conseqüentemente, a quantidade de erros na *cache* chega aumentar em duas ordens de grandeza.

Ao se comparar a aceleração obtida durante uma execução utilizando compilação dinâmica (tabela 4.1) com o tempo de execução do programa, pode-se perceber que a aceleração é limitada por três fatores:

- *acessos à memória*;
- *percentual de tempo gasto em bibliotecas nativas*; e
- *operações de entrada e saída*.

Benchmark	**Aceleração**		***Benchmark***	**Aceleração**	
Java Grande Forum	**Cliente**	**Servidor**	***DaCapo***	**Cliente**	**Servidor**
Crypt	5,17	5,63	*Antlr*	11,06	2,65
FFT	8,20	10,23	*Batik*	2,23	1,42
LU	21,78	44,68	*Bloat*	4,47	3,18
Heap Sort	10,98	14,32	*Chart*	9,37	9,72
Sparse	3,04	3,12	*Fop*	4,44	1,55
Euler	28,40	25,29	*Hsqldb*	12,55	13,01
MolDyn	15,74	26,88	*Jython*	9,86	15,64
MonteCarlo	4,08	2,31	*Pmd*	10,34	9,47
RayTracer	21,42	48,11	*Xalan*	14,35	14,26

Tabela 4.1: Aceleração ao se utilizar um Compilador JIT.

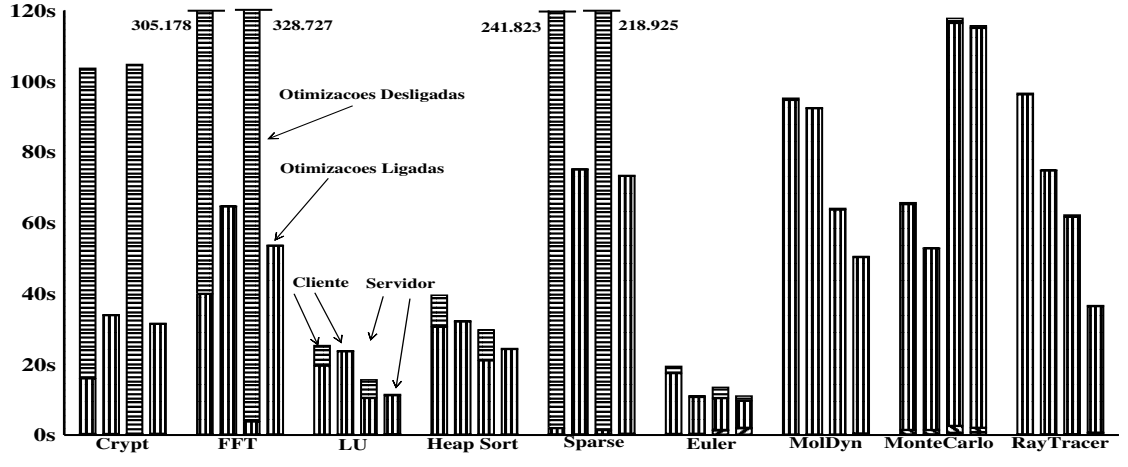
Os programas do *Java Grande Forum* obtiveram uma maior aceleração devido ao fato de não possuirem componentes gráficos e operações de entrada e saída. Além disto, estes programas utilizam uma pequena quantidade de chamadas a bibliotecas nativas, o que aumenta o potencial de otimização. Pode-se observar que o programa que obteve a menor aceleração, *MonteCarlo*, é aquele que mais possui chamadas a bibliotecas nativas.

Outros tipos de programas podem utilizar bibliotecas nativas e realizar operações de entrada e saída, limitando o processo de otimização. Observe que em geral, a faixa de aceleração obtida pelo pacote *DaCapo* é menor do que aquela obtida pelo *Java Grande Forum*.

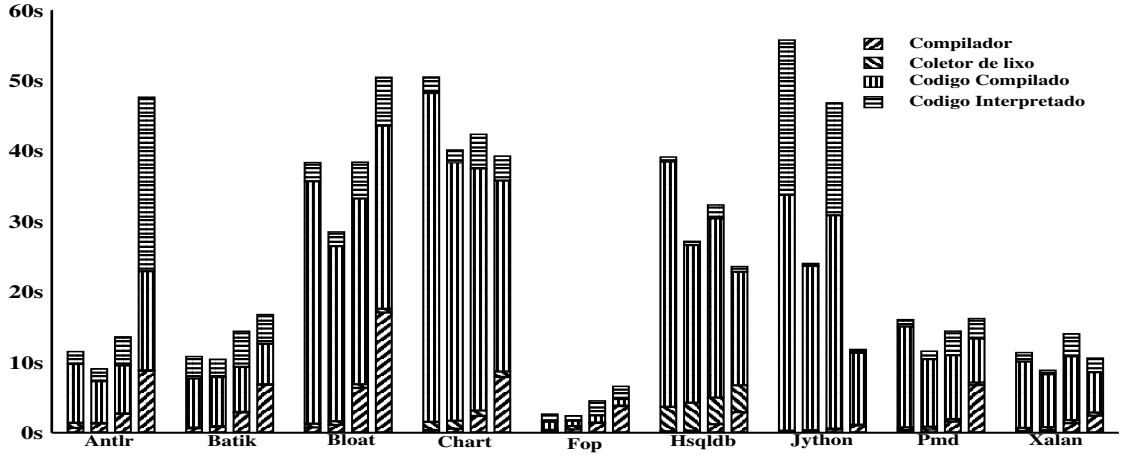
O compilador *servidor* obteve um melhor desempenho (tabela 4.1) para quase todos os programas científicos, exceto para os programas *Euler* e *MonteCarlo*. Por outro lado, para os programas não científicos o desempenho é equilibrado entre os dois compiladores. A próxima seção esclarece alguns pontos importantes na diferença de desempenho.

4.4 Análise dos Compiladores

A figura 4.4 apresenta o tempo de execução dos programas com e sem um conjunto de otimizações. Esta figura possui quatro colunas por programa: as duas da esquerda representam a execução com o compilador *cliente*, e as duas da direita representam a execução com o compilador *servidor*. A primeira coluna representa o tempo de execução com as otimizações estudadas desligadas, enquanto a segunda coluna (por compilador) representa o tempo de execução com as otimizações estudadas ligadas.



(a) Programas Científicos



(b) Programas Não-Científicos

Figura 4.4: Tempo de execução dos programas.

Os tempos de execução totais variam consideravelmente, entre segundos para *Fop* e minutos no caso de *Euler*. O tempo de execução é geralmente dominado pelo tempo

de execução de código nativo. Exceções ocorrem para alguns programas não científicos quando executados com o compilador *servidor*, por exemplo *Pmd*, *Fop* e *Batik* o tempo é dominado pelo sistema de compilação.

Não existe uma vantagem clara entre os compiladores. Os programas numéricos *Crypt, FFT, Euler* e *Sparse*, como também o programa não científico *Xalan* possuem similar desempenho. No caso dos programas científicos, *Monte Carlo* é o programa que possui a maior variação no tempo de execução. O compilador *servidor* possui o pior desempenho, pois este gera uma grande quantidade de falhas na *cache*. Por outro lado, a maior variação no tempo de execução para os programas não científicos é devido ao alto tempo gasto pelo sistema de compilação. Um fato importante a observar é que o tempo de compilação na maioria dos casos é desprezível para o compilador *cliente*.

As otimizações estudadas têm um significante impacto no desempenho. Exceções são os programas que requerem *on-stack-replacement* para obter um bom desempenho do compilador.

Os resultados dos programas *Java Grande Forum* são bem diferentes daqueles obtidos com os programas *DaCapo*. Primeiro, pode-se observar que o compilador *servidor* geralmente não possui um bom desempenho como o compilador *cliente*. Isto ocorre devido a três razões:

1. *Tempo de compilação alto: de 35% a 70% do tempo total de execução, por exemplo para Bloat e Fop*. Em geral, o tempo de compilação é consideravelmente maior para os programas *DaCapo* do que para os programas *Java Grande Forum*. Programas não cientícos não chegam a consumir 10% do tempo total de execução no processo de compilação. Por outro lado, programas científicos consomem até 70% do tempo total de execução no processo de compilação. Além disto, o tempo de compilação é mais longo para o compilador *servidor*, que no programa *Pmd* chega a ser três vezes mais lento do que o compilador *cliente*.

2. *Os programas DaCapo realizam operações de entrada e saída, e utilizam bibliotecas do sistema*. Como também gastam uma significante quantidade de tempo em métodos nativos. Um exemplo típico é o método ZIP que processa um arquivo compactado. A execução deste método consome aproximadamente um quarto do tempo total de execução do programa *Chart*.

3. *Em programas pequenos, uma maior porcentagem de tempo total pode ser gasta interpretando métodos para o compilador servidor, porque ele leva mais tempo*

para ser acionado do que o compilador cliente. Isto ocorre com *Fop*, que gasta aproximadamente 10% do tempo total de execução executando métodos da biblioteca Java. Este problema deve desaparecer quando aumentar o tempo de execução dos programas.

O tempo gasto na coleta de lixo é desprezível nestes programas, exceto para *Hsqldb*, onde o tempo total é aproxidamente o mesmo para ambas as versões. Os programas *Char*, *Pmd* e *Xalan* são exemplos onde os compiladores possuem desempenho semelhante. *Jython* é o único programa *DaCapo* onde o compilador *servidor* consegue um desempenho melhor.

O tempo de execução dos programas é geralmente melhor para o compilador *servidor* (exceto para *Antlr*, *Bloat*, *Batik* e *Fop*), mas o tempo de compilação é uma importante questão para o compilador *servidor* nestes programas, e pode obter considerável queda de desempenho.

As tabelas 4.2 e 4.3 mostram a comparação de baixo nível entre os compiladores *cliente* e *servidor*. É interessante comparar o número de instruções executadas e o IPC em ambos os compiladores. O compilador *servidor* geralmente executa um menor número de instruções e tem um IPC similar para a maioria dos programas, com os melhores resultados para *LU* e *Jython*. Existem poucas surpresas: *Sparse* executa um número menor de instruções no compilador *cliente*. Por outro lado, *Sparse* também possui o pior IPC para o compilador *cliente*, resultando em desempenho similar. *MonteCarlo* possui o resultado mais estranho: o compilador *servidor* geralmente executa um número menor de instruções, mas tem o pior IPC causado pelas falhas na *cache*. Os programas *DaCapo* mostram o custo da compilação extra: *Antlr* e *Fop* mostram um aumento substancial no número das instruções executadas.

O compilador *servidor* é geralmente eficaz em reduzir o número total de acessos de leitura à memória. *LU* é impressionante: o número de leituras reduz por um fator de cinco. Isto também é verdade para os acessos de escrita, porém por um fator menor. O compilador *servidor* executa muito bem para *FFT*, *MolDyn*, *RayTracer*, e *Jython*. Por outro lado, *Sparse* é uma exceção. Para *Fop* e *Antlr*, o compilador *servidor* realiza um número muito menor de acessos à memória, porém ele executa uma quantidade maior de instruções do que o compilador *cliente*.

Benchmark	**Compilador Cliente**			
	Instruções (10^9)	Acessos à Memória (10^6)	Falhas na *Cache* (10^9)	Instruções por Ciclo
Crypt	81,29	52,52	4,91	0,86
FFT	21,86	9,35	1465,68	0,24
LU	62,14	34,85	1536,04	0,85
Heap Sort	35,39	20,03	122,89	0,89
Sparse	16,50	9,50	2148,55	0,08
Euler	20,91	12,03	388,77	0,67
MolDyn	204,26	65,48	5796,12	0,79
MonteCarlo	57,84	37,24	234,77	0,42
RayTracer	151,20	82,63	438,66	0,72
Antlr	16,23	8,6	92,62	0,95
Batik	17,34	11,45	239,74	0,89
Bloat	35,29	18,95	492,87	0,55
Chart	78,95	52,22	993,51	0,77
Fop	2,68	0,86	19,67	1,24
Hsqldb	46,15	26,4	397,19	0,97
Jython	73,79	33,71	210,26	1,14
Pmd	18,79	10,65	270,18	0,69
Xalan	17,47	10,41	196,61	0,87

Tabela 4.2: Impacto no *hardware* para o compilador *cliente*.

Benchmark	**Compilador Servidor**			
	Instruções (10^9)	Acessos à Memória (10^9)	Falhas na *Cache* (10^9)	Instruções por Ciclo
Crypt	69,82	47,14	6,39	0,80
FFT	11,17	5,03	1482,04	0,15
LU	13,63	8,14	3988,50	0,40
Heap Sort	19,98	6,53	148,21	0,87
Sparse	19,50	13,50	2154,82	0,09
Euler	14,86	9,31	498,74	0,62
MolDyn	109,39	34,82	4721,97	0,78
MonteCarlo	50,93	30,24	427,26	0,16
RayTracer	89,49	42,30	197,80	0,91
Antlr	24,14	4,81	87,27	1,92
Batik	18,23	11,04	228,02	0,95
Bloat	29,26	13,60	445,37	0,53
Chart	60,43	36,82	947,45	0,76
Fop	3,21	0,58	20,60	1,67
Hsqldb	30,00	12,96	252,11	1,07
Jython	29,68	13,45	172,94	1,09
Pmd	16,34	7,04	269,19	0,71
Xalan	15,74	8,02	164,43	0,94

Tabela 4.3: Impacto no *hardware* para o compilador *servidor*.

4.5 Impacto das Otimizações Estudadas

Para entender os resultados anteriores foi estudado, de forma individual, o impacto de algumas otimizações mais importantes, a saber:

- *integração de métodos*;
- *numeração de valores*;
- *eliminação de expressões condicionais*;
- *eliminação de checagem de array*;
- *coalescing*; e
- *peepholing*.

Além destas otimizações, também foi estudado o impacto de *on-stack replacement*.

Os resultados mostrados foram obtidos apenas da diferença no tempo de execução do programa, não do tempo total de execução.

4.5.1 Integração de Métodos

Integração de métodos é a mais importante otimização estudada. A figura 4.5 mostra o impacto de ativar esta otimização. A integração de métodos é muito efetiva para o compilador *cliente*, com aceleração de 35% para *Hsqldb*. Integração de métodos obtem um melhor desempenho para os programas *DaCapo*, pois eles são escritos em um estilo mais orientado a objetos: quase todos programas obtiveram uma aceleração superior a 15%. Os programas *Java Grande Forum* se beneficiaram menos, apenas *MonteCarlo*, *RayTracer* e *Euler* obtiveram aceleração significante.

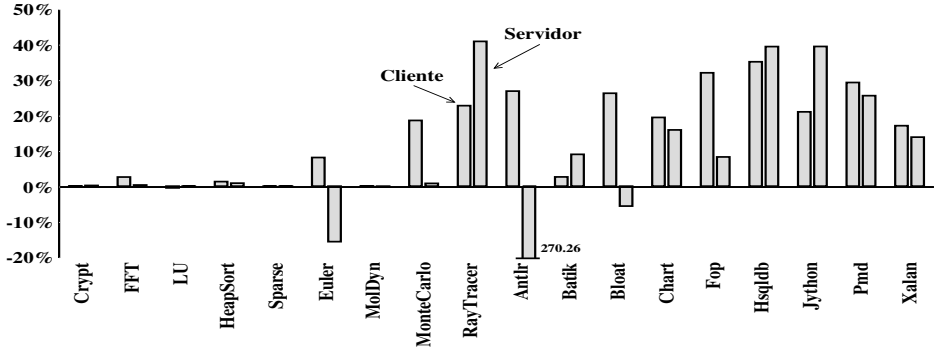


Figura 4.5: Impacto da integração de métodos.

A figura 4.5 mostra uma situação mais complexa para o compilador *servidor*. Enquanto cada programa *DaCapo* se beneficiou no compilador *cliente*, *Antlr* e *Bloat* obtiveram uma queda no desempenho. Por outro lado, *Jython* beneficia 40% contra 20% no compilador *cliente*. Os programas *Java Grande Forum* possuem uma situação similar. *Euler* possui uma significante queda no desempenho, e *MonteCarlo* não possui nenhum benefício. Apenas *RayTracer* obtém benefício da integração de métodos.

O problema do compilador *servidor* é a agressividade em integrar métodos. Isto ocasiona dois problemas:

1. *um aumento no tempo de compilação, que está incluso no tempo total de execução*;
2. *excessiva integração de métodos aumenta os acessos à memória e resultam em uma degradação no desempenho*.

A análise de baixo nível para *Euler* e *MonteCarlo* indicam este ser o caso, como indicado na tabela 4.3.

Integração de métodos é controlada por diversos parâmetros. Foi estudado ajustar o valor de *MaxInlineSize*, que fornece o tamanho máximo do método a ser integrado. Isto

ocasionou um impacto grande para diversos programas. *Euler* obteve aceleração de 15%, *Antlr* 21% e *Bloat* 13%. Infelizmente, este ajuste ocasionou a degradação no desempenho para outros programas, como por exemplo *Batik* onde o desempenho piorou de 9% para 2%. Estes resultados sugerem que integração de métodos deve ser ajustada de acordo com as características de cada programa.

4.5.2 Numeração de Valores

O impacto de usar numeração de valores com o compilador *cliente* é mostrado na figura 4.6. Numeração de valores obtém um melhor impacto para programas numéricos: *Euler*, *FFT* e *HeapSort*. Ela também obtém aceleração para *Fop*. Esta otimização melhora o desempenho em 30% em *Euler*. *Euler* é um programa particularmente propício para este tipo de otimização: *ele realiza computações complexas reusando o mesmo valor em diversos pontos*. Programas não numéricos tendem a se beneficiar pouco com numeração de valores.

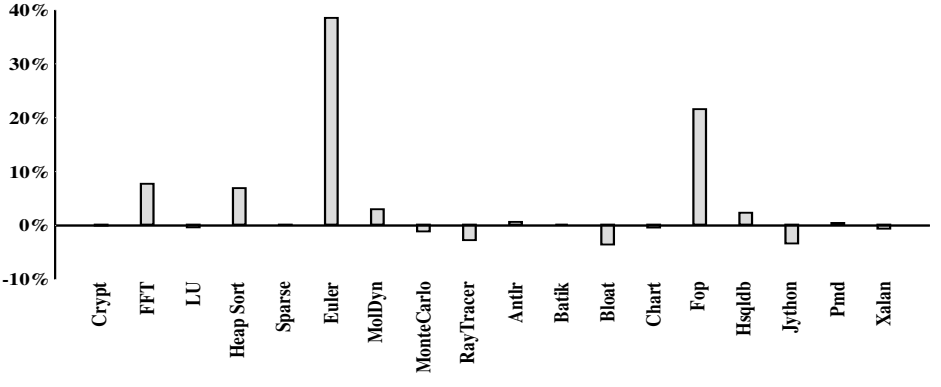


Figura 4.6: Impacto da numeração de valores.

4.5.3 Eliminação de Expressões Condicionais

Eliminação de expressões condicionais somente foi estudada no compilador *cliente*. O impacto é mostrado na figura 4.7. Eliminação de expressões condicionais é surpreendente em *Fop*, onde sozinha obtém uma aceleração de 20%. Ela também beneficia *Xalan* e *Euler*, porém bem menos. É interessante que para *Bloat*, *Hsqldb* e *Jython* esta otimização degrada o desempenho. Em geral, eliminação de expressões condicionais têm pouco efeito no desempenho, apesar de diminuir o número de falhas na *cache*.

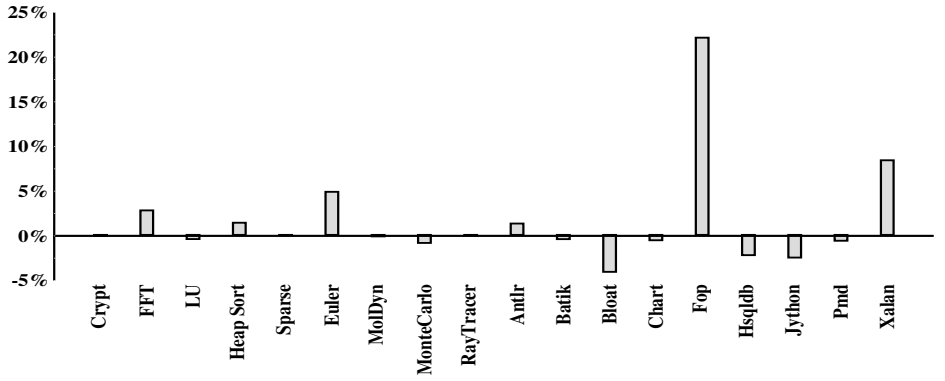


Figura 4.7: Impacto da eliminação de expressão condicional.

4.5.4 Eliminação de Checagem de Faixa

A figura 4.8 mostra o impacto da eliminação de checagem de faixa. Esta otimização obtém pouco impacto na maioria dos programas. Exceções são *Fop* e *FFT* para o compilador *cliente*, e *LU* e *Batik* para o compilador *servidor*. Tanto *FFT* como *LU* se beneficiam da eliminação de checagem de faixa. O compilador *cliente* obtém aceleração para *FFT* mas não para *LU*, e o compilador *servidor* obtém aceleração para *LU* mas não para *FFT*.

A eliminação de checagem de faixa ocasiona uma degradação de desempenho significante no compilador *servidor*, a saber para *Euler*. Isto parece ser causado pela interação com integração de métodos: *a combinação das duas parece obter uma degração do desempenho, devido ao aumento do tempo de compilação e a má localidade*.

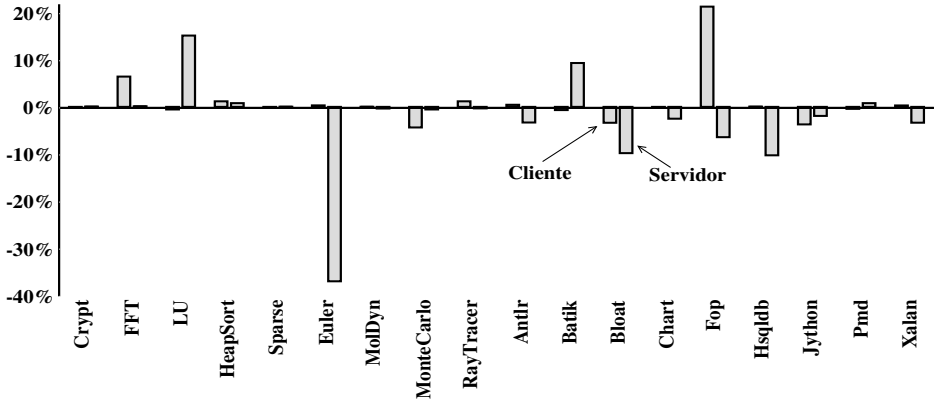


Figura 4.8: Impacto da eliminação de checagem de faixa.

4.5.5 *Coalescing*

Coalescing somente foi estudada no compilador *servidor*. Seu impacto é mostrado na figura 4.9. Quatro aplicações se beneficiam de *coalescing*, com aceleração entre 1,11% e 6,28%. Nestes casos, esta otimização reduz o número de instruções executadas e o número de falhas na *cache*. Infelizmente, *coalescing* pode degradar o desempenho, novamente para aqueles programas onde o tempo de compilação é um problema.

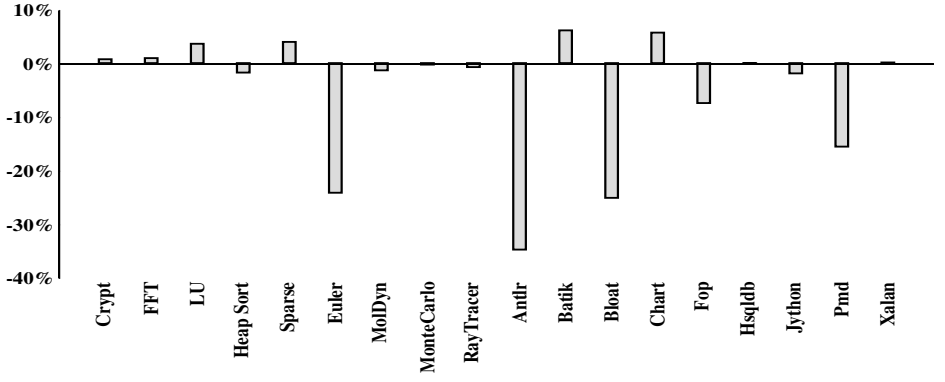


Figura 4.9: Impacto de *coalescing*.

4.5.6 Otimização *Peephole*

O compilador *cliente* e o compilador *servidor* implementam diferentes formas de *peepholing*. No compilador *cliente peepholing* obtém um bom desempenho para *Fop* e *FFT*, acima de 6%. Por outro lado, *peepholing* obtém diferentes impactos para diferentes programas. *Peepholing* diminui o IPC para *Fop*, enquando aumenta o IPC para *FFT*. Em ambos os casos *peepholing* tende a aumentar o número de falhas de *cache*, mas diminui o número de instruções de *hardware* executadas.

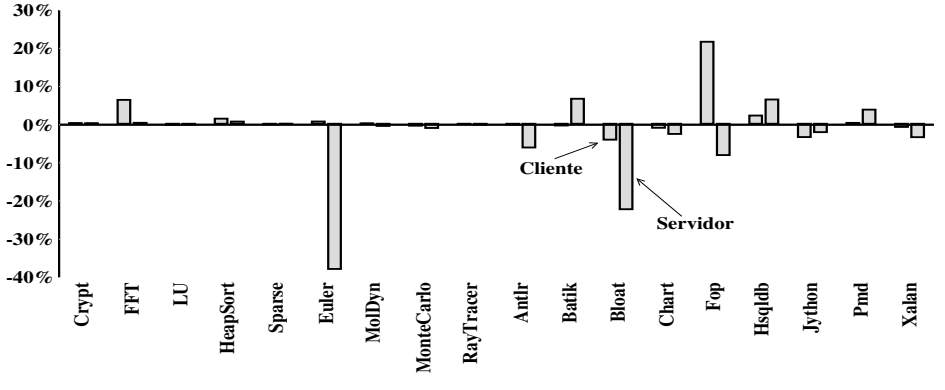


Figura 4.10: Impacto da otimização *peephole*.

4.5.7 *On-Stack Replacement*

On-stack replacement é uma otimização específica de compiladores JIT, e como os resultados mostram, é a segunda otimização mais efetiva, que melhora em até 80% o desempenho em ambos os compiladores. A figura 4.11 mostra o impacto de *on-stack replacement*.

On-stack replacement é crítico se o programa gasta muito tempo em um único método. Este é o caso dos progamas *Crypt*, *FFT*, *Sparse* e *Jython*. Esta otimização requer mais variáveis vivas, o que aumenta a pressão por registradores e pontencialmente degrada o desempenho. Este não é o caso para o compilador *cliente*, mas para o compilador *servidor*, onde *Bloat* e *Pmd* tem um mau desempenho. Nestes casos, a degradação do desempenho corresponde a um significante aumento do tempo de compilação.

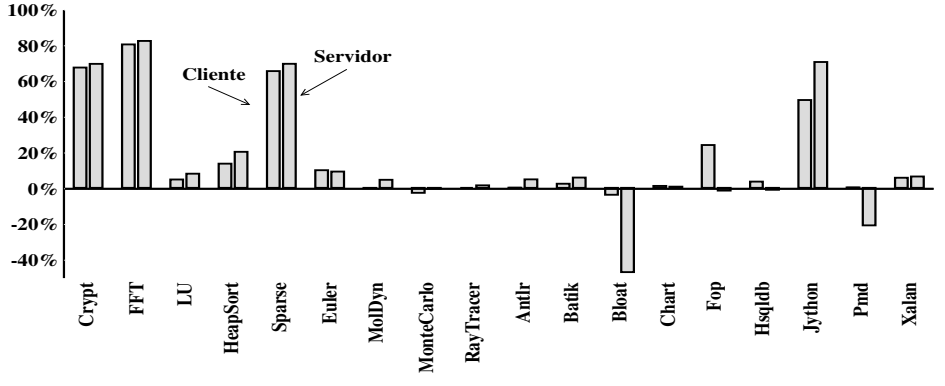


Figura 4.11: Impacto de *on-stack replacement*.

4.6 O Impacto no Tamanho do Código

A tabela 4.4 apresenta a variação do tamanho do código nativo gerado. Os programas científicos tendem a ser mais simples e possuir um código menor. Neste caso, o código gerado pelo compilador *servidor* chega a ser três vezes mais compacto, com as exceções de *Euler* e *MonteCarlo*. Por outro lado, o compilador *servidor* geralmente produz um código maior para os programas não-científicos. Este crescimento pode ser explicado pela agressividade ao aplicar integração de procedimentos. Infelizmente, esta agressividade nem sempre resulta em melhor código. Uma exceção e um caso interessante é *Jython*, onde o compilador *servidor* possui o melhor desempenho e gera um código duas vezes mais compacto do que o compilador *cliente*.

Programa	*Bytecodes* Compilados		Programa	*Bytecodes* Compilados	
	Cliente	Servidor		Cliente	Servidor
Crypt	3.38	1.27	*Antlr*	247.36	362.21
FFT	3.27	1.73	*Batik*	146.47	109.44
LU	3.07	1.53	*Bloat*	174.44	273.69
Heap Sort	2.50	0.83	*Chart*	86.12	106.49
Sparse	2.56	1.17	*Fop*	78.21	96.86
Euler	26.42	41.64	*Hsqldb*	46.67	62.03
MolDyn	5.27	4.45	*Jython*	35.31	16.89
MonteCarlo	15.62	16.18	*Pmd*	80.28	122.13
RayTracer	4.68	3.62	*Xalan*	60.20	61.42

Tabela 4.4: Tamanho do código gerado (em *Kbytes*).

4.7 Consideração Gerais

A tecnologia JIT introduziu significantes melhorias sobre a interpretação pura, e possui um bom desempenho quanto comparado com compiladores tradicionais [106]. Por outro lado, esta tecnologia preserva a principal vantagem do uso de *bytecodes*: a portabilidade. Ficou verificado que Java compilar a partir de *bytecodes* não introduz um significante *overhead*. Em muitos casos o tempo de compilação é uma fração desprezível do tempo total de compilação.

Um problema que ainda existe nos compiladores da Sun é que algumas otimizações [108] podem resultar em uma degradação do desempenho. Em particular integração de métodos pode aumentar o tamanho do código e a quantidade de acessos à memória, ocasionando um desempenho sub-ótimo.

Um fino controle do ambiente JIT é necessário para se obter o melhor desempenho. Controle manual poderia ser permitido, contudo automatizar o sistema é a opção ideal. Essa necessidade é muito clara na aplicação da otimização integração de métodos. Esta otimização é muito importante em linguagens orientadas a objetos, devido tais programas freqüentemente chamarem métodos pequenos. Contudo, esta otimização pode ocasionar em uma degradação do desempenho, quando aplicada sem se levar em consideração as características do programa, como é o exemplo do programa *Antlr*.

A tendência corrente é ajustar o ambiente de execução às características do programa [108]. Os ambientes Java disponíveis não se comportam desta maneira. Por exemplo, o sistema de compilação da máquina virtual Java da Sun possui um conjunto de otimizações (com parâmetros fixos) que são aplicadas indiscriminadamente a qualquer programa.

Esta deve ser a área de pesquisa nos próximos anos, *não apenas compilar código mais freqüente mas também obter, em tempo de execução, informações sobre o comportamento do programa, e usá-las para ajustar o sistema de compilação às características do programa sendo executado*.

Após estudar os conceitos fundamentais em compilação dinâmica e o desempenho dos compiladores JIT da Sun, pode-se partir para a descrição detalhada da tese proposta. Este é o objetivo do próximo capítulo.

Capítulo 5

Compilação *Just-In-Time* Para Prolog

Baseado na idéia de usar informações sobre o tipo dos dados para otimizar mais eficientemente o código gerado, foi desenvolvido o compilador YAPc: *um compilador dinâmico para Prolog*. O projeto do compilador possui dois objetivos:

- ***Desempenho.*** O principal objetivo foi obter desempenho mesmo em situações onde o tempo total de execução do programa fosse consideravelmente pequeno.
- ***Manutenabilidade.*** O sistema deveria ter baixo custo de manuteção porque um problema com os trabalhos anteriores é que eram sistemas muito complexos, conseqüentemente o custo de manutenção acabou por não justificar o benefício de desempenho.

A idéia principal é focar nos componentes mais importantes do programa e inferir quais as características do programa nestes componentes. A abordagem de interpretar primeiro e depois compilar garante a possibilidade de coletar tais características, com o objetivo de utilizá-las para gerar um código otimizado. Outro ponto importante é o fato dos algoritmos utilizados serem simples e eficientes, o que torna o sistema fácil de se manter.

Compilação dinâmica tem várias vantagens importantes que a tornam particularmente importante para Prolog. Primeiro, permite saber quais os componentes (regiões) mais utilizados no código. Na prática, a compilação dinâmica permite reduzir substancialmente o *overhead* de compilação, concentrando o processo de compilação nos componentes mais importantes (*hot-spots*). Segundo, a compilação dinâmica possibilita o compilador ter informações sobre os argumentos de cada procedimento, o que é particularmente difícil no caso de uma linguagem não tipada e com vários modos de uso como Prolog [111]. Nomeadamente, em Prolog é importante saber:

- Se um argumento é uma variável livre ou se está ligado a um termo;
- Se um argumento instanciado é uma constante ou um termo com um certo funtor.

Este capítulo descreve o processo de compilação dinâmica para Prolog, mais precisamente o compilador dinâmico YAPc. Inicialmente é descrito o ambiente de execução, para em seguida ser descrita a compilação dinâmica e a arquitetura do compilador. Este finaliza apresentando um exemplo de compilação.

5.1 Ambiente de Execução

O ambiente de execução (YAP+) é a versão do YAP [39, 102, 130] que faz uso de um compilador JIT. Inicialmente o ambiente interpreta todas as cláusulas e, baseado em informações coletadas em tempo de execução identifica as cláusulas que são freqüentemente executadas. Após, o interpretador aciona o compilador para gerar código nativo somente para estas cláusulas.

O YAP é um ambiente Prolog desenvolvido na Universidade do Porto. Altamente portável, como outras implementações o código Prolog é compilado para uma máquina abstrata, a *Yet Another Abstract Machine* (YAAM), sendo depois interpretada. A YAAM [74] possui uma linguagem intermediária baseada na WAM com algumas otimizações para melhorar o desempenho.

A principal diferença entre a YAAM e a WAM está no modo como é realizada a unificação de estruturas compostas. O código da WAM corresponde a uma busca em largura dos termos, enquanto que o código da YAAM corresponde a uma busca em profundidade.

A YAAM possui as seguintes vantagens sobre a WAM:

1. *Gera um número menor de instruções*. A instrução *pop* corresponde a uma instrução *unify_var* na WAM. Nesta é sempre necessário gerar os *unify_var* correspondentes a cada subestrutura. Enquando na YAAM é possível agregar várias instruções *pop* na instrução *popn*, e em alguns casos, é possível não gerá-las. Por exemplo, a lista [1,2,3] dará origem ao seguinte código na YAAM:

```
get_list Xi
unify_atom 1
unify_last_list
unify_atom 2
unify_last_list
```

```
unify_atom 3
unify_atom []
```

Neste caso não é necessário gerar nenhuma instrução *pop* porque não existe a necessidade de continuar com uma das estruturas superiores. Note ainda que as instruções *unify_last_list* e *unify_last_struct* diferem das instruções *unify_list* e *unify_struct* por não precisarem fazer um *push* do registrador `S`.

2. *Outra vantagem é a propagação do modo de leitura ou de escrita*. Considere a estrutura k(k(X),k(X)) na figura 5.1.

```
get_struct Aj, k/2
       unify_var Rx
       unify_var Rl
get_struct k/1, Rx
       unify_var Rm
get_struct R1, k/1
       unify_var Rm
```

(a) WAM

```
get_struct Aj, k/2
        unify_struct k/1
                    unify_var Ri
                    pop
        unify_struct k/1
                   unify_val Ri
```

(b) YAAM

Figura 5.1: Código WAM e YAAM para a estrutura k(k(X),k(X)).

Se o argumento *Aj* unificar com uma variável, entrando portanto no modo de escrita, a YAAM usa essa informação para simplificar a construção de ambas as subestruturas. Por outro lado, na WAM isso não é possível, o ambiente deve testar o valor de *Rk* para voltar a determinar se está em modo de leitura ou de escrita.

A YAAM ainda difere da WAM na execução das instruções *get_struct*, *get_list*, *unify_struct* e *unify_list*. Na YAAM se o modo de execução for alterado para escrita durante a execução destas instruções, a *heap* é inicializada, fato que não ocorre na WAM. Com isso, a YAAM consegue evitar que as instruções *unify_void* e *unify_var* sejam geradas. A desvantagem é que existem casos onde essas instruções não são geradas, tornando a inicialização da *heap* uma operação supérflua.

A principal vantagem da YAAM está no uso da instrução *pop* que recupera o modo de execução, enquanto que na WAM é necessário realizar uma desreferenciação para decidir em que modo a execução deve continuar.

As cláusulas de um programa Prolog são compiladas isoladamente pelo YAP. Cada

cláusula é transformada em código para a máquina abstrata YAAM passando por cinco fases, a saber:

- *compilação da cabeça de uma cláusula*;
- *compilação do corpo de uma cláusula*;
- *classificação de variáveis e otimização das variáveis temporárias*;
- *eliminação das instruções pop e unify_var supérfluas*;
- *preparação do código para ser executado pela máquina abstrata*.

O código de indexação é gerado posteriormente, quando todas as cláusulas do programa forem compiladas. O YAP ao compilar o código para as cláusulas separado do código de indexação, torna o processo de asserção de novas cláusulas relativamente simples. Esta é uma das vantagens dos sistemas Prolog baseado em interpretadores, a fácil implementação das instruções: *assert* e *retract*. Nos sistemas com geração de código nativo a implementação destes predicados é mais complexa.

5.2 Compilação Dinâmica

YAP+ utiliza a compilação dinâmica [11] não apenas para poder usar informações sobre tipo dos argumentos de um predicado, mas também para determinar quais partes da aplicação deverão ser otimizadas. A figura 5.2 mostra o processo de compilação dinâmica do sistema.

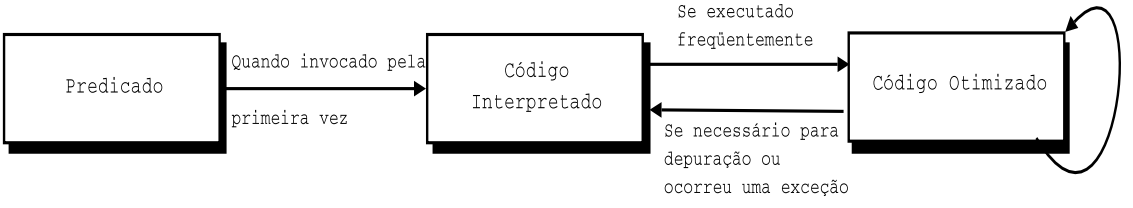


Figura 5.2: Processo de Compilação.

Quando um predicado é invocado pela primeira vez, ele é sempre interpretado. Se ele é executado freqüentemente, ele é compilado e conseqüentemente otimizado utilizando informações sobre tipos de dados. Ter um compilador rápido é essencial para reduzir as pausas de compilação em um sistema interativo que utiliza a compilação dinâmica.

O sistema deve descobrir oportunidades para a compilação sem a intervenção do programador. Em particular, ele deve decidir:

- ***Quando compilar***. Quanto tempo deve ser esperado para que as informações sobre o tipo dos argumentos estejam consistentes?
- ***O que compilar***. Quais predicados se beneficiarão das informações coletadas?
- ***Quais caminhos compilar***. Quais os caminhos mais executados?

As subseções seguintes discutem cada uma destas questões.

5.2.1 Quando Compilar

O sistema de compilação utiliza contadores para detectar candidatos a serem compilados [106]. Cada cláusula interpretada possui seu próprio contador. Na entrada de seu código, o ambiente incrementa o contador e o compara com um determinado limite. Se o contador excedeu o limite, o sistema de compilação decide se a cláusula deveria ser compilada.

Contadores foram propostos apenas como um primeiro passo, na proposta original. Porém, o momento do acionamento do sistema de compilação (*"quando"*) é menos importante para um bom resultado de compilação do que o mecanismo de seleção (*"o que"*). Desde que a técnica baseada em contadores tem se mostrado uma boa opção [106], no desenvolvimento de YAP+ não tem sido investigado novos mecanismos. Contudo, existem algumas questões relativas a utilização de contadores, que merecem ser observadas:

- *Idealmente, o sistema deveria compilar apenas aqueles predicados cujo custo de otimização fosse menor do que o benefício obtido através das futuras invocações do predicado otimizado*. Naturalmente, o sistema não conhece quão freqüente um predicado será executado no futuro, mas em uma medida baseada em taxas também o passado é esquecido: um predicado que executa com menor freqüência que uma taxa de execução mínima nunca acionará o processo de compilação, ao menos se ele for executado muitas vezes.
- *O limite de invocação não deveria ser uma constante, pelo contrário, ele deveria depender de um predicado em particular*. O que os contadores estão realmente tentando medir é quanto tempo de execução é gasto executando um código interpretado. Então, um predicado que se beneficiaria muito da otimização deveria incrementar mais rapidamente seu contador (ou ter um baixo limite) do que um

predicado que não se beneficiaria do processo de otimização. Na prática é difícil estimar o impacto de otimizar um predicado em particular.

- *Como o tempo de vida deveria ser adaptado quando a aplicação fosse executada em uma máquina mais rápida (ou mais lenta)*? Supondo que o limite do contador original seja 1000 chamadas, mas que o sistema agora é executado em uma máquina que é duas vezes mais rápida. Deveria este parâmetro ser alterado, e se sim, como? Pode-se ver a máquina mais rápida como um sistema onde o tempo real seja reduzido a metade, e reduzir o limite para 500 chamada. Mas, pode-se argumentar que a taxa limite de invocação é absoluta: se um predicado executa menos que `N` vezes por segundo, não tem de ser compilado.

- *Similarmente, deveria o limite de invocação do compilador ser tempo real, tempo de CPU, ou alguma unidade específica de máquina*? Intuitivamente, usar tempo real parece um erro, dado que interferências de outras tarefas ou do usuário influenciariam no processo de compilação. Usar tempo de CPU também tem problemas: por exemplo, se a maior parte do tempo é gasta na coleta de lixo ou no processo de compilação, o tempo de vida é efetivamente diminuído desde que predicados interpretados gastam pouco tempo para executar e incrementam seus contadores de invocação. Por outro lado, este efeito pode ser desejável: se pouco tempo é gasto em código Prolog compilado, otimizar este código não acarretará em um aumento de desempenho significante.

5.2.2 O Que Compilar

Quando o contador atinge um determinado limite, o sistema de compilação é invocado para decidir se a cláusula deve ser compilada. Uma estratégia simples seria sempre compilar a cláusula cujo contador atingiu o determinado limite, pois é óbvio que ela foi invocada freqüentemente. Porém, está estratégia não é sempre a mais eficiente. Por exemplo, supondo que a cláusula que excedeu o limite seja formada apenas pelas instruções:

```
get_num 1, A1
proceed
```

otimizar esta cláusula não geraria o melhor *ganho*, a solução ideal é integrar a cláusula com a cláusula que a invocou. Em geral, *para encontrar um boa candidata à compilação o corpo da cláusula deve ser inspecionada*.

Inspecionando a clásula que excedeu o limite, o sistema de compilação verifica se esta é uma boa candidada para a compilação. Caso verdadeiro, o compilador é invocado para otimizá-la. Após gerar código nativo para esta cláusula, o sistema de compilação instala no ambiente de execução esta nova versão. Se nenhuma candidata é encontrada, o interpretador continua a execução. O sistema seleciona a cláusula a ser compilada examinando diversas métricas. Para uma cláusula $\boldsymbol{c}$, os seguintes valores são definidos:

- $\boldsymbol{c}.contador$: número de vezes que o predicado foi invocado.
- $\boldsymbol{c}.tamanho$: número de instruções YAAM.

O compilador compila uma cláusula de cada vez, e seleciona a que satisfaz as seguintes condições:

- $\boldsymbol{c}.contador > MinInvocações$. Esta condição assegura que o predicado foi executado vezes suficiente para considerar o tipo dos seus argumentos.
- $\boldsymbol{c}.tamanho >= TamLimite$. Esta última, elimina os predicados que não obteriam nenhum ganho de desempenho.

Estas regras assumem que as cláusulas executadas freqüentemente são boas candidatas à otimização. Além disto, estas regras fornecem um excelente ambiente para a detecção das regiões freqüentemente executadas do programa.

5.2.3 Quais Caminhos Compilar

Programas geralmente contêm caminhos raramente ou nunca executados. Compilá-los pode causar efeitos adversos tais como reduzir a eficiência do código gerado. Por exemplo, um compilador otimizador pode gastar muito tempo aplicando agressivas otimizações em código raramente executado. O problema é que procedimentos fornecem uma maneira conveniente de particionar o processo de compilação, mesmo que não sejam necessariamente a melhor unidade para realizar otimizações. Uma abordagem diferente é tentar compilar apenas aquelas partes que são executadas freqüentemente.

Em um sistema de compilação dinâmica, esta estratégia é usual. Primeiro, compiladores dinâmicos podem se utilizar da vantagem de obter em tempo de execução informações estruturais e comportamentais da aplicação e usá-las no processo de seleção de regiões. Segundo, eles são sensíveis ao *overhead* de compilação, e esta técnica pode significantemente reduzir o tempo total de compilação e o tamanho do código. Terceiro,

eles podem evitar gerar código fora das regiões selecionadas até o código ser realmente executado.

O sistema de compilação do YAP+ utiliza a técnica de compilação baseada em regiões. Procedimentos (ou melhor, cláusulas) não são tratados como a unidade de compilação. Ao invés disto, YAPc seleciona apenas aquelas porções que são identificadas como caminhos freqüentes. O termo região se refere a nova unidade de compilação, que resulta da exclusão de todas porções raramente ou nunca executadas.

O compilador otimizador seleciona da cláusula que está sendo compilado, os caminhos freqüentes observando as informações sobre o tipo dos seus argumentos. Desta forma, o compilador reduz e otimiza o grafo de fluxo de controle da clásula que está sendo compilada.

5.3 Arquitetura do Compilador

O sistema YAP+ possui um compilador otimizador, o YAPc, que é acionado para gerar código nativo otimizado para as partes do programa que são executadas freqüentemente. O compilador otimizador YAPc possui as seguintes características:

- ***Emprega algoritmos simples***. O objetivo é ter um ambiente de compilação leve que reduza as pausas do processo de compilação.
- ***É adaptável às características do programa***. As características do programa determinam o comportamento do compilador, ou seja, o fluxo de execução entre as suas fases. O objetivo é obter um sistema que alcance um bom desempenho.
- ***Diminui as pausas do sistema de compilação***. A medida que YAPc compila apenas os caminhos executáveis, o processo de compilação além de mais efetivo e leve, se torna mais rápido.
- ***Gera código compacto***. Como o compilador YAPc apenas gera código para as porções do programa que são realmente executadas, o código é altamente compacto.

A figura 5.3 apresenta a arquitetura do compilador. O compilador é dividido em *frontend* e *backend*. O *frontend* chama o *parser*. Este representa o código YAAM como uma árvore sintática abstrata otimizada, que contém apenas regiões executáveis. Após o *parser*, o compilador irá decidir se transforma a árvore sintática abstrata em uma representação baseada na forma *static single assignment* (SSA) [27]. No caso desta decisão ser

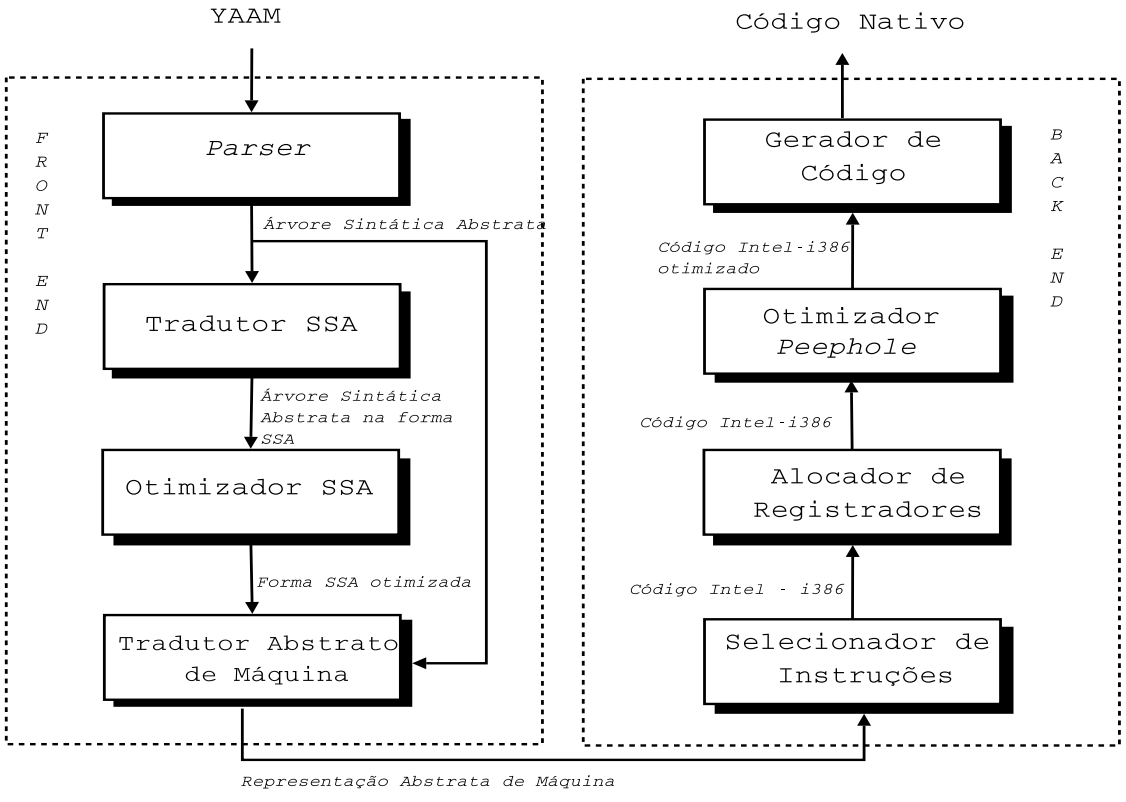


Figura 5.3: Arquitetura do Compilador YAPc.

tomada, o compilador otimiza esta representação. A última fase do *frontend* transforma a árvore sintática abstrata (que pode estar ou não na representação SSA) em uma representação abstrata de máquina. Inicialmente, o *backend* realiza a seleção de instruções gerando código específico para a arquitetura alvo. Em seguida registradores são alocados, para posteriormente o otimizador *peephole* otimizar o código gerado. Finalmente, o *backend* gera código nativo.

Quando o compilador termina o processo de compilação, o código nativo é instalado no ambiente de execução e escalonado para execução. As subseções a seguir descrevem cada fase do compilador.

5.3.1 O *Parser*

Quando o ambiente de execução aciona o sistema de compilação, este fornece ao compilador o conjunto de instruções YAAM do predicado que excedeu seu contador. Fica a cargo do *parser* gerar uma árvore sintática abstrata otimizada a partir deste conjunto.

No momento em que o ambiente de execução aciona o sistema de compilação, as informações sobre os tipos dos dados estão disponíveis no ambiente. Desta forma o pro-

cesso de criar uma árvore sintática abstrata otimizada é simples. Basta apenas inspecionar o tipo dos argumentos do predicado. O *parser* realiza as seguintes tarefas:

1. ***Decide qual será o fluxo do processo de compilação***. Durante a criação da árvore sintática abstrata, o *parser* decide se aplicar otimizações de alto nível ocasionará em ganho de desempenho. Esta decisão é tomada inspecionando o tipo de cada instrução inserida na árvore. O objetivo em se tomar tal decisão é tornar o processo de compilação adaptável às características do programa. Esta decisão determinará o fluxo de execução entre as fases do compilador.

2. ***Gera uma árvore sintática abstrata para a cláusula do predicado em execução***. As instruções YAAM da cláusula são representadas por um conjunto de instruções simples através de uma árvore sintática abstrata. Esta árvore contém apenas os caminhos executáveis da cláusula, desta maneira o processo de compilação é mais efetivo ao otimizar apenas código realmente executado. Durante o processo de criação da árvore sintática abstrata, o *parser* baseado nas informações sobre o tipo dos dados seleciona apenas as partes de cada instrução YAAM que efetivamente serão executadas. Isto torna o processo de compilação mais efetivo.

3. ***Reduz a granularidade das instruções YAAM***. As instruções YAAM realizam tarefas complexas, contudo estas podem ser implementadas por um conjunto de instruções simples.

4. ***Especializa unificação***. O *parser* usa duas versões para unificação (uma para cada modo de execução), que são otimizadas separadamente.

5. ***Realiza integração de funções nativas***. Reduzir a granularidade de cada instrução YAAM pode ocasionar em chamadas de funções do ambiente de execução. Portanto, o *parser* pode integrar tais funções, eliminando o *overhead* ocasionado por chamadas de funções. Por exemplo, árvore sintática gerada pelo *parser* para o programa *append* possui uma chamada à função *AbsPair* para determinar o próximo termo a ser unificado. Neste caso, ao invés de realizar uma chamada à biblioteca nativa, o *parser* integra esta função.

5.3.1.1 Criação da Árvore Sintática Abstrata

Em YAP+ cada instrução YAAM possui um *template* contendo um conjunto de instruções simples, da representação intermediária de alto nível descrita na figura 5.4. Portanto, cada

template corresponde a uma sub-árvore abstrata.

```
HIR_Dec         = HIR_DecVar

HIR_DecVar      = (HIR_Indentificador, HIR_Type)

HIR_Type        = int | unsigned | pointer

LIR_Inst        = HIR_Seq | HIR_Label | HIR_Goto | LIR_Assign | HIR_If | HIR_Return |
                  HIR_Call

HIR_Seq         = HIR_InstrList
HIR_Label       = HIR_Identificador
HIR_Goto        = Goto(HIR_Identificador)
HIR_Assign      = Assign(HIR_Exp, HIR_Exp)
HIR_If          = If(HIR_Exp, Goto(HIR_Identificador))
HIR_Return      = Return(HIR_Exp)
HIR_Call        = Call(HIR_identificador, HIR_ExpList)


HIR_Exp         = HIR_Var | LIR_Content | HIR_Address | HIR_Op | HIR_Int | HIR_Unsigned

HIR_Var         = HIR_Identificador
HIR_Content     = Content(HIR_Identificador)
HIR_Address     = Address(HIR_Identicador)
HIR_Op          = Op(HIR_oper, HIR_Exp, HIR_Exp)

HIR_Int         = [+|-] [HIR_Digito]+
LIR_Unsigned    = [+] [HIR_Digito]+

HIR_Oper          = HIR_add | HIR_sub | HIR_mul | HIR_div | HIR_and | HIR_and | HIR_or |
                    HIR_shl | HIR_shr | HIR_xor

HIR_InstrList     = (HIR_Instr, HIR_InstrList)

HIR_ExpList       = (HIR_Exp, HIR_ExpList)

HIR_Letra         = a | ... | z | A | ... | Z
HIR_Digito        = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

HIR_Identificador      = HIR_Letra { HIR_Letra | HIR_Digito }*
```

Figura 5.4: Representação de alto nível utilizada pelo *parser*.

O *parser* cria a árvore sintática abstrata de um predicado agrupando as sub-árvores de cada instrução YAAM. Neste momento, duas otimizações são realizadas:

1. *seleção de regiões*; e
2. *integração de funções auxiliares*.

A seleção de regiões ocorre juntamente com a criação da árvore sintática abstrata. Para cada instrução YAAM, o *parser* agrupa à árvore sintática abstrata do predicado somente os caminhos da sub-árvore (da instrução YAAM) que realmente serão executados. Este processo é realizado inspecionando o tipo dos argumentos da cláusula.

A integração de funções auxiliares também ocorre juntamente com a criação da árvore sintática abstrata. Se o caminho que será executado contém uma chamada a uma função auxiliar, o *parser* integrará a chamada.

Durante a execução do programa *append*, descrito na figura 5.5, gerar código para a instrução `get_list` é o processo de selecionar do *template* desta instrução as regiões que realmente podem ser executadas.

```
append/3

switch_nl_list [ ], [H|T], fail

[ ]:        get_atom [ ], A1
            get_val  X2, A3
            proceed

[H|T]:      get_list A1
            unify_var X4
            unify_last_var X1
            get_list A3
            unify_val X4
            unify_last_var X3
            execute append/3
```

Figura 5.5: Programa *append*.

A figura 5.6 mostra o *template* da instrução `get_list`. Este código primeiramente lê o valor do argumento acessível a partir de $P \rightarrow u.x.x$. Se este argumento for uma referência ocorre um salto para `L1` e entra-se num ciclo até provar um de dois casos. Se o argumento está instanciado, neste caso salta-se para `L2`. Ou se o argumento não está instanciado, então ele é uma variável livre, neste caso a execução continua na linha 16. O código em `L2` primeiro verifica se o argumento é realmente uma lista. Se for, coloca um ponteiro para a lista em `sreg` e salta para `L3`. Caso contrário gera uma falha. O parser terá que lidar com estas diferentes questões.

O *parser* inspeciona o tipo dos registradores da máquina abstrata para determinar o tipo dos dados do programa em execução. Após identificar as informações sobre o tipo

```
01.       d0 = *P->u.x.x;
02.       if (IsVarTerm(d0))
03.               goto L1;
04. L2:
05.       if (!IsPairTerm(d0))
06.               FAIL();
07.       sreg = RepPair(d0);
08.       goto L3
09.       do {
10.               if (!IsVarTerm(d0)
11.                     goto L2;
12.        L1:
13.               pt0 = (CELL *) d0;
14.               d0  = *(CELL *) d0;
15,       } while (Unsigned(A) != (D));
16.       s_sreg = H;
17.       d0 = AbsPair(s_sreg);
18.       H = s_sreg + 2;
19.       SREG = s_sreg;
20. L3:
```

Figura 5.6: *Template* da instrução get_list.

dos dados, o *parser*, utilizando estas informações, seleciona as regiões que potencialmente serão executadas.

Durante a execução da pergunta:

```
?- append([1,2,...,9999],[],_),
```

o *parser* identifica que o tipo do argumento `A1` é uma lista não vazia. Desta forma, ele gera para a primeira instrução `get_list` uma árvore abstrata contendo apenas as instruções das linhas 1 e 7. A figura 5.7 mostra a árvore sintática abstrata gerada pelo *parser* para a instrução `get_list`.

O compilador realiza este mesmo processo para todas as outras instruções. Primeiro, ele identifica o tipo dos argumentos, para em seguida, baseado nesta informação, selecionar do *template* somente os caminhos executáveis.

Observe que para a mesma instrução o compilador é capaz de lidar com situações diferentes. Este é o caso das instruções get_list. A segunda ocorrência é diferente da anterior. Nesta o o parser deverá provar que o argumento `A1` é uma lista não vazia, enquanto naquela deverá provar que o argumento `A3` é uma lista vazia ou uma variável livre.

O exemplo completo do processo de compilação do programa *append* é apresentado no final deste capítulo.

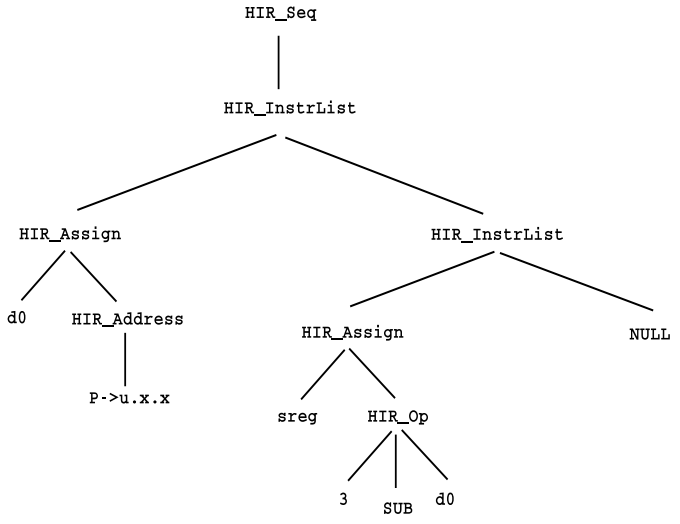


Figura 5.7: Árvore sintática abstrata para a instrução `get_list`.

Aplicar otimizações durante o *parser* também é realizado por outros compiladores, como por exemplo os compiladores da Sun descritos na seção 3.3.3.

5.3.1.2 Anotação da Árvore Sintática Abstrata

É durante a execução da primeira fase do compilador que se decide o fluxo de execução entre as fases de compilação.

Durante a construção da árvore sintática abstrata, o *parser anota* a árvore sintática com as seguintes informações:

1. *número de cópias*;
2. *número de constantes*; e
3. *número de saltos condicionais*.

Se cada uma destas informações exceder um limite pré-definido, o fluxo de execução continuará na segunda fase de compilação. Isto indica que o programa fonte possivelmente se beneficiará com a aplicação de otimizações de alto nível.

Para os casos onde estas informações não excedem o limite pré-definido, o fluxo de execução salta a fase de transformação para a representação SSA e a fase de otimização de alto nível.

5.3.1.3 A Finalização do *Parser*

A finalização do *parser* ocorre quando este encontra uma das seguintes instruções:

- *proceed*;
- *execute*; ou
- *call*.

A árvore abstrata construída até uma instrução *procceed* indica que o compilador irá gerar código para toda a cláusula do predicado. Suponha que para o programa *append* da figura 5.5, seja feita a pergunta:

```
?- append([],[],_)
```

Neste caso, as instruções:

```
get_atom [],A1
get_val X2, A3
proceed
```

serão selecionadas para compilação. Em um caso como este, YAPc compila todos as instruções YAAM, gerando uma versão otimizada para uma cláusula completa.

Por outro lado, uma parada decorrente da instrução *execute* somente indicará que o compilador irá gerar código para toda a cláusula, se o *parser* identificar que a chamada realizada pela instrução *execute* é relizada na mesma cláusula que está sendo compilada. Suponha agora, que para o mesmo programa seja feita a seguinte pergunta:

```
?- append([1,2,...,9999],[],_).
```

Diferente do caso anterior, as instruções:

```
name append/3
get_list A1
unify_var X4
unify_last_var X1
```

```
get_list A3
unify_val X4
unify_last_var X3
execute append/3
```

serão selecionadas para compilação. No momento de traduzir a instrução *execute*, baseado nas informações sobre o tipo dos dados, o *parser* identifica que esta instrução pode recursivamente chamar o mesmo conjunto de instruções. Portanto, o *parser* traduz a instrução *execute* como um teste condicional contendo um salto para o início do código. Mais precisamente, no seguinte algoritmo:

```
1 if A1 = [H|T] then
2 |  goto início;
3 else
4 |  return ;
5 end
```

Algoritmo 5.1: Tradução da instrução *execute*.

Se a instrução *execute* chama uma cláusula do predicado compilado, porém com tipo de argumentos diferente, ou se *execute* chama um predicado diferente, o *parser* simplesmente emite um ***return***, deixando o interpretador executar a chamada.

A tradução de uma instrução *call*, é similar ao caso anterior. No momento, este último indica uma compilação parcial da cláusula do predicado. Isto significa que, o conjunto de instruções YAAM, descritas na figura 5.8, para a execução da pergunta:

```
?- nreverse([1,2,...,9999],_).
```

será parcialmente executado em código nativo e parcialmente interpretado. Pois o parser irá parar ao incontrar a instrução *call*. Significando, que as instruções restantes serão executadas pelo interpretador.

Esta abordagem possui dois objetivos:

1. ***Deixar o compilador leve***. As decisões tomadas pelo compilador são as mais simples possíveis. Desta forma, não é gasto muito tempo decidindo por exem-

```
nreverse/2

switch_nl_list [ ], [H|T], fail

[ ]:        get_atom        [ ], A1
            get_atom        [ ], A2
            proceed

[H|T]:      get_list        A1
            unify_var       Y1
            unify_last_var  X1
            get_var         Y0, A2
            allocate
            put_var         Y2, A2
            call            nreverse/2
            put_unsafe      Y2, A1
            put_list        A2
            write_val       Y1
            write_atom      [ ]
            put_val         Y0, A3
            deallocate
            execute         append/3
```

Figura 5.8: Programa *nreverse*.

plo: *qual predicado compilar*. Conseqüentemente, o compilador reduz as pausas de compilação.

2. ***Eliminar os saltos entre o código compilado e o interpretador***. Se um predicado que contém uma instrução *call* em seu corpo fosse completamente compilado, a instrução *call* seria traduzida por uma chamada ao interpretador. Neste caso, quando o predicado chamado pela instrução *call* terminasse sua execução, o fluxo de execução retornaria para o código compilado. No caso do predicado ser executado freqüentemente, isto ocasionaria o *overhead* da transição entre interpretador e código nativo. Uma abordagem mais ambiciosa no futuro será integrar o predicado chamado pela instrução *call*.

Como será visto no capítulo 6, a compilação parcial se mostrou uma boa escolha para deixar o sistema leve e com bom desempenho.

5.3.2 O Tradutor SSA

Muitas análises de fluxo de dados precisam encontrar os usos de cada definição de variável ou a definição de casa uso de uma variável em cada expressão. O canal *def-uso* é uma

estrutura de dados que torna este acesso mais eficiente: *para cada declaração no grafo de fluxo, o compilador mantém uma lista de ponteiros para todos os usos da variável definida nesta declaração, e para cada instrução uma lista de ponteiros para todas as definições das variáveis usadas nela*. Desta maneira o compilador pode rapidamente saltar da definição para o uso ou vice-versa.

Uma melhoria nesta idéia é o uso da representação *static single-assignment* (SSA), uma representação intermediária na qual cada variável possui apenas uma definição no programa. A única (estática) definição pode estar em um laço que é executado diversas (*dinâmicas*) vezes, logo o nome representação *static single-assignment* ao invés de representação *single assignment* (onde variáveis nunca são redefinidas).

A representação SSA é popular por diversas razões:

1. Análises de fluxo de dados e algoritmos de otimização podem ser implementados de maneira simples quando cada variável possui apenas uma definição.

2. Se a variável tem `N` usos e `M` definições, o espaço necessário para representar as estruturas *def-uso* é proporcional a $N \times M$. Para a maioria dos programas reais, o tamanho da representação SSA é linear no tamanho do programa original.

3. Usos e definições das variáveis na representação SSA simplificam algoritmos como construção de um grafo de interferência.

4. Usos não relacionados da mesma variável em um programa tornam-se diferentes variáveis na representação SSA, eliminando relacionamentos desnecessários. Um exemplo é o algoritmo:

1 **for** $x = 1$ **to** N **do** $K[x] = 0$;
2 **for** $x = 1$ **to** M **do** $a = a + W[x]$;

Algoritmo 5.2: Usos não relacionados da mesma variável.

Neste exemplo, a variável x dos dois laços não se relacionam entre si. Portanto, não existe a necessidade de criar um relacionamento entre elas.

A escolha por esta representação está baseada nas razões descritas acima. Contudo, entender o comportamento da aplicação é crucial para se obter um bom desempenho. Como descrito, YAPc decide o quão efetivo será transformar a árvore sintática abstrata

em uma representação SSA, e em seguida otimizá-la. O *parser* possui informações suficientes para decidir a efetividade das otimizações de alto nível.

Embora o uso desta representação torne os algoritmos de otimizações mais leves e simples (o algoritmo utilizado por YAPc para propagação de cópia possui apenas cinco linhas), ela possui o problema de aumentar o tamanho do código gerado e a pressão por registradores. O tamanho do código aumenta devido ao uso de uma única definição para cada variável, acarretando o aumento dos acessos à memória. Conseqüentemente, o aumento da quantidade de variáveis aumenta a pressão por registradores. Em uma arquitetura com poucos registradores isto pode acarretar uma quantidade expressiva de *spills*.

A abordagem utilizada por YAPc reduz ao máximo os acessos à memória. Desta forma, o alocador de registradores tenta utilizar ao máximo os registradores da arquitetura alvo e cria *spills* necessários para as variáveis que excedem a quantidade de registradores.

5.3.2.1 Conversão da Árvore Sintática Abstrata para a Representação SSA

Em um programa composto por um simples bloco básico, como por exemplo:

1 $x = a + b;$
2 $b = 7;$
3 $x = b * y;$
4 $b = x * b;$
5 $x = c + 5;$

Algoritmo 5.3: Programa contendo apenas um bloco básico.

é fácil verificar que cada instrução pode *definir* uma nova variável ao invés de *redefinir* uma variável. Neste caso, cada nova *definição* de uma variável é modificada para *definir* uma nova variável, e cada uso da variável é modificado para usar a mais recente *definição*. Portanto, o programa acima pode ser transformado em:

1 $x0 = a0 + b0;$
2 $b0 = 7;$
3 $x1 = b0 * y0;$
4 $b1 = x1 * b0;$
5 $x2 = c0 + 5;$

Algoritmo 5.4: Representação SSA do programa anterior.

Porém, quando dois fluxos de controle se encontram, por exemplo através de um *if-then-else*, não é claro como ter uma única atribuição para uma variável. Um exemplo

desta situação é mostrado na figura 5.9(a). Se uma nova versão da variável x é definida no bloco 1 e outra no bloco 3, qual versão deverá ser utilizada no bloco 4?

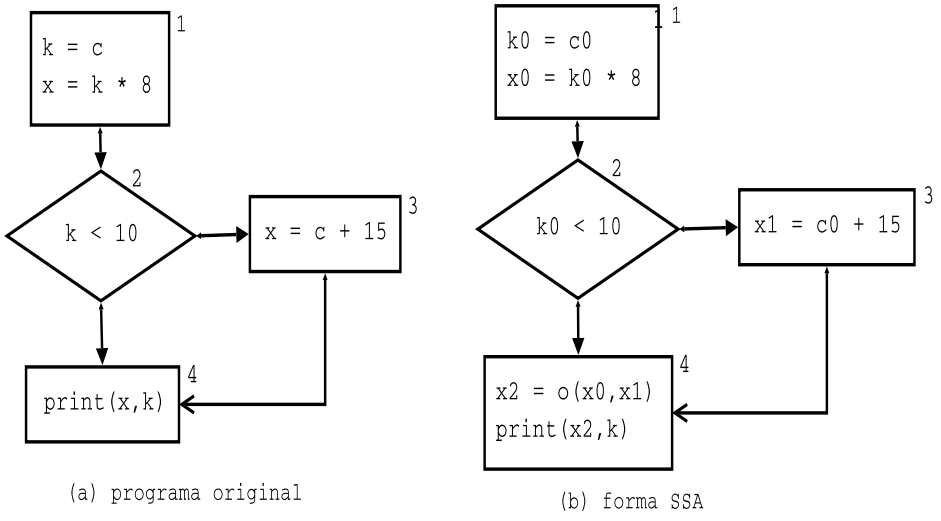


Figura 5.9: Um programa com *if-then-else*.

Esta situação é resolvida introduzindo a notação fictícia, função-ϕ. A figura 5.9(b) mostra que $x0$ e $x1$ são combinados na função

$x2 = \phi(x0, x1)$.

A função-ϕ indica que o valor da variável é determinado pelo fluxo de execução. No exemplo anterior, $x2$ terá o valor de $x0$ se a condição $k0 < 10$ for verdadeira, senão $x2$ terá o valor de $x1$.

Critério para Inserir as Funções-ϕ

O algoritmo que converte o programa para a representação SSA primeiro adiciona as funções-ϕ para as variáveis, e então renomeia todas as definições e usos das variáveis.

O tradutor SSA deve adicionar uma função-ϕ para cada variável contida em pontos de junção de fluxo, mais precisamente, para as variáveis contidas no nodo do grafo de fluxo de controle que possui mais de um predecessor. Mas isto é desnecessário. Por exemplo, na figura 5.9(b) o valor da variável k no bloco 4 independe do fluxo da execução. Portanto a variável k não necessita de uma função-ϕ.

Deve existir um critério para inserir no programa funções-ϕ. Um algoritmo eficiente

utiliza a noção de árvore de dominadores baseado em um grafo de fluxo de controle [84, 9].

O tradutor SSA de YAPc utiliza o seguinte critério para inserir funções-ϕ: *para cada nodo x que contenha uma definição para alguma variável a, insira uma função-ϕ em cada nodo z na fronteira de dominadores de x.* O objetivo de utilizar este critério é inserir apenas funções-ϕ para satisfazer o critério da fronteira de dominadores.

Inserindo Funções-ϕ

YAPc utiliza o algoritmo descrito por Appel [9] para inserir funções-ϕ. O algoritmo 5.5 descreve o algoritmo de Appel. O objetivo deste algoritmo é calcular o conjunto de variáveis ($F[n]$) que precisam de uma função-ϕ no nodo n.

Para a representação SSA tornar um compilador rápido, a representação SSA deve ser computada rapidamente. Observe que antes da execução do algoritmo 5.5, deve ser calculada a fronteira de dominadores (linha 11) de cada nodo. A fronteira de dominadores pode ser calculada de forma simples através de um método iterativo [9], contudo Lengauer e Tarjan [70] descrevem um algoritmo muito mais eficiente. Como um dos objetivo de YAPc é ser o mais rápido possível, ele utiliza o algoritmo de Lengauer e Tarjan para calcular a fronteira de dominadores de cada nodo. Uma descrição detalhada do algoritmo de Lengauer e Tarjan pode ser encontrada em [9, 70].

Renomeando Variáveis

Após inserir todas as funções-ϕ necessárias, o próximo passo é renomear as diferentes definições de cada variável e seus usos. O algoritmo para renomear as variáveis primeiramente renomeia todas as definições de a, e renomeia em seguida cada uso de a pela definição mais recente de a. Em um programa com *if-then-else*, o algoritmo renomeia cada uso de a com a definição d que está imediatamente acima de a na árvore de dominadores. Este processo é descrito nos algoritmos 5.6 e 5.7.

Algoritmo: Insere-ϕ()

Entrada: O grafo de fluxo de controle do programa.
Entrada: Para cada nodo n do grafo, o conjunto ($Vd[n]$) das variáveis nele definidas.
Entrada: A fronteira de dominadores de cada nodo.

```
foreach node n do
    foreach variável a ∈ Vd[n] do
        defsites[a] = defsites[a] ∪ {n};
    end
end
foreach variável a do
    W = defsites[a];
    while W ≠ {} do
        remova algum nodo n de W;
        foreach Y na fronteira de dominadores de n do
            if a ∉ F[Y] then
                insira a atribuição a = φ(a, a, ..., a) no topo do bloco Y. O número
                de argumentos da função−φ deve ser o número de predecessores de
                Y;
            end
            F[Y] = F[Y] ∪ {a};
            if Y ∉ W then
                W = W ∪ [Y];
            end
        end
    end
end
```

Algoritmo 5.5: Inserir funções-ϕ.

Algoritmo: Inicializa()

Entrada: Conjunto de variáveis do programa
Saída: *Count* e *stack* para cada váriavel

```
foreach variável a do
    count[a] = 0;
    stack[a] = {};
    empilhe 0 em stack[a]
end
```

Algoritmo 5.6: Inicialização.

```
1  Algoritmo:Renomear(n)
   Entrada: Nodo n
2  foreach instrução S no bloco n do
3      if S ≠ funcao − phi then
4          foreach uso de alguma variável x em S do
5              i = top(stack[x]);
6              troque o uso x por xi em S;
7          end
8      end
9      foreach definição de alguma variável a em S do
10         count[a] = count[a] + 1;
11         i = count[a];
12         empilhe i em stack[a];
13         troca a definição a pela definição ai em S;
14     end
15 end
16 foreach cada sucessor Y do bloco n do
17     foreach cada instrução em Y do
18         if instrução = função-φ then
19             arg = jth operando de Y;
20             i = top(stack[arg]);
21             troque o jth operando por argi;
22         end
23     end
24 end
25 foreach filho X de n do
26     Renomear (X);
27 end
28 foreach definição de alguma variável a em S original do
29     desempilhe stack[a];
30 end
```

Algoritmo 5.7: Renomear variáveis.

5.3.2.2 Ponteiros

A representação SSA foi inicialmente projetada para trabalhar com escalares. Trabalhos posteriores adicionaram o conceito de vetores, estruturas [36, 66] e ponteiros [37].

Construir uma representação SSA requer conhecer as variáveis *definidas* e *usadas* por uma instrução. Contudo, uma instrução pode definir ou usar uma variável que não está explicitamente referenciada, como por exemplo ponteiros. Para obter uma representação SSA, todas as referências das variáveis, sejam elas explícitas ou implícitas, devem ser conhecidas.

No algoritmo:

```
1 k = 9;
2 if b then
3 |   x = k;
4 else
5 |   *p = 8;
6 |   x = k;
7 end
8 z = x * k;
```

Algoritmo 5.8: Algoritmo com ponteiro.

a instrução da linha 5 possui um uso (do ponteiro p) e uma definição (da localização $*p$). Uma situação como esta deve ser clara durante a construção da representação SSA. A questão é a seguinte: *como renomear as variáveis de maneira a adequar o uso de referências implícitas*?

O algoritmo utilizado por YAPc é uma simplificação do algoritmo desenvolvido por Cytron e Gershbetin [37]. O algoritmo desenvolvido por Cytron e Gershbetin define para cada referência implícita do tipo: $*p = 8$, os seguintes conjuntos de dados:

1. *MayAlias(p,S)*: contém os nomes pontencialmente referenciados por $*p$ na instrução S. Por exemplo, se na linha 5 do algoritmo 5.8 p contiver o endereço da variável k, o valor da variável k será alterado quanto a instrução desta linha for processada, em outros casos o valor da variável k será inalterado. Para representar esta dualidade, Cytron e Gershbetin inserem no programa a função *IsAlias()* (descrita no algoritmo 5.9).

2. *MustAlias(p,S)*: contém os nomes certamente referenciados por $*p$ na instrução S. Se a variável $k \in$ *MustAlias(p,S)* (para a linha 5), então deve ser inserida a instrução $k = *p$ logo em seguida da instrução da linha 5. Se a variável k não é afetada na linha 5, então nenhuma atribuição para k é necessária, após a linha 5.

```
1  Algoritmo:IsAlias(a,b)
2  if a = b then
3  |  r0 = *a;
4  else
5  |  r1 = *b;
6  end
7  r2 = φ(r0, r1);
8  return (r2);
```

Algoritmo 5.9: Função *IsAlias()*.

O algoritmo de Cytron e Gershbetin é iterativo. Ele insere quantas funções *IsAlias*, ou ainda quantas atribuições, forem necessárias, até que um ponto fixo seja atingido. Após aplicar o algoritmo de Cytron e Gershbetin ao algoritmo com ponteiro, ele é transformado no seguinte:

```
1  k0 = 9;
2  if b then
3  |  x0 = k0;
4  else
5  |  *p = 8;
6  |  k1 = IsAlias(p, &k0);
7  |  x1 = k1;
8  end
9  x2 = φ(x0, x1);
10 k2 = φ(k0, k1);
11 z0 = x2 * k2;
```

Algoritmo 5.10: Representação SSA utilizando ponteiro.

O algoritmo utilizado por YAPc não necessita do conjunto de dados *MayAlias*, pois no momento que o *parser* cria a árvore sintática abstrata, o compilador tem conhecimento sobre cada referência (ponteiro) contida na árvore. Mais precisamente, o tradutor SSA sabe precisamente que o ponteiro *P* é uma referência à variável *V*. Portanto, em todo o programa, o ponteiro *P* **sempre e somente** referencia a variável *V*.

A tarefa do algoritmo desenvolvido é percorrer a árvore sintática abstrata e inserir instruções do tipo: $V = *P$, para a variável que *P* referencia, após cada instrução $*P = ?$.

5.3.2.3 Retorno da Representação SSA

Após aplicar algumas otimizações, o compilador deve remover as funções$-\phi$ do programa. A instrução $k3 = \phi(k0, k1, k2)$ no nodo `N` pode ser transformada em uma das

seguintes atribuições:

- $k3 = k0$ *se o fluxo procede do primeiro predecessor do nodo* `N`; ou
- $k3 = k1$ *se o fluxo procede do segundo predecessor do nodo* `N`; ou
- $k3 = k2$ *se o fluxo procede do terceiro predecessor do nodo* `N`.

O compilador YAPc remove as funções$-phi$ da seguinte maneira: para cada argumento i contido na função$-\phi$ da instrução $Vk = \phi(a0, a1, ..., ak)$, o compilador insere a instrução $Vk = xi$ no final do ith predecessor do bloco que contém a função-ϕ.

5.3.2.4 Exemplo Completo

O algoritmo utilizado por YAPc para transformar a árvore abstrata sintática em uma representação SSA possui as seguintes fases:

1. *criar um grafo de fluxo de controle*;
2. *ajustar os ponteiros existentes*;
3. *criar a árvore de dominadores e a fronteira de dominadores do grafo*;
4. *inserir as funções*$-\phi$;
5. *renomear as variáveis*.

A figura 5.10 mostra as fases da transformação SSA do seguinte programa:

```
1  i = 1; j = &i; k = 0;
2  while k < 100 do
3  |   if *j < 20 then
4  |   |   *j = k + 2; k = k + 1;
5  |   else
6  |   |   *j = k; k = k + 1;
7  |   end
8  |   print(i, *j, k);
9  end
10 return *j
```

Algoritmo 5.11: Exemplo SSA.

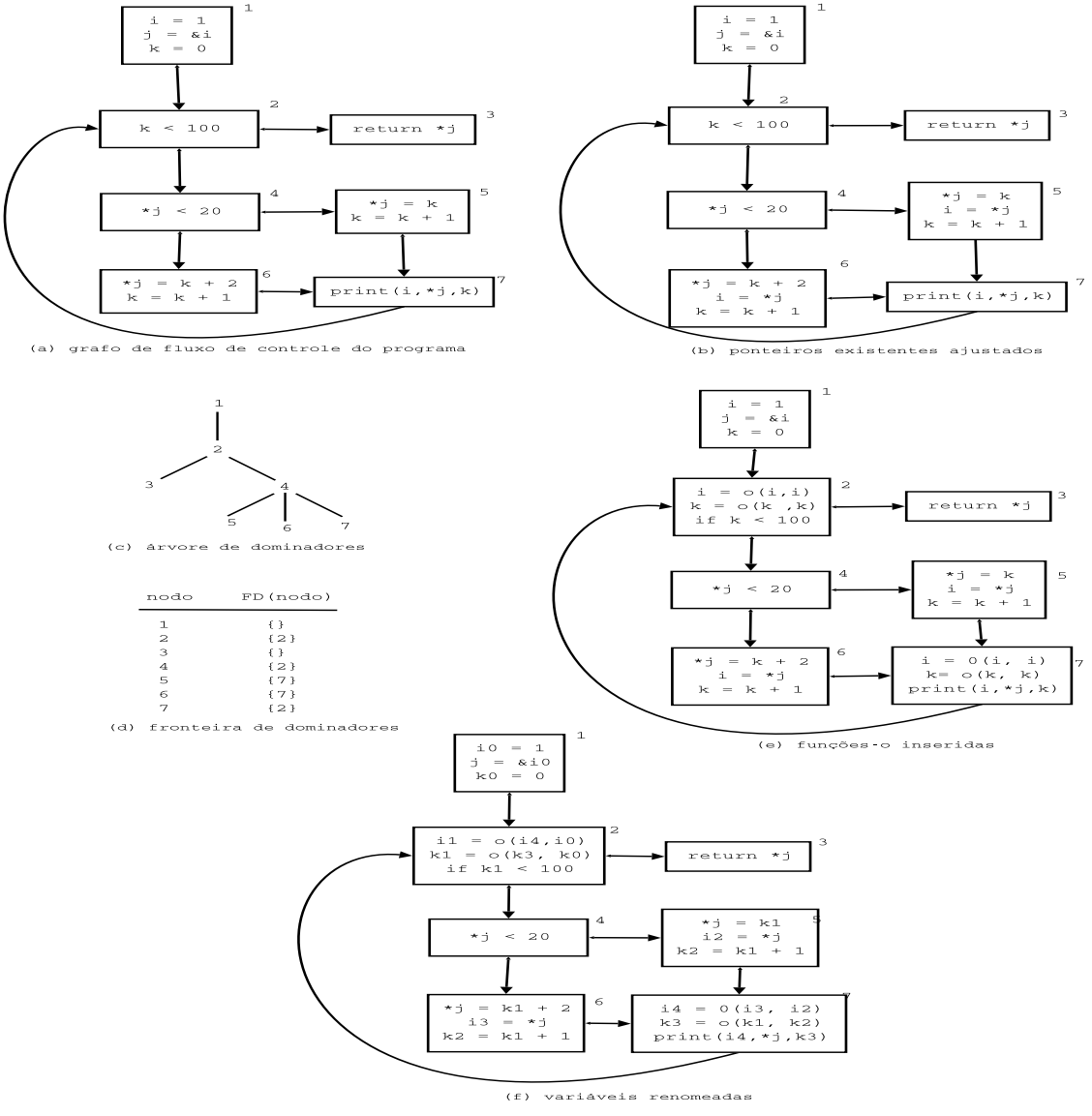


Figura 5.10: Exemplo da transformação SSA.

5.3.3 O Otimizador de Alto Nível

O otimizador de alto nível facilmente acessa informações de fluxo de dados, de representação SSA. Portanto, as otimizações por ele aplicadas não consomem um tempo considerável. Mesmo assim, o otimizador de alto nível somente será acionado se durante o *parser* for verificado que as otimizações deste nível serão efetivas para o aumento do desempenho do programa em execução.

Neste momento, o otimizador de alto nível aplica três otimizações:

1. *propagação de cópias (explicada na seção 3.2.1.1)*;

2. *propagação de constantes esparsas condicionais (será explicada na seção 5.3.3.2)*; e

3. *eliminação de código morto (explicada na seção 3.2.5.1).*

5.3.3.1 Propagação de Cópia

O algoritmo que propaga cópias pode ser sintetizado em dois passos:

1. *para cada variável construa uma lista de seus usos*; e

2. *identifique as cópias e propague-as*.

Como pode ser visto, o algoritmo para propagar cópias, utilizando a representação SSA, é bem simples. YAPc utiliza o algoritmo 5.12 para aplicar propagação de cópias.

Entrada: Lista de instruções do programa
Algoritmo: PropagaCópias()

```
foreach instrução N do
    if N é do tipo: v1 = v2 then
        foreach uso de v1 do
            troque-o por v2;
            Usos[v2] = N;
        end
    end
end
```

Algoritmo 5.12: Propagação de cópias.

5.3.3.2 Propagação de Constantes Esparsas Condicionais

Observe o seguinte programa da figura 5.11 representado como um grafo de fluxo de controle.

Se neste programa j for sempre igual a 1, então o bloco 5 nunca será executado. Se as vezes $j > 20$, então o bloco 5 será eventualmente executado. Na prática, no exemplo a variável j nunca possui um valor diferente de 1. Em casos como este, um algoritmo simples de propagação de constantes assume que o bloco 5 pode ser executado e que o valor da variável j não é constante. O problema destes algoritmos é que eles encontram um ponto fixo que não é na realidade o último ponto fixo.

Propagação de constantes esparsas baseada na representação SSA encontra o último ponto fixo: *este algoritmo não assume que um bloco pode ser executado até existir a*

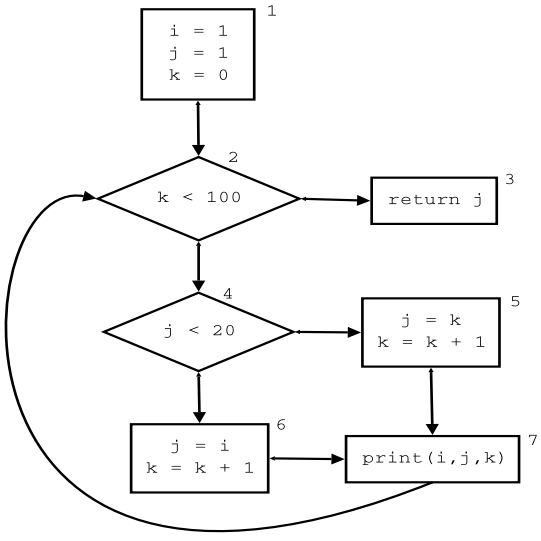


Figura 5.11: Grafo de fluxo de controle exemplo.

evidência de que ele será executado. Além disto, este algoritmo *não assume que uma variável não é uma constante até existir a evidência deste fato*.

Propagação de constantes esparsas inferi o valor da variável V como sendo:

- $V = \bot$*: indicando que não existe evidência que o programa atribuirá um valor para a variável* V; ou
- $V = \top$*: indicando que a variável* V *terá pelo menos um valor no decorrer do programa, porém este valor não pode ser calculado em tempo de compilação*; ou
- $V = 7$*: indicando que o valor* 7 *é atribuido a variável* V *no decorrer do programa*.

A figura 5.12 mostra o programa da figura 5.11 após a propagação de constantes esparssas.

O algoritmo utizado pelo compilador YAPc realiza uma execução simbólica do programa utilizando as arestas do grafo de fluxo de controle e as arestas SSA (canais *def-uso*) para transmitir informações. Durante a execução simbólica, o algoritmo marca os nodos como *executáveis* apenas quando as condições necessárias para a sua execução forem satisfeitas. Apenas os nodos *executáveis* são processados e nodos que possuam seus predecessores SSA processados. Este algoritmo pode ser analisado com mais detalhes em [84].

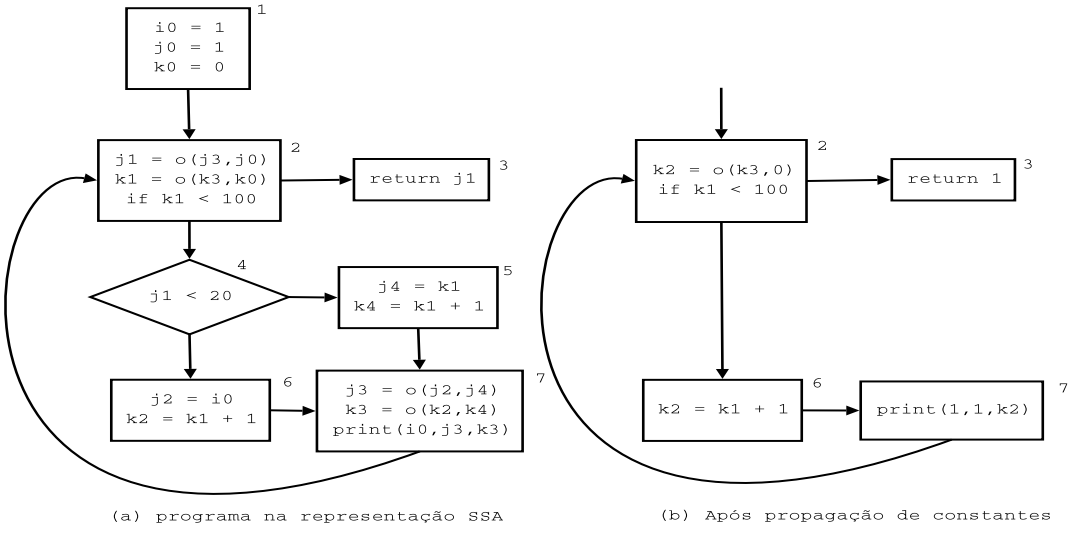


Figura 5.12: Exemplo de propagação de constantes esparssas condicionais.

5.3.3.3 Eliminação de Código Morto

Aplicar propagação de constantes esparsas, como também propagação de cópias, pode gerar código que não contribui para o resultado do programa. O objetivo da eliminação de código morto é eliminar do programa estas porções de código.

O algoritmo utilizado por YAPc pode ser sintetizado em dois passos:

1. *Marque como **viva** toda instrução que*:

   - *realiza entrada/saída, armazena dados em memória, é o retorno de uma função*; ou
   - *define uma variável v que é usada por outra instrução viva*; ou
   - *é um condicional*.

2. *Após, elimine todas as instruções não marcadas*.

Uma descrição mais detalhada do algoritmo utilizado por YAPc pode ser visto em [9].

5.3.4 O Tradutor Abstrato de Máquina

A última fase do *frontend* traduz a árvore sintática abstrata em uma representação abstrata de máquina. Embora seja possível traduzir diretamente a árvore sintática para código real de máquina, a abordagem utilizada visa obter modularidade e portabilidade. Desta maneira pode-se trocar facilmente o *backend* para uma nova arquitetura alvo.

```
LIR_Stm         = LIR_Seq | LIR_Label | LIR_Jump | LIR_CJump | LIR_Move | LIR_Expr

LIR_Seq         = SEQ(LIR_Stm, LIR_Stm)
LIR_Label       = LIR_Identificador
LIR_Jump        = JUMP(LIR_Exp, LIR_Label_List)
LIR_CJump       = CJUMP(LIR_RelOp, LIR_Exp, LIR_exp, LIR_Label_True, LIR_Label_False)
LIR_Move        = MOVE(LIR_Exp, LIR_Exp)
LIR_Expr        = LIR_Exp


LIR_Exp         = LIR_Global | LIR_Binop  | LIR_Mem | LIR_Temp | LIR_Eseq | LIR_Name |
                  LIR_Const  | LIR_UConst | LIR_Call

LIR_Global      = LIR_Identificador
LIR_Binop       = BINOP(LIR_BinOp, LIR_Exp, LIR_Exp)
LIR_Mem         = MEM(LIR_Exp)
LIR_Temp        = T[LIR_Digito]+
LIR_Eseq        = ESEQ(LIR_Stm, LIR_Exp)
LIR_Name        = LIR_Identificador
LIR_Const       = [+|-] [LIR_Digito]+
LIR_UConst      = [+] [LIR_Digito]+
LIR_Call        = CALL(LIR_Exp, LIR_ExpList)

LIR_RelOp       = LIR_a   | LIR_ae | LIR_b   | LIR_be | LIR_e   | LIR_g  | LIR_ge   | LIR_l   |
                  LIR_le  | LIR_na | LIR_nae | LIR_nb | LIR_nbe | LIR_ne | LIR_ng   | LIR_nge |
                  LIR_nl  | LIR_nle

LIR_Label_True    = LIR_Identificador

LIR_Label_Fals    = LIR_Identificador

LIR_BinOp         = LIR_add | LIR_sub | LIR_mul | LIR_div | LIR_and | LIR_and | LIR_or |
                    LIR_shl | LIR_shr | LIR_xor

LIR_Label_List    = (LIR_Label, LIR_Label_List)

LIR_Exp_List      = (LIR_Exp, LIR_Exp_List)

LIR_Letra         = a | ... | z | A | ... | Z
LIR_Digito        = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

LIR_Identificador      = LIR_Letra { LIR_Letra | LIR_Digito }*
```

Figura 5.13: Representação abstrata de máquina.

A representação abstrata de máquina implementada possui duas vantagens:

1. *esconde do front-end a complexidade sobre os detalhes específicos de máquina*; e
2. *é um tipo de linguagem de máquina abstrata que expressa as operações da máquina destino sem se preocupar com detalhes específicos de máquina*.

A representação abstrata, mostrada na figura 5.13, é conveniente para ser produzida, além de ser conveniente para ser traduzida em linguagem real de máquina, para qualquer arquitetura alvo desejada. Cada construção da representação abstrata possui um significado simples e claro, desta forma otimizações que reescrevam a representação intermediária são facilmente especificadas e implementadas.

Ela possui componentes que descrevem apenas tarefas simples tais como: *leitura da memória*, *escrita na memória*, *instrução para movimentação de dados*, *instruções aritméticas* e *saltos*. Desta forma, qualquer grupo de instruções da árvore sintática abstrata é transformado no correto conjunto de instruções da arquitetura alvo. Além disto, grupos de instruções abstratas de máquina são agrupadas para formar instruções reais de máquina.

5.3.5 O Selecionador de Instruções

O papel do selecionador de instruções é encontrar as instruções de máquina que implementem a representação abstrata de máquina em uma arquitetura real. Mais precisamente, o objetivo é encontrar um conjunto mínimo de instruções de máquina [56].

Em um processo de selecionar instruções que forneçam um custo a cada tipo de instrução, pode-se definir facilmente uma solução que é *optimum* e aquela que é apenas *optimal*. Uma solução *optimum* é aquela cuja soma das intruções fornecem o menor valor possível. Por outro lado, uma *optimal* é aquela onde dois custos adjacentes não podem ser combinados em uma simples instrução de custo mínimo. Uma solução *optimum* é sempre *optimal*, contudo o contrário não é verdadeiro.

Existem bons algoritmos para encontrar os dois tipos de soluções, contudo algoritmos para uma solução *optimal* são mais simples. Como um dos propósitos do YAPc é utilizar apenas algoritmos simples, o selecionador de instruções utiliza o algoritmo *Maximal Munch* [9] que encontra uma solução *optimal*.

Maximal Munch é um algoritmo guloso. Iniciando na raiz da representação intermediária, este algoritmo seleciona a instrução que melhor a representa. Esta tarefa irá criar várias sub-árvores. Agora, basta repetir o mesmo algoritmo para cada sub-árvore.

Este algoritmo gera as instruções em ordem reversa. A instrução da raiz é a primeira a ser gerada, mas ela somente pode ser instalada após as outras instruções terem produzido operandos em registradores.

5.3.6 O Alocador de Registradores

O alocador de registradores determina quais dos valores (*variáveis*, *valores temporários* ou *constantes*) deverão ser armazenados em registradores durante a execução do programa. Alocação de registradores é importante porque registradores são quase sempre um recurso escasso, especialmente na arquitetura *x86*. Geralmente, não exite uma quantidade de registradores suficiente para manter todos os valores do programa em execução.

Coloração de grafos é uma abordagem altamente efetiva para realizar a alocação de registradores em âmbito global. Esta é a implementação utilizada em YAPc.

O algoritmo implementado [9] é conhecido por fornecer bons resultados. Este possui as seguintes fases: *construa*, *simplifique*, *combine*, *congele*, *derrame* e *selecione*.

Construa cria um grafo de interferência. É utilizado análise de fluxo de dados para calcular o conjunto de temporários que estão simultaneamente vivos a cada ponto do programa, e se adiciona um vértice ao grafo para cada par de temporários no conjunto. Isto é repetido para todos pontos do programa. Cada nodo do grafo é caracterizado como *move-related* ou *non-move-related*. Um nodo *move-related* é aquele que contém a origem ou o destino de uma instrução *move*.

Simplifique colore o grafo usando uma heurística simples. Suponha um grafo `G` que contém um nodo `M` com menos de `K` vizinhos, onde `K` é a quantidade de registradores da arquitetura alvo. Suponha também que `G'` representa o grafo `G - M` obtido removendo `M` de `G`. Se `G'` pode ser colorido, então `G` também pode, quando `M` é adicionado ao grafo colorido `G'`, os vizinhos de `M` possuem pelo menos `K-1` cores então uma cor livre pode ser encontrada para `M`. Isto resulta em um algoritmo recursivo: *repetidamente são removidos nodos com graus menores que `K`*. Cada simplificação reduzirá o grau dos outros nodos, gerando mais oportunidades para simplificações.

Combine realiza uma combinação (*coalescing*) conservativa para reduzir o grafo obtido na fase anterior. Desde que o grau de vários nodos foram reduzidos na fase *simplifique*, a estratégia conservativa é encontrar uma quantidade maior de instruções (*moves*) para combinar, do que a quantidade existente no grafo de interferência inicial. Após dois nodos serem combinados (e a instrução *move* ser eliminada), se o resultante nodo não for do tipo *move-related* ele é disponibilizado para a próxima rodada da fase *simplifique*. *Simplifique* e *combine* são repetidas até apenas nodos de graus significantes ou nodos do tipo *move-related* permanecerem.

Congele. Se nem *simplifique* nem *combine* podem ser aplicadas, o algoritmo procura por nodos do tipo *move-related* de baixo grau. Esta fase *congela* os *moves* nos quais este nodo é envolvido: *o objetivo é combinar estes nodos*. Isto causa o nodo (e talvez outros nodos relacinados aos *moves* congelados) ser considerado como um

nodo *non-move-related*, o que possibilitará mais simplificações. Neste momento, *simplifique* e *combine* são retomadas.

Derrame. Se durante a simplificação o grafo possui apenas graus significantes (nodos de graus >= K), então a heurística de simplificação falha, e algum nodo é marcado para derramar (*spilling*). Este nodo é então representado na memória e não em registrador durante a execução do programa. O nodo escolhido é aquele que não interfere com os outros nodos que permanecerão no grafo. Este é removido do grafo e armazenado em uma pilha. Após este processo, a fase *simplifique* continua.

Selecione atribui cores as nodos do grafo. Partindo do grafo vazio, o algoritmo constrói o grafo original adicionando repetidamente o nodo do topo da pilha, neste momento cada nodo recebe uma determinada cor (ou registrador).

Quando nodos que potencialmente serão colocados em memória são retirados da pilha não existe garantia que eles serão coloridos: *seus vizinhos no grafo já podem estar coloridos com K cores*. Neste caso, o nodo será realmente representado em memória. Nenhuma cor é atribuída a este nodo, contudo a fase *selecione* continuará para identificar outros nodos que também serão representados em memória.

Em alguns casos, os vizinhos de um potencial nodo possuem a mesma cor e além disto existe no grafo uma quantidade menor que K cores, então este nodo pode ser colorido não tendo uma representação em memória.

Se a fase *selecione* não é capaz de encontrar uma cor para um nodo, então o programa é reescrito para buscá-lo da memória imediatamente antes de cada *uso*, e armazená-lo na memória após cada *definição*. Desta maneira, um registrador temporário representado em memória será transformado em diversos novos temporários com pequenas faixas vivas. Estes poderão interferir com os temporários no grafo original. Portanto, o algoritmo é então repetido para o programa reescrito. Este processo continua até *simplifique* suceder com nenhuma representação em memória.

5.3.7 O Otimizador *Peephole*

O otimizador *peephole* busca a seqüência de instruções que pode ser trocada por outra mais eficiente, para uma arquitetura particular. Em geral, o objetivo do otimizador *peephole* é diminuir o tamanho do código gerado e conseqüentemente melhorar o desempenho do programa. Esta otimização é aplicada após a geração de código específico para a arquitetura alvo.

O algoritmo empregado em YAPc é padrão. Ele percorre as instruções de máquina procurando oportunidades para trocar um conjunto de instruções por uma seqüência melhor. Percorrer a seqüência de instruções geradas significa mover uma *janela* sobre ela. Esta *janela* define a quantidade de instruções que serão inspecionadas.

5.3.8 O Gerador de Código

A função do gerador de código é emitir código binário para a arquitetura Intel. Como os outros algoritmos implementados em YAPc, o gerador de código é bem simples e possui um tempo linear.

Inicialmente esta fase do compilador seria implementada através de chamadas ao sistema. Contudo, com o objetivo de deixar o sistema o mais simples e leve possível, foi desenvolvido o seguinte algoritmo de três passos:

1. *Leia uma instrução*.
2. *Gere código binário para ela*.
3. *Se existirem mais instruções volte ao passo 1*.

Após o gerador de código gerar o código binário para uma determinada cláusula do predicado que excedeu seu contador, o código gerado é instalado no ambiente de execução.

Na inicialização do sistema, o ambiente de execução cria na memória a *área de códigos nativos*, onde são armazenadas as versões otimizadas dos predicados. Instalar um código nativo para posterior execução, é o processo de atualizar a estrutura do predicado indicando que para este predicado existe uma *versão otimizada* na *área de códigos nativos*. Em posteriores chamadas, o sistema pode executar apenas código nativo.

5.4 Execução de Código Otimizado

O mecanismo para selecionar o código otimizado é simples. Ele é descrito no algoritmo 5.13.

Durante a execução da cláusula, o interpretador verifica se existe para esta cláusula uma versão otimizada. Se a versão otimizada existe, o ambiente de execução faz uma chamada a esta versão. Caso contrário, o ambiente de execução verifica o valor do contador e a quantidade de instruções YAAM do predicado para determinar se este predicado

```
1  foreach instruções da cláusula C do
2      switch tipo da instrução do
3          case get_list
4              ...;
5          end
6          case native_me
7              if C.native then          /* Se cláusula já otimizada */
8                  (void)(*func)(void);
9                  &func = C.native;    // Execução de código nativo.
                   func();
10             else
11                 if P.contador > Limite E C.tamanho >= TamLimite then /* Se
                   cláusula excedeu limite */
12                     code = YAPc(clausula);          // YAPc otimiza a
                       cláusula.;
13                     P.native = code;          // Instalação do código
                       otimizado.
14                 else
15                     P.contador++;    // Incremento do contador do
                       predicado.
16                 end
17             end
18         end
19         case ...
20             ...;
21         end
22     end
23 end
```

Algoritmo 5.13: Código do interpretador.

precisa ser otimizado. Se o contador do predicado excedeu o limite e a quantidade de instruções YAAM for maior que *TamLimite*, o ambiente de execução chama o compilador otimizador para gerar uma versão otimizada do predicado. Por outro lado, se o contador do predicado não excedeu o limite, o ambiente de execução simplesmente incrementa o contador do predicado e continua executando a versão não otimizada do predicado.

O retorno do código nativo pode ocorrer devido a três casos:

1. Uma saída normal.
2. Uma falha.
3. Ou na ocorrência de uma exceção.

No caso de um retorno devido a ocorrência de uma exceção, o estado do ambiente é restaurado e o interpretador trata a exceção. A próxima seção fornece mais detalhes sobre o tratamento de exceções.

A versão não otimizada do predicado não é descartada, pois o código otimizado poderá ser desotimizado, quando existir a necessidade de depuração. Neste caso, o ambiente de execução torna a executar a versão não otimizada. Para este fim, o ambiente altera a estrutura do predicado para indicar que uma cláusula específica deve ser executada na versão não otimizada. Além disto, o estado do ambiente de execução é restaurado e uma parte da computação é refeita.

5.5 Tratamento de Exceções

Exceções podem aparecer no código compilado se argumentos são chamados com uma instanciação inesperada ou com tipos inesperados. Felizmente, o algoritmo utilizado pelo YAPc é muito simples, porque este sistema compila código puro que tem *transparência referencial*. Em outras palavras, nós assumimos que se o código Prolog suceder para G com as ligações θ, então o código sucede para $G\theta$. Isso significa que o ambiente pode executar o código compilado com G, e depois o interpretador com $G\theta$, obtendo os mesmos resultados.

Existem dois problemas técnicos. Primeiro a instrução de corte é problemática porque se perde a transparência referencial. O programa:

```
a(X) :- var(X), !, X=2, exception.
a(X).
```

poderia gerar problemas porque a primeira chamada (compilada) colocaria $X = 2$. Contudo, na segunda chamada (interpretada), o interpretador usaria $a(2)$ que entraria pela segunda cláusula sucedendo.

A solução atual é não compilar a instrução de corte, a não ser que se prove que o código antes desta instrução não instancia os argumentos de entrada. Nesse caso, o código é seguro [101]. Alternativamente, o usuário pode declarar esta instrução como segura.

Um segundo problema é o fato da YAAM reusar registradores. Por exemplo, no código da figura 5.14 o registrador `A1` é destruído antes de se unificar com `A3`. Se o código tiver que voltar para o interpretador (ou seja, porque `A3` foi chamado instanciado e se

esperava que ele estivesse livre), o interpretador encontraria `A1` apenas com a cauda da lista e `A3` com o argumento original, resultando em execução errada. Portanto, tem que se ter cuidado para que o interpretador encontre sempre chamadas consistentes.

Há pelo menos duas soluções para esse problema. Primeiro, a instrução *native_me* copia os registradores da YAAM antes de chamar o código compilado, e se houver uma exceção esses resgistradores são restaurados. Ou, o código compilado garante que os registradores externos são consistentes em todos os pontos de uma possível exceção. Esta solução é similar ao que é usado em outros sistemas JIT [88, 112]. No exemplo, o registrador `A1` seria guardado na pilha e recuperado se `A3` faltar. Alternativamente, o registrador `A1` não seria destruído pelo código compilado, em vez disso, os registradores de trabalho seriam apenas copiados para registradores YAAM no retorno para código interpretado.

A primeira solução é mais simples de implementar, mas a segunda solução diminui o *overhead* na interface dos dois sistemas. Contudo, nesta primeira versão optou-se por implementar a primeira solução e deixar a segunda como um trabalho futuro.

5.6 Exemplo de Compilação

A figura 5.14 descreve o programa exemplo, que será utilizado nesta seção para demonstrar passo a passo o processo de compilação.

```
append([ ], L, L).
append([X|L1], L2, [X|L3]) :-
               append(L1, L2, L3).
```

(a) código Prolog do programa exemplo

```
name            append/3,3
get_atom        [ ], A1
get_val         X2, A1
proceed

name            append/3,3
get_list        A1
unify_var       X4
unify_last_var  X1
get_list        A3
unify_val       X4
unify_last_var  X3
execute         append/3
```

(b) código YAAM do programa exemplo

Figura 5.14: Código do programa *append*.

Em um determinado momento da execução da pergunta:

```
?- append([1,2,3,...,9999],[],L)
```

o predicado *append* excederá o limite do contador.

Neste momento, o interpretador está executando a segunda cláusula do programa *append*, pois o valor do primeiro argumento é uma lista não vazia. Portanto, YAPc verificará se esta cláusula deve ser otimizada.

Como o número de instruções YAAM da segunda cláusula excede *TamLimite*, YAPc irá gerar uma versão otimizada da segunda cláusula. Este processo pode ser visto na figura 5.15.

Após, YAPc gerar uma versão otimizada da segunda cláusula, o ambiente de execução instala esta nova versão. A partir deste momento, YAP+ executa a versão otimizada da segunda cláusula.

A compilação detalhada da segunda cláusula do programa *append* é mostrada nas figuras 5.16, 5.17, 5.18 e 5.19.

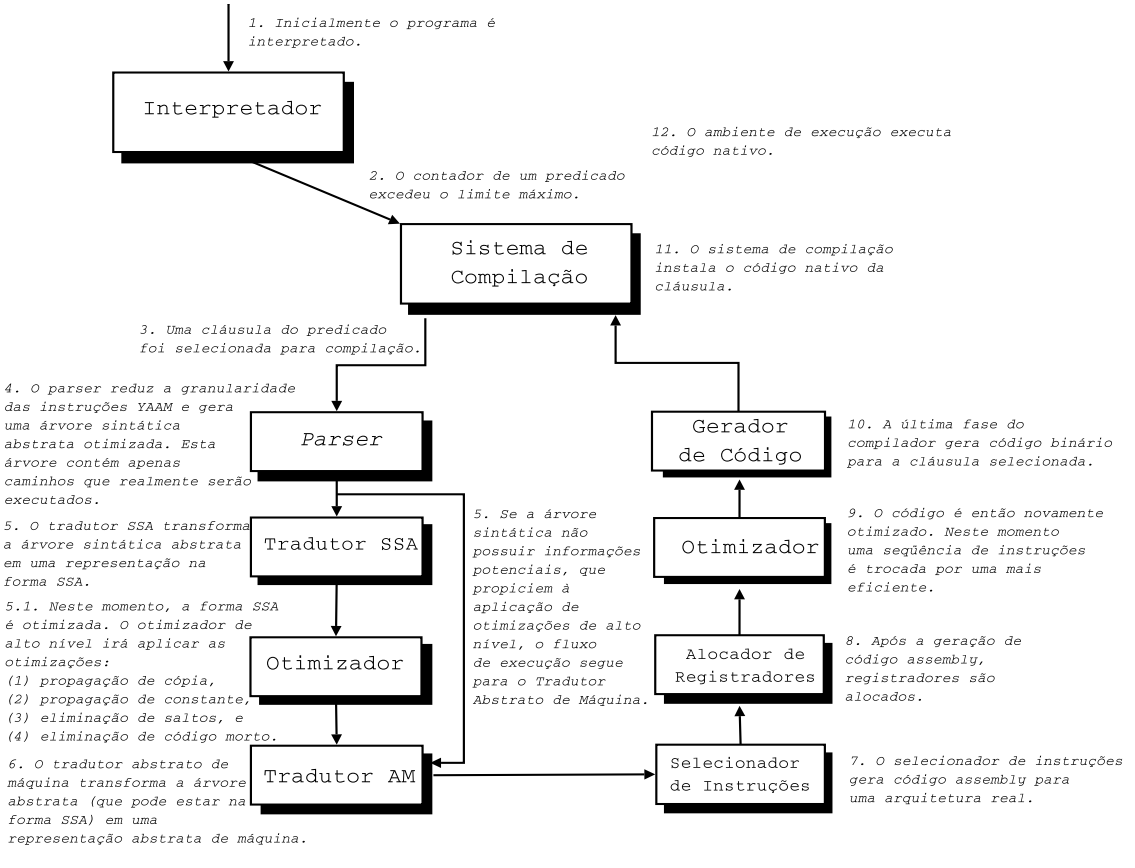
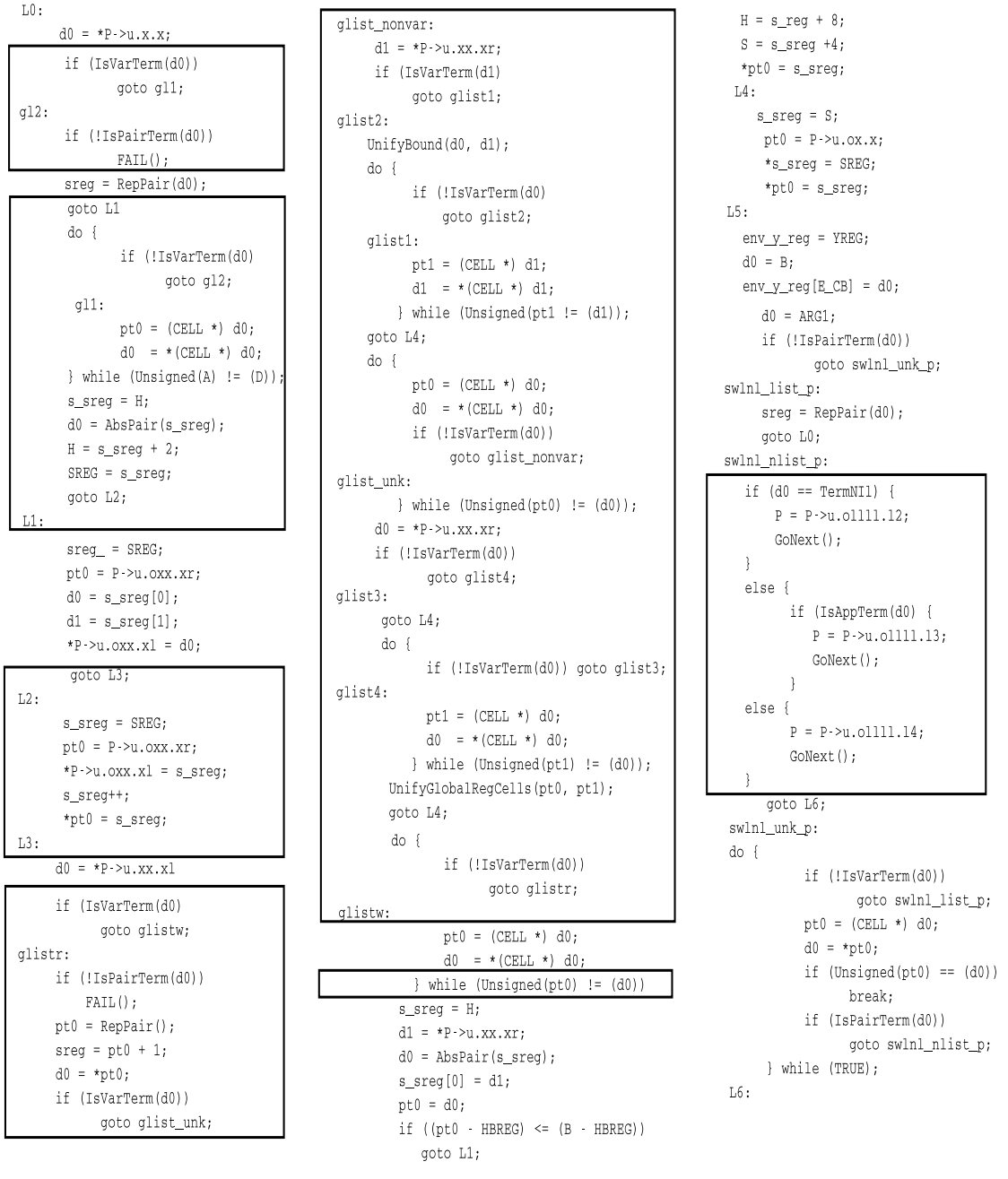


Figura 5.15: Ambiente de compilação dinâmica.



Este código representa a parte do emulador que executa a cláusula que foi selecionada para compilação.
No momento em que o parser inicia sua execução, ele inspeciona o tipo dos argumentos da cláusula para determinar quais porções deste código serão efetivamente executadas. Portanto, o parser fornece à próxima fase do compilador, uma árvore abstrata sintática contendo apenas caminhos executáveis.

É fácil perceber que uma significante porção do código será eliminada nesta fase. Isto torna o processo de compilação mais efetivo.

Nota: o código marcado é a porção eliminada.

Figura 5.16: Compilação do programa *append* - execução do *parser*.

```
L0:
     d0 = *P->u.x.x;
     sreg = 3 - d0;
     s_sreg = sreg;
     pt0 = P->u.oxx.xr;
     d0 = *s_sreg;
     d1 = *(s_sreg + 4);
     *P->u.oxx.xl = d0;
     *pt0 = d1;
     d0 = *P->u.xx.xl
     pt0 = d0;
     d0 = *d0;
     s_sreg = H;
     d1 = *P->u.xx.xr;
     d0 = 3 + s_sreg;
     *s_sreg = d1;
     *pt0 = d0;
     if ((pt0 - HBREG) <= (B - HBREG))
             goto L1;
     H = s_sreg + 8;
     S = s_sreg + 4;
L1:
     s_sreg = S;
     pt0 = P->u.ox.x;
     *s_sreg = sreg;
     *pt0 = s_sreg;
     env_yreg = Y;
     d0 = B;
     (env_y_reg + (-12)) = d0;
     d0 = ARG1;
     if ((d0 & 3) != 3)
         goto L2;
L3:
     sreg = 3 - d0;
     goto L0;
L4:
     goto L5;
L2:
     if (d0 & 3) != 0)
        goto L4;
     pt0 = d0;
     d0 = *pt0;
     if (pt0 == d0)
        goto L5;
     if ((d0 & 3) == 3)
        goto L3;
     goto L2;
L5:
     return 0;
```

(a) após parser

```
LABEL(L0)
MOVE(T0, MEM(8144184))
MOVE(SREG, (3 - T0))
MOVE(T1, SREG)
MOVE(T2, 8144184)
MOVE(T0, MEM(T1))
MOVE(T3, MEM(T1 + 4))
MOVE(MEM(8144190), T0)
MOVE(MEM(T2), T3)
MOVE(T0, MEM(814418C))
MOVE(T2, T0)
MOVE(T1, H)
MOVE(T3, MEM(8144190))
MOVE(T0, (3 + T1))
MOVE(MEM(T1), T3)
MOVE(MEM(T2), T0)
CJUMP(((T2 - HB) LE (B - HB)), L1, L7)
LABEL(L7)
MOVE(H, (T1 + 8))
MOVE(S, (T1 + 4))
LABEL(L1)
MOVE(T1, S)
MOVE(T2, 814418C)
MOVE(MEM(T1), T1)
MOVE(MEM(T2), T1)
MOVE(T4, Y)
MOVE(T0, B)
MOVE(MEM(T4 + (-12)), TO)
MOVE(T0, ARG1)
CJUMP(((T0 & 3) NE 3), L2, L8)
LABEL(L8)
LABEL(L3)
MOVE(S, (3 - T0))
JUMP(L0)
LABEL(L4)
JUMP(L5)
LABEL(L2)
CJUMP(((T0 & 3) NE 0), L4,  L9)
LABEL(L9)
MOVE(T2, T0)
MOVE(T0, MEM(T2))
CJUMP((T2 E T0), L5, L10)
LABEL(L10)
CJUMP(((TO & 3) E 3), L3, L11)
LABEL(L11)
JUMP(L2)
LABEL(L5)
MOVE(%EAX, 0)
```

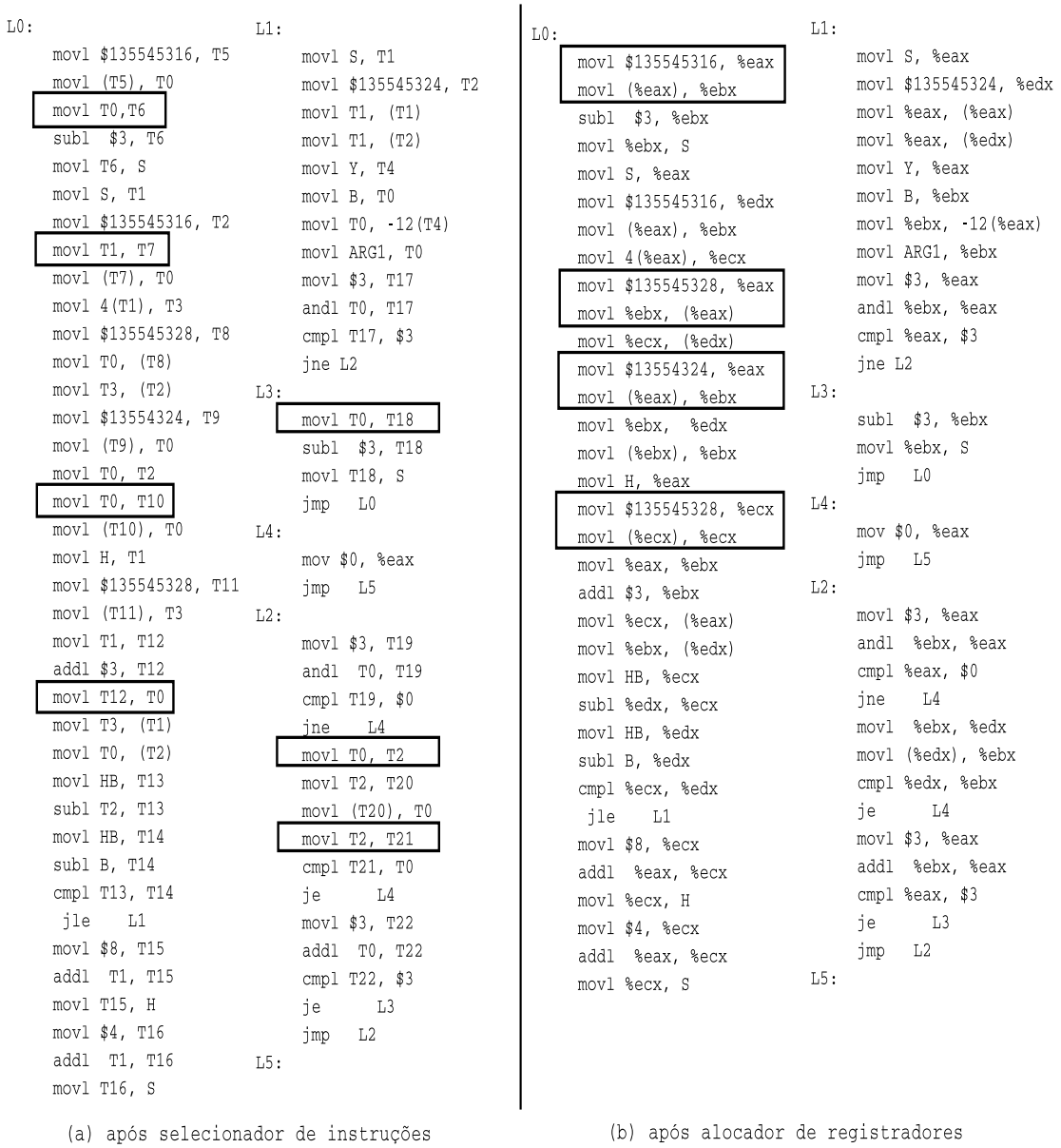
(b) após tradutor abstrato de máquina

Como pode ser observado nesta figura, o código gerado pelo parser é um código otimizado que contém apenas caminhos executáveis. Durante a criação da árvore sintática abstrata anotada, o parser realizou duas otimizações:

1. integração de bibliotecas nativas, e
2. eliminação de chamada recursiva.

Além disto, durante a criação da árvore sintática abstrata, o parser decidiu que não existem oportunidades à aplicação das otimizações de alto nível. Desta maneira, o fluxo de execução segue no selecionador de instruções.

Figura 5.17: Compilação do programa *append* - transição entre as fases: *parser*, *tradutor abstrato de máquina* e *selecionador de instruções*.



O selecionador de instruções gera código nativo para a arquitetura Intel. Após as instruções da arquitetura alvo serem geradas, o alocador de registradores atribui registradores reais aos temporários.

O compilador YAPc explora ao máximo a arquitetura alvo. Ao invés de armazenar todas as variáveis no "frame", YAPc tenta armazenar todas as variáveis do programa em registradores. Contudo, em alguns casos isto não é possível. Neste caso, YAPc gerará "spills" e "fetchs" para as variáveis que serão representadas em memória.

O programa append é um caso ótimo, onde nenhuma variável precisou ser representada em memória.

Nota: as instruções marcadas serão removidas (ou combinadas) na fase posterior.

Figura 5.18: Compilação do programa *append* - transição entre as fases: *selecionador de instruções*, *alocador de registradores* e *otimizador peephole*.

```
L0:
     movl 0x81441E4, %ebx
     subl  $3, %ebx
     movl %ebx, S
     movl S, %eax
     movl $135545316, %edx
     movl (%eax), %ebx
     movl 4(%eax), %ecx
     movl %ebx, (0x81441F0)
     movl %ecx, (%edx)
     movl 0x81441EC, %ebx
     movl %ebx,  %edx
     movl (%ebx), %ebx
     movl H, %eax
     movl 0x81441F0, %ecx
     movl %eax, %ebx
     addl $3, %ebx
     movl %ecx, (%eax)
     movl %ebx, (%edx)
     movl HB, %ecx
     subl %edx, %ecx
     movl HB, %edx
     subl B, %edx
     cmpl %ecx, %edx
      jle    L1
     movl $8, %ecx
     addl  %eax, %ecx
     movl %ecx, H
     movl $4, %ecx
     addl  %eax, %ecx
     movl %ecx, S
```

```
L1:
     movl S, %eax
     movl $135545324, %edx
     movl %eax, (%eax)
     movl %eax, (%edx)
     movl Y, %eax
     movl B, %ebx
     movl %ebx, -12(%eax)
     movl ARG1, %ebx
     movl $3, %eax
     andl %ebx, %eax
     cmpl %eax, $3
     jne L2
L3:
     subl  $3, %ebx
     movl %ebx, S
     jmp   L0
L4:
     mov $0, %eax
     jmp   L5
L2:
     movl $3, %eax
     andl  %ebx, %eax
     cmpl %eax, $0
     jne    L4
     movl  %ebx, %edx
     movl (%edx), %ebx
     cmpl %edx, %ebx
     je      L4
     movl $3, %eax
     addl  %ebx, %eax
     cmpl %eax, $3
     je      L3
     jmp   L2
L5:
```

(a) após otimizador peephole

```
55
89 E5
8B 1D E4 41 14 08
83 EB 03
89 1D E8 6C 14 08
A1 E8 6C 14 08
BA E4 41 14 08
8B 18
8B 48 04
89 1D F0 41 14 08
89 0A
8B 1D EC 41 14 08
89 DA
8B 1B
A1 D0 6C 14 08
8B 0D F0 41 14 08
89 C3
83 C3 03
89 08
89 1A
8B 0D C4 6C 14 08
29 D1
8B 15 C4 6C 14 08
2B 14 D4 6C 14 08
39 D1
7E 1A
B9 08 00 00 00 00
01 C1
89 0D D0 6C 14 08
B9 04 00 00 00
01 C1
89 0D E8 6C 14 08
```

```
A1 E8 6C 14 08
BA EC 41 14 08
89 00
89 02
A1 E4 6C 14 08
8B 1D D4 6C 14 08
B8 03 00 00 00
21 D8
83 F8 03
75 15
83 EB 03
89 1D E8 6C 14 08
E9 4E FF FF FF
B8 00 00 00 00
EB 22
B8 03 00 00 00
21 D8
83 F8 00
75 ED
89 DA
8B 1A
39 DA
74 E5
B8 03 00 00 00
21 D8
83 F8 03
74 CB
EB DE
C9
C3
```

(b) após gerador de código

O otimizador peephole ao otimizar o código trocou algumas seqüências de instruções por uma única instrução. Ao final do processo de compilação, YAPc gerou um código nativo que pode ser considerado ótimo, contendo apenas caminhos executáveis e sem instruções redundantes. Finalmente, o código nativo é instalado. Após o código ser instalado, o ambiente de execução não mais interpretará esta cláusula do predicado, simplemente fará uma chamada ao seu código nativo.

Figura 5.19: Compilação do programa append - transição entre as fases: *otimizador peephole* e *gerador de código*.

Capítulo 6

Avaliação de *Just-In-Time* para Prolog

Este capítulo apresenta uma avalição do desempenho do sistema de compilação desenvolvido. Para isto, são apresentados e analisados os resultados obtidos pela execução de um conjunto de programas de teste. Todos os resultados foram obtidos em um Pentium(R) Intel(R) 4 CPU 2.80 GHz, com 1GB de memória, rodando em Linux.

A avaliação é dividida em três partes:

1. ***Avaliar o desempenho do YAP+ comparando-o com o YAP***: durante esta avaliação o ambiente YAP+ possui todas as suas características habilitadas: *seleção de regiões e otimizações*. Esta avaliação possui como objetivo *identificar a aceleração obtida através de chamadas a código nativo*.

2. ***Avaliar o compilador otimizador YAPc***: esta avaliação tem por objetivo identificar o impacto do compilador ao ambiente e o desempenho de cada fase do compilador.

3. ***Avaliar o desempenho do YAP+ comparando-o com os sistemas CIAO e Mercury***: por fim se objetiva comparar o sistema desenvolvido com outros sistemas.

Inicialmente é descrito o conjunto de programas teste, em seguida é apresentado a avaliação do sistema e o capítulo encerra com algumas considerações gerais.

6.1 Programas de Teste

O conjunto de programas usado é constituído por *kernels* de programas completos. Estes *kernels* são habitualmente usados pela comunidade científica do Prolog, tendo eles sido usados na avaliação de outros sistemas, como por exemplo Aquarius [117] e Parma [114]. Segue-se uma breve descrição de cada um dos programas de teste usados (no apêndice é apresentado o código Prolog e o código YAAM).

- ***append***: inserir uma lista de `N` elementos no início de outra.
- ***naive reverse***: inverter uma lista com `N` elementos.
- ***zebra***: quebra cabeça lógico baseado em restrições que é resolvido por procura exaustiva.
- ***hanoi***: torre de *Hanoi* contendo 3 pinos e `N` discos.
- ***quick sort***: ordenar uma lista com `N` elementos pelo método *quick-sort*.
- ***tak***: programa altamente recursivo. A recursão é controlada por operações inteiras.

A tabela 6.1 descreve as características de cada programa, em termos da quantidade de predicados, quantidade de cláusulas por predicado e a quantidade de instruções YAAM de cada cláusula. O apêndice descreve o código completo de cada programa teste.

Programa	**Predicados**	**Cláusulas**	
		Cabeça	**Intruções**
Append	*append*	append([],L,L)	4
		append([X|L1],L2,[X|L3])	8
Naive Reverse	*nreverse*	nreverse([],[])	4
		nreverse([X|L0],L)	15
	append	append([],L,L)	4
		append([X|L1],L2,[X|L3])	8
Zebra	*houses*	houses	44
	right_of	right_of(A, B, [B, A | _])	7
		right_of(A, B, [_ | Y])	5
	next_to	next_to(A, B, [A, B | _])	7
		next_to(A, B, [B, A | _])	7
		next_to(A, B, [_ | Y])	5
	my_member	my_member(X, [X|_])	5
		my_member(X, [_|Y])	5
Hanoi	*han*	han(N,_,_,_)	6
		han(N,A,B,C)	22
Quick Sort	*qsort*	qsort([X|L],R,R0)	22
		qsort([],R,R)	4
	partition	partition([X|L],Y,[X|L1],L2)	11
		partition([X|L],Y,L1,[X|L2])	8
		partition([],_,[],[])	5
Tak	*tak*	tak(X,Y,Z,A)	6
		tak(X,Y,Z,A)	36

Tabela 6.1: Características dos programas.

6.2 YAP+ versus YAP

Esta avaliação possui como objetivo *identificar a aceleração obtida através de chamadas a código nativo*, para este fim foi utilizada a seguinte configuração:

- Cada programa teste foi invocado 100.000 vezes. Portanto, o tempo de execução para cada programa é $100.000 \times tempo\ do\ programa$.
- O tempo de execução mostrado é a média entre cinco execuções.
- O limite de invocação para invocação da compilação dinâmica foi de 1.000 chamadas.
- Todas as otimizações estavam ligadas para avaliar o desempenho total.
- O coletor de lixo estava ligado.

A tabela 6.2 apresenta o tamanho da entrada para cada programa teste.

Programa	**Entrada**
Append	1000 elementos
Naive Reverse	1000 elementos
Zebra	5 casas
Hanoi	7 discos
Quick Sort	1000 elementos
Tak	x=12, y=7 e z=5

Tabela 6.2: Entrada dos programas.

Na tabela 6.3 é apresentado os tempos que cada programa de teste obteve nos ambientes YAP e YAP+. Esta tabela também apresenta a aceleração obtida utilizando código nativo. Quanto maior for a aceleração melhor é o desempenho.

Com os dados apresentados na tabela 6.5 pode-se concluir que o ambiente de excução obtém uma aceleração média de 7 ao utilizar YAPc. Para nenhum programa houve uma queda no desempenho.

Como esperado, executar código nativo obtém um melhor desempenho comparado com interpretar um conjunto de instruções. Executar código nativo obteve uma aceleração que variou entre 4,35 e 10,20.

O tempo gasto pelo sistema de compilação, para os programas testes, não ultrapassa 40 milisegundos. Conseqüentemente, para estes programas o sistema de compilação não impacta o ambiente de execução.

Programa	Tempo de Execução		Aceleração
	YAP	YAP+	
Append	37,49	8,62	4,35
Naive Reverse	672,81	66,61	10,10
Zebra	294,65	44,78	6,58
Hanoi	182,36	17,88	10,20
Quick Sort	203,79	32,82	6,21
Tak	120,17	25,73	4,67

Tabela 6.3: Tempo de execução das aplicações em segundos e a aceleração obtida utilizando o compilador YAPc.

A figura 6.2 apresenta as quatro componentes que compõem o tempo relativo de execução:

1. *tempo executando código nativo*,
2. *tempo executando código interpretado*,
3. *tempo gasto pelo coletor de lixo*, e
4. *tempo gasto pelo compilador*.

Nesta figura a barra da esquerda representa o tempo de execução obtido por YAP, enquanto a barra da direita o tempo de execução obtido por YAP+.

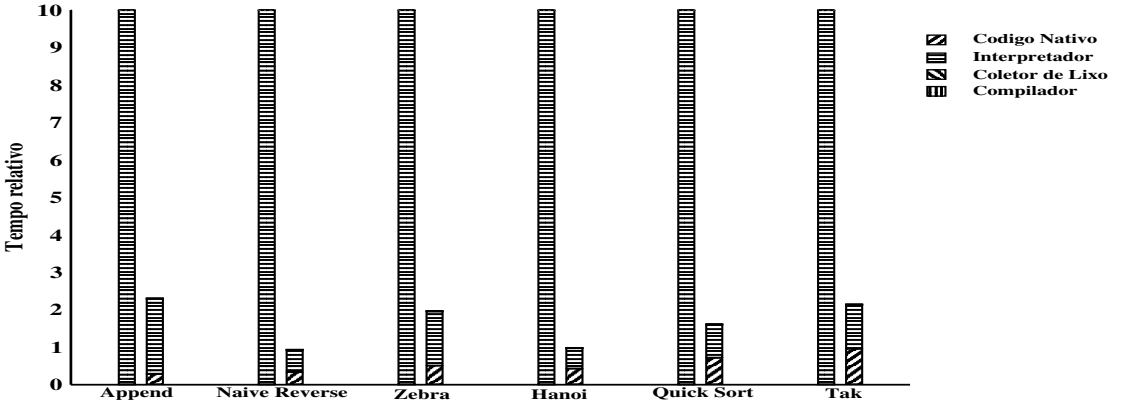


Figura 6.1: Tempo de execução normalizado de YAP e YAP+.

Como pode ser observado na figura , o tempo de execução dos programas é dominado pelo tempo gasto pelo interpretador e pela execução de código nativo. O tempo gasto pelo compilador e pelo coletor de lixo são desprezíveis.

Como dito anteriormente, não existe impacto do sistema de compilação ao ambiente de execução para tais tempo de execução. Para avaliar o impacto do sistema de compilação é necessário diminuir o tempo de execução dos programas, pois um tempo de execução alto tende a esconder o tempo gasto pelo compilador. A próxima seção utiliza uma configuração diferente para os programas, como o objetivo de avaliar o impacto do compilador.

6.3 Impacto do YAPc no Ambiente de Execução

Nesta avaliação foi utilizada a seguinte configuração:

- Cada programa teste foi invocado apenas uma vez.
- O tempo de execução mostrado é a média entre cinco execuções.
- Cada programa teste foi instrumentado para coletar informações de *hardware*. Foi utilizado a biblioteca Program Counter que coleta informações de *hardware* durante a execução do programa.
- O limite de invocação para invocação da compilação dinâmica foi de 1.000 chamadas.
- Todas as otimizações estavam ligadas para avaliar o desempenho total.
- Para avaliar apenas uma otimização, todas as outras estavam desligadas.
- O coletor de lixo estava ligado.

Programa	**Entrada**
Append	100.000 elementos
Naive Reverse	5.000 elementos
Zebra	5 casas
Hanoi	20 discos
Quick Sort	100.000 elementos
Tak	x=18, y=12 e z=6

Tabela 6.4: Entrada dos programas.

A tabela 6.4 apresenta o tamanho da entrada para cada programa teste. A diferença na configuração tem por objetivo reduzir o tempo de execução. Isto expõe o tempo de compilação, e conseqüentemente avalia de forma mais justa o impacto do compilador. O ambiente com esta configuração é o pior caso para YAPc.

6.3.1 Desempenho do YAP+

Na tabela 6.5 é apresentado os tempos que cada programa de teste obteve nos ambientes YAP e YAP+. Esta tabela também apresenta a aceleração obtida utilizando código nativo. Quanto maior for a aceleração melhor é o desempenho.

Programa	**Tempo de Execução**		**Aceleração**
	YAP	**YAP+**	
Append	10,80	5,00	2,16
Naive Reverse	704,40	222,40	3,17
Zebra	4,00	38,58	**-9,64**
Hanoi	504,80	109,86	4,59
Quick Sort	321,20	104,07	3,09
Tak	15,00	16,53	**-1,10**

Tabela 6.5: Tempo de execução das aplicações em milisegundos e a aceleração obtida utilizando o compilador YAPc.

Como pode ser observado, o tempo de execução dos programas de teste é muito reduzido. Isto facilita verificar o impacto do sistema de compilação ao ambiente de execução, pois programas com altos tempo de execução tendem a esconder o processo de compilação.

Com os dados apresentados na tabela 6.5 pode-se concluir que o ambiente de excução obtém uma aceleração média de 3,25 ao utilizar YAPc. Excluindo os programas *zebra* e *tak*, para os quais houve uma queda no desempenho.

Deve-se notar o fato do compilador ser bem leve. Embora o programa *append* obtivesse a menor aceleração, este é um caso de desempenho considerável. Observe que sendo a aplicação com um tempo de execução reduzido, o YAP+ ainda obteve aceleração. O mesmo não ocorre com a aplicação *tak*. Isto ocorre pelo fato das duas cláusulas do programa serem selecionadas para compilação. Enquanto para o programa *append* somente uma cláusula é selecionada para compilação, em *tak* o sistema de compilação selecionou para compilação as duas cláusulas que compõem o programa.

Como esperado, executar código nativo obtém um melhor desempenho comparado com interpretar um conjunto de instruções. Executar código nativo obteve uma aceleração que variou entre 2,16 e 4,59.

Estes dados mostram que as técnicas utilizadas, embora simples, são a melhor escolha para deixar o sistema de compilação leve, contudo eficiente. Gerar código apenas para uma parte do programa se mostrou uma ótima opção.

A figura 6.2 apresenta as quatro componentes que compõem o tempo relativo de execução:

1. *tempo executando código nativo*,
2. *tempo executando código interpretado*,
3. *tempo gasto pelo coletor de lixo*, e
4. *tempo gasto pelo compilador*.

Nesta figura a barra da esquerda representa o tempo de execução obtido por YAP, enquanto a barra da direita o tempo de execução obtido por YAP+.

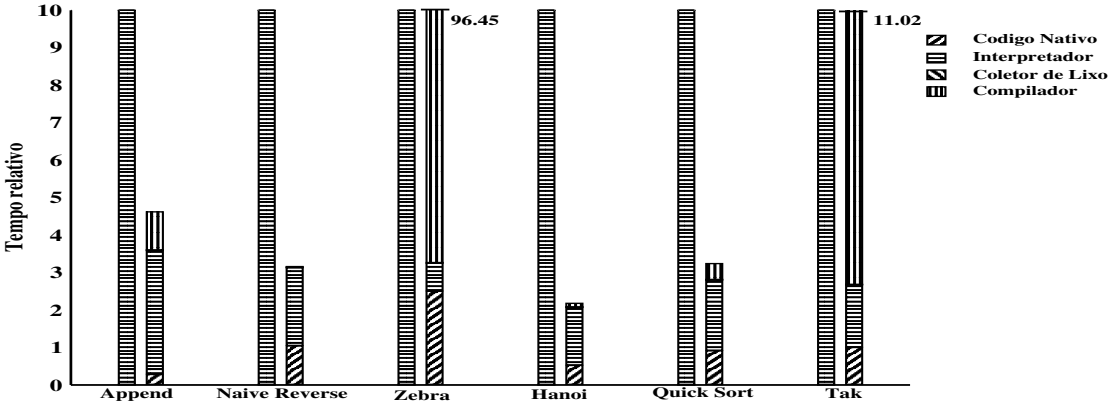


Figura 6.2: Tempo de execução normalizado de YAP e YAP+.

Para os casos nos quais YAP+ obteve um bom desempenho, o tempo de execução é dominado pelo tempo gasto pelo interpretador. Entre estes, o programa *append* é o que possui um tempo de compilação relativamente maior.

O programa *zebra* é o único caso onde a execução da aplicação passa mais tempo executando código nativo do que interpretando código.

O impacto do sistema de compilação ao ambiente de execução obteve um efeito negativo para os programas *zebra* e *tak*. Nestes dois programas, o sistema de compilação gasta 93,2% e 38,53% do tempo total de execução. Isto explica a falta de desempenho para estes programas.

6.3.2 Comparação Detalhada entre YAP e YAP+

Para um estudo detalhado, os programas foram instrumentados através da biblioteca *Program Counter Library* [94] para coletar as seguintes informações de hardware:

- *quantidade de instruções de hardware*;
- *acessos à memória*;
- *falhas na cache*; e
- *instruções por ciclo (IPC)*

A tabela 6.6 montra a comparação de baixo nível entre YAP e YAP+. É interessante comparar o número de instruções executadas e o número de acessos à memória em ambos os ambientes. YAP geralmente executa um maior número de instruções e possui um número maior de acessos à memória para a maioria dos programas. Como pode ser observado, em YAP+ a tendência é reduzir o número de instruções executadas, como também o número de acessos à memória. O melhor exemplo deste caso é o programa *append*, onde YAP+ reduziu por um fator de 8 o número de instruções executadas e de acessos à memória. Uma conseqüência da redução do número de acessos à memória é a redução do número de falhas na *cache*. É surpreendente YAP e YAP+ possuirem um IPC similar na maioria dos casos.

Programa	**YAP**			
	Instruções (10^9)	Acessos à Memória (10^7)	Falhas na *Cache* (10^7)	Instruções por Ciclo
Append	1,61	100,66	1,13	0,62
Naive Reverse	1,88	156,98	0,42	1,31
Zebra	1,61	107,37	1,95	0,66
Hanoi	1,96	52,01	1,42	0,41
Quick Sort	1,46	62,91	0,21	0,88
Tak	1,34	80,53	0,89	0,62
Programa	**YAP+**			
	Instruções (10^9)	Acessos à Memória (10^7)	Falhas na *Cache* (10^7)	Instruções por Ciclo
Append	0,20	13,42	0,15	0,62
Naive Reverse	1,34	90,59	0,40	1,31
Zebra	2,11	77,21	1,58	0,60
Hanoi	1,33	46,97	1,02	0,46
Quick Sort	0,59	36,90	0,13	0,87
Tak	1,48	87,24	2,36	0,44

Tabela 6.6: Impacto no *hardware* para os sistemas YAP e YAP+.

Em ambientes interpretados existe o *overhead* intrínseco da interpretação. Ambientes que utilizam um compilador dinâmico reduzem este *overhead* através da geração de código nativo. Este é o caso de YAP+, como apresentado nos resultados desta seção.

A redução do número de acessos à memória é devido também ao fato do compilador YAPc utilizar ao máximo os registradores de máquina, tentando eliminar ao máximo o número de acessos à memória.

Zebra e *Tak* são os dois programas nos quais YAP+ não obteve bom desempenho. O problema nestes casos (como descrito na seção 6.3.1) é devido ao tempo gasto pelo sistema de compilação. Principalmente para *zebra*, YAP+ gasta muito tempo na compilação. Nestes casos o aumento do número de instruções executadas e nos acessos à memória são decorrentes da execução do compilador. A tendência ocasionada pelo mau desempenho de YAP+ é não somente aumentar o número de instruções executadas, mas também o número de acessos à memória e falhas na *cache*, resultando numa redução do IPC.

6.3.3 Análise do Compilador YAPc

A tabela 6.7 apresenta o tempo gasto pelo compilador YAPc em milisegundos. O tempo de execução do compilador varia bastante, conforme as características do programa teste.

Programa	**Cláusula Compiladas**	**Tempo de Execução (milisegundos)**	**Código Gerado (bytes)**
Append	1	1,11	224
Naive Reverse	2	2,11	464
Zebra	5	37,28	4476
Hanoi	2	5,56	828
Quick Sort	3	14,13	1723
Tak	2	12,53	1427

Tabela 6.7: Dados utilizados na análise do compilador.

Os programas *append*, *hanoi* e *tak* possuem características similares. Contudo, no caso de *hanoi* e *tak* todas as cláusulas do programa foram compiladas. O que não ocorre em *append*, isto diminui o tempo de compilação, além deste possuir um código mais simples. *Hanoi* e *tak* são casos de aplicações com bastante retrocesso, nestas o único predicado atinge o limite estabelecido para acionar o sistema de compilação, quase simultaneamente para as duas cláusulas.

A figura 6.3 discrimina o tempo de execução gasto pelas fases do compilador. As fases são:

- *parser*,
- *tradutor SSA*,
- *otimizador de alto nível*,
- *tradutor abstrato de máquina*,
- *selecionador de instruções*,
- *alocador de registradores*,
- *otimizador peephole*, e
- *gerador de código*.

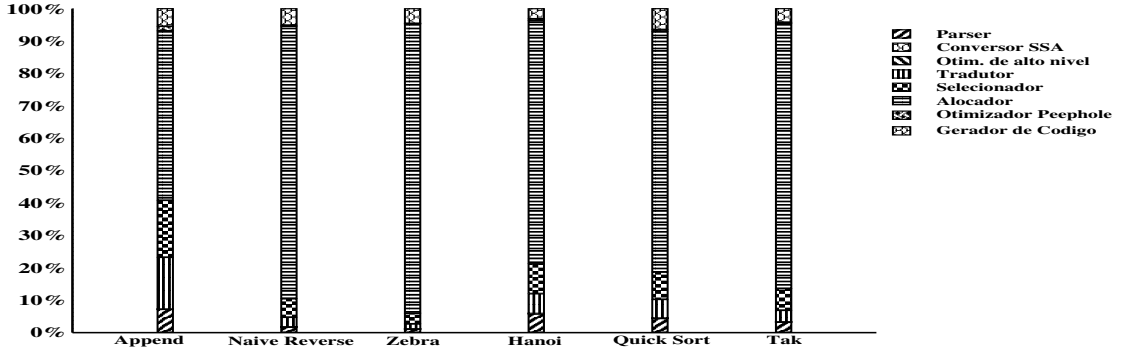


Figura 6.3: Percentual do tempo de execução das fases do compilador.

O processo de selecionar as regiões de código que realmente serão executadas e transformá-las em uma árvore sintática abstrata anotada não gasta mais que 10% do tempo de compilação. Isto mostra que a técnica de seleção de regiões não possui um impacto considerável ao sistema de compilação.

O fluxo de execução entre as fases do compilador passou do *parser* diretamente para o tradutor em todos os programas. Portanto, o compilador não aplicou nenhuma otimização de alto nível ao código dos programas. Isto ocorre porque as anotações da árvore sintática abstrata gerada pelo *parser*, para cada programa, informa que as características do programa não efetivam a aplicações das otimizações de alto nível. Desta forma, o compilador altera o fluxo de execução entre suas fases para se adaptar às características do programa.

Isto não indica que as otimizações implementadas não sejam aplicáveis à programas lógicos, apenas indica que não foram aplicáveis aos programas de teste.

Dois fatores explicam este fato. Primeiro, uma porção considerável de código foi eliminada na primeira fase, o que eliminou as chances da aplicação das otimizações de alto nível. Por exemplo, o programa *append* possui apenas 64 instruções de máquina, o que indica que a árvore sintática abstrata é bem pequena. Além disto, os programas no conjunto de teste não contém estruturas complexas. Contudo, muitos programas Prolog podem ter controle complexo.

A fase de tradução da árvore sintática abstrata anotada para uma representação abstrata de máquina e a fase de seleção de instruções consomem entre 5% e 15% do tempo de compilação. A última fase, geração de código nativo, não consome mais do que 5% do tempo de compilação.

O maior custo do compilador está na fase de alocação de registradores. O alocador de registradores consome entre 50,12% a 83,72% do tempo de compilação. Isto é um tempo muito alto, comparado com o tempo gasto pelas outras fases do compilador. O tempo gasto nesta fase poderia ainda ser mais alto se durante a execução do *parser* fosse verificado a viabilidade de se aplicar otimizações de alto nível, porque a representação SSA geralmente aumenta a pressão por registradores.

6.3.3.1 Análise da Técnica de Seleção de Regiões

Assim como a figura 6.2, a figura 6.4 possui o tempo de execução decomposto em quatro componentes, e representa o tempo relativo. O objetivo desta análise é mostrar o impacto da técnica de seleção de regiões ao ambiente.

Na figura 6.4 a barra a esquerda representa o tempo de execução obtido por YAP. A barra central representa o tempo de execução obtido por YAP+ sem a técnica de seleção de regiões. E a barra a direita representa o tempo de execução obtido por YAP+ com a técnica de seleção de regiões ligada.

Os dados mostram que programas com um tempo de execução superior a 300 milisegundos tendem a esconder o tempo gasto pelo sistema de compilação. Podemos concluir que o compilador é bem leve, pois 300 milisegundos é um tempo de execução bastante rápido para um compilador.

Por outro lado, para programas com tempo de execução inferior a 15 milisegundos a melhor solução é otimizar a menor quantidade de predicados possível, mais especificamente uma única cláusula de um predicado. Este é o caso de *append*. Possivelmente se ocorresse a mesma situação para *tak*, este obteria um melhor desempenho. Para estes casos a técnica de compilação dirigida, ou seleção de regiões mostrou-se mais efetiva.

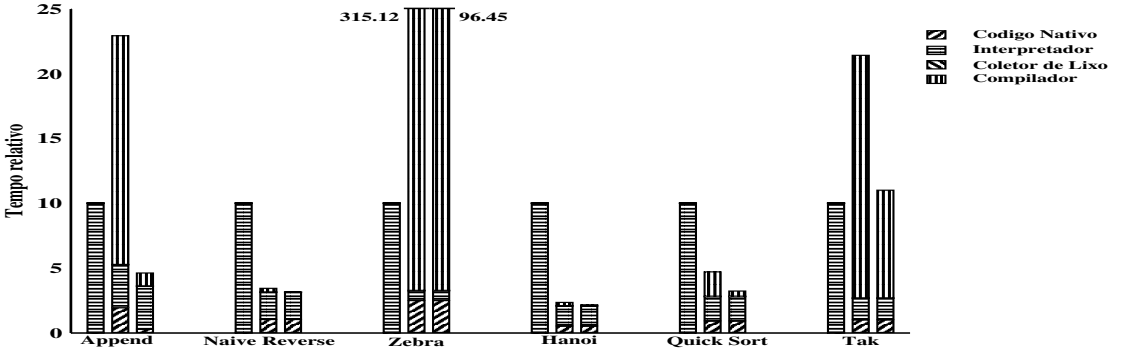


Figura 6.4: Tempo de execução normalizado com a tecnica de seleção de regiões ligado e desligada.

Quando menor o tempo de execução do programa, mais eficiente deve ser o processo de compilação.

A técnica de selecionar regiões acelerou o tempo de compilação de 2,25 a 17,22 vezes. Mesmo para os casos onde o nosso sistema não obteve bom desempenho, a técnica se mostrou eficiente.

6.3.3.2 Análise do uso da Forma SSA

Como mostrado anteriormente, o fluxo de execução do compilador não passa pela fase de tradução para a forma SSA. Contudo, é interessante medir o custo desta transformação. Para este fim, esta fase foi habilitada independentemente das decisões tomadas durante o *parser*.

A tabela 6.8 mostra o tempo de execução da conversão da árvore sintática abstrata para a forma SSA. Os valores desta tabela mostram que a transformação é muito rápida. Deve ser observado que o impacto no tempo de compilação não passa de 10% do tempo total. Esta fase é uma das mais rápidas do compilador.

Uma análise das otimizações de alto nível somente será possível utilizando outro conjunto de programas. A medida que o sistema desenvolvido for avaliado com programas maiores, espera-se que exista a possibilidade de medir o impacto das otimizações de alto nível. Além disto, poderá ser identificada a necessidade de implementar outras otimizações.

É significante não aplicar otimizações em alguns casos. Geralmente os sistemas de compilação dinâmica aplicam de forma indescriminada um conjunto de otimizações a qualquer programa em execução.

Este fato acarreta sérios problemas, como os descritos no capítulo 4. Nem sempre

Programa	Tempo de Execução do Tradutor (milisegundos)	Acréscimo no Tempo de Compilação
Append	0,07	6,31%
Naive Reverse	0,12	5,69%
Zebra	0,56	1,50%
Hanoi	0,27	4,86%
Quick Sort	0,54	3,81%
Tak	0,34	2,71%

Tabela 6.8: Tempo de execução do tradutor SSA.

aplicar uma otimização ocasionará em ganho de desempenho. Além disto, mesmo que uma otimização seja aplicável a um programa, para um bom ganho de desempenho os parâmetros da otimização devem ser ajustados às características do programa.

6.3.3.3 Análise da Integração de Funções Nativas

A tabela 6.9 mostra o impacto da integração de funções nativas. Esta tabela apresenta duas informações: *o percentual de aumento do tamanho do código e o percentual de redução do tempo de execução*.

Programa	Impacto	
	Tamanho do Código	Tempo de Execução
Append	18,75%	12,25%
Naive Reverse	19,04%	14,35%
Zebra	1,76%	0,87%
Hanoi	13,33%	9,12%
Quick Sort	3,84%	7,56%
Tak	4,82%	6,89%

Tabela 6.9: Impacto da integração de funções nativas.

A técnica da integração de funções nativas acelera o tempo de execução do programa, contudo tende a aumentar o tamanho do código gerado. Esta técnica aumentou de 1,76% a 20% o tamanho do código. O percentual de 20% é um aumento considerável do tamanho do código gerado. Contudo, o ganho de desempenho obtido através da integração é muito bom.

YAPc não deixa o tamanho do código crescer consideravelmente. A técnica de integração de funções nativas somente é aplicada para as funções nativas com poucas instru-

ções de *hardware*. Além disto, esta técnica somente é aplicada após a seleção de regiões.

A integração de funções nativas é a otimização mais efetiva para os programas de teste. Nos melhores casos, a integração de funções nativas acelera em até 10% o tempo de execução. *Zebra* é o único caso onde a integração de funções nativas não obteve um bom desempenho.

6.3.3.4 Análise de *Coalescing*

A tabela 6.10 mostra o impacto de remover instruções *move* desnecessárias. Esta otimização é realizada juntamente com a alocação de registradores. Esta tabela apresenta duas informações: *o percentual de redução do tamanho do código e o percentual de redução do tempo de execução*.

Programa	**Impacto**	
	Tamanho do Código	**Tempo de Execução**
Append	9,38%	10%
Naive Reverse	9,52%	7,52%
Zebra	2,12%	1,72%
Hanoi	5,33%	4,12%
Quick Sort	11,54%	9,7%
Tak	2,97%	1,02%

Tabela 6.10: Impacto de *coalescing*.

Coalescing possui um efeito contrário ao da integração de funções nativas em relação ao tamanho do código gerado. Enquanto a integração de funções aumenta o tamanho do código, *coalescing* reduz em até 10% o tamanho do código. Quanto maior a redução do tamanho do código, maior o impacto na aceleração e vice-versa.

Coalescing obteve um desempenho menor do que a integração de funções nativas. Enquanto para alguns casos a integração de funões nativas obteve um melhoramento superior a 10%, o mesmo não ocorre com *coalescing*. O melhor caso desta otimização é para o programa *append*, onde esta técnica acelera em 10% o tempo de execução. Assim como na integração de funções nativas, o programa *zebra* foi o que obteve o desempenho mais baixo.

6.3.3.5 Análise do Otimizador *Peephole*

A tabela 6.11 mostra o impacto da aplicação da otimização *peephole*. Esta tabela apresenta duas informações: *o percentual de redução do tamanho do código e o percentual*

de redução do tempo de execução.

Programa	Impacto	
	Tamanho do Código	Tempo de Execução
Append	4,69%	4,39%
Naive Reverse	6,35%	5,85%
Zebra	2,48%	2,28%
Hanoi	5,33%	5,11%
Quick Sort	4,04%	3,72%
Tak	3,86%	3,55%

Tabela 6.11: Impacto do otimizador *peephole*.

Esta otimização reduziu de 2,48% a 6,35% o código de cada programa, ocasionando uma aceleração de 2,28% a 5,85%. Como em *coalescing*, quanto maior a redução do tamanho do código, maior o impacto na aceleração e vice-versa.

A porção de código eliminada nesta fase pode conter instruções redundantes decorrentes da redução da granularidade das instruções YAAM. Como o *parser* não verifica os blocos de código que conectam duas instruções YAAM, a junção pode conter instruções redundantes. Por exemplo, uma instrução YAAM pode terminar escrevendo em um registrador, que a próxima instrução irá inicialmente ler. Neste caso, existe uma leitura imediatamente após uma escrita.

Casos como este são otimizados pelo otimizador *peephole*, que além de eliminar instruções redundantes, troca um conjunto de instruções por outro mais eficiente.

6.3.3.6 Qualidade do Código Gerado

A tabela 6.12 apresenta as características do código gerado pelo compilador YAPc. O código gerado é muito compacto, *append* e *naive reverse* são os casos ótimos. Para estes, todos os registradores temporários foram mantidos em registradores de máquina, não precisando representar nenhum temporário em memória. Fato que não ocorre para os outros programas. O pior caso ocorre para o programa *zebra*, porque vários predicados são compilados.

A técnica implementada elimina uma porção considerável de código. Observe que algumas instruções YAAM podem ser executadas em diferentes modos. Neste caso, quanto maior a quantidade de instruções deste tipo maior pode ser a porção de código eliminada. Isto irá depender do tipo dos argumentos do predicado, pois a quantidade de instruções difere para cada modo de execução.

Programa	Código Gerado			
	Bytes	Instruções	*Spills*	*Fetchs*
Append	1623(**224**)	498(**64**)	37(**0**)	106(**0**)
Naive Reverse	2265(**464**)	679(**126**)	58(**0**)	140(**0**)
Zebra	8293(**4476**)	2273(**1138**)	142(**64**)	434(**177**)
Hanoi	1596(**828**)	460(**225**)	33(**19**)	60(**19**)
Quick Sort	5657(**1723**)	1743(**520**)	142(**23**)	348(**72**)
Tak	2340(**1427**)	696(**415**)	44(**31**)	116(**55**)

Tabela 6.12: Características do código gerado. Os valores a esquerda foram obtidos com a técnica de seleção de regiões desligada. Enquanto, os da direita com a técnica ligada.

Pode-se concluir que a técnica de seleção de regiões torna o processo de compilação mais eficiente, reduzindo o impacto do sistema de compilação ao ambiente de execução, além de gerar um código altamente compacto.

6.3.3.7 Análise Detalhada dos Programas *Zebra* e *Tak*

Como mostrado anteriormente, os programas *zebra* e *tak* não obtiveram aceleração. A análise detalhada destes programas tem por objetivo identificar o problema da falta de desempenho.

Zebra é o programa teste mais rápido, que executa em apenas 4 milisegundos. É difícil obter desempenho através de um sistema de compilação dinâmica, para programas muito rápidos. Nestes casos, a tendência é ter um tempo de compilação maior que o tempo de execução do programa. Este é o caso de *zebra*.

Observando o tempo gasto pelo compilador em outros programas, por exemplo *append*, nota-se que *zebra* consome 37 vezes mais tempo. Isto é devido ao fato dos programas possuirem características diferentes. Em *append* o sistema de compilação otimiza apenas uma cláusula do predicado *append*. Enquanto que em *zebra* são otimizadas todas as cláusulas dos predicados *next_to* e *my_member*.

O predicado *next_to* passa metade do tempo sendo interpretado. Isto não é uma boa escolha, devido ao custo da interpretação. O limite de 1.000 chamadas não se mostrou uma boa escolha neste caso. Um limite superior seria a melhor escolha. Isto eliminaria a invocação do sistema de compilação pelo predicado *next_to*. Ocasionando em uma redução signicativa do tempo de compilação, em um fator de 5,4.

A figura 6.5 apresenta o tempo de compilação (em milisegundos) gasto por cada predicado otimizado. Observe que o tempo de compilação é dominado pela otimização do predicado *next_to*. Por outro lado, mesmo com um limite superior de invocação do sis-

tema de compilação, o programa *zebra* ainda não obteria aceleração. Otimizar o predicado *my_member* consome um tempo maior do que interpretar o programa.

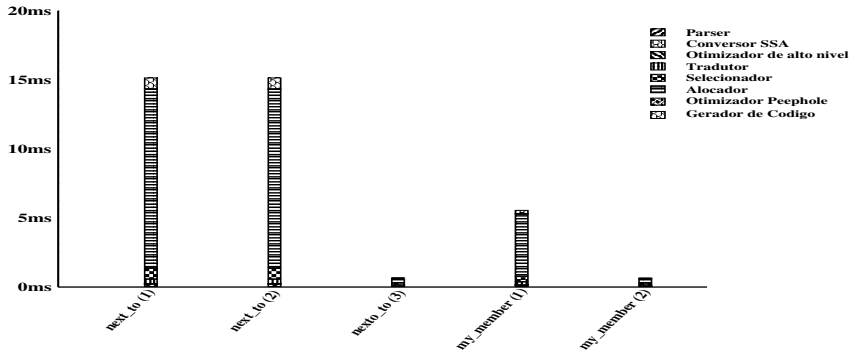


Figura 6.5: Tempo de execução das fases do compilador para o programa *Zebra*.

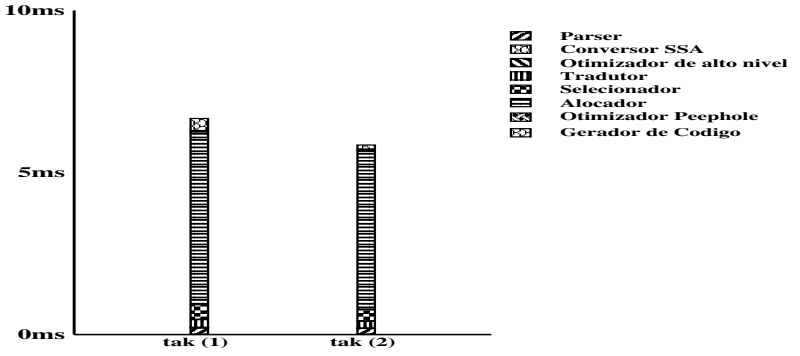
Além de aumentar o limite de invocação do sistema de compilação, existe a necessidade de utilizar um alocador de registradores mais veloz. Para todos os predicados otimizados, o compilador passa a maior parte do tempo na fase de alocação de registradores. Ainda que o predicado *next_to* não fosse otimizado, levaria 4,5 milisegundos somente para alocar registradores para o predicado *my_member*.

Considerando apenas o tempo de execução da aplicação, otimizar o programa *tak* obtém uma aceleração de fator 3. Contudo, em um ambiente de compilação dinâmica o tempo de execução inclui o tempo gasto em otimizações. Neste caso, otimizar *tak* ocasiona em uma aceleração negativa.

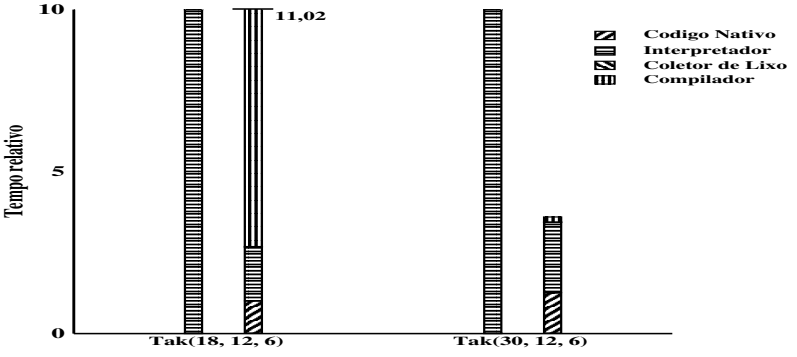
Neste programa, o interpretador consome 62,5% do tempo de execução (excluindo coletor de lixo e compilador). Este percentual é devido ao sistema de compilação otimizar apenas 45% do programa. Além disto, a execução alterna constantemente entre o interpretador e chamadas ao código nativo. Porém este chaveamento não ocorre apenas em *tak*, ocorre também com os programas *naive reverse*, *quick sort* e *hanoi*. Para os programas mais rápidos, alternar freqüentemente entre interpretador e código nativo é um problema significante.

O mau desempenho do programa *tak* é devido ao alto custo do alocador de registradores. A figura 6.6(a) mostra o tempo de compilação gasto por cada cláusula do programa *tak*. Como pode ser observado, a alocação de registradores demora 10 milisegundos, 66,67% do tempo total de execução.

Como indicado nesta mesma seção, o algoritmo utilizado pelo alocador de registradores precisa ser revisto.



(a) Tempo de execução das fases do compilador.



(b) Tempo de execução com entradas diferentes.

Figura 6.6: Análise do programa *Tak*.

No caso do programa *tak*, aumentar o tamanho da entrada melhora o desempenho. Neste caso, com um tempo de execução maior, o percentual de tempo gasto pelo compilador se torna menor.

A figura 6.6(b) mostra os resultados obtidos alterando o tamanho da entrada do programa *tak*. Nesta figura, o tempo de execução também está normalizado. A barra a esquerda mostra o tempo de execução obtido por YAP, e a barra a direita mostra o tempo de execução obtido por YAP+.

Com uma entrada maior o tempo do compilador passa a ser negligenciável para o programa *tak*, e o tempo de execução passa a ser dominado pelo tempo gasto pelo emulador e pela execução do código nativo. Com esta entrada maior YAP+ obteve uma aceleração de 2,78 para o programa *tak*.

6.4 Comparação de YAP+ com CIAO e Mercury

Uma comparação direta entre YAP+, CIAO e Mercury não é possível porque CIAO e Mercury geram código nativo para o programa Prolog todo, enquanto o YAP+ gera código nativo somente para as cláusulas. De qualquer forma, o objetivo desta seção é comparar o desempenho da compilação dinâmica de YAP+ com um sistema de compilação estática.

Nesta seção os resultados com o programa *tak* foram obtidos com o tamanho de entrada para o qual YAP+ obteve aceleração sobre YAP. Para os programas *append*, *zebra* e *quick sort* não são apresentados resultados para Mercury. Isto devido a não termos conseguido usar estes programas em Mercury. Além disto, o ambiente possui a configuração descrita na seção 6.3.2.

A figura 6.7 mostra o tempo de execução normalizado dos sistemas: YAP+, CIAO e Mercury. Nesta figura a barra a esquerda mostra o tempo de execução obtido por YAP+, a barra central mostra o tempo de execução obtido por CIAO, e a barra a direita mostra o tempo de execução obtido por Mercury.

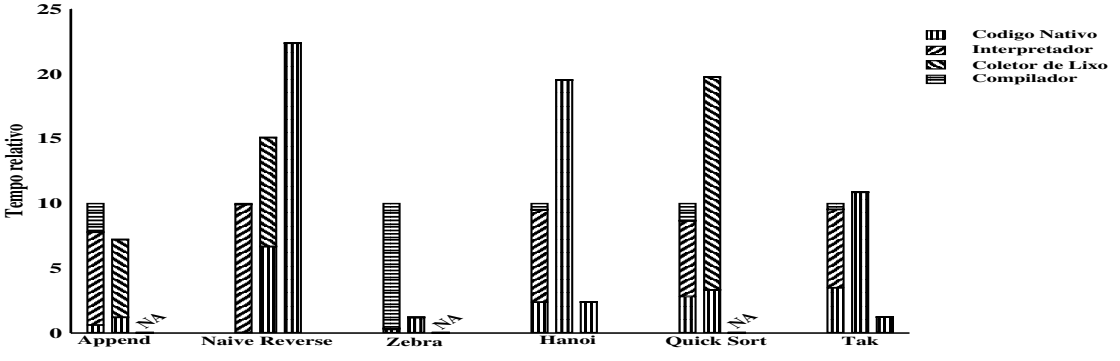


Figura 6.7: Comparação entre YAP+, CIAO e Mercury.

YAP+ possue um desempenho semelhante a CIAO para os programas *append* e *tak*. Para os outros programas, YAP+ obteve um melhor desempenho que CIAO. Para os programas *naive reverse* e *quick sort*, o problema do sistema CIAO está no coletor de lixo. Em CIAO, o tempo gasto pelo coletor de lixo domina o tempo total de execução nestes dois programas. Desconsiderando o tempo gasto pelo coletor de lixo, CIAO obtém um melhor desempenho para estes programas.

A coleta de lixo em YAP+ é muito mais eficiente, para todos os programas o tempo gasto pelo coletor de lixo é negligenciável.

YAP+ também obteve um bom desempenho para o programa *hanoi*. Neste programa não houve coleta de lixo. O problema com CIAO é o aumento do número de instruções

executadas pelo *hardware*. CIAO executa o dobro de instruções quando comparado com YAP+.

O programa *zebra* é um caso excepcional. O desempenho de CIAO é semelhante ao desempenho de YAP. Enquanto o código gerado por CIAO leva 4,60 milisegundos para executar *zebra*, YAP executa esta mesma aplicação em 4 milisegundos.

Mercury possui um bom desempenho para os programas *hanoi* e *tak*. Para o programa *naive reverse*, YAP+ obteve um melhor desempenho do que Mercury. O código nativo gerado por Mercury ocasiona um número de instruções executadas pelo *hardware* maior do que o número instruções executadas em YAP+.

A tabela 6.13 mostra o tempo de cada compilador. YAPc é muito mais rápido do que CIAO e Mercury em todos os casos.

	Sistema		
Programa	**YAP+**	**CIAO**	**Mercury**
Append	1,11	412	NA
Naive Reverse	2,11	160,80	18825
Zebra	37,28	158	NA
Hanoi	5,56	149,80	810
Quick Sort	14,13	352,60	NA
Tak	12,53	148,80	1498

Tabela 6.13: Tempo de compilação em milisegundos.

Para o programa *append*, onde YAP+ obteve um desempenho semelhante a CIAO, o compilador YAPc chega a ser 400 vezes mais rápido. Em alguns casos, o alto tempo de compilação de CIAO não justifica o desempenho obtido pelo código gerado. CIAO utiliza análises globais, o que acarreta um aumento do tempo gasto pelo processo de compilação.

Naive reverse é o caso onde Mercury obteve o maior tempo de compilação. Contudo, este programa obteve o pior desempenho de Mercury. Assim como CIAO, o alto tempo de compilação é devido ao custo das análises globais.

A tabela 6.14 mostra o tamanho do código gerado pelos compiladores. O código gerado por CIAO e Mercury são maiores porque estes compilam todo o programa Prolog, enquanto YAPc compila apenas as cláusulas. Porém, vale observar que mesmo gerando código nativo apenas para as cláusulas, YAP+ obteve em alguns casos um desempenho melhor (ou semelhante) a CIAO ou Mercury. Isto indica que a técnica de *seleção de regiões* e o uso de *informações sobre o tipo dos dados* são técnicas competitivas com as técnicas de análises globais.

Programa	Sistema		
	YAP+	**CIAO**	**Mercury**
Append	0,22	3815,23	NA
Naive Reverse	0,45	311,93	1690,54
Zebra	4,37	152,78	NA
Hanoi	0,81	149,90	1651,35
Quick Sort	1,68	3032,13	NA
Tak	1,39	150,135	1651,45

Tabela 6.14: Tamanho do código gerado em *Kbytes*.

6.5 Considerações Gerais

Esta seção mostrou a avaliação de desempenho do sistema de compilação desenvolvido e seu impacto sobre o ambiente de execução.

O tempo de execução dos programas de teste facilitou verificar o impacto do sistema de compilação ao ambiente de execução. Mesmo para programas rápidos, o ambiente de execução ao utilizar YAPc obteve um bom desempenho para a maioria dos programas. Isto é devido ao fato do compilador desenvolvido ser significantemente leve e adaptável às características do programa.

Executar código nativo obteve um melhor desempenho comparado com interpretar um conjunto de instruções. Ao utilizar YAPc, o ambiente de execução acelerou as aplicações entre 2,78 e 4,84 vezes. Gerar código apenas para uma parte do programa mostrou ser uma ótima opção. Porém, na maioria dos casos o tempo de execução ainda é dominado pelo tempo gasto pelo interpretador.

Em alguns casos YAP+ possui um desempenho semelhante ao obtido por sistemas estáticos. Isto devido, as técnicas utilizadas por YAP+ deixarem o sistema de compilação leve e otimizarem de forma efetiva os programas em execução.

O desempenho ruim de alguns programas é ocasionado pelo fato destes acionarem muitas vezes o sistema de compilação, que por sua vez, consome muito tempo alocando registradores.

Os resultados também mostraram que compilar apenas porções de código, ocasiona um menor impacto ao ambiente de execução. Em alguns casos, o código gerado chega a ser ótimo, por ser compacto e não precisar representar nenhum registrador em memória.

A avaliação do sistema de compilação mostrou alguns trabalhos futuros que podem ser desenvolvidos para melhorar o ganho de desempenho. Estes trabalhos são descritos no próximo capítulo, juntamente com as considerações finais.

Capítulo 7

Conclusões

Diversas técnicas para implementar Prolog de uma maneira eficiente tem sido propostas desde o original interpretador de Prolog [31], muitas destas visam obter mais velocidade. Existem duas abordagens principais para implementar Prolog eficientemente:

1. *compilar para bytecode e então interpretá-los*; ou
2. *compilar para código nativo*.

Interpretadores possuem tempo de carga e compilação pequenos e são uma boa solução pela sua simplicidade quando a velocidade não é uma prioridade. Compiladores são mais complexos do que interpretadores, e a diferença é ainda mais acentuada se alguma forma de análise [84] é realizada como parte da compilação, o que impacta o tempo de desenvolvimento. A geração de código de baixo nível promete programas rápidos através do uso de recursos caros durante as fases de compilação.

Nos últimos anos ocorreram melhorias significativas na compilação de Prolog, resultando no retorno da idéia da execução em código nativo. São duas as razões principais que levaram alguns sistemas recentes evitar a interpretação:

1. ***Redução do overhead***: a compilação para código nativo elimina o *overhead* intrínseco em interpretar as instruções de uma máquina abstrata, que ocupa um alto percentual do tempo total de execução [106].
2. ***Aumentar a flexibilidade***: a compilação para código nativo permite uma maior especialização das instruções WAM.

Nesta tese foi desenvolvido um sistema de compilação dinâmica que utiliza *informações sobre o tipo dos dados e otimizações adaptáveis*.

O principal objetivo desta tese foi o desenvolvimento de um ambiente de compilação dinâmica que se adapta às características do programa em execução. Um segundo objetivo foi demonstrar como algumas técnicas utilizadas por linguagens orientadas a objetos poderiam ser aplicadas em outro tipo de linguagem de programação. Como conseqüência destes objetivos tem-se uma eficiente implementação de Prolog.

A seguir é apresentado uma síntese do trabalho, para em seguida serem enumerados os trabalhos futuros. E finalmente são fornecidas as considerações finais.

7.1 Síntese do Trabalho

O uso da compilação dinâmica em linguagens orientadas a objetos reduz significantemente o *overhead* imposto pela interpretação. As técnicas utilizadas por estas linguagens podem também garantir uma implementação eficiente de linguagens declarativas, tais como a linguagem de programação em lógica Prolog.

Esta tese demonstra que a compilação dinâmica pode realizar otimizações que se adaptam ao comportamento do programa e que além disto, são aplicadas de maneira tardia. Desta forma, obtém-se um sistema de compilação leve e eficiente que possibilita um ganho de desempenho aos programas.

Três técnicas principais foram utilizadas para este fim: *uso de informações sobre os tipos dos dados*, *otimizações adaptativas* e *desotimização dinâmica*.

No projeto do compilador otimizador foi proposto gerar código nativo para Prolog, através da técnica de compilação dinâmica. Naturalmente, o objeto não é apenas implementar Prolog eficientemente, mas também uma larga classe de linguagens lógicas. Além disto, como as técnicas utilizadas não são específicas da linguagem Prolog, elas podem ser aplicadas em outros tipos de linguagens.

Para que o ambiente de execução gerasse código nativo de forma eficiente foi necessário acoplar a este um sistema de compilação. A partir do desenvolvimento deste sistema, o ambiente de execução passou a ter as seguintes características:

- ***Utiliza informações sobre o tipo dos dados***. Utilizando tais informações o ambiente de compilação apenas otimizada as porções de código que realmente serão executadas.

- ***Utiliza um mecanismo de compilação dinâmica***. O sistema é capaz de descobrir oportunidades para a compilação sem a intervenção do programador. Em particular, ele deve decidir: *quando compilar, o que compilar e quais caminhos compilar*.

- ***Emprega um modo misto de execução***. Inicialmente o programa é interpretado e automaticamente o ambiente detecta as partes do programa que deverão ser otimizadas. Assim, o ambiente gera código nativo apenas para as regiões executadas com grande freqüência.

- ***Explora o determinismo***. Através do uso de informações sobre os tipos de dados o ambiente de execução determina quais cláusulas devem ser otimizadas.

- ***Explora as características do programa***. O ambiente de compilação é capaz de adaptar-se às características de cada programa. Desta maneira, uma otimização somente é aplicada quando o ganho real for maior do que o tempo gasto em aplicá-las e o comportamento do programa favorecer a sua aplicação.

O compilador YAPc é conservativo, ele compila apenas aquelas partes do programa que são executadas freqüentemente. Além disto, ele apenas executa otimizações de maneira tardia, explorando as características do programa. Adicionalmente, muitos casos que poderiam ocorrer em princípio mas que na prática ocorrem raramente nunca são compilados, por exemplo *overflow* de inteiros ou tipos ilegais de argumento. Isto permite minimizar o tempo de compilação e o tamanho do código gerado, além de permitir otimizar mais eficientemente as partes realmente executadas.

Os resultados da comparação entre YAP e YAP+ mostraram que mesmo gerando código nativo apenas para partes do programa, o YAP+ conseguiu obter uma aceleração entre 2,16 e 4,60, para o pior e o melhor caso respectivamente. Embora, em dois casos YAP+ obtivesse um tempo de execução maior, devido ao tempo consumido pelo sistema de compilação, acredita-se que YAP+ mostrou ser uma ótima opção para execução de alto desempenho de programas Prolog.

7.2 Trabalhos Futuros

Embora o objetivo deste trabalho esteja alcançado, algumas questões devem ser novamente revistas enquanto outras devem ser abordadas, tais como:

- *melhorar o algoritmo de alocação de registradores*,

- *utilizar um limite dinâmico*,

- *integrar predicados*,

- *aumentar a quantidade de otimizações de alto nível.*

Os trabalhos futuros serão realizados na ordem mencionada. As próximas subseções descrevem cada uma destas questões.

7.2.1 Alocação de Registradores

Para melhorar o desempenho do compilador é necessário trocar o algoritmo de alocação de registradores. O uso de um algoritmo baseado na coloração de grafo consome de 80% a 90% do tempo gasto no processo de compilação.

O uso da técnica de compilação dirigida mostrou-se uma técnica útil para reduzir o tamanho do código gerado, conseqüentemente, diminuindo a pressão por registradores. Além disto, para os programas utilizados na avaliação de desempenho o *parser* identificou que a melhor escolha seria não transformar a árvore sintática abstrata em uma forma SSA, o que também reduziu a pressão por registradores.

Para programas complexos, embora a técnica de compilação dirigida reduza o tamanho do código, a pressão por registradores aumentará a medida que o compilador utilize a forma SSA.

Pereira e Palsberg [89] descrevem um algoritmo mais rápido do que os algoritmos tradicionais de coloração de grafo. Este e o algoritmo utilizado pelo compilador *cliente* da Sun serão os algoritmos estudados para uma possível implementação em uma futura versão do compilador YAPc.

7.2.2 Limite Dinâmico

Como mencionado anteriormente, o sistema de compilação utiliza contadores para detectar candidatos a serem compilados. Cada predicado interpretado possui um próprio contador, que é incrementado a cada chamada. Se o contador excede um determinado limite, o sistema de compilação é invocado para decidir se o predicado deverá ser compilado.

A técnica do uso de contadores é uma boa opção no desenvolvimento do sistema de compilação. Contudo o limite de invocação não deveria ser o mesmo em todos os casos, pelo contrário, ele poderia ser diferente para cada predicado em particular. Desta forma, um predicado poderia se beneficiar muito mais da otimização.

Pode ser difícil estimar o impacto de otimizar um predicado particular. Contudo, será investigado como utilizar informações de controle para inferir o limite de um determinado

predicado. O objetivo por exemplo, é adiantar o momento da compilação daqueles predicados cujo tempo de compilação podem ser elevados. Objetivando desta forma, alcançar o melhor desempenho possível.

7.2.3 Integração de Predicados

Integração de procedimentos [121] é uma técnica bem conhecida que troca as chamadas aos procedimentos por cópias de seus corpos. Esta otimização reduz o *overhead* das chamadas aos procedimentos, que pode ser significante em alguns tipos de linguagens.

Integração aumenta o escopo de compilação, explorando mais oportunidades para otimizações e eliminando o *overhead* de se criar segmentos de pilha, passagem de argumentos e valores de retorno. Por outro lado, excessivas aplicações da integração pode degradar o desempenho. Isto é devido ao aumento do tamanho do código, que pode reduzir a localidade, conseqüentemente reduzindo a quantidade de instruções por ciclo de máquina.

O crescimento explosivo do código pode ser evitado concentrando apenas em procedimentos executados freqüentemente. Heurísticas podem auxiliar na decisão sobre quais métodos integrar. Experiências tem mostrado que escolher pequenos métodos é uma boa heurística [131].

Uma otimização que será implementada em YAPc é alterar o retorno ao interpretador durante chamadas de predicados, por uma chamada ao código nativo do predicado quando este já tiver sido compilado. Em um caso ótimo, por exemplo no caso do programa *hanoi*, a execução seguirá apenas em código nativo.

Acreditamos que neste caso, o desempenho pode ser melhorado eliminado o custo das chamadas nativas. Além disto, integrar apenas predicados anteriormente compilados, garante que o sistema de compilação otimizará apenas as partes do programa executadas freqüentemente.

Integrar predicados que ainda não foram compilados pode favorecer a aplicação de outras otimizações, além de antecipar a otimização daqueles predicados que futuramente seriam selecionados para compilação. Por outro lado, os predicados integrados podem ser predicados que nunca acionarão o sistema de compilação.

Duas abordagem diferentes serão implementadas: (1) *integrar apenas predicados anteriormente compilados* e (2) *integrar apenas predicados que não foram compilados*.

A primeira abordagem visa otimizar apenas as partes do programa executadas freqüentemente, enquanto a segunda visa antecipar possíveis acionamentos do sistema de

compilação. Integrar predicados anteriormente compilados será implementado primeiro devido a sua simplicidade. Para implementar a segunda abordagem, um estudo detalhado deve ser feito para definir a heurística que será utilizada.

7.2.4 Otimizações de Alto Nível

Uma das características do compilador otimizador é ser adaptável às características individuais de cada programa. Desta forma, ele somente aplica uma determinada otimização quando as características do programa propiciarem a sua aplicação.

Os programas utilizados na avaliação do sistema de compilação não possuem as características necessárias para a aplicação das otimizações implementadas: *propagação de cópia, propagação de constante, eliminação de saltos e eliminação de código morto*.

Para programas mais complexos, outras otimizações podem ser necessárias. Desta forma, as novas versões do compilador otimizador possivelmente possuirá outras otimizações.

As técnicas implementadas foram aplicadas em uma linguagem de programação lógica. Contudo, um trabalho futuro será aplicá-las ao contexto de outras linguagens, por exemplo linguagens funcionais. A medida que se utiliza o sistema de compilação em diferentes contextos, passa a existir a necessidade da implementação de outras otimizações.

Neste sentido temos duas frentes de trabalho: *otimizações de alto nível específicas para programas lógicos, e aquelas específicas para programas funcionais*.

Primeiramente serão analisados programas Prolog complexos, com o objetivo de especificar quais otimizações implementar. Em seguida, planeja-se atacar as linguagens funcionais.

7.3 Considerações Finais

A principal contribuição desta tese foi a implementação de um sistema de compilação adaptável para o sistema YAP. Para atingir este objetivo foi desenvolvido um compilador *Just-In-Time* para Prolog.

Os bons resultados obtidos constituem uma forte motivação para uma continuidade do trabalho realizado. Além disto, existe o objetivo de aplicar as mesmas técnicas em outros tipos de linguagens. Todo o projeto do sistema de compilação visava esta última questão.

A motivação em se utilizar uma linguagem de programação lógica foi a inovação em se aplicar técnicas de compilação dinâmica em um contexto ainda não aplicado. Ficou

claro que estas técnicas são aplicáveis às linguagens lógicas. Por outro lado, também ficou demonstrado que um bom desempenho somente é obtido quando o sistema de compilação adapta-se às características do programa em execução.

Referências Bibliográficas

[1] ABBOY, M. B., PETERSON, L. L. "A Language-Based Approach to Protocol Implementation". *Computer Communications Review*, v. 22, n. 4, pp. 27–37, August 1992.

[2] ADL-TABATABAI, A., CIERNIAK, M. "Fast, Effective Code Generation in a Just-in-Time Java Compiler". In: *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 280–290, Montreal, Canada, June 1998.

[3] AGESEN, O., DETLEFS, D. *Mixed-mode Bytecode Execution*. Technical Report SMLI TR-2000-87, Sun Microsystems, October 2000.

[4] AHO, A. V., JOHNSON, S. C., ULLMAN, J. D. "Code Generation for Expressions with Common Subexpressions". *ACM*, v. 24, n. 1, pp. 146–160, January 1977.

[5] AHO, A. V., SETHI, R., ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. 1 ed., New York, USA, Addison Wesley, 1986.

[6] AIT-KACI, H. *Warren's Abstract Machine*. 1 ed., London, England, MIT Press, 1991.

[7] ALPERN, B., AUGART, S., BLACKBURN, S. M., BUTRICO, M., COCCHI, A., CHENG, P., DOLBY, J., FINK, S., GROVE, D., HIND, M., McKINLEYy, K. S., MERGEN, M., MOSS, J. E. B., NGO, T., SARKAR, V. "The Jikes Research Virtual Machine Project: Building an Open-source Research Community". *IBM Syst. J.*, v. 44, n. 2, pp. 399–417, October 2005.

[8] ALPERN, B., WEGMAN, M. N., ZADECK, F. "Detecting Equality of Values in Programs". In: *Proceedings of the Symposium on Principles of Programming Languages*, pp. 1–11, San Diego, California, USA, January 1988.

[9] APPEL, A. W. *Modern Compiler Implementation in C*. 1 ed., New York, USA, Cambridge University Press, 1998.

[10] ARNOLD, K., GOSLING, J., HOLMES, D. *The Java Programming Language*. 2 ed., California, USA, Addison Wesley, 2000.

[11] AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., BERSHAD, B. N. "Fast, Effective Dynamic Compilation". In: *Proccedings of Conference on Programming Language Design and Implementation*, pp. 149–159, Philadelphia, Pennsylvania, USA, May 1996.

[12] BACON, D. F., GRAHAM, S. L., SHARP, O. J. "Compiler Transformations for High-Performance Computing". *ACM Computing Surveys*, v. 26, n. 4, pp. 345–420, December 1994.

[13] BALL, J. E. "Predicting the Effects of Optimization on a Procedure Body". In: *Proceedings of the Symposium on Compiler Construction*, pp. 214–200, Denver, Colorado, USA, August 1979.

[14] BEER, J. "The Occur-Check Problem Revisited". *Journal of Logic Programming*, v. 5, n. 3, pp. 243–261, September 1988.

[15] BERNSTEIN, D., GOLUMBIC, M., MANSOUR, Y., PINTER, R., KRAWCZYK, H., NAHSHON, I., GOLDIN, D. Q. "Spill Code Minimization Techniques for Optimizing Compilers". In: *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 258–263, Portland, Oregon, June 1989.

[16] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., McKINLEY, K. S., BENTZUR, R., DIVAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., von DINCKLAGE, D., WIEDERMANN, B. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, New York, NY, USA, October 2006. ACM Press.

[17] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., McKINLEY, K. S., BENTZUR, R., DIWAN, A., FEIBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGEe, D., WIEDERMANN, B. *The DaCapo Benchmarks: Java Benchmarking*

Development and Analysis (Extended Version). Technical Report TR-CS-06-01, 2006. http://www.dacapobench.org.

[18] CALLAHAN, D., COOPER, K. D., KENNEDY, K., TORCZON, L. "Interprocedural Constant Propagation". In: *Proceedings of Symposium on Compiler Construction*, pp. 152–161, California, USA, July 1986.

[19] CARLSSON, M. "On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog". In: *Proceedings of the International Conference on Logic Programming*, pp. 3–16, Lisbon, Portugal, June 1989.

[20] CARLSSON, M. *The SICStus Emulator*. Technical Report T91:15, Swedish Institue of Computer Science, February 1996.

[21] CASANOVA, M. A., GIORNO, F. A. C., FURTADO, A. L. *Programação em Lógica e a Linguagem Prolog*. 1 ed., São Paulo, Brasil, Edgard Blucher LTDA, 1987.

[22] CHAMBERLIN, D. D. "Support for Repetitive Transactions and Ad Hoc Queries in System R". *ACM Transactions on Database Systems*, v. 6, n. 1, pp. 70–94, March 1981.

[23] CHAMBERS, C. *The Design and Implementation of the Self Compiler: an Optimizing Compiler for Objetct-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, April 1992.

[24] CHAMBERS, C., UNGAR, D. "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language". In: *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 174–185, New York, NY, USA, June 1989.

[25] CHIEN, A. A., KARAMCHETI, V., PLEVYAK, J. *The Concert System: Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Objet-Oriented Programs*. Technical Report UIUC DCS-R-93-1815, University of Illinois at Urbana-Champaign, 1993.

[26] CIERNIAK, M., LUEH, G. Y., STICHONOTH, J. M. "Practicing JUDO: Java Under Dynamic Optimizations". In: *Proceedings of the ACM Conference on Pro-*

gramming Language Design and Implementation, pp. 13–26, Vancouver, British Columbia, Canada, June 2000.

[27] CLICK, C., PALECZNY, M. "A Simple Graph-Based Intermediate Representation". In: *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, pp. 35–49, Vancouver, Canada, January. 1995.

[28] COCKE, J. "Global Common Subexpression Elimination". In: *Proceedings of the Symposium on Compiler Optimization*, pp. 20–24, Urbana-Champaign, Illinois, USA, July 1970.

[29] CODOGNET, P., DIAZ, D. "WAMCC: Compiling Prolog to C". In: *Proceedings of the International Conference on Logic Programming*, pp. 317–331, Tokyo, Japan, June 1995.

[30] COHEN, J., HICKEY, T. J. "Parsing and Compilation Using Prolog". *Transactions on Programming Languagens and Systens*, v. 9, n. 6, pp. 125–1693, April 1987.

[31] COLMERAUER, A. "The Birth of Prolog". In: *Proceedings of the Second History of Programming Languages Conference*, pp. 37–52, Cambridge, Massachusetts, USA, April 1993.

[32] COLMERAUER, A., KANOUI, H., PASERO, R., ROUSSEL, P. *Un Systeme de Communication Homme-machine en Fracais*. Technical Report 72-18, Groupe Intelligence Artificille, Universite Aix-Marseille II, October 1973.

[33] CONWAY, T., HENDERSON, F., SOMOGYI, Z. "Code Generation for Mercury". In: *Proceedings of International Logic Programming Symposium*, pp. 242–256, Portland, Oregon, USA, December 1995.

[34] COUSOT, P., COUSOT, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Fixed Points". In: *Proceedings of Symposium on Principles of Programming Languages*, pp. 78–88, Los Angeles, California, USA, 1977.

[35] COUSOT, P., COUSOT, R. "Abstract Interpretation and Application to Logic Programs". *Journal of Logic Programming*, v. 13, n. 2-3, pp. 103–179, July 1992.

[36] CYTRON, R., FERRANTE, J., ROSEN, B. K., WWGMAN, M. N., ZADECH, F. K. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Transactions on Programming Languages and Systems*, v. 13, n. 4, pp. 451–490, October 1991.

[37] CYTRON, R., GERSHBEIN, R. "Efficient Accommodation of May-alias Information in SSA Form". *ACM SIGPLAN Notices*, v. 28, n. 6, pp. 36–45, June 1993.

[38] "DACAPO BENCHMARK SUITE". *http://dacapobench.org*; accessed December 2, 2005.

[39] DEMOEN, B., NGUYEN, P.-L. "So Many WAM Variations, So Little Time". In: *LNAI 1861, Proceedings Computational Logic - CL 2000*, pp. 1240–1254, London, UK, July 2000. Springer-Verlag.

[40] DETLEFS, D., AGESEN, O. "Inlining of Virtual Methods". *Lecture Notes in Computer Science*, v. 1628, pp. 258–277, June 1999.

[41] DEUTSCH, L. P., SCHIFFMAN, A. "Efficient Implementation of the Smalltalk-80 System". In: *Symposium on the Principles of Programming Languages*, pp. 297–302, Salt Lack City, USA, January 1984.

[42] DIAZ, D., CODOGNET, P. "Design and Implementation of the GNU Prolog System". *Journal of Functional and Logic Programming*, v. 13, n. 4, pp. 451–490, October 2001.

[43] DIAZ, D., CODOGNET, P. "GNU Prolog: Beyond Compiling Prolog to C". *Lecture Notes in Computer Science*, v. 1753, pp. 81–91, January 2000.

[44] DIAZ, D., CODOGNET, P. "The GNU Prolog System and its Implementation". In: *Proceedings of the 2000 ACM symposium on Applied computing*, pp. 728–732, Como, Italy, March 2000.

[45] DONGARRA, J., HIND, A. R. "Unrolling Loops in Fortran". *Software - Practice and Experience*, v. 9, n. 3, pp. 219–226, March 1979.

[46] ELLIS, J. R. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, New Haven, Connecticut, USA, August 1984.

[47] FINK, S., QIAN, F. "Design, Implementation and Evaluation of Adaptive Recompilation With On-Stack-Replacement". In: *Proceeding of the International Symposium on Code Generation and Optimization*, pp. 421–252, San Francisco, California, USA, 2003.

[48] FRANZ, M. "Technological Steps Toward a Software Component Industry". *Lecture Notes on Computer Science*, v. 782, n. 4, pp. 111–125, March 1994.

[49] GOLDBERG, A., ROBSON, D. *Smalltalk-80: The Language and Its Implementation*. 1 ed., Massachusets, USA, Addison-Wesley, 1983.

[50] GREENBLATT, R. D. *The Lisp Machine*. Technical Report 79, MIT Artificial Intelligence Laboratory, October 1979.

[51] GREENBLATT, R. D., KNIGHT, T. F. "A LISP Machine". In: *Proceedings of the Workshop on Computer Architecture for Non-Numeric Processing*, pp. 137–138, California, USA, March 1980.

[52] GUPTA, R. "Optimizing Array Bounds Checks Using Flow Analysis". *ACM Letters on Programming Languages and Systems*, v. 2, n. 1-4, pp. 135–150, December 1993.

[53] HAYGOOD, R. C. *Native Code Compilation in SICStus Prolog*. 1 ed., New York, USA, MIT Press, 1994.

[54] HENDERSON, F., SOMOGYI, Z. "Compiling Mercury to High-Level C Code". In: *Proceedings of Computational Complexity*, pp. 197–212, Montreal, Canada, May 2002.

[55] HENNESSY, J. L., PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 1 ed., Morgan Kaufman, 2006.

[56] HENRY, R. R., FRASER, C. W., PROEBSTING, T. A. "Burg - Fast Optimal Instruction Selection and Tree Parsing". In: *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 36–44, Sao Francisco, USA, June 1992.

[57] HO, W. W., OLSSON, R. A. "An Approach to Genuine Dynamic Linking". *Software - Pratice and Experience*, v. 21, n. 4, pp. 31–40, April 1991.

[58] HÖLZLE, U. *Adaptative Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Department of Computer Science, Stanford University, March 1995.

[59] HÖLZLE, U., UNGAR, D. "Optimizing Dynamically-dispatched Calls with Runtime Type Feedback". In: *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 326–336, Orlando, Florida, United States, June 1994.

[60] "JAVA GRANDE FORUM BENCHMARCK". *http://www.epcc.ed.ac.uk/ javagrande/*; accessed July 2, 2004.

[61] "JIKES RVM HOMEPAGE". *http://www-124.ibm.com/developerworks/oss/ jikesrvm/*; accessed November 2, 2004.

[62] "KAFFE VIRTUAL MACHINE". *http://www.kaffe.org/*; accessed July 2, 2004.

[63] KAVAHITO, M., KOMATSU, H., NAKATAMI, T. "Effective Null Pointer Check Elimination Utilizing Hardware Trap". *ACM SIGARCH Computer Architecture New*, v. 28, n. 5, pp. 139–149, November 2000.

[64] KESSLER, P. "Fast Breakpoints: Design and Implementation". In: *Proccedings of Conference on Programming Language Design and Implementation*, pp. 78–84, June 1990.

[65] KILDALL, G. "A Unified Approach to Global Program Optimization". In: *Proceedings of Symposium on Principles of Programming Languages*, pp. 194–602, Boston, USA, October 1973.

[66] KNOBE, K., SARKAR, V. "Array SSA Form and its Use in Parallelization". In: *Proceedings of the Symposium on Principles of Programming Languages*, pp. 107–120, San Diego, CA, USA, January 1998.

[67] KNOOP, J., RUTHING, O., STEFFEN, B. "Partial Dead Code Elimination". In: *Proceddings of Conference on Programming Language Design and Implementation*, pp. 147–158, Orlando, Florida, USA, June 1994.

[68] KOMATSU, H., TAMURA, N., ASAKAWA, Y., KUROKAWA, T. "An Optimizing Prolog Compiler". In: *Proceedings of the Logic Programming*, pp. 104–115, June 1986.

[69] KRALL, A., GRAFL, R. "CACAO - A 64 bit JavaVM Just-in-Time Compiler". In: *Proceedings of the ACM Workshop on Java for Science and Engineering Computation*, pp. 362–387, Las Vegas, Nevada, USA, June 1997.

[70] LENGAUER, T., TARJAN, R. E. "A Fast Algorithm for Finding Dominators in a Flowgraph". *ACM Transactions on Programming Languages and Systems*, v. 1, n. 1, pp. 121–141, June 1979.

[71] LIANG, S., BRACHA, G. "Dynamic Class Loadding in Java Virtual Machine". In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 36–44, Vancouver, Canada, October 1998.

[72] LINDHOLM, T., YELLIN, F. *The Java Virtual Machine Specification Second Editon*. 2 ed., California, USA, Addison Wesley, 1999.

[73] LLOYD, J. W. *Foundations of Logic Programming*. 1 ed., New York, USA, Springer-Verlay, 1987.

[74] LOPES, R. N. S. *Execução de Prolog com Alto Desempenho*. Master's thesis, Universidade do Porto, Porto, Portugal, Master thesis, Universidade do Porto, Portugal, Jul. 1996.

[75] MARIEN, A. *An Optimal Intermediate Code for Structure Creation in a WAM-based Prolog Implementation*. Technical Report T1988:01, Katholicke Universiteit Leuven, March 1988.

[76] MARIEN, A. *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*. PhD thesis, Katholieke Universiteit Leuven, Leuven, Nederland, 1993.

[77] MARIEN, A., DERMOEN, B. "On the Management of Choicepoint and Environment Frames in the WAM". In: *Proceedings of North American Conference on Logic Programming*, pp. 1030–1047, Cleveland, Ohio, USA, October 1989.

[78] MEIER, M. "Compilation of Compound Terms in Prolog". In: *Proceedings of North American Conference on Logic Programming*, pp. 63–79, Cambridge, MA, USA, October 1990.

[79] MEIER, M., AGGOUN, A. "SEPIA - An Extendible Prolog System". In: *Proceedings of the World Computer Congress*, pp. 1127–1132, San Francisco, USA, August 1989.

[80] MICROSYSTEMS, S. *The Java HotSpot Virtual Machine*. Technical Report 43001, Sun Developer Network Community, April 2003.

[81] MIKHEEV, V., FEDOSEEV, S., SUKHAREV, V., LIPSKY, N. "Effective Enhancement of Loop Versioning in Java". In: *Proceedings of the International Conference Compiler Construction*, pp. 293–306, Grenoble, France, April 2002.

[82] MORALES, J., CARRO, M., HERMENEGILDO, M. "Improved Compilation of Prolog to C Using Moded Types and Determinism Information". In: *Proceedings of the Colloquium on Implementation of Constraint and Logic Programming Systems*, pp. 197–212, 2003.

[83] MOREL, E., RENVOISE, C. "Global Optimization by Suppression of Partial Redundancies". *CACM*, v. 22, n. 2, pp. 96–103, February 1979.

[84] MUCHNICK, S. S. *Advanced Compiler Design And Implementation*. 1 ed., San Francisco, CA, USA, Morgan Kaufmann, 1997.

[85] NAISH, L. "Negation and Quantifiers in NU-Prolog.". In: *Proceedings of International Conference in Logic Programming*, pp. 624–634, London, United Kingdom, 1986.

[86] NASSEN, H. *Optimizing the SICStus Prolog Virtual Machine Instruction Set*. Technical Report T2000:01, Intelligent Systems Laboratory, Uppsala University, March 2001.

[87] OGAWA, H., SHIMURA, K. "OpenJIT, An Open-Ended, Reflective JIT Compiler Framework for Java". In: *Proceedings of the ECOOP*, pp. 362–387, Cannes, France, June 2000.

[88] PALECZNY, M., VICH, C., CLICK, C. "The Java HotSpot Server Compiler". In: *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pp. 1–12, Monterey, CA, USA, April 2001.

[89] PALSBERG, J., PEREIRA, F. M. Q. “Register Allocation via Coloring of Chordal Graphs”. In: *Proceedings of Asian Symposium on Programming Languages and Systems*, pp. 315–329, Tsukuba, Japan, 2005.

[90] PELEGRÍ-LLOPART, E., GRAHAM, S. L. “Optimal Code Generation for expression Trees: An Application BURS Theory”. In: *Proceedings of the Conference on Principles of Programming Languages*, pp. 294–308, Sao Francisco, USA, June 1988.

[91] PLAISTED, D. A. “A Simplified Problem Reduction Format”. *Artificial Intelligence*, v. 18, n. 6, pp. 227–261, March 1982.

[92] PLEZBERT, M. P., CYTRON, R. K. “Does Just-In-Time = Better Late Than Never?”. In: *Proceedings of the Symposium on Principles of Programming Language*, pp. 120–131, Paris, France, January 1997.

[93] POLLOCK, L., BIVENS, M., SOFFA, M. L. “Debugging Optimized Code Via Tailoring”. In: *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 201–222, New York, NY, USA, 1994. ACM Press.

[94] “PROGRAM COUNTER LIBRARY”. *http://www.fz-juelich.de/zam/PCL/*; accessed July 2, 2004.

[95] QUINTANO, L., RODRIGUES, I. “Using a Logic Programming Framework to Control Database Query Dialogues in Natural Language”. In: *Proceedings of ICLP International Conference in Logic Programming*, pp. 210–216, Seattle, WA, USA, 2006.

[96] RAMAKRISHNAN, C. R., RAMAKRISHNAN, I. V., WARREN, D. S. “Deductive Spreadsheets Using Tabled Logic Programming.”. In: *Proceedings of ICLP International Conference in Logic Programming*, pp. 391–405, Seattle, WA, USA, 2006.

[97] RAU, B. R. “Levels of Representation of Programs and the Architecture of Universal Host Machines”. In: *Proceedings of Micro 11*, pp. 67–79, California, USA, November 1978.

[98] REIF, J. H., LEWIS, H. R. “Efficient Symbolic Analysis of Programs”. *Computer Systems*, v. 32, n. 3, pp. 280–313, June 1986.

[99] ROMER, T. H., LEE, D., VOELKER, G. M., WOLMAN, A., WONG, W. A., BAER, J.-L., BERSHAD, B. N., LEVY, H. M. "The Structure and Performance of Interpreters". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, v. 31, pp. 150–159, Cambridge, Massachusetts, USA, October 1996. ACM Press.

[100] SAGONAS, K. F., SWIFT, T., WARREN, D. S. "The XSB Programming System". In: *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS*, pp. 164–195, Vancouver, British Columbia, Canada, October 1993.

[101] SANTOS COSTA, V. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University Of Bristol, UB, GrÃ£-Bretanha, August 1993.

[102] SANTOS COSTA, V. "Optimizing Bytecode Emulation for Prolog". In: *LNCS 1702, Proceedings of the Principles and Practice of Declarative Programming*, pp. 261–267, Paris, France, September 1999. Springer-Verlag.

[103] SCHEIFLER. "An Analysis of Inline Substitution for a Structured Programming Language". *Commun ACM*, v. 20, n. 9, pp. 647–654, September 1977.

[104] "SICSTUS PROLOG". *http://www.sics.se/isl/sicstuswww/site/index.html*; accessed January 12, 2006.

[105] SILICON GRAPHICS. *MIPS Assembly Language Programmer's Guide*. Technical Report ASM-01-DOC, Silicon Graphics, October 1992.

[106] SILVA, A. F., SANTOS COSTA, V. "An Experimental Evaluation of JAVA JIT Technology". In: *Proceedings on 9th Brazilian Symposium on Programming Languages*, pp. 35–50, Recife, Brasil, May 2005.

[107] SILVA, A. F., SANTOS COSTA, V. "Design of the YAPc Compiler: An Optimizing Compiler for Logic Programming Languages". In: *Proceedings on 10th Brazilian Symposium on Programming Languages*, pp. 35–50, Itatiaia-RJ, Brasil, May 2006.

[108] SILVA, A. F., SANTOS COSTA, V. "Our Experiences with Optimizations in Sun's Java Just-in-time Compilers". In: *Proceedings on 10th Brazilian Symposium on Programming Languages*, pp. 51–65, Itatiaia-RJ, Brasil, May 2006.

[109] SPINELLIS, D. "Declarative Peephole Optimization Using String Pattern Matching". In: *Proceedings of ACM SIGPLAN Notices*, pp. 47–51, Vancouver, Canada, February 1999.

[110] STALLINGS, W. *Computer Organization and Architecture*. 1 ed., New York, USA, Prentice Hall, 2003.

[111] STARLING, L., SHAPIRO, E. *The Art of Prolog*. 1 ed., New York, USA, MIT Press, 1986.

[112] SUGANUMA, T., OGASAWARA, T. "Overview of the IBM Java Just-in-Time Compiler". *IBM Systems Journal*, v. 39, n. 1, pp. 66–76, December 2000.

[113] TAMURA, N. "Knowledge-Based Optimization in Prolog Compiler". In: *Proceedings of the Computer Society Fall Joint Conference*, pp. 237–240, California, USA, November 1986.

[114] TAYLOR, A. "Parma - Bridging the Performance GAP Between Imperative and Logic Programming". *Journal of Logic Programming*, v. 29, n. 1-3, pp. 5–16, October 1996.

[115] TURK, A. K. "Compiler Optimizations for the WAM". In: *Proceedings of International Conference on Logic Programming*, pp. 657–662, July 1986.

[116] VAN ROY, P. "An Intermediate Language to Support Prolog's Unification". In: *Proceedings of North American Conference on Logic Programming*, pp. 1148–1164, Cleveland, Ohio, USA, October 1989.

[117] VAN ROY, P. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California, Berkeley, California, USA, December 1990.

[118] VAN ROY, P. "1983-1993: The Wonders Year of Sequential Prolog Implementation". *Journal of Logic Programming*, v. 29, n. 1-3, pp. 5–16, December 1994.

[119] VAN ROY, P., DESPAIN, A. "High Performance Logic Programming with the Aquarius Prolog Compiler". *IEEE Computer Magazine*, v. 39, n. 1, pp. 54–68, December 1992.

[120] VENNERS, B. *Inside the Java 2 Virtual Machine*. 2 ed., New York, USA, Mc Graw Hill, 1999.

[121] WADDELL, O., DYBIG, R. K. "Fast and Effective Procedure Inlining". In: *Proceedings of Static Analysis Symposium*, pp. 35–52, Paris, France, September 1997.

[122] WARREN, D. H. D. *Applied Logic - Its Use and Implementation as a Programming Tool*. PhD thesis, University of Edinburgh, April 1977.

[123] WARREN, D. H. D., PEREIRA, F. C. N. "An Efficient Easily Adaptable System for Interpreting Natural Language Queries". *American Journal of Computational Linguistics*, v. 8, n. 3-4, pp. 110–122, July 1982.

[124] WARREN, D. H. D. *Implementing Prolog - Compiling Predicate Logic Programs*. Technical Report 39-40, Department of Artificial Intelligence, University of Edinburgh, October 1977.

[125] WARREN, D. H. D. *An Abstract Prolog Instruction Set*. Technical Report 4776, Artificial Intelligence Center, SRI International, October 1983.

[126] WARREN, R., HERMENEGILDO, M., DEBRAY, S. K. "On the Practicality of Global Flow Analysis of Logic Programs". In: *Proceedings of the International Conference and Symposium on Logic Programming*, pp. 684–699, Seattle, Washington, USA, 1988.

[127] WEGMAN, M. N., ZADECK, F. K. "Constant Propagation with Conditional Branches". *ACM Transactions on Programming Languages and Systems*, v. 13, n. 2, pp. 181–210, April 1991.

[128] WIELEMAKER, J. "An Overview of the SWI-Prolog Programming Environment". In: *Proceedings of the 13th International Workshop on Logic Programming Environments*, pp. 1–16, Heverlee, Belgium, December 2003. Katholieke Universiteit Leuven. CW 371.

[129] YANG, B.-S., MOON, S.-M. "LaTTe - A Java VM Just-in-Time Compiler whith Fast and Efficient Register Allocation". In: *Proceedings of International Conference on Parallel Architecture and Compilation Technique*, pp. 362–387, Newport Beach, California, USA, October 1999.

[130] "YAP PROLOG SYSTEM". *http://www.ncc.up.pt/ vsc/Yap/*; accessed January 12, 2006.

[131] ZHAO, P., AMARAL, J. N. "To Inline or Not to Inline? Enhanced Inlining Decisions". In: *Proceedings of 16th Workshop on Languages and Compilers for Parallel Computing*, pp. 405–419, Texas, USA, October 2003.

Apêndice A

Append

?- append([1,2,3,...,100000]),[],_).

Prolog	**YAAM**	
append([],L,L).	name	append/3,3
	get_atom	[],A1
	get_val	X2,A3
	proceed	
append([X|L1],L2,[X|L3]):- append(L1,L2,L3).	name	append/3,3
	get_list	A1
	unify_var	X4
	unify_last_var	X1
	get_list	A3
	unify_val	X4
	unify_last_var	X3
	execute	user:append/3

Apêndice B

Naive Reverse

?- nreverse([1,2,3,...,5000]),_).

Prolog	**YAAM**	
nreverse([],[]).	name	nreverse/2,2
	get_atom	[],A1
	get_atom	[],A2
	proceed	
nreverse([X|L0],L):-	nreverse/2,2	
nreverse(L0,L1),	get_list	A1
append(L1,[X],L).	unify_var	Y1
	unify_last_var	X1
	get_var	Y0,A2
	allocate	
	put_var	Y2,A2
	call	user:nreverse/2,3,1
	put_unsafe	Y2,A1
	put_list	A2
	write_val	Y1
	write_atom	[]
	put_val	Y0,A3
	deallocate	
	execute	user:append/3

Prolog	**YAAM**	
append([],L,L).	name	append/3,3
	get_atom	[],A1
	get_val	X2,A3
	proceed	
append([X|L1],L2,[X|L3]):-	name	append/3,3
append(L1,L2,L3).	get_list	A1
	unify_var	X4
	unify_last_var	X1
	get_list	A3
	unify_val	X4
	unify_last_var	X3
	execute	user:append/3

Apêndice C

Zebra

```
?- houses(Houses),
    my_member(house(red, english, _, _, _), Houses),
    my_member(house(_, spanish, dog, _, _), Houses),
    my_member(house(green, _, _, coffee, _), Houses),
    my_member(house(_, ukrainian, _, tea, _), Houses),
    right_of(house(green,_,_,_,_), house(ivory,_,_,_,_), Houses),
    my_member(house(_, _, snails, _, winstons), Houses),
    my_member(house(yellow, _, _, _, kools), Houses),
    Houses = [_, _, house(_, _, _, milk, _), _,_],
    Houses = [house(_, norwegian, _, _, _)|_],
    next_to(house(_,_,_,_,chesterfields), house(_,_,fox,_,_), Houses),
    next_to(house(_,_,_,_,kools), house(_,_,horse,_,_), Houses),
    my_member(house(_, _, _, orange_juice, lucky_strikes), Houses),
    my_member(house(_, japanese, _, _, parliaments), Houses),
    next_to(house(_,norwegian,_,_,_), house(blue,_,_,_,_), Houses),
    my_member(house(_, _, zebra, _, _), Houses),
    my_member(house(_, _, _, water, _), Houses).
```

Prolog

```
houses([house(_, _, _, _, _),
   house(_, _, _, _, _),
   house(_, _, _, _, _),
   house(_, _, _, _, _),
   house(_, _, _, _, _)]).
```

YAAM

```
name             houses/1,1
get_list         A1
unify_struct     house/5
unify_var        X0
unify_var        X0
unify_var        X0
unify_var        X0
unify_last_var   X0
pop              L1
unify_last_list
unify_struct     house/5
unify_var        X0
unify_var        X0
unify_var        X0
unify_var        X0
unify_last_var   X0
pop              L1
unify_last_list
unify_struct     house/5
unify_var        X0
unify_var        X0
unify_var        X0
unify_var        X0
unify_last_var   X0
pop              L1
unify_last_list
unify_struct     house/5
unify_var        X0
unify_var        X0
unify_var        X0
unify_var        X0
unify_last_var   X0
pop              L1
unify_last_list
unify_struct     house/5
unify_var        X0
unify_var        X0
unify_var        X0
unify_var        X0
unify_last_var   X0
pop              L1
unify_last_atom  []
proceed
```

Prolog	**YAAM**	
right_of(A, B, [B, A | _]).	name	right_of/3,3
	get_list	A3
	unify_local	X2
	unify_last_list	
	unify_local	X1
	unify_last_var	X0
	proceed	
right_of(A, B, [_ | Y]) :- right_of(A, B, Y).	name	right_of/3,3
	get_list	A3
	unify_var	X0
	unify_last_var	X3
	execute	user:right_of/3
next_to(A, B, [A, B | _]).	name	next_to/3,3
	get_list	A3
	unify_local	X1
	unify_last_list	
	unify_local	X2
	unify_last_var	X0
	proceed	
next_to(A, B, [B, A | _]).	name	next_to/3,3
	get_list	A3
	unify_local	X2
	unify_last_list	
	unify_local	X1
	unify_last_var	X0
	proceed	
next_to(A, B, [_ | Y]) :- next_to(A, B, Y).	name	next_to/3,3
	get_list	A3
	unify_var	X0
	unify_last_var	X3
	execute	user:next_to/3
my_member(X, [X|_]).	name	my_member/3,3
	get_list	A2
	unify_local	X1
	unify_last_var	X0
	proceed	

Prolog	**YAAM**	
my_member(X, [_|Y]) :-	name	my_member/3,3
my_member(X, Y).	get_list	A2
	unify_var	X0
	unify_last_var	X2
	execute	user:my_member/3

Apêndice D

Hanoi

?- han(20,1,2,3).

Prolog	**YAAM**	
han(N,_,_,_) :- N =< 0.	name	han/4,4
	put_var	X2,A0
	get_num	0,A0
	fetch_args_for_ bccall	X1
	binary_cfuc	X2,prolog:=</2
	proceed	
han(N,A,B,C) :- N > 0,	name	han/4,4
N1 is N-1,	get_var	Y0,A2
han(N1,A,C,B),	get_var	Y1,A3
han(N1,C,B,A).	get_var	Y2,A4
	allocate	
	put_var	X2,A0
	get_num	0,A0
	fetch_args_for_bccall	X1
	binary_cfunc	X2,prolog:>/2
	fetch_reg_constant	0x0xffffffff,X1/0
	function_to_var	Y3,plus
	put_val	Y3,A1
	put_val	Y0,A2
	put_val	Y2,A3
	put_val	Y1,A4
	call	user:han/4,4,1
	put_unsafe	Y3,A1
	put_val	Y2,A2
	put_val	Y1,A3
	put_val	Y0,A4
	deallocate	
	execute	user:han/4

Apêndice E

Quick Sort

?- qsort([27,74,17,33,94,18,46,...,100000],_,[]).

Prolog	**YAAM**	
qsort([X|L],R,R0) :-	name	qsort/3,3
partition(L,X,L1,L2),	get_list	A1
qsort(L2,R1,R0),	unify_var	Y1
qsort(L1,R,[X|R1]).	unify_last_var	X1
	get_var	Y2,A2
	get_var	Y4,A3
	allocate	
	put_val	Y1,A2
	put_var	Y3,A3
	put_var	Y5,A4
	call	user:partition/4,6,1
	put_unsafe	Y5,A1
	put_var	Y0,A2
	put_val	Y4,A3
	call	user:qsort/3,4,2
	put_unsafe	Y3,A1
	put_val	Y2,A2
	put_list	A3
	write_val	Y1
	write_local	Y0
	deallocate	
	execute	user:qsort/3
qsort([],R,R).	name	qsort/3,3
	get_atom	[],A1
	get_val	X2,A3
	proceed	

Prolog	**YAAM**	
partition([X|L],Y,[X|L1],L2) :-	name	partition/4,4
X =< Y, !,	get_list	A1
partition(L,Y,L1,L2).	unify_var	X5
	unify_last_var	X1
	get_list	A3
	unify_val	X5
	unify_last_var	X3
	fetch_args_for_bccall	X5
	binary_cfunc	X2,prolog:=</2
	cut	
	execute	user:partition/4
partition([X|L],Y,L1,[X|L2]) :-	name	partition/4,4
partition(L,Y,L1,L2).	get_list	A1
	unify_var	X5
	unify_last_var	X1
	get_list	A4
	unify_val	X5
	unify_last_var	X4
	execute	user:partition/4
partition([],_,[],[]).	name	partition/4,4
	get_atom	[],A1
	get_atom	[],A3
	get_atom	[],A4
	proceed	

Apêndice F

Tak

?- tak(18,12,6,_).

Prolog	**YAAM**	
tak(X,Y,Z,A):-	name	tak/4,4
X =< Y,	fetch_args_for_bccall	X1
Z = A.	binary_cfunc	X2,prolog:=</2
	put_val	X3,A0
	get_val	X4,A0
	proceed	

Prolog	YAAM	
tak(X,Y,Z,A):-	name	tak/4,4
X > Y,	get_var	Y5,A1
X1 is X - 1,	get_var	Y5,A2
tak(X1,Y,Z,A1),	get_var	Y5,A3
Y1 is Y - 1,	get_var	Y5,A4
tak(Y1,Z,X,A2),	allocate	
Z1 is Z - 1,	fetch_args_for_bccall	Y5
tak(Z1,X,Y,A3),	binary_cfunc	Y4,prolog:>/2
tak(A1,A2,A3,A).	fetch_reg_constant	0x0xffffffff,Y58977361/0
	function_to_var	X2,plus
	put_val	X2,A1
	put_val	Y4,A2
	put_var	Y3,A4
	call	uuser:tak/4,7,1
	fetch_reg_constant	0x0xffffffff,Y48977364/0
	put_val	Y4,A1
	function_to_var	X2,plus
	put_val	X2,A1
	put_val	Y6,A2
	put_val	Y5,A3
	put_var	Y2,A4
	call	user:tak/4,7,2
	fetch_reg_constant	0x0xffffffff,Y68977368/0
	put_val	Y6,A1
	function_to_var	X2,plus
	put_val	X2,A1
	put_val	Y5,A2
	put_val	Y4,A3
	put_var	Y1,A4
	call	user:tak/4,4,3
	put_unsafe	Y3,A1
	put_unsafe	Y2,A2
	put_unsafe	Y1,A3
	put_val	Y0,A4
	deallocate	
	execute	user:tak/4