**Xingó - Compilação para uma Representação Intermediária Executável**

*Wesley Attrot*

**Dissertação de Mestrado**

Instituto de Computação
Universidade Estadual de Campinas

# Xingó - Compilação para uma Representação Intermediária Executável

**Wesley Attrot**[1]

Abril de 2004

**Banca Examinadora:**

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Profa. Dra. Mariza A. Silva Bigonha
  Departamento de Ciência da Computação - UFMG
- Profa. Dra. Cecília Mary Fischer Rubira
  Instituto de Computação - UNICAMP
- Prof. Dr. Arnaldo Vieira Moura (Suplente)
  Instituto de Computação - UNICAMP

[1]**Trabalho com suporte do CNPq, processo 131094/2002-5 e FAPESP, processo 00/15083-9**

# Xingó - Compilação para uma Representação Intermediária Executável

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Wesley Attrot e aprovada pela Banca Examinadora.

Campinas, 16 de Abril de 2004.

Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

“Toda grande jornada começa com um primeiro passo.”

Provérbio Oriental

“Whether you think you can or think you can’t - You are right.’

Henry Ford (1863 – 1947)

# Resumo

O aumento da complexidade dos novos projetos de microprocessadores aumentou consideravelmente a necessidade de compiladores que sejam capazes de gerar código altamente otimizado. Isto resultou em uma grande demanda por novas otimizações de código que possam fazer uso dos novos recursos da arquitetura do processador. Projetar tais otimizações é um trabalho muito complexo que requer uma plataforma de compilação flexível e simples de usar.

O compilador Xingó foi desenvolvido como uma ferramenta para auxiliar o trabalho de pesquisa em áreas como otimização de código e arquitetura de computadores. Xingó é um compilador otimizante capaz de gerar código C a partir de sua representação intermediária. Tal característica auxilia o desenvolvedor a avaliar a corretude e o desempenho de novas otimizações de código. Xingó também é um compilador redirecionável, isto é, pode ser facilmente portado para várias arquiteturas.

O compilador Xingó possui oito otimizações independentes de máquina (meados de março de 2004) e que foram avaliadas com a ajuda do *benchmark* NullStone, o qual é um *benchmark* para compiladores de produção de alta qualidade, amplamente utilizado por companias da indústria da computação. Do total de 6611 testes realizados pelo NullStone, o Xingó produziu código correto para 6581 (99,54%) sem a aplicação de nenhuma otimização. Depois da aplicação de todas as otimizações disponíveis, o Xingó produziu código correto para 6497 testes (98,27%). Estes números refletem a qualidade da plataforma de compilação Xingó, abrindo uma nova gama de oportunidades de pesquisa nas áreas de tecnologias de compilação e projeto de arquiteturas no Laboratório de Sistemas de Computação (IC-UNICAMP). Pretende-se tornar o Xingó de domínio público em meados de 2005.

# Abstract

The increasing complexity of the new microprocessor designs has considerably increased the pressure for compilers that are capable of generating highly optimized code. This has resulted in a great demand for new code optimizations which can make an effective usage of the processor architectural resources. Designing such optimnizations is usually a very complex task that requires a flexible and easy to use compiler platform.

The Xingó compiler was designed as a tool to help researchers working in areas like compiling optimization and computer architecture. Xingó is an optimizing compiler capable of generating compilable C code from its intermediate representation. Such feature considerably helps the designer in evaluating the correcteness and performance of new code optimization techniques. Xingó is also a retargetable compiler, that is, it can be easily ported for several architectures.

The Xingó compiler has eight machine independent code optimizations (circa March 2004) and was evaluated with the help of the NullStone *benchmark*, a production-quality compiler *benchmark*, largely used by companies in the computer industry. Out of the 6611 programs available at Nullstone, Xingó produced correct code for 6581 (99,54%) without the application of any optimization. After applying all the available optimizations, Xingó produced correct code for 6497 programs (98,27%). These numbers reflect the quality of the Xingó compiling platform, opening up a new set of research opportunities in areas like compiling technology and architecture design at the Computer Systems Laboratory, (IC-UNICAMP). Xingó is planned to go on public domain by early 2005.

# Agradecimentos

Primeiramente eu gostaria de agradecer ao meu orientador, o Prof. Dr. Guido Costa Souza de Araújo. Ele foi fundamental para a realização deste trabalho, pois ele me motivou e auxiliou em todos os momentos, nunca me deixando desanimar, sempre dando sugestões e sugerindo caminhos a seguir quando tudo parecia nebuloso. Obrigado Guido.

Eu também gostaria de agradecer aos amigos que conheci no Instituto de Computação e na Unicamp pelos bons e felizes momentos que vivemos juntos. A amizade é um grande tesouro.

Eu também quero agradecer aos amigos que conheci na pensão durante o período em que morei em Campinas. Duarte, Gregório (gregs), Marcelo (Mr. Bomba) e Márcio Pardo (meus companheiros não só de pensão, mas também de graduação na UEL). Raúl (mestre argentino da matemática), Alan (você fez suas escolhas), Walkíria (bebíria) e Karime (peps, nunca vou esquecer das nossas conversas), os grandes amigos que ganhei em Campinas. Pessoal, um obrigado muito especial para vocês.

Quero agradecer também a todos aqueles que direta ou indiretamente ajudaram na realização deste trabalho.

E finalmente eu quero agradecer a meus pais, pois eles me ajudaram e deram todo o apoio necessário para que eu pudesse realizar mais esta etapa em minha vida. Se na etapa anterior eles foram fundamentais, nesta eles se superaram. Somente eles conhecem todas as dificuldades pelas quais tive que passar antes de chegar a este momento. Obrigado.

# Sumário

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

Devido ao aumento da complexidade dos novos processadores, compiladores de alto desempenho e qualidade são essenciais para se conduzir experimentos em Arquitetura de Computadores e para se testar novas idéias de otimização e geração de código.

A construção de um compilador é algo extremamente trabalhoso, e pode se tornar o "gargalo" de um projeto de pesquisa. Para testar novas arquiteturas, é necessário desenvolver um gerador de código para a mesma; para se testar uma nova otimização é preciso que haja um compilador onde a mesma possa ser inserida, e muitas vezes tal compilador não está disponível. Estes dois exemplos mostram situações onde pode ser necessário escrever um compilador completo de modo a testar apenas pequenos módulos, o que fatalmente acarretará um acréscimo no tempo de pesquisa e na complexidade do projeto.

De modo a tentar resolver tais problemas, o presente trabalho apresenta o compilador Xingó. O Xingó é um compilador ANSI C, desenvolvido em C++, que tem o objetivo de facilitar o desenvolvimento de pesquisas em otimização de código e novas técnicas de geração de código, bem como facilitar o desenvolvimento de novas arquiteturas.

O Xingó possui uma representação intermediária de leitura simples, com sintaxe próxima à da linguagem C. Tal representação pode ainda ser convertida em um código fonte C e compilada como outro programa qualquer. Essa característica visa facilitar e tornar rápido o processo de validação de novas otimizações, pois permite acompanhar cada transformação que é realizada na representação intermediária. O foco deste trabalho é o processo de conversão da representação intermediária do Xingó para código C, onde são apresentados os diversos problemas de tal conversão e as soluções para os mesmos.

No tocante à parte de geração de código, o Xingó utiliza um gerador de gerador de código para implementar o seu *back-end*, de modo a poder ser portado para diversas arquiteturas. Ele possui ainda um conjunto de otimizações independentes de máquina, bem como um conjunto de classes abstratas que permitem a realização de *post-pass data*

*flow analysis*, dando suporte à implementação de otimizações dependentes de máquina.

O restante deste trabalho está organizado como segue: o capítulo 2 apresenta um conjunto de trabalhos relacionados, bem como uma breve descrição dos mesmos. Atenção especial é dada a um deles (LCC), por fazer parte do Xingó. O capítulo 3 apresenta o compilador Xingó, a sua representação intermediária, bem como todos os passos para convertê-la em um código fonte C compilável (foco deste trabalho). O capítulo 4 apresenta as otimizações que integram o Xingó. O capítulo 5 apresenta os resultados obtidos no processo de conversão da representação intermediária do compilador para código C e finalmente o capítulo 6 lista as conclusões e propõe um guia de trabalhos futuros.

# Capítulo 2

# Trabalhos Relacionados

## 2.1 SPAM

O compilador SPAM [29] é um *retargetable compiler*, isto é, um compilador redirecionável para *fixed-point* DSPs[1] desenvolvido em Princeton. O SPAM utiliza o Stanford University Intermediate Format (SUIF) compiler [19] para implementar o seu *front-end*. SUIF, que inicialmente foi desenvolvido para estudar questões relativas à paralelização de código, traduz código C em uma representação intermediária ou *Intermediate Representation* (IR). SPAM toma como entrada esta IR e aplica uma série de otimizações independentes de máquina.

O *back-end* do SPAM, denominado TWIF, realiza otimizações dependentes de máquina. O TWIF é composto de dois componentes: uma biblioteca de estruturas de dados que encapsulam as várias representações do programa fonte e uma biblioteca de algoritmos que realizam a geração de código e as *post-pass optimizations* (otimizações dependentes de máquina) através da análise e manipulação de tais estruturas. A IR do TWIF possui uma estrutura hierárquica, propagando informações de alto nível do código fonte para níveis mais baixos. Um exemplo é a manutenção de construções de alto nível como laços *for* e a indexação de *arrays*. Tal estrutura pode ser útil para o otimizador, visto que ela fornece melhores oportunidades de otimização; em contrapartida, ela pode revelar-se muito complexa de se manipular e modificar em determinados casos.

Os algoritmos de geração de código do TWIF traduzem a IR já otimizada em um código *assembly* preliminar. Entre os algoritmos presentes no TWIF está incluído o gerador de gerador de código Olive [31], que é baseado nos geradores de geradores de código Twig [3] e Iburg [15]. O Olive toma como entrada uma gramática baseada na descrição da arquitetura da máquina alvo, e então constrói automaticamente o gerador de código

[1] Digital Signal Processors

que executa, em tempo linear, a seleção de instruções para árvores de expressões. A descrição da arquitetura alvo consiste em padrões de árvores da IR, onde cada padrão corresponde a uma instrução da máquina alvo e tem associado a si um custo. O algoritmo de seleção de instruções do Olive trabalha então com casamento de padrões e com técnicas de programação dinâmica, utilizando uma versão modificada do algoritmo de Aho-Johnson [4].

Uma vez que o código *assembly* tenha sido gerado, o TWIF aplica os algoritmos que realizam a *post-pass optimization* com o intuito de reduzir o tamanho do código gerado. Essas otimizações provêem suporte para as características especializadas da arquitetura do processador alvo. Elas são executadas após a geração do código *assembly*, ao invés de serem executadas durante a seleção das instruções, devido ao fato de que as informações sobre a arquitetura, que são necessárias às otimizações, estarem presentes no código *assembly* e não na IR.

## 2.2 LANCE

O compilador LANCE [24] é um *retargetable compiler*. Ele foi feito com o objetivo de ser uma infra-estrutura de desenvolvimento de compiladores C para novos processadores e como plataforma de pesquisa para técnicas de otimização de código.

O *front-end* utilizado pelo LANCE reconhece o C padrão ANSI. Este *front-end* recebe como entrada um programa C e constrói uma IR que é independente da máquina alvo. Somente o tamanho e o alinhamento dos tipos de dados presentes na linguagem C é que devem ser especificados por meio de um arquivo de configuração. A IR utilizada pelo LANCE consiste em código de três endereços, isto é, ao menos três operandos por operação (dois argumentos e um resultado). Todas as construções de alto nível como expressões aritméticas complexas, desvios condicionais e aritmética implícita para o acesso a *arrays* não estão presentes na IR, pois são quebradas em seqüências de operações simples. Adicionalmente todas as conversões de tipo implícitas no código fonte original são traduzidas em operações explícitas de *cast* e o código necessário para a inicialização de variáveis locais é automaticamente inserido. Tal representação facilita a implementação de ferramentas que operam sobre a IR. Uma característica especial da IR utilizada pelo LANCE é que ela é mantida em um *assembly* de baixo nível com sintaxe próxima a do C. Desta forma a IR pode ser convertida em um programa C e ser compilada da mesma forma que o programa fonte original. Tal característica torna a representação adotada pelo LANCE fácil de entender e é uma forte auxiliar no processo de validação do *front-end* e de novas otimizações.

O LANCE contém uma biblioteca de otimizações independentes de máquina, tais como: *constant folding*, *dead code elimination*, bem como otimizações em *loops*. Dependendo

do nível de otimização desejado, tais otimizações podem ser chamadas de forma separada ou executadas via *shell script*.

O fluxo de controle e dos dados de um programa C pode ser analisado e visualizado através de uma ferramenta de visualização de grafos. Essa ferramenta auxilia o entendimento das estruturas do programa e facilita o desenvolvimento de *back-ends* para geradores de código *assembly*.

O *back-end* do compilador LANCE transforma o código de três endereços em *data-flow trees* (DFTs). Cada DFT representa um fragmento da computação do código C e compreende argumentos, operações, posições de memória, bem como as dependências de dados existentes entre elas. O formato de uma DFT gerada pelo LANCE é compatível com geradores de geradores de código como o Iburg e o Olive. Desta forma *back-ends* para arquiteturas específicas podem ser desenvolvidos de forma rápida.

## 2.3 The Zephyr Compiler Infrastructure

O objetivo do projeto Zephyr [7] é possibilitar a construção de sistemas completos de compilação através da substituição ou modificação somente dos módulos que são importantes para o projeto em foco. O Zephyr pretende reduzir significativamente os custos de se realizar pesquisas em sistemas de computação no que se refere a tempo e dinheiro, encorajando novas pesquisas e aumentando a freqüência com que novos resultados são produzidos.

O projeto Zephyr inclui o *front-end* SUIF, que propicia um formato comum no qual as diferentes linguagens de programação podem ser expressas e fornece suporte a otimizações de "alto-nível"em áreas como paralelização automática.

Toda a infra-estrutura do Zephyr é construída em torno do *Very Portable Optimizer* (VPO). O VPO aceita otimizações de baixo nível em programas representados ao nível de instruções de máquina. Ao invés de focar em um compilador, o Zephyr foca em *estratégias* de compilação, que podem ser utilizadas para se construir diferentes compiladores. A estratégia básica do Zephyr consiste em construir compiladores através de módulos, onde:

- O desenvolvedor fornece um *front-end*.
- O Zephyr fornece um conjunto de ferramentas, entre elas:
  - otimizador independente de máquina, VPO;
  - componentes para *Back-ends*;
  - *Glue* para conectar todos os componentes do compilador.
- O desenvolvedor escolhe uma IR como alvo do *front-end*.

- O Zephyr fornece um guia para conectar a IR ao otimizador e ao *back-end*.
- Os componentes dependentes de máquina são gerados através de especificações da máquina alvo.

O VPO é o elemento crucial do Zephyr. Sem o VPO, os demais módulos conseguem gerar apenas um código muito simples, enquanto que a inserção do mesmo torna o código competitivo com o GCC [1] e com outros compiladores comerciais, sendo que em algumas ocasiões o código gerado pelo VPO é melhor e em outras o código é pior. Para ser otimizado pelo VPO, um programa deve ser expressado como um conjunto de procedimentos, onde cada procedimento deve ser representado como um grafo de fluxo de controle ou *control flow graph* (CFG), onde os elementos individuais são as Zephyr RTLs (*Register-Transfer Lists*). Dado um programa qualquer, o VPO o otimizará repetidamente até que nenhuma melhora seja possível.

Jung e Paek [22] utilizaram a estrutura do Zephyr para desenvolver um compilador para DSPs, demonstrando, desta forma, a capacidade de portabilidade desta infra-estrutura.

## 2.4 LCC

O LCC [16] é um compilador redirecionável para ANSI C, que foi desenvolvido por David Hanson e Christopher Fraser na Universidade de Princeton e nos laboratórios da AT&T. Ele é um compilador de produção que vem sendo utilizado diariamente desde 1988 por centenas de programadores C. O LCC será mais detalhado do que os demais projetos anteriormente apresentados, pois ele é utilizado como *front-end* do Xingó.

O programa apresentado na Figura 2.1 será utilizado para ilustrar o funcionamento do LCC em cada um de seus passos. Cada passo muda a forma com que o programa é representado: código pré-processado, *tokens*, árvores, *directed acyclic graphs* (DAGs - Grafos Dirigidos Acíclicos) e listas de DAGs.

```
int round(f) float f;
{
        return f + 0.5;
}
```

Figura 2.1: Programa exemplo

O primeiro passo no processo de compilação consiste em realizar o pré-processamento do programa, o que irá expandir macros, incluir arquivos de cabeçalho e selecionar código

da compilação condicional. O resultado do pré-processamento é apresentado na Figura 2.2.

```
# 1 "sample.c"
int round(f) float f;
{
        return f + 0.5;
}
```

Figura 2.2: Programa pré-processado

O próximo passo consiste em realizar a análise léxica do programa fonte, a qual irá quebrá-lo em uma seqüência de *tokens*. A Tabela 2.1 mostra a seqüência de *tokens* criada pelo LCC para o programa da Figura 2.2.

A coluna da esquerda corresponde ao código (um valor inteiro) do *token*. A coluna da direita contém o valor associado, caso exista algum. Por exemplo, o valor associado com a palavra reservada *int* é *inttype*. *Tokens* formados por apenas um único caractere utilizam o seu valor ASCII como código.

| | |
|---|---|
| INT | inttype |
| ID | "round" |
| '(' | |
| ID | "f" |
| ')' | |
| FLOAT | floattype |
| ID | "f" |
| ';' | |
| '{' | |
| RETURN | |
| ID | "f" |
| '+' | |
| FCON | "0.5" |
| ';' | |
| '}' | |
| EOI | |

Tabela 2.1: *Tokens* gerados pelo LCC

A próxima fase no processo de compilação é a análise sintática, na qual é verificada se a seqüência de *tokens* obedece às regras sintáticas da linguagem C. Ao mesmo tempo em

que a análise sintática ocorre, o LCC também realiza a análise semântica, a qual verifica o tipo dos operandos em cada uma das operações presentes no programa. A saída gerada pelas fases de análise sintática e semântica formam as *Abstract Syntax Trees* (Árvores Sintáticas Abstratas). A Figura 2.3 mostra as *Abstract Syntax Trees* geradas pelo LCC para o programa da Figura 2.2.

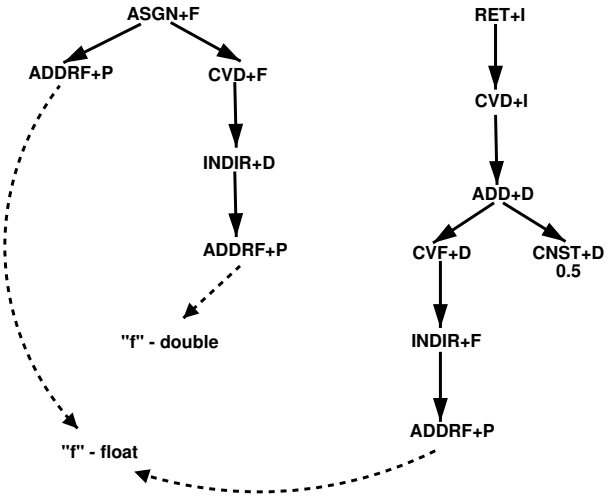


Figura 2.3: *Abstract Syntax Trees* geradas pelo LCC

Cada nó representa uma operação básica. A primeira árvore reduz um double para um float. Ela atribui um valor float (ASGN+F) para a célula com o endereço &f (ADDRF+P à esquerda). Ela computa o valor a ser atribuído convertendo para float (CVD+F) o valor double carregado em (INDIR+D) do endereço &f (ADDRF+P a direita). A segunda árvore retorna um int (RET+I). O valor é computado carregando-se o float (INDIR+F) vindo da célula com o endereço &f (ADDRF+P), convertendo-o para double, somando-se (ADD+D) à constante double 0.5 (CNST+D) e truncando-se o resultado para um inteiro (CVD+I). Estas árvores tornam explícitos muitos fatos que estão implícitos na linguagem fonte, como por exemplo as conversões de tipo.

Depois que as *Abstract Syntax Trees* estão construídas, o LCC as transforma em *DAGs*. A Figura 2.4 mostra os *DAGs* gerados para o programa da Figura 2.2.

Os *DAGs* 1 e 2 são gerados a partir das duas árvores na Figura 2.3. Os operadores são escritos sem o sinal + para identificar as estruturas como *DAGs* ao invés de árvores. A transição de árvores para *DAGs* torna explícito alguns fatos, como por exemplo, a constante 0.5, que aparecia em um nó CNST+D na árvore, aparece agora no *DAG* como o valor de uma variável do tipo *static* de nome "2". Outra alteração pode ser observada no operador CNST+D, que foi substituído por operações que computam o endereço da variável (ADDRGP) e retornam o seu valor (INDIRD). O terceiro *DAG* define um *label* de nome "1". Operações de *return* são transformadas em desvios para este *label*.

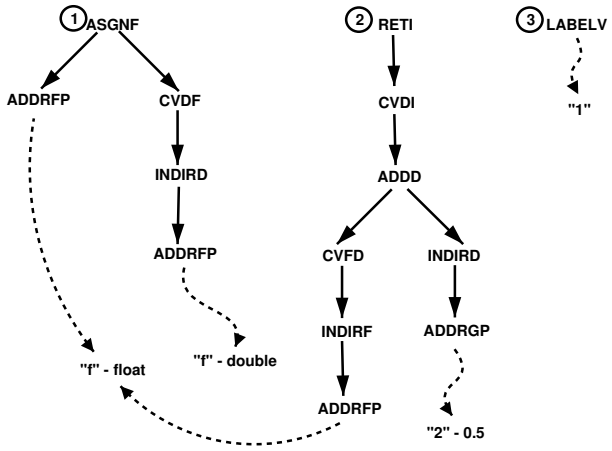


Figura 2.4: DAGs gerados pelo LCC

Outro fato advindo da transformação de árvores em *DAGs*, é a eliminação de subexpressões comuns[2], o que pode ser considerado uma forma preliminar de otimização do código gerado.

### 2.4.1 Escolha do LCC

Embora o compilador LANCE também seja um compilador ANSI C e já gere uma representação intermediária com sintaxe próxima a da linguagem C (o que teoricamente tornaria o desenvolvimento do Xingó mais simples), ele não foi escolhido como *front-end* do Xingó pois ele não é um compilador de código aberto e ainda está sob desenvolvimento. Desta forma ele não atende as necessidades do Xingó, o qual é um compilador que pretende-se tornar de domínio público.

O LCC foi ecolhido como *front-end* do Xingó pelo fato de ser razoavelmete estável e rápido (como dito anteriormente, ele vem sendo desenvolvido e aperfeiçoado desde 1988), por ser um compilador de código aberto e pelo fato de que sua documentação é gerada utilizando-se *literate programming*, isto é, ela é feita por meio de um livro. Tal livro [16], mostra todo o funcionamento e a estrutura interna do compilador. Desta forma torna-se muito simples entender, usar e modificar o LCC.

[2]Esse conceito será apresentado adiante.

# Capítulo 3

# O Compilador Xingó

O Xingó é um compilador otimizante para a linguagem C. Ele foi desenvolvido na linguagem C++. O seu *front-end* é constituído pelo LCC, detalhado anteriormente. A representação intermediária do Xingó é obtida convertendo-se a representação em forma de DAGs do LCC para uma representação linear em forma de quádruplas utilizada pelo Xingó. Tal representação é muito próxima da linguagem C, e por isso pode ser convertida para um código fonte C e compilada.

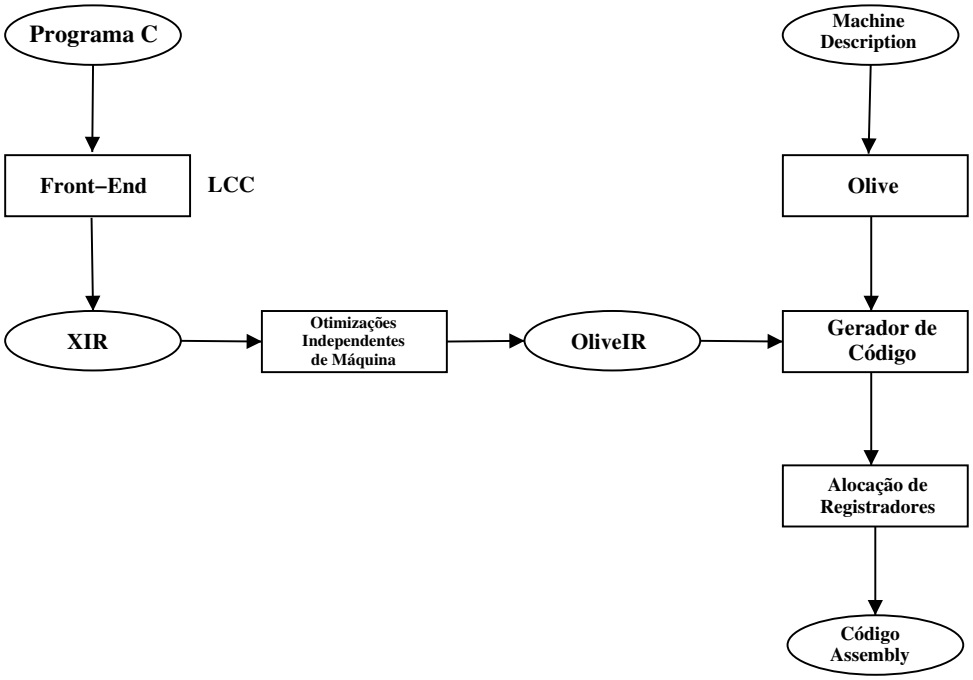


Figura 3.1: Estrutura do compilador Xingó

A Figura 3.1 mostra a estrutura do compilador Xingó, onde pode ser visualizado o

fluxo de funcionamento do mesmo através de seus diversos módulos. A contribuição deste trabalho está focada nos módulos com compõem a XIR e as otimizações independentes de máquina. O restante deste capítulo será dedicado a descrever o funcionamento interno do compilador Xingó e o processo de conversão da sua IR para código C.

## 3.1 A Representação XIR

A representação intermediária utilizada pelo Xingó é chamada de XingoIR (Xingó *Intermediate Representation*), ou simplesmente XIR. Tal representação é constituída por várias classes implementadas em C++ que correspondem às mais diversas estruturas de um programa, como instruções, variáveis, funções e tipos. A partir destas classes, o Xingó constrói uma estrutura que será usada durante todo o processo de compilação. A classe principal desta estrutura é a classe *File*. Esta classe corresponde ao arquivo que está sendo compilado. Ela armazena diversas informações sobre o arquivo, como por exemplo a lista de funções que o arquivo implementa, a lista de funções externas (que estão presentes em outros arquivos) ao arquivo e que são utilizadas internamente, bem como uma tabela de símbolos que armazena todos os símbolos globais, como variáveis e tipos estruturados.

A próxima classe nessa hierarquia é a classe *Procedure*. Cada função presente no arquivo tem associada a si um objeto *Procedure*, mesmo as funções que são *extern*, ou seja, definidas em outros arquivos. Tal classe contém informações como o nome da função, o seu tipo de retorno, o número de argumentos, a tabela de símbolos e o número identificador da mesma. O tipo de retorno da função é armazenado na forma de um número que corresponde ao identificador do objeto que representa tal tipo, que também é representado por uma classe. Os parâmetros da função são armazenados sob a forma de uma lista que contém os seus identificadores. Através de tais identificadores, os parâmetros podem ser localizados na tabela de símbolos local à função.

Se um determinado objeto *Procedure* representa uma função que está implementada em outro arquivo ou uma função presente em alguma biblioteca (como por exemplo a função *printf*), então nada mais resta para ser armazenado. Caso contrário tal objeto corresponde a uma função que está sendo implementada no arquivo sendo compilado, e neste caso esse objeto irá conter uma referência para um outro objeto que é representado pela classe *ProcedureBody*. A classe *ProcedureBody* corresponde ao corpo da função, ou seja, às suas instruções. Esta classe armazena o identificador numérico do objeto, a lista de instruções proveniente da conversão da representação em forma de DAGs do LCC para as instruções da XIR, e uma possível referência para um grafo de fluxo de controle.

A Figura 3.2 mostra o relacionamento entre as classes do Xingó. Tal hierarquia corresponde a um primeiro estágio do processo de compilação.

Cada instrução na XIR constitui uma quádrupla, onde existe um *opcode* e no máximo

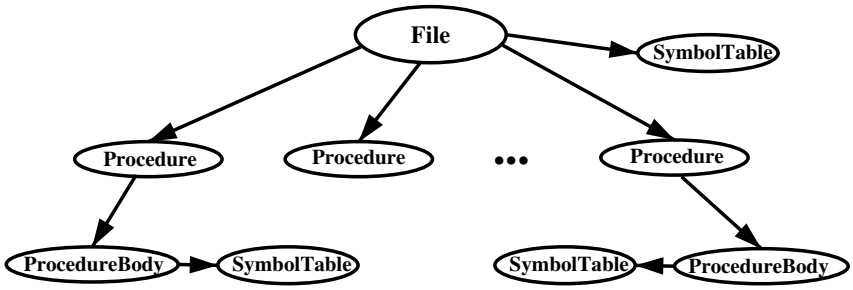


Figura 3.2: Hierarquia de classes do Xingó

três operandos. O *opcode* corresponde ao tipo de operação que a instrução executa, como por exemplo uma soma ou uma multiplicação. Cada um dos operandos corresponde a uma variável ou uma constante.


Figura 3.3: Instrução da XIR

Uma instrução na XIR é implementada pela classe *Instruction*, que engloba todas as informações necessárias para a correta interpretação da mesma, bem como outras informações do projetista. Nesta classe, o *opcode* é representado por um identificador numérico e os operandos são armazenados em um vetor de três posições. Cada variável na XIR possui associada a si um identificador numérico único, assim como todo objeto. A Figura 3.3 ilustra o formato básico de uma instrução da XIR.

O vetor presente no modelo é utilizado para armazenar as variáveis que são utilizadas pela instrução. Ao invés de armazenar a variável, cada posição do vetor armazena o seu identificador numérico. Caso a instrução manipule alguma constante, o seu valor será armazenado diretamente no vetor. O modelo da Figura 3.3 pode ser visualizado como:

$$\mathrm{DST} = \mathrm{SRC1} \oplus \mathrm{SRC2}$$

A posição DST do vetor corresponde a variável destino, ou seja, aquela que vai receber ao resultado da operação. A posição SRC1 corresponde ao primeiro operando fonte, podendo ser uma variável ou uma constante. A posição SRC2 corresponde ao segundo operando fonte, podendo ser também, uma variável ou uma constante. O operador $\oplus$ corresponde a uma operação genérica. Em instruções que envolvam constantes, o campo *ConstPosition* é utilizado para saber se a constante está armazenada na posição SRC1 ou SRC2. Nunca os dois campos irão conter simultaneamente uma constante, pois tal tipo de expressão é avaliada pelo compilador e transformada em uma simples operação

de atribuição. O campo *ConstPosition* pode conter os valores CONST_LEFT (constante localizada na posição SRC1) ou CONST_RIGHT (constante localizada na posição SRC2).

Em instruções comutativas que envolvam constantes, como por exemplo uma soma ou uma multiplicação, a constante sempre será armazenada na posição SRC2 do vetor. Em instruções que não utilizam constantes, *ConstPosition* sempre irá conter o valor CONST_RIGHT. Se uma instrução qualquer fizer uso de alguma constante, sempre que possível, essa constante será armazenada na posição SRC2.

Em instruções que utilizam menos de três operandos, como uma atribuição, os campos não utilizados irão conter o valor zero, o qual não é utilizado como identificador por nenhum objeto.

Na XIR uma variável também é representada por uma classe, a classe *Variable*. Esta classe contém todas as informações importantes relativas a variável, como o seu nome, o seu alinhamento, o segmento a que ela pertence, se em algum momento do programa o endereço da variável foi computado e o tipo da mesma. Uma variável deve pertencer a pelo menos um dos quatro segmentos que são definidos na XIR. Estes segmentos são:

- D_LIT - segmento de constantes.
- D_CODE - segmento de código. Se alguma variável pertencer a este segmento, ela provavelmente será uma variável local.
- D_DATA - segmento de variáveis globais que estão inicializadas.
- D_BSS - segmento de variáveis globais que não estão inicializadas.

O tipo de todas as variáveis, bem como o tipo de retorno de todas as funções na XIR é representado pela classe *Type*. Os tipos que a classe *Type* pode representar estão presentes na Tabela 3.1.

| | | | |
|---|---|---|---|
| • float | • int | • struct | • enum |
| • double | • unsigned | • union | • long |
| • char | • pointer | • function | • const |
| • short | • void | • array | • volatile |

Tabela 3.1: Tipos utilizados pelo Xingó

Tipos *compostos*, como por exemplo um ponteiro para inteiros, são formados agrupando-se vários objetos *Type*, de tal forma que uma lista de tipos é construída. Uma variável que seja do tipo ponteiro para inteiros, terá o seu tipo representado por uma lista de tipos, onde o primeiro elemento da lista é um objeto *Type* representando o tipo **pointer**

e o segundo elemento da lista é um objeto *Type* representando o tipo **int**. Um *array* de inteiros também tem o seu tipo representado como uma lista de objetos *Type*, onde o primeiro representa o tipo **array** e o segundo o tipo **int**. Tal esquema se aplica a todos os tipos que se precise representar internamente no compilador Xingó.

Nunca são criados dois objetos *Type* que representem o mesmo tipo, ou seja, uma vez que existe um objeto *Type* para, por exemplo, representar o tipo char, não será mais necessário criar outro objeto *Type*. O Xingó faz uma reutilização dos objetos *Type*, de forma que um único objeto *Type* representando o tipo char, será utilizado para representar todas as variáveis que sejam do tipo char ou que sejam de algum tipo composto envolvendo char, como um *char∗*, *unsigned char* ou *char[]*.

Certas instruções e variáveis podem ter associadas a si alguma informação extra que pode ser útil em algum momento do processo de compilação, como a geração do código C ou durante a geração de código *assembly*. Para armazenar estas informações a XIR se utiliza da classe *Note*. A classe *Note* pode armazenar o valor com que uma determinada variável do segmento D_DATA será inicializada, o valor das cadeias de um *array* de *strings*, como também pode estar associada a uma instrução do programa e armazenar informações relativas a uma *Jump Table*. Um objeto *Note* pode representar um dos seguintes tipos:

- D_ADDRESS: definição de endereço
- D_SKIP: definição de espaço
- D_BYTES: definição de bytes
- D_FLOAT: definição de float
- D_DOUBLE: definição de double
- D_STRING: definição de *string*
- D_JUMP_TABLE: definição de *Jump Table*

## 3.2 Control Flow Graph

Uma vez que a estrutura da Figura 3.2 tenha sido construída, o compilador está pronto para seguir para o próximo estágio do processo de compilação. Todos os objetos *ProcedureBody* terão as suas listas de instruções transformadas em CFGs (Control Flow Graphs). Um CFG é um grafo onde os nós representam computações e as arestas representam o fluxo de controle. Os nós de um CFG são chamados de blocos básicos. Um bloco básico é uma seqüência de instruções onde o fluxo de controle entra somente no início do bloco e

sai ao fim do mesmo, sem que haja a possibilidade de ocorrer algum desvio para alguma outra parte do programa.

Uma seqüência de instruções é transformada em blocos básicos utilizando-se o seguinte algoritmo:

***Algoritmo 3.1:*** Criação de Blocos Básicos

1. Determinar o conjunto de *líderes*, que correspondem à primeira instrução de cada bloco básico, utilizando as seguintes regras:
   (i) A primeira instrução é um líder.
   (ii) Qualquer instrução que é alvo de um desvio condicional ou incondicional é um líder.
   (iii) Qualquer instrução que venha depois de uma instrução de desvio condicional ou incondicional é um líder.
2. Para cada *líder*, o seu bloco básico consiste do líder e de todas as instruções que o seguem, indo até mas não incluindo o próximo líder.

■

A Figura 3.4 ilustra o funcionamento do algoritmo para a criação de blocos básicos em um fragmento de código qualquer. A porção esquerda da figura mostra o código antes da marcação dos líderes e a porção direita da figura mostra quais instruções foram escolhidas como líder. Pode-se observar que o fragmento de código apresentado irá conter 7 blocos básicos.

Uma vez que os líderes tenham sido encontrados, os blocos básicos já podem ser construídos. Ao construí-los, o Xingó irá inserir, quando necessário, instruções adicionais de modo que a primeira instrução de um bloco básico sempre seja um *label* e a última sempre seja uma instrução de salto (condicional ou incondicional). Depois que os blocos básicos estão construídos é necessário realizar o ajuste das arestas que irão indicar o fluxo de controle no programa, pois após a criação dos blocos básicos, o CFG se parece com uma lista duplamente encadeada. A Figura 3.5 mostra o aspecto inicial do CFG criado pelo Xingó. Neste CFG inicial, de forma a facilitar a visualização do mesmo, só são mostradas a primeira e a última instrução de cada bloco básico, e os mnemônicos das instruções de salto foram substituídos respectivamente por *goto* em saltos incondicionais e *if (...) goto* em saltos condicionais.

A Figura 3.5 mostra dois tipos de arestas. Um tipo é representado por um conjunto de arestas sólidas e o outro por um conjunto de arestas tracejadas. O conjunto de arestas sólidas corresponde aos sucessores do bloco básico enquanto que as arestas tracejadas

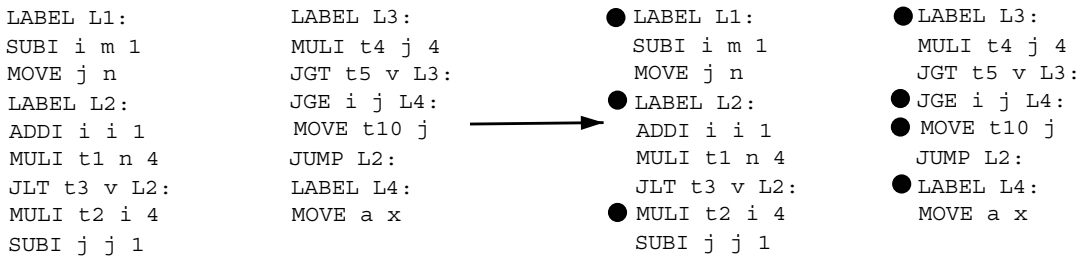


Figura 3.4: Marcação dos *líderes*

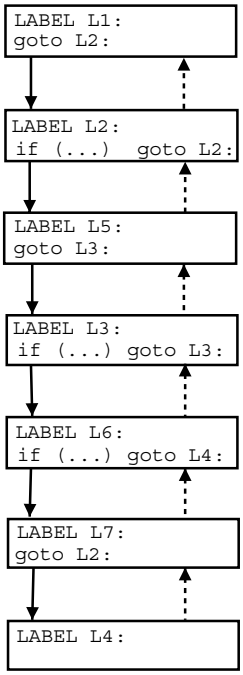


Figura 3.5: Formação inicial do CFG no Xingó

correspondem aos predecessores do bloco básico. Desta forma é possível caminhar no grafo de modo ascendente ou descendente.

Para acertar as arestas do CFG, o Xingó percorre todos os nós do grafo e verifica a instrução de salto do nó e realiza uma das seguintes ações:

- Se a instrução de salto for um desvio incondicional, então o Xingó remove do nó corrente $n$ a aresta que o está ligando a algum sucessor $s$, e remove de $s$ a aresta que torna $n$ um predecessor seu. Verifica-se então qual o bloco básico $b$ que é o alvo da instrução de salto em $n$ e adiciona-se uma aresta de $n$ para $b$ (indicando que $b$ é um sucessor de $n$) e outra aresta de $b$ para $n$ (indicando que $n$ é um predecessor de $b$). Tal operação é realizada pois podem haver inconsistências na formação inicial do CFG. A Figura 3.5 apresenta uma inconsistência, pois o nó cujo *label* é L7, realiza

um desvio incondicional para o *label* L2, mas tal bloco possui como sucessor o nó de *label* L4.

- Se a instrução de salto for um desvio condicional, então o Xingó verifica qual o bloco básico $b$ que é alvo da instrução de salto e adiciona uma aresta de $n$ (nó corrente) para $b$ (indicando que $b$ é um sucessor de $n$) e outra aresta de $b$ para $n$ (indicando que $n$ é um predecessor de $b$).

Os conjuntos de arestas que indicam sucessores e predecessores são distintos. De outra forma a criação do CFG seria inconsistente. A Figura 3.6 mostra o CFG gerado para o programa da Figura 3.4. Na figura só são apresentadas as arestas que indicam os sucessores de um nó.

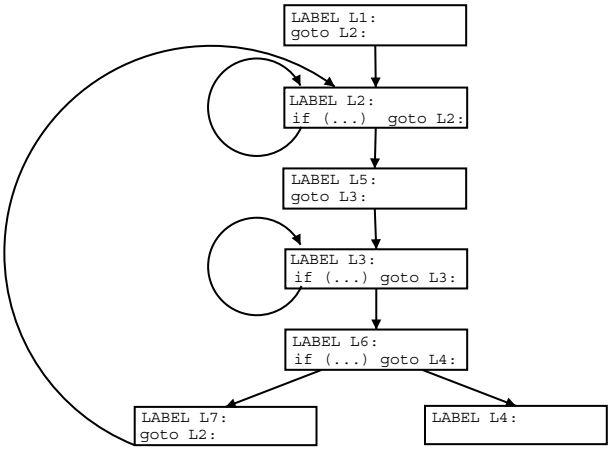


Figura 3.6: *Control Flow Graph* depois do ajuste das arestas

Durante a montagem do CFG, o Xingó acrescenta dois blocos básicos especiais ao mesmo, um bloco chamado *Header* e outro chamado *Tailer*. O bloco *Header* só possui duas instruções, uma que cria um *label* chamado $H$ e outra que realiza um salto incondicional para o primeiro bloco básico obtido após a determinação dos líderes. O bloco *Tailer* possui somente uma instrução, que cria um *label* chamado $T$. Os demais blocos terão o seu *label* renomeado de forma que o mesmo tenha um nome da forma B1, B2, ... B$n$. O último bloco básico obtido após a determinação dos líderes será o predecessor do bloco *Tailer*. A função dos blocos *Header* e *Tailer* é servir respectivamente de ponto de entrada e saída do procedimento/função. Desta forma o único bloco que não possui nenhum predecessor é *Header* e o bloco que não possui nenhum sucessor é *Tailer*. Tal abordagem torna mais simples a realização de buscas no grafo, pois o mesmo possui apenas um ponto de entrada e um de saída previamente conhecidos.

Depois que o CFG está construído e os blocos *Header* e *Tailer* foram adicionados, o Xingó realiza uma busca em profundidade no grafo para encontrar *unreachable code*, isto é, código que nunca será executado. Os nós que não forem visitados durante a busca, são considerados como *unreachable code* e podem ser eliminados.

No Xingó, um CFG é implementado pela classe *CFGraph*. Tal classe será a base para a realização das otimizações independentes de máquina, bem como para a geração da representação intermediária compilável. Os nós de um CFG são representados pela classe *CFGNode*, que é uma classe abstrata. Um bloco básico é implementado pela classe *CFGBBlock*, que herda a classe *CFGNode*. O Xingó utiliza tal esquema de forma a permitir a existência de tipos diferentes de nós no CFG, tal recurso pode ser interessante para se criar uma representação que permita o estudo de questões relativas a paralelização de código ou representar estruturas de alto nível como *while*, *if*, etc.

Depois que o CFG é construído, são computadas diversas informações relativas ao mesmo, as quais serão utilizadas em outros estágios do compilador, como por exemplo a construção da árvore de dominadores.

## 3.3 Geração de Código C a partir da XIR

A representação intermediária utilizada pelo compilador Xingó possui uma característica especial: pode ser *compilada*. Para tal, a XIR é convertida em um código fonte C e compilada como outro programa qualquer.

O processo de conversão da XIR em código C compilável envolve diversas etapas, como a recuperação das estruturas utilizadas pelo programa, a declaração de protótipos de funções externas ao programa e protótipos de funções de bibliotecas. Em um segundo momento são recuperadas todas as funções do programa. A Figura 3.7 mostra as etapas do processo de geração do código fonte C a partir da XIR.

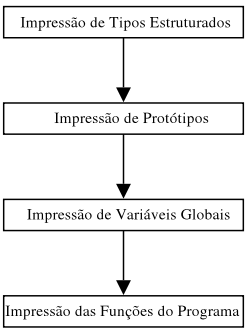


Figura 3.7: Etapas da geração do código C

A possibilidade de se poder gerar código C a partir da representação intermediária

do programa é uma excelente forma de acompanhar todas as mudanças que o mesmo sofre durante o processo de compilação. Outra finalidade é a possibilidade de se poder validar as otimizações disponíveis no compilador, pois é possível gerar código C a partir da representação intermediária cada vez que uma otimização é aplicada. Para se validar uma otimização, basta compilá-la e comparar a sua saída com a saída gerada pelo código fonte original. Se a otimização não estiver correta, o código C gerado pela mesma não preservará a semântica do código C gerado antes da otimização, facilitando a identificação do ponto onde ela está errada.

A geração do código C também é uma valiosa ferramenta de validação da própria representação intermediária gerada pelo Xingó. A Figura 3.8 mostra o funcionamento do processo de validação utilizado para se testar a XIR.

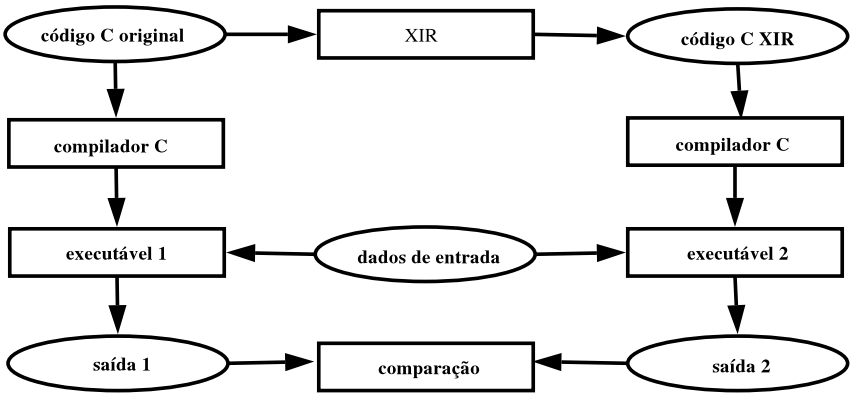


Figura 3.8: Processo de validação da XIR

A partir do programa original a XIR é construída e a partir da mesma ou após alguma etapa de otimização, um programa fonte C é gerado. O programa C original e o programa C obtido a partir da XIR são compilados utilizando-se um compilador C de referência (como por exemplo o GCC). A mesma entrada é então fornecida para os dois arquivos executáveis obtidos e as suas respectivas saídas geradas são comparadas. Se ambas as saídas estiverem iguais, então a XIR está correta, caso contrário deve haver algum erro na XIR ou em alguma otimização aplicada. Tal esquema é bem simples e foi amplamente utilizado durante o processo de desenvolvimento e teste do Xingó, bem como em cada uma das otimizações implementadas no mesmo.

O primeiro passo no processo de geração do código C a partir da XIR consiste em recuperar todos os tipos estruturados utilizados pelo programa, como **struct** e **union**. As tabelas de símbolos utilizadas pelo Xingó não armazenam apenas variáveis e nomes de funções, mas também tipos. Como dito anteriormente, cada tipo é representado por um objeto *Type*. Os tipos estruturados struct e union não fogem a essa regra. A classe *Type* é muito versátil, pois para tipos estruturados, ela armazena internamente uma lista

de objetos da classe *Field*. Cada objeto *Field* representa um campo da struct ou union que o objeto *Type* representa. Desta forma é possível recuperar todos os campos do tipo estruturado, bastando para isso percorrer a lista de objetos *Field*. Cada objeto *Field* armazena basicamente o nome e o tipo do campo, bem como outras informações que podem ser úteis em outros estágios do processo de compilação. Os objetos *Field* são armazenados em uma lista de forma a respeitar a ordem original em que foram criados.

Para coletar todos os tipos estruturados presentes em um programa, o Xingó percorre as tabelas de símbolos de todas as funções bem como a tabela de símbolos global. Em cada uma dessas tabelas ele irá verificar quais objetos *Type* são do tipo struct ou union e armazená-los em uma lista $L$.

A simples impressão dos tipos estruturados na ordem em que foram encontrados pode causar problemas de compilação, visto que algumas estruturas podem referenciar internamente outras estruturas. Para contornar esse problema o Xingó *ordena* a lista de estruturas de forma que todas as dependências sejam satisfeitas. Tal ordenação é basicamente realizada utilizando-se o conceito de uma árvore de dependência. Inicialmente as estruturas que não fazem referência a nenhuma outra estrutura são movidas para uma outra lista $S$. Na lista $L$ só restaram, se for o caso, estruturas que referenciam outras estruturas. A partir daí basta percorrer a lista $L$ e para cada elemento $e$ verificar se todas as estruturas internamente referenciadas já se encontram na lista $S$. Em caso afirmativo, $e$ é removido de $L$ e inserido ao fim de $S$, caso contrário o próximo elemento é verificado. Essa verificação é repetida até o momento em que a lista $L$ se torne vazia. O algoritmo a seguir ilustra o processo básico de ordenação dos tipos struct e union empregado pelo Xingó, de modo a realizar a impressão dos mesmos na ordem correta.

***Algoritmo 3.2:*** Ordenação de tipos estruturados

*Entrada:*

Lista $L$ contendo todos os tipos **union** e **struct** do programa.
Lista $S$ inicialmente vazia.
**enquanto** $L$ não estiver vazia **faça**
Type $t \leftarrow$ primeiro elemento de $L$
boolean mover $\leftarrow$ **true**
**para** cada campo $f$ de $t$ **faça**
**se** $f$ é do tipo **struct** ou **union** e $f$ não está em $S$ **então**
mover $\leftarrow$ **false**
**fim**
**fim**
**se** mover = **true então**
insira $t$ ao fim de $S$
**senão**

```
            insira t ao fim de L
        fim
    fim
```

■

Depois de obter a lista, já corretamente ordenada, das estruturas utilizadas pelo programa, o Xingó irá então realizar a impressão das mesmas. O processo de impressão se dá com a utilização de dois algoritmos mostrados a seguir. O algoritmo para impressão dos tipos estruturados basicamente verifica se o tipo é uma struct ou union, imprimindo o identificador apropriado e depois percorre os campos do mesmo. Para cada campo encontrado ele imprime o tipo do campo e depois o seu nome.

***Algoritmo 3.3:*** Impressão de tipos estruturados

*Entrada:*

Lista $S$ contendo os tipos **union** e **struct** na ordem correta para impressão.

```
enquanto S não estiver vazia faça
        Type t ← primeiro elemento de S
        se t é do tipo struct então
            imprima("struct{")
        senão
            imprima("union{")
        fim
        para cada campo f de t faça
            ImprimaTipo(f → Type)
            Imprima(f → Nome)
            Imprima(";");
        fim
        imprima("};")
fim
```

■

O algoritmo de impressão de tipos mostrado a seguir, ilustra o processo básico empregado pelo Xingó para imprimir o tipo de um dado objeto. Tal algoritmo é recursivo, uma vez que os tipos compostos no compilador Xingó são representados como uma lista ligada de tipos. O algoritmo percorre a lista imprimindo o identificador dos tipos na ordem adequada. O tipo *enum* definido na linguagem C é tratado internamente pelo Xingó como uma variável do tipo inteiro. Outros compiladores dão tratamento semelhante a este tipo de dado [16].

***Algoritmo 3.4:*** Impressão de Tipos

```
ImprimaTipo(t)
    escolha t
        caso _ARRAY
            ImprimaTipo(t → Type)
        caso _STRUCT
            imprima("struct")
            imprima(t → Nome)
        caso _UNION
            imprima("union")
            imprima(t → Nome)
        caso _POINTER
            ImprimaTipo(t → Type)
            imprima("*")
        caso _CONST
            imprima("const")
            ImprimaTipo(t → Type)
        caso _ENUM
            imprima("int")
        senão
            imprima(t → Nome)
    fim
```

■

Uma vez que todos os tipos estruturados tenham sido impressos, o próximo passo na geração do código C consiste em imprimir o protótipo de todas as funções que são **extern**, ou seja, são definidas em outros arquivos do programa ou são funções presentes em bibliotecas padrão da linguagem C, como a função *printf* por exemplo.

O processo de impressão de tais protótipos é bem simples, pois estas funções possuem associadas a si um objeto *Procedure*, mas não um objeto *ProcedureBody*. Desta forma, percorrendo-se a lista de objetos *Procedure* que está armazenada no objeto *File* e verificando-se quais deles não possuem um objeto *ProcedureBody*, descobrem-se todas as funções que são *extern*. Todos os possíveis tipos estruturados que os argumentos de tais funções possam utilizar já foram impressos, o que evita qualquer problema de compilação. O algoritmo a seguir ilustra o processo de impressão de tais protótipos.

***Algoritmo 3.5:*** Impressão de protótipos

*Entrada*: Lista $P$ com todos os objetos *Procedure* que não possuem um objeto *ProcedureBody* associado

**para** cada objeto $o$ em $P$ **faça**
    imprima("extern")
    ImprimaTipo($o \rightarrow Type$)
    imprima($o \rightarrow Nome$)
    imprima("(")
    **para** cada parâmetro $p$ na lista de parâmetros de $o$ **faça**
        ImprimaTipo($p \rightarrow Type$)
        imprima($o \rightarrow Nome$)
        **se** houver mais parâmetros **então**
            imprima(",")
        **fim**
    **fim**
    imprima(");")
**fim**

■

Depois de imprimir os protótipos das funções que não são implementadas pelo arquivo sendo compilado, o Xingó realiza a impressão dos protótipos das funções que são de fato implementadas pelo programa. Tal processo é realizado de forma a evitar que possam ocorrer erros de compilação, pois a simples impressão das funções pode trazer problemas caso existam situações como a apresentada na Figura 3.9, onde tem-se duas funções mutuamente recursivas.

```
void foo()
{
        ...
        bar();
        ...
}

void bar()
{
        ...
        foo();
        ...
}
```

Figura 3.9: Referência mútua entre funções

A impressão dos protótipos das funções implementadas pelo programa é realizada por

um algoritmo semelhante ao apresentado para a impressão das funções *extern*.

Com os protótipos das funções *extern* e das funções do programa já impressos, o Xingó parte agora para a impressão das variáveis globais do programa. Tal como os tipos struct e union, a impressão de tais elementos não pode ser realizada de forma desordenada, pois existem variáveis que podem estar inicializadas com valores que foram inicialmente atribuídos a outras variáveis. Desta forma, é preciso classificar as variáveis antes que elas sejam impressas.

As primeiras variáveis a serem impressas são as que pertencem ao segmento D_LIT, que como dito anteriormente, é o segmento de constantes. *Strings* que são definidas dentro do programa, como por exemplo em chamadas da função *printf*, são tratadas como variáveis pertencentes a este segmento. Depois que as constantes estão impressas, o Xingó imprime as variáveis globais que não estão inicializadas, ou seja, as que pertencem ao segmento D_BSS. Por último, o Xingó imprime as variáveis globais que estão inicializadas, ou seja, as que pertencem ao segmento D_DATA, pois elas podem referenciar alguma constante pertencente ao segmento D_LIT ou alguma variável pertencente ao segmento D_BSS.

O programa C da Figura 3.10 será utilizado para ilustrar o processo utilizado pelo Xingó para imprimir as variáveis. Este programa apresenta quatro variáveis globais, das quais duas estão inicializadas e também uma *string* dentro da função *main*.

```
int j;
int i = 23;
int k;
int l = 48;

void main()
{
        printf("This is a string.");
}
```

Figura 3.10: Código analisado pelo Xingó

A *string* "This is a string" presente na função *main* é considerada como uma variável pertencente ao segmento de constantes, e como tal será a primeira a ser impressa. As variáveis globais $j$ e $k$ não inicializadas pertencerão ao segmento D_BSS e serão as próximas a serem impressas. Por último, as variáveis globais inicializadas $i$ e $l$ irão pertencer ao segmento D_DATA e serão as últimas a serem impressas. A Figura 3.11 mostra o resultado da impressão das variáveis gerado pelo Xingó.

Depois que todos os tipos estruturados, protótipos de todas as funções e todas as variáveis estão impressas, o Xingó está pronto para partir para a impressão das funções

do programa propriamente ditas. A impressão das funções constitui-se basicamente de três etapas: impressão do protótipo da função, impressão das variáveis locais e impressão das instruções.

```
static char _C3[ ] = "This is a string.";
int k;
int j;
int l = 48;
int i = 23;
```

Figura 3.11: Seqüência de impressão das variáveis globais

A impressão do protótipo da função segue o mesmo processo já citado anteriormente e é bem simples. No tocante à impressão das variáveis locais, alguns cuidados devem ser tomados. As variáveis locais de uma função pertencem ao segmento D_CODE. Variáveis locais que precisam de inicialização também pertencem a este segmento, sendo que o código apropriado de inicialização é inserido junto com o programa.

Durante a impressão das variáveis locais, um tratamento diferenciado é dado às variáveis que são ponteiros. O Xingó imprime todas as variáveis que são do tipo ponteiro (tanto locais como globais) como *void** (com excessão, possivelmente, dos parâmetros das funções). Essa mudança de tipos deve ser realizada de forma a garantir que o programa seja executado corretamente após a geração do código C.

```
int array[ ] = {1,2,3,4,5,6,7,8,9,10};
void main()
{
        int* p;
        int i;

        p = &(array[0]);
        for(i=0;i<10;i++)
        {
                printf("%d ",*p);
                p++;
        }
}
```

Figura 3.12: Programa C contendo ponteiros

A Figura 3.12 mostra um programa C que é a motivação da conversão de tipo utilizada pelo Xingó. Neste programa existe um *array* de inteiros, que são impressos por meio de uma variável $p$ que é um ponteiro para inteiros. Na linha $p = \&(array[0])$, a variável $p$ recebe o endereço da primeira posição do vetor. A linha *printf("%d ",*p)* faz com que o conteúdo da posição de memória apontada por $p$ seja impresso. A linha $p{+}{+}$ incrementa o valor de $p$ de modo que ele aponte para o próximo valor inteiro do vetor, o laço *for* faz com que todos os elementos do vetor $array$ sejam impressos.

Atenção especial deve ser dada à linha $p{+}{+}$. Esta linha faz com que o valor de $p$ seja incrementado para que ele aponte para o *próximo inteiro*. A semântica do incremento é feita de acordo com o tipo de dado que $p$ pode apontar e não com o valor que ele armazena internamente. Isto faz com que o valor interno de $p$ seja acrescido não por uma unidade, mas por um valor que pode ser bem maior do que 1. No caso específico da Figura 3.12, onde $p$ é um ponteiro para inteiros, o valor interno de $p$ será incrementado em um valor igual ao número de bytes necessários para representar uma variável do tipo int. Supondo que sejam necessários 4 bytes para representar um inteiro e que $p$ esteja apontando para o endereço de memória $E$, após o incremento realizado por $p{+}{+}$, $p$ irá conter o valor $E + 4$. A semântica da linha $p{+}{+}$ torna este fato implícito, mas o mesmo se torna explícito após a construção da XIR, ou seja, a operação de incremento $p{+}{+}$ se torna $p = p + 4$. Se a variável $p$ for declarada como um *int**, ao se gerar o código C a partir da XIR, a instrução $p = p + 4$ faria com que o valor armazenado em $p$ fosse incrementado em 16 unidades, o que certamente iria levar a uma execução incorreta do programa.

Ao se considerar todas as variáveis do tipo ponteiro como sendo *void**, tal problema é eliminado, uma vez que a operação $p = p + 4$, irá incrementar o conteúdo de $p$ em 4 unidades e não em 16, pois o tipo *void* não corresponde a nenhum tipo de dado em particular.

Embora considerar todos os ponteiros como sendo *void** resolva o problema relativo ao incremento dos mesmos, isto insere um novo problema. A linha *printf("%d ",*p)* realiza a impressão do conteúdo do endereço apontado por $p$. Embora não esteja explícito na instrução, $*p$ faz com que sejam lidos bytes da memória, de modo a recuperar o valor de $*p$. Se o tipo de $p$ é *int**, então internamente serão lidos 4 bytes (novamente supondo que um inteiro ocupe 4 bytes) a partir do endereço apontado por $p$, mas se $p$ é do tipo *void**, então será lido apenas 1 byte, o que fatalmente resultará na impressão de um valor errado.

A solução deste novo problema introduzido pela mudança de tipos é realizada inserindo-se operações explícitas de *cast* no momento de ler ou escrever variáveis do tipo *void**. O Xingó realiza internamente tal operação, pois o compilador sabe qual o tipo real da variável. Desta forma o programa da Figura 3.12 seria transformado pelo Xingó em algo semelhante ao que é apresentado na Figura 3.13.

```
int array[ ] = {1,2,3,4,5,6,7,8,9,10};
void main()
{
        void* p;
        int i;
        int t1;

        p = &(array[0]);
        for(i=0;i<10;i++)
        {
                t1 = *((int*)p);
                printf("%d ",t1);
                p = p + 4;
        }
}
```

Figura 3.13: Transformação de tipos realizada pelo Xingó

A variável $p$ é do tipo *void** e uma variável temporária $t1$ é criada. O valor apontado por $p$ é lido utilizando-se uma operação de *cast*, fazendo assim com que a quantidade apropriada de bytes seja lida.

As demais variáveis do programa, isto é, as que não são apontadores, são impressas sem qualquer alteração em seu tipo. Uma vez que todas as variáveis tenham sido impressas o Xingó parte para a impressão das instruções da XIR.

As instruções da XIR estão armazenadas em blocos básicos e o conjunto de blocos básicos formam um CFG. Desta forma, para imprimir as instruções da XIR que compõem o programa, deve-se imprimir as instruções que cada bloco básico contém. Os blocos básicos do CFG devem então ser impressos seguindo-se uma ordem que corresponda à seqüência correta de execução das instruções. Isto é, não se pode escolher aleatoriamente qual bloco básico vai ser impresso. Para determinar a seqüência correta de blocos básicos a serem impressos, o Xingó utiliza o conceito de *trace*.

Um *trace* é uma seqüência de instruções que podem ser executadas consecutivamente, durante a execução de um programa. Um *trace* pode incluir desvios condicionais [6]. Um programa pode ter vários *traces* diferentes. O algoritmo a seguir, apresentado por [6], ordena blocos básicos em *traces*:

***Algoritmo 3.6:*** Geração de *Traces*
Insira todos os blocos básicos do programa em uma lista $Q$
**enquanto** $Q$ não estiver vazia **faça**

Crie um novo *trace* $T$, vazio
Remova o primeiro elemento $b$ de $Q$
**enquanto** $b$ não estiver marcado **faça**
Marque $b$, insira $b$ ao fim do *trace* corrente $T$
Examine os sucessores de $b$ (os blocos para os quais $b$ salta)
**se** algum dos sucessores $c$ de $b$ não estiver marcado **então**
$b \leftarrow c$
**fim**
**fim**
Fim do *trace* corrente
**fim**

■

O algoritmo inicia em algum bloco básico (no Xingó, o bloco inicial é o *Header*) e segue uma cadeia de desvios, marcando cada bloco e inserindo-o no final do *trace* corrente. Em algum momento ele encontrará um bloco cujos sucessores estão todos marcados, desta forma o *trace* corrente chega ao seu fim e um novo *trace* é iniciado a partir de algum bloco básico não marcado.

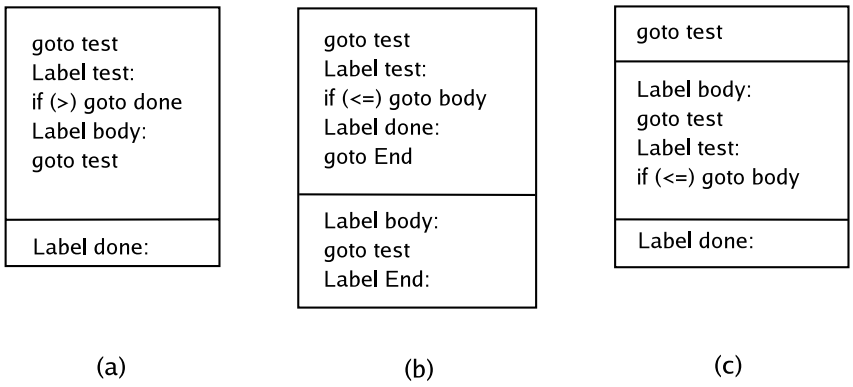


Figura 3.14: Diferentes *traces* para um mesmo programa

A Figura 3.14 mostra exemplos de *traces* diferentes para um mesmo programa escrito em pseudo-código. Desvios condicionais podem ser alterados de forma a garantir que a ordem de execução do programa seja correta. Em ($a$) observa-se que caso o *if* seja verdadeiro, o fluxo irá saltar para o *label done*. Em ($b$), caso o *if* seja verdadeiro, o fluxo irá saltar para o *label body*.

O Xingó realiza as devidas alterações nas instruções de salto condicional de tal forma que a seqüência de blocos básicos determinados pelo algoritmo de geração de *traces* seja executada corretamente. O compilador também insere novos *labels* e instruções de salto

caso seja necessário, como na Figura 3.14 (*b*), onde um novo *label* chamado *End* foi inserido. Tal fato pode ser necessário caso o nó de saída do programa não tenha ficado por último. Neste caso o *label End* atua como um novo nó de saída do programa. Esse nó só é criado no arquivo C gerado pelo Xingó, não sendo necessário inserí-lo no CFG.

Tendo os *traces* do CFG, o Xingó percorre os blocos básicos na ordem adequada e converte cada uma das instruções da XIR na estrutura apropriada da linguagem C. A conversão das instruções é praticamente um mapeamento um-para-um, visto que os *opcodes* que compõem a XIR representam operações bem simples, onde quase todas estão presentes na linguagem C.

O programa apresentado na Figura 3.15 será utilizado para ilustrar o processo de conversão. Inicialmente ele é analisado pelo LCC e então convertido para a XIR. As instruções resultantes da transformação do programa da Figura 3.15 são apresentadas na Figura 3.16.

```
int fat(int n)
{
        if(n==0)
        {
                return 1;
        }
        else
        {
                return n*fat(n-1);
        }
}
```

Figura 3.15: Função em C para calcular o fatorial

Depois de obter esta lista de instruções, o Xingó constrói o CFG para a mesma e depois determina os *traces* do programa. O Xingó irá percorrer os blocos básicos e converter cada uma das instruções em uma declaração equivalente na linguagem C. A Tabela 3.2 mostra o mapeamento utilizado pelo Xingó para converter cada uma das instruções da XIR.

A Figura 3.17 mostra o programa C final gerado pelo Xingó, que é equivalente ao programa apresentado na Figura 3.15. Pode-se observar que o programa gerado contém os seus *labels* renomeados em relação aos que são apresentados na Tabela 3.2, tal processo é realizado após a construção do CFG.

O programa gerado pelo Xingó apresentado na Figura 3.17, ainda não foi otimizado, por isso ele emprega diversos passos para realizar simples computações. Os diversos passos acabam sendo introduzidos como resultado da conversão da representação em forma de

```
MOVE .t3 n          CALL [.t10] fat
MOVEI .t4 0         ARG .t9
JNE .L2 .t3 .t4     MOVE .l1 .t10
MOVEI .t5 1         MOVE .t13 n
RET .t5             MOVE .t15 .l1
JUMP .L1            MUL .t16 .t13 .t15
LABEL .L2           MOVE .l2 .t16
MOVE .t7 n          MOVE .t19 .l2
MOVEI .t8 1         RET .t19
SUB .t9 .t7 .t8     LABEL .L1
```

Figura 3.16: Instruções do Xingó geradas para a função da Figura 3.15

| | | | | | |
|---|---|---|---|---|---|
| MOVE .t3 n | ⇒ | t3 = n; | CALL [.t10] fat | ⇒ | t10 = fat(t9); |
| MOVEI .t4 0 | ⇒ | t4 = 0; | ARG .t9 | ⇒ | //argumento da função |
| JNE .L2 .t3 .t4 | ⇒ | if ( t3 != t4) goto L2; | MOVE .l1 .t10 | ⇒ | l1 = t10; |
| MOVEI .t5 1 | ⇒ | t5 = 1; | MOVE .t13 n | ⇒ | t13 = n; |
| RET .t5 | ⇒ | return t5; | MOVE .t15 .l1 | ⇒ | t15 = l1; |
| JUMP .L1 | ⇒ | goto L1; | MUL .t16 .t13 .t15 | ⇒ | t16 = t13 * t15; |
| LABEL .L2 | ⇒ | L2: | MOVE .l2 .t16 | ⇒ | l2 = t16; |
| MOVE .t7 n | ⇒ | t7 = n; | MOVE .t19 .l2 | ⇒ | t19 = l2; |
| MOVEI .t8 1 | ⇒ | t8 = 1; | RET .t19 | ⇒ | return t19; |
| SUB .t9 .t7 .t8 | ⇒ | t9 = t7 - t8; | LABEL .L1 | ⇒ | L1: |

Tabela 3.2: Conversão das instruções do Xingó para código C

DAGs do LCC para as quádruplas da XIR. A aplicação de algumas etapas de otimização diminui o número de etapas em cada computação.

Durante a conversão para código C, as únicas instruções que requerem um pouco mais de análise são LOAD, STORE e STOREI, pois as variáveis ponteiro são declaradas como tendo tipo *void∗*. Quando o Xingó encontra uma destas três instruções na XIR, ele insere as operações de *cast* necessárias de forma a garantir que a quantidade correta de bytes seja lida da (ou escrita na) memória, tal como mostrado na Figura 3.13.

As únicas instruções da XIR que não podem ser mapeadas diretamente em operações da linguagem C, são possivelmente instruções de salto. Uma instrução de salto só não terá um mapeamento um-para-um se estiver associada a uma *Jump Table*. Neste caso o Xingó constrói um *switch* para transformar a *Jump Table* em código C compilável.

*Jump Tables* surgem como uma forma de se tentar otimizar estruturas *switch* onde

```
int fat(int n)
{
        int _t8;
        int _t7;
        int _t10;
        int _t9;
        int _t16;
        int _t15;
        int _t13;
        int _l2;
        int _l1;
        int _t5;
        int _t4;
        int _t3;
        int _t19;
H:
        goto B1;
B1:
        _t3 = n;
        _t4 = 0;
        if( _t3 != _t4 ) goto B3;
B2:
        _t5 = 1;
        return _t5;
        goto B4;
B3:
        _t7 = n;
        _t8 = 1;
        _t9 = _t7 - _t8;
        _t10 = fat(_t9);
        _l1 = _t10;
        _t13 = n;
        _t15 = _l1;
        _t16 = _t13 * _t15;
        _l2 = _t16;
        _t19 = _l2;
        return _t19;
        goto B4;
B4:
        goto T;
T:
        ;
}
```

Figura 3.17: Programa C gerado pelo Xingó a partir das instruções da Figura 3.16

as opções de escolha seguem alguma ordem lógica (como uma progressão aritmética). O programa da Figura 3.18 mostra um programa C onde as opções de escolha seguem uma numeração crescente que sempre é incrementada de um. Este tipo de programa, fatalmente irá gerar uma *Jump Table*.

O Xingó armazena as informações sobre as *Jump Tables* utilizando objetos da classe *Note*. Um objeto *Note* contendo informação sobre alguma *Jump Table* vai estar associado a uma instrução de salto na XIR. A Figura 3.19 mostra o código C gerado pelo Xingo para o programa da Figura 3.18.

Com as informações da *Jump Table* o Xingó constrói uma estrutura *switch* presente na linguagem C. Esta é a única estrutura de controle de nível mais alto, junto com o *if*, que o Xingó recupera. As demais estruturas de controle (*for*, *while*, *do*) são todas transformadas em *if's* e *goto's*.

```
void main()
{
        int i;

        switch(i)
        {
                case 1:printf("%d\n",i*2);break;
                case 2:printf("%d\n",i*3);break;
                case 3:printf("%d\n",i*4);break;
                case 4:printf("%d\n",i*5);break;
        }
}
```

Figura 3.18: Programa C contendo uma estrutura *switch*

## 3.4 Geração da Representação XAR

Depois que a XIR está construída, é possível realizar a geração de código para a máquina alvo. A geração do código *assembly* pode se dar antes ou depois da fase de otimização. O código *assembly* inicialmente gerado pelo Xingó recebe o nome de XAR (Xingó *Assembly Representation*).

A geração de código do Xingó é realizada a partir das listas de instruções obtidas da conversão da representação em forma de DAGs do LCC para instruções da XIR. Desta forma, se o CFG foi construído, o Xingó o desmonta e reconstrói as listas de instruções, as quais são entregues ao gerador de código. O Xingó utiliza o gerador de gerador de código Olive [31] para implementar parte do seu *back-end*.

O Xingó converte as listas de instruções da XIR em padrões de árvores que são reconhecidos pelo Olive e através de uma descrição da máquina alvo ele gera um conjunto de instruções *assembly* que irão compor a nova representação chamada de XAR.

De modo a ser relativamente independente da máquina alvo, o Xingó implementa as instruções *assembly* por meio de classes genéricas. Uma instrução *assembly* é implementada pela classe *AsmOperation*. Tal classe armazena o *opcode* da operação, os seus operandos e possivelmente outras informações caso seja necessário, como por exemplo os registradores que a operação utiliza.

Os operandos da instrução *assembly* são todos derivados da classe *AsmOperand*, a qual é uma classe abstrata. Todos os operandos que a instrução for utilizar de fato devem ser implementados por classes que herdam a classe *AsmOperand*.

O Xingó define alguns operandos base como *AsmRegister*, *AsmVariable*, *AsmLabel*,

```
extern int printf(const char*,...);
void main();
static char _C17[ ] = "%d\n";
static int _C23[ ] = {0, 1, 2, 3};

void main()
{
        int _t2;
        int _t41;
        void* _t12;
        int _t22;
        int _t21;
        void* _t16;
        void* _t15;
        void* _t14;
        const void* _t28;
        int _t26;
        const void* _t43;
        int i;
        int _t46;
        int _t19;
        const void* _t17;
        const void* _t32;
        int _t31;
        int _t30;
        int _t37;
        int _t36;
        int _t35;
        int _t34;
H:
        goto B1;
B1:
        if( i < 1 ) goto B5;
B2:
        if( i > 4 ) goto B5;
B3:
        _t2 = i << 2;
        _t12 = &_C23;
        _t14 = ((void*) _t12 ) + -4;
        _t15 = ((void*) _t14 ) + _t2;
        _t16 = *((void**)_t15);
        switch ((int)_t16)
        {
                case 0: goto B9;break;
                case 1: goto B6;break;
                case 2: goto B8;break;
                case 3: goto B4;break;
        };
B4:
        _t41 = i * 5;
        _t43 = &_C17;
        _t46 = printf(_t43,_t41);
        goto B5;
B5:
        goto B7;
B6:
        _t26 = i * 3;
        _t28 = &_C17;
        _t31 = printf(_t28,_t26);
        goto B7;
B7:
        goto B10;
B8:
        _t32 = &_C17;
        _t37 = printf(_t32,_t2);
        goto B7;
B9:
        _t17 = &_C17;
        _t21 = i << 1;
        _t22 = printf(_t17,_t21);
        goto B7;
B10:
        goto T;
T:
        ;
}
```

Figura 3.19: Programa gerado pelo Xingó na ocorrência de uma *Jump Table*

*AsmImmediate*, *AsmAddrRegister*, *AsmBaseRegister* e *AsmExprRegister*. De acordo com as necessidades da arquitetura alvo, este conjunto pode ser ampliado de forma a refletir a nova situação.

O Xingó contém um arquivo chamado *asmconfig.h*, que é utilizado para armazenar informações que são específicas da arquitetura alvo, tal como o *opcode* das instruções, a quantidade de registradores disponíveis, o tipo dos registradores (int ou float) e se os mesmos podem ser utilizados para alocação.

Uma vez que as instruções *assembly* tenham sido geradas, o Xingó novamente irá construir um CFG. Em um primeiro momento, a alocação de registradores ainda não foi efetuada, desta forma o Xingó utiliza registradores virtuais na geração do código *assembly*. Para determinar quais objetos são candidatos a residir em registradores, o Xingó utiliza o conceito de *web* [26].

Uma *web* é uma máxima união de *Du-Chains*[1] (cadeias definição-uso) tal que, para cada definição $d = d_0$,..., $u_0$, $d_n$, $u_n = u$, e para cada $i$, $u_i$ pertence a cadeia definição-uso de $d_i$ e $d_{i+1}$. Cada *web* é associada a um registrador virtual. Segundo [26] a vantagem de se utilizar *webs* ao invés de variáveis como candidatas para permanecerem em registradores, reside no fato de que uma mesma variável pode ser utilizada em uma função para diversos propósitos totalmente disjuntos. Embora o programador possa pensar que os vários usos da variável estejam ligados, eles de fato são separáveis para a alocação de registradores.

Através da utilização de *webs*, o código *assembly* gerado pelo Xingó pode utilizar um número ilimitado de registradores virtuais. Além de "separar" os diversos usos que uma variável pode ter em um programa, as *webs* tendem a ter uma *live range* limitada, isto é, não ficam vivas ao longo de diversas instruções. Tal fato diminui a quantidade de interferências que um dado registrador virtual possa ter em relação a outros registradores virtuais, ocasionando uma alocação mais eficiente de registradores físicos, na medida em que diminui a possibilidade de ocorrência de *spills*.

Para poder alocar registradores, o Xingó precisa de informação sobre a longevidade dos registradores virtuais, isto é, informações de *Liveness*. A maneira como uma instrução *assembly* usa ou define registradores é específica da arquitetura alvo, por isso a *Liveness Analysis* deve ser implementada de acordo com essa arquitetura. De forma a fornecer os meios de realizar *Data Flow Analysis* no código *assembly*, o Xingó possui um conjunto de classes base que fornecem o suporte a *Post-Pass Data Flow Analysis*, que corresponde a realização de *Data Flow Analysis* no código *assembly* da arquitetura alvo. Além de fornecer informações sobre o fluxo da informação, a *Post-Pass Data Flow Analysis* pode servir como base para a realização de otimizações dependentes de máquina. As classes que compõem a *Post-Pass Data Flow Analysis* no Xingó, são implementadas como classes abstratas, de modo que o programador deve fornecer a implementação para

[1]Este conceito será apresentado adiante

alguns métodos, que dependem de características da arquitetura alvo, como por exemplo quais instruções são definições, quais registradores uma instrução está utilizando, etc.

Para a alocação de registradores, o Xingó utiliza informações provenientes das classes *PostPassDFAInfo* e *PostPassLivenessClass*. O programador deve criar novas classes que irão herdar estas duas classes e implementar os métodos abstratos que tratam de aspectos específicos da arquitetura alvo.

## 3.5 Alocação de Registradores

O alocador de registradores do Xingó utiliza um algoritmo de coloração de grafos. John Cocke foi o primeiro a reconhecer, em 1971, que a alocação de registradores poderia ser mapeada em problemas de coloração de grafos. Uma primeira implementação utilizando esta abordagem foi feita por Chaitin [11] em 1981, mas o algoritmo mais bem sucedido foi o desenvolvido por Briggs [8].

A alocação de registradores via coloração de grafos é realizada basicamente em 3 etapas:

1. Construir um grafo, chamado de grafo de interferência, onde os nós do mesmo representam objetos alocáveis e registradores físicos da arquitetura alvo; e as arestas representam as interferências, onde dois objetos que são alocáveis interferem entre si se eles estão simultaneamente vivos, e um objeto alocável interfere com um registrador físico se o objeto não puder ser atribuído a este registrador.

2. Colorir o grafo de interferência com $R$ cores, onde $R$ corresponde ao número de registradores físicos disponíveis para alocação, de tal forma que quaisquer dois nós adjacentes do grafo tenham cores diferentes.

3. Atribuir a cada objeto alocável o registrador que corresponda a cor recebida pelo objeto.

O Xingó utiliza a informação computada pela *Post-Pass Liveness Analysis* para determinar as interferências entre os objetos alocáveis (*webs*) e os registradores físicos da arquitetura alvo. O algoritmo de coloração precisa constantemente verificar o grafo de interferência e realizar uma das duas consultas a seguir:

1. Descobrir todos os nós adjacentes a um dado nó.

2. Descobrir se dois nós quaisquer são adjacentes.

A utilização de uma lista de adjacência pode responder a questão 1 de forma rápida, mas não a questão 2. Uma matriz de adjacência pode responder a questão 2 de forma rápida, mas não a questão 1. Desta forma é mais eficiente utilizar as duas estruturas para se representar o grafo de interferência, que é justamente o que o Xingó faz.

O processo de coloração do grafo de interferência é um problema NP-completo [6], por isso sua resolução é feita por meio de uma aproximação do resultado. A heurística da resolução é bem simples e se baseia na remoção de nós do grafo.

Supondo que um grafo de interferência $G$ contenha um nó $m$ com um grau menor do que $R$, onde $R$ é o número de registradores disponíveis para alocação, constrói-se o grafo $G'$ que é $G - \{m\}$ o qual é obtido removendo-se o nó $m$ de $G$. Se $G'$ puder ser colorido, então $G$ também o poderá, pois quando o nó $m$ é inserido em $G'$, seus vizinhos estarão coloridos com no máximo $R - 1$ cores, e desta forma sempre será possível encontrar uma cor livre para $m$. Desta forma o processo de coloração do grafo consiste em repetidamente remover os nós que contenham um grau menor que o número de cores disponíveis. Cada vez que um nó é removido, o grau de seus vizinhos diminui podendo gerar novos candidatos a remoção do grafo.

Depois que todos os nós do grafo tiverem sido removidos, eles devem ser reinseridos no grafo na ordem inversa a que saíram, isto é, o último nó removido será o primeiro a ser reinserido e o primeiro nó removido será o último a ser reinserido. Cada vez que um nó é inserido no grafo, ele deve ser colorido com uma cor que seja distinta das cores de seus vizinhos. Quando todo o grafo estiver remontado e colorido, basta modificar o código *assembly* trocando cada registrador virtual pelo registrador físico correspondente a cor recebida pelo nó no grafo de interferência.

Podem existir situações em que durante o processo de remoção dos nós do grafo $G$, não exista um nó $m$ com um grau menor do que $R$, neste caso algum nó do grafo não poderá ser mapeado em um registrador físico, devendo assim permanecer em memória. Quando um nó do grafo de interferência deve ser mapeado em memória, dizemos que o mesmo sofreu *spill*.

Se for necessário realizar *spill* em algum nó do grafo de interferência, deve-se escolher um nó para que o mesmo seja removido. Para se escolher qual nó do grafo será removido, utiliza-se um valor chamado *spill cost*, que corresponde ao custo de se representar um dado nó do grafo em memória. Para computar o *spill cost* para cada registrador virtual o Xingó utiliza a seguinte fórmula:

$$def \cdot \sum_{def \in web} 10^{depth(def)} + use \cdot \sum_{use \in web} 10^{depth(use)} \tag{3.1}$$

Para cada *web* do programa a fórmula realiza o somatório do total de definições e usos da mesma. Se um dado uso ou definição ocorrer dentro de um *loop* então o seu peso

é multiplicado por 10 elevado a profundidade do *loop* (a profundidade da definição ou uso da *web* é dada pela função *depth*), isto é, se uma *web* não ocorrer dentro de nenhum *loop*, então o seu peso será igual a 1. Se ela estiver dentro de um *loop*, terá peso 10; se ocorrer dentro de um *loop* que está dentro de outro *loop* então terá peso 100, e assim sucessivamente.

A Fórmula 3.1 faz com que *webs* que estejam dentro de *loops* tenham um *spill cost* maior, sendo desta forma mais dispendioso selecioná-las para sofrer *spill*. Ela é apresentada de maneira semelhante em [26] e [6], onde os mesmos a apresentam como sendo a mais comum para se computar o *spill cost* dos nós do grafo de interferência.

De posse do *spill cost* de cada nó candidato a *spill*, seleciona-se o nó a sofrer *spill* calculando-se o valor *spill priority* para cada nó. O escolhido será o nó que contiver o menor valor *spill priority*, o qual é calculado dividindo-se o *spill cost* do nó pelo seu grau.

Uma vez que o nó selecionado para *spill* é removido do grafo, o processo de remoção dos demais nós continua. Deve-se observar que o nó selecionado para *spill* durante o processo de remoção dos nós do grafo poderá eventualmente ser colorido durante o processo de reinserção dos nós, no caso de, mesmo tendo um grau maior que $R$, os seus vizinhos sejam coloridos com no máximo $R - 1$ cores. Se durante o processo de reinserção do nós, o nó selecionado para *spill* não puder ser colorido, significa que o número de registradores disponíveis para alocação não é o suficiente para colorir o grafo. Desta forma o programa deve ser modificado de forma a tentar tornar o grafo colorível. A modificação do programa consiste em inserir operações de escrita na memória (STORE) depois de cada definição do registrador virtual no programa e operações de leitura da memória (LOAD) antes de cada uso do mesmo. Tal processo é chamado de *Gen Spill Code*.

Depois que o *spill code* é inserido no programa, o processo de alocação de registradores é reiniciado, recomputando-se a *Post-Pass Liveness Analysis* [30] (o intuito do *spill code* é diminuir a longevidade do registrador virtual selecionado para *spill*), reconstruindo-se o grafo de interferência e repetindo-se o processo de coloração do grafo através da remoção dos nós do mesmo. Se durante a nova tentativa de coloração do grafo for necessário realizar *spill*, então novamente *spill code* é gerado e todo o processo é repetido, até que em um dado momento, não seja mais necessário gerar *spill code* e o grafo de interferência possa ser colorido.

Uma otimização que o Xingó tenta realizar durante o processo de alocação de registradores é a realização de *coalescing* [6]. O *coalescing* é uma operação que tenta eliminar instruções de MOVE com o auxílio do grafo de interferência. A operação de *coalescing* consiste em verificar para cada instrução de MOVE do programa se os registradores virtuais origem e destino contêm uma aresta que os ligue no grafo de interferência. Se eles contiverem uma aresta implica que eles interferem entre si. Se não existir uma aresta que os ligue, então eles podem ser unidos e transformados em um único registrador no grafo

de interferência, fazendo com que o mesmo registrador físico seja atribuído a ambos, o que irá fazer com que a instrução de MOVE tenha o mesmo registrador físico como origem e destino e desta forma possa ser eliminada do programa. Em princípio quaisquer dois nós do grafo de interferência que não tenham uma aresta que os ligue podem sofrer *coalescing*, mas é preciso observar o grau do nó resultante, pois se ele for maior que o número de registradores físicos, o *coalescing* pode transformar um grafo colorível em um grafo não colorível.

De forma a não transformar um grafo colorível em um grafo não colorível, ao realizar *coalescing* o Xingó utiliza a seguinte regra, proposta por Briggs [8]:

**Briggs:** dois nós $a$ e $b$ podem sofrer *coalescing* se o nó $ab$ resultante contiver menos que $R$ vizinhos de grau significante (isto é, possuir um total de arestas $\geq R$). Com isso o *coalescing* não irá transformar um grafo $R$ colorível em um grafo que não seja $R$ colorível, pois depois que todos os nós que contiverem um grau menor que $R$ tiverem sido removidos do grafo, o nó resultante será adjacente somente a nós que tenham grau significativo, mas como o total deles é inferior a $R$, o nó resultante será selecionado para ser removido do grafo. Assim se o grafo original for colorível, o grafo resultante desta estratégia conservativa de *coalescing* também o será.

Esta estratégia é conservativa, pois não altera as propriedades de colorabilidade do grafo de interferência. A Figura 3.20 mostra o pseudo-código da função principal de alocação de registradores que o Xingó utiliza. A função *PruneGraph* presente na figura é responsável por remover do grafo de interferência os nós com grau não significativo, marcando quando necessário, os nós candidatos a *spill*. A função *AssignRegs* é a responsável por reinserir os nós no grafo e os colorir. Caso ela tenha conseguido colorir todo o grafo sem realizar *spill* o valor de retorno é *true*, caso contrário o valor é *false*, implicando que é necessário inserir *spill code* no programa.

A alocação de registradores do Xingó é implementada pela classe *RegAlloc*, que é uma classe abstrata. O programador que deseje portar o Xingó para alguma arquitetura e utilizar o algoritmo de coloração de grafos do Xingó, deve criar novas classes para a alocação de registradores as quais devem herdar de *RegAlloc*. A classe *RegAlloc* possui alguns métodos abstratos, que são dependentes do formato do código *assembly* da arquitetura alvo, como por exemplo o método responsável pela geração de *spill code* e o responsável por transformar os registradores virtuais em registradores físicos.

```
sucesso ← false
enquanto sucesso ! = true faça
        Compute Liveness Analysis
        Construa a Matriz de Adjacência
        Realize Coalescing
        Construa a Lista de Adjacência
        Compute o spill cost para cada registrador virtual
        PruneGraph
        sucesso ← AssignRegs
        se sucesso = true então
                Modifique o programa convertendo registradores virtuais em físicos
        else
                GenSpillCode
        fim
fim
```

Figura 3.20: Pseudo-código do algoritmo de alocação de registradores do Xingó

# Capítulo 4

# Otimização de Código no Xingó

Um compilador otimizante realiza transformações no programa fonte com o objetivo de melhorar a sua eficiência sem mudar o que o mesmo computa. Para realizar tais transformações o compilador faz uso de otimizações. Existem diversas otimizações como *dead code elimination*, *copy propagation* e *common subexpressions elimination*.

Segundo Appel [6] não existe uma lista completa com todas as otimizações, pois de fato tal lista nunca existirá. A teoria da computabilidade mostra que sempre será possível criar novas otimizações, pois é possível mapear o problema de otimização ideal em compiladores para o problema da parada [6], ou seja, dado qualquer compilador otimizante, sempre existe um melhor.

Este capítulo tem por objetivo descrever as otimizações que estão implementadas na presente versão do Xingó bem como seu funcionamento básico. Tais otimizações estão constantemente sendo aperfeiçoadas e comparadas com o compilador GCC através do *benchmark* NullStone [2].

## 4.1 Data Flow Analysis

Para que um compilador possa otimizar um programa e realizar um bom trabalho na geração de código, ele precisa coletar informações sobre o programa que ele está compilando e distribuir essas informações pelos blocos básicos. Tal coleta de informações é chamada de *Data Flow Analysis* [5].

O propósito da *Data Flow Analysis* (DFA) é fornecer informação sobre como um procedimento (ou um grande segmento de um programa) manipula os seus dados. As possíveis análises de fluxo de dados vão desde a execução abstrata de um procedimento, até outras mais simples como *reaching definitions*.

Em todos os casos, deve-se ter certeza de que a DFA fornece informação que não deturpe o que o programa sob análise computa, pois ela é quem irá dizer se é ou não

seguro realizar uma determinada transformação no programa. A DFA deve ser projetada de forma que a sua solução, se não for uma exata representação de como o programa manipula os seus dados, que ao menos seja uma aproximação conservativa. A análise é conservativa se a sua solução, caso não seja a exata, não leve a realização de transformações indevidas no programa. Entretanto, para se obter o máximo benefício possível de uma otimização é necessário utilizar análises de fluxo de dados que sejam ao mesmo tempo conservativas e o mais agressivas possível. Desta forma, existe uma linha tênue entre ser o mais agressivo possível e ser conservativo, de forma a conseguir obter o melhor benefício das transformações em um programa sem mudar o que o mesmo computa.

A DFA é realizada estabelecendo-se e resolvendo sistemas de equações que relacionam a informação em vários pontos de um programa. Uma equação típica tem a forma

$$out[S] = gen[S] \cup (in[S] - kill[S]) \tag{4.1}$$

e pode ser lida como, "a informação ao fim de uma instrução é gerada por esta instrução, ou entra no início da mesma e não é morta quando o fluxo de informação a atravessa" [5]. Tais equações são chamadas de *data-flow equations* ou equações de fluxo de dados.

Os detalhes relativos a como as *data-flow equations* são estabelecidas e resolvidas dependem de três fatores:

1. A noção de *gerar* e *matar* depende da informação desejada, isto é, do problema de DFA que se deseje resolver. Para alguns problemas, ao invés de seguir o fluxo de controle, definindo $out[S]$ em termos de $in[S]$, deve-se seguir o caminho inverso ao fluxo de controle.

2. Como os dados seguem o caminho do fluxo de controle, a DFA é afetada pelas estruturas de controle dos programas.

3. Existem sutilezas ao longo do fluxo de controle, como chamadas de procedimentos, atribuições por meio de variáveis do tipo ponteiro e atribuições a variáveis do tipo *array*.

A DFA poderia ser computada no CFG de um procedimento, mas em geral é mais eficiente dividí-la em uma fase local, feita a nível de bloco básico, e outra global, feita em todo o CFG. Para tal, o efeito que cada bloco básico causa na execução do programa é utilizado para computar a informação local, que será utilizada na análise global, a qual produz a informação sobre o que acontece na entrada e saída de cada bloco básico. A informação global é então combinada com a informação local, de modo a se produzir a informação que reflita o que acontece em cada uma das instruções do bloco básico. O emprego desta técnica reduz o número de passos necessários para se computar a DFA.

Os problemas de DFA podem ser classificados em basicamente dois tipos: *forward* e *backward* [5]. Os problemas *forward* são os que seguem o sentido de execução do programa. Em tais problemas os conjuntos *out* são computados em termos dos conjuntos *in*. Os problemas *backward* são os que seguem o sentido oposto ao da execução do programa. Neles os conjuntos *in* são computados em termos dos conjuntos *out*.

Em problemas *forward* a informação presente em um conjunto *out* de um bloco básico será a entrada dos conjuntos *in* de seus sucessores. Se for possível *ordenar* os blocos básicos de um CFG de modo que cada nó tenha o seu conjunto *out* computado antes de seus sucessores, então será possível resolver o problema em apenas uma passagem por cada bloco básico.

A aplicação da *ordenação topológica* a um grafo faz com que um nó do grafo venha antes de seus sucessores. Tal técnica pode fazer com que em problemas *forward*, todos os conjuntos *out* necessários ao cômputo de um conjunto *in* sejam computados antes de serem necessários, evitando assim uma nova iteração por parte do algoritmo que resolve as *data-flow equations*.

A ordenação topológica só se aplica a grafos que não contêm ciclos, mas grafos de fluxo de controle em geral possuem ciclos, o que não permite a aplicação direta da ordenação topológica. Para contornar este problema, os nós de um CFG são ordenados utilizando-se um *ordenamento quase-topológico* através de uma busca em profundidade, reduzindo assim o número de iterações necessárias para se resolver um problema de DFA, pois como muitos nós virão antes de seus sucessores, a informação irá fluir o mais distante possível [6].

A busca em profundidade ordena topologicamente um grafo acíclico e ordena quase-topologicamente um grafo cíclico. Os algoritmos a seguir mostram como realizar um ordenamento topológico ou quase-topológico em um grafo [6]:

***Algoritmo 4.1:*** Ordenação Topológica

```
OrdenaçãoTopologica
    N ← número de nós
    para todos os nós i faça
        mark[i] ← false
    fim
    DFS(nó-inicial)
fim

função DFS(i)
    se mark[i] = false então
        mark[i] ← true
        para cada sucessor s do nó i faça
```

```
            DFS(s)
        fim
        sorted[ N ] ← i
        N ← N − 1
    fim
fim
```

■

Uma vez que um CFG tenha sido ordenado quase-topologicamente, os seus nós podem ser visitados seguindo-se tal ordem. A Figura 4.1 mostra como utilizar a ordenação dos nós do CFG para se computar um problema *forward*.

```
repita
    para i ← 1 até N faça
        n ← sorted[i]
        in ← ∪_{p∈pred[n]} out[p]
        out[n] ← gen[n] ∪ (in − kill[n])
    fim
até que nenhum conjunto out sofra qualquer alteração
```

Figura 4.1: Emprego da ordenação topológica em um *forward data-flow problem*

Problemas de DFA que são *backward*, tal como *Liveness Analysis* também podem ser tratados de forma mais eficiente utilizando-se a ordenação quase-topológica, bastando apenas que a ordenação se inicie pelo nó de saída do CFG. No Xingó, a classe *CFGraph* realiza o ordenamento quase-topológico, iniciando a busca em profundidade pelo bloco básico *Header* de modo a obter a ordenação adequada para problemas *forward*. Para problemas *backward* a busca em profundidade se inicia no bloco básico *Tailer*. Quando a busca em profundidade se inicia em *Header*, somente as arestas que indicam um sucessor são percorridas, ao passo que quando a busca em profundidade se inicia em *Tailer*, somente as arestas que indicam um predecessor são percorridas. Desta forma, o Xingó armazena os ordenamentos na ordem direta e inversa ao sentido de execução do programa.

Outra forma de fazer com que a solução das *data-flow equations* seja mais rápida é através da utilização de *work-lists*. O algoritmo da Figura 4.1 permite observar que se apenas um dos conjuntos *out* sofrer alguma alteração, todas as equações serão recalculadas, recomputando assim todos os conjuntos *out*. Em muitos casos a alteração de apenas um único conjunto *out* em uma iteração do algoritmo não requer que todos sejam recalculados, pois em geral grande parte dos conjuntos *out* que não tiveram o seu valor alterado permanecerão assim durante a próxima iteração. Uma forma mais eficiente de se resolver

as equações seria recomputar somente os conjuntos que sofreram alguma alteração, e é justamente isso que o *work-list algorithm* faz. Ele armazena somente nós cujos conjuntos *out* (em problemas *backward* os conjuntos *in* é que são considerados) sofrem alteração e os recomputa. Caso o conteúdo de algum conjunto se altere, o nó a que o conjunto *out* pertence e todos os seus sucessores (predecessores em problemas *forward*) são armazenados na *work-list*. O processo é repetido até que a *work-list* esteja vazia.

***Algoritmo 4.2:*** Work-List Algorithm

```
W ← todos os nós do CFG
enquanto W não estiver vazio faça
    remova um nó n de W
    old ← out[n]
    in ← ∪_{p∈pred[n]} out[p]
    out[n] ← gen[n] ∪ (in − kill[n])
    se old ≠ out[n]
        para cada sucessor s de n faça
            se s ∉ W então
                ponha s em W
            fim
        fim
    fim
fim
```

■

Se o algoritmo de *work-list* for empregado em conjunto com a ordenação quase-topológica do CFG, a resolução do problema irá convergir de forma mais rápida. O Xingó utiliza esta combinação de modo a tornar o mais rápido possível a resolução dos sistemas de equações que compõem os problemas da DFA.

No Xingó a DFA é implementada pela classe *CFGraphInfo*. Tal classe realiza uma primeira coleta de informações, que serão exaustivamente utilizadas no cômputo da DFA, bem como durante o processo de otimização. Dentre as diversas análises de fluxo de dados conhecidas, o Xingó realiza quatro análises: *Reaching Definitions*, *Liveness Analysis*, *Available Expressions* e *Alias Analysis*.

De forma a preparar o cálculo da DFA, o objeto que representa a classe *CFGraphInfo*, percorre todo o CFG numerando todas as instruções do grafo. Em um segundo momento, tal objeto percorre todas as tabelas de símbolos necessárias e numera todas as variáveis que estejam visíveis ao procedimento que o CFG representa. Esse conjunto de informações é armazenado de forma que se possa consultar qual o número que uma determinada instrução ou variável recebeu, e dado um número, a qual instrução ou variável

ele corresponde. Essa organização visa diminuir o tempo necessário para se recuperar informações.

Uma vez que todas as instruções e variáveis tenham sido *catalogadas*, através de sua numeração, o Xingó vai inicializar todos os usos e definições de cada variável numerada. Para cada variável, o Xingó determina quais instruções atribuem um valor para a mesma e quais instruções fazem uso da mesma, ou seja, a utilizam como um operando. Também de modo a acelerar a pesquisa de informações, para cada instrução, o Xingó determina quais as variáveis que ela está definindo, ou seja, atribuindo um valor, e quais variáveis ela está usando, ou seja, utilizando como um operando.

Ao mesmo tempo em que usos e definições de variáveis são inicializados, as instruções que correspondem a expressões são igualmente inicializadas. Para cada instrução que é considerada como sendo uma expressão, o Xingó atribui um número único para aquela expressão. Isto quer dizer que instruções como $a = x * y$ e $*p = x * y$ receberão o mesmo número, pois correspondem à mesma expressão $(x * y)$. Para cada expressão o Xingó armazena quais instruções correspondem àquela expressão; a qual expressão uma instrução em particular corresponde, e para uma dada expressão, quais as variáveis utilizadas pela mesma, isto é, os seus operandos.

De modo a considerar os *alias* das variáveis de forma adequada, o Xingó também armazena informações de modo a poder lidar com ambigüidades no programa. As informações são armazenadas considerando-se possíveis usos e definições de variáveis. Para cada variável, o Xingó armazena quais são as instruções que podem potencialmente definí-la, e quais as instruções que podem potencialmente utilizá-la. Para cada instrução o Xingó armazena quais as variáveis que ela potencialmente pode definir e quais variáveis que ela potencialmente pode utilizar. A inicialização de tais informações é realizada com o auxílio de *Alias Analysis* [5], a qual é a primeira análise aplicada ao CFG.

### 4.1.1 Alias Analysis

*Alias analysis* [5, 6, 26] refere-se à determinação de locais de armazenamento que podem ser acessados de duas ou mais formas diferentes. Por exemplo, na linguagem C uma variável pode ter o seu endereço computado e desta forma ser-lhe atribuído um valor via o seu nome ou através de um ponteiro. A Figura 4.2 mostra um simples exemplo onde uma variável pode ser acessada de duas formas diferentes, uma através de seu próprio nome $n$ e outra através de uma referência indireta utilizando a forma $*p$.

Existem dois tipos de relações de *alias* a se considerar: *may alias* e *must alias*. A primeira indica o que pode ocorrer em algum caminho do CFG, a outra indica o que vai ocorrer em todos os caminhos do CFG. Se, por exemplo, todo caminho em um procedimento incluir a atribuição do endereço da variável $x$ para $p$ e a atribuição de tal valor for

```
void main()
{
        int *p;
        int n;

        p = &n;
        n = 4;
        printf("%d\n",*p);
}
```

Figura 4.2: Simples relação de *alias* com ponteiro em C

somente para $p$, então a relação "$p$ aponta para $x$" é considerada como uma informação do tipo *must alias*. Por outro lado, se o procedimento inclui caminhos que podem atribuir o endereço de uma variável $y$ para a variável ponteiro $q$ e outros que podem atribuir o endereço da variável $z$ para $q$, então a relação "$q$ pode apontar para $y$ ou para $z$" é considerada como uma informação do tipo *may alias*, pois não se sabe com certeza para qual das duas variáveis $q$ estará apontando.

Outro tipo de informação que se deve distinguir se refere a *flow-sensitive* e *flow-insensitive*. A informação do tipo *flow-insensitive* é independente do fluxo de controle encontrado em um procedimento, enquanto que a informação do tipo *flow-sensitive* depende do fluxo de controle do procedimento. A Figura 4.3 mostra como o fluxo de controle pode afetar a informação de *alias*.

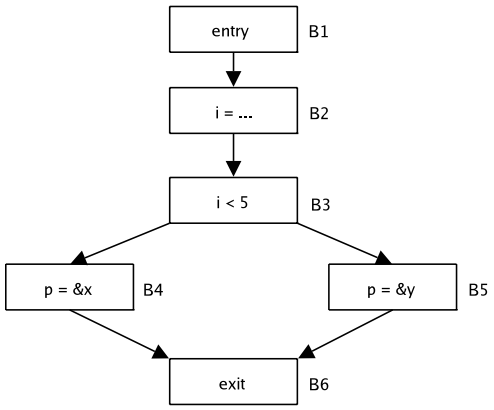


Figura 4.3: Influência do fluxo de controle sobre a informação de *alias*

A informação do tipo *flow-insensitive* dirá que $p$ poderá apontar para $x$ ou para $y$, enquanto que a informação do tipo *flow-sensitive* dirá que $p$ aponta para $x$ no bloco básico

B4 e que $p$ aponta para $y$ no bloco básico $B5$.

Este exemplo torna claro que dependendo do tipo de análise realizado, a precisão da informação obtida também é alterada. Os algoritmos de *alias analysis* que são *flow-sensitive* são os mais precisos, embora tenham um custo computacional e de memória muito elevado. Os algoritmos que são *flow-insensitive* são bem mais rápidos do que os *flow-sensitive* e têm um custo computacional bem menor em termos de tempo e memória. Variações sobre estes métodos podem ser encontradas em [27, 14, 18, 20, 28, 13].

No compilador Xingó estão presentes as duas formas de *alias analysis*, isto é, a *flow-sensitive* e a *flow-insensitive*. Desta forma é possível escolher qual das duas análises será utilizada como fornecedora das informações de *alias*, dependendo da agressividade desejada nas otimizações, pois uma informação mais precisa sobre *alias* pode fazer com que mais oportunidades de otimização de código sejam encontradas, ao passo que uma informação menos precisa pode restringir as possibilidades de transformações no código. Para computar *alias analysis* o Xingó estabelece e resolve *data-flow equations*.

Seja $in[B]$, para o bloco $B$, a função que dá para cada ponteiro $p$ o conjunto de variáveis para as quais $p$ pode apontar ao início de B. Formalmente, $in[B]$ é um conjunto de pares da forma ($p$,$a$), onde $p$ é um ponteiro e $a$ uma variável qualquer, significando que $p$ pode apontar para $a$. O conjunto $out[B]$ é definido de forma similar ao fim de $B$.

Especifica-se então, uma função de transferência $trans_B$, que define o efeito do bloco $B$, isto é, $trans_B$ toma como entrada um conjunto de pares $S$, onde cada par é da forma ($p$,$a$) e produz como saída um outro conjunto $T$. A função $trans_B$ é aplicada a $in[B]$ e gera $out[B]$, onde $B$ consiste em instruções $s_1, s_2, ...s_k$. Desta forma tem-se:

$$trans_B(S) = trans_{s_k}(trans_{s_{k-1}}(...(trans_{s_2}(trans_{s_1}(S)))...)) \tag{4.2}$$

As regras que determinam o comportamento da função $trans$ é que determinam quão complexo e custoso será o processo de cálculo de *alias analysis*. O Xingó utiliza o seguinte conjunto de regras para computar a função $trans$:

1. Se $p$ é um ponteiro, e $p = NULL$ ou $p = 0$, então:
   - Flow-Sensitive - $p$ não aponta para nada.
   - Flow-Insensitive - $p$ também pode não apontar para nada.
2. Se $p$ é um ponteiro, e $p = C$, onde $C$ é uma constante, então:
   - Flow-Sensitive - $p$ aponta para um local desconhecido.
   - Flow-Insensitive - $p$ também pode apontar para um local desconhecido.

3. Se $p$ é um ponteiro e $a$ uma variável qualquer, e $p = \&a$, então:
   - Flow-Sensitive - $p$ aponta somente para $a$.
   - Flow-Insensitive - $p$ também aponta para $a$.
4. Se $p$ e $q$ são ponteiros, e $p = q$, então:
   - Flow-Sensitive - $p$ aponta para tudo aquilo que $q$ pode apontar.
   - Flow-Insensitive - $p$ também aponta para tudo aquilo que $q$ pode apontar.
5. Se $p$ é um ponteiro, e $p = malloc(...)$ ou $p = calloc(...)$, então:
   - Flow-Sensitive - $p$ aponta para um local desconhecido.
   - Flow-Insensitive - $p$ também pode apontar para um local desconhecido.
6. Se $p$ é um ponteiro, e $p = p2 \rightarrow p3$, onde $p2$ é um ponteiro e $p3$ é um campo do tipo ponteiro, então:
   - Flow-Sensitive - $p$ aponta para tudo aquilo que $p3$ pode apontar.
   - Flow-Insensitive - $p$ também aponta para tudo aquilo que $p3$ pode apontar.
7. Se $p$ e $q$ são ponteiros, e $*q = p$, então:
   - Flow-Sensitive - Todas as variáveis para as quais $q$ aponta e que são ponteiros, passam a apontar para tudo aquilo que $p$ pode apontar.
   - Flow-Insensitive - Todas as variáveis para as quais $q$ aponta e que são ponteiros, também podem apontar para tudo aquilo que $p$ pode apontar.
8. Se $p$ e $q$ são ponteiros e $c$ uma constante, e $p = q + c$ ou $p = q - c$, então:
   - Flow-Sensitive - $p$ aponta para tudo aquilo que $q$ pode apontar.
   - Flow-Insensitive - $p$ também pode apontar para tudo aquilo que $q$ pode apontar.
9. Se $p$ e $q$ são ponteiros e $i$ uma variável não ponteiro, e $p = q + i$, então:
   - Flow-Sensitive - $p$ aponta para tudo aquilo que $q$ pode apontar.
   - Flow-Insensitive - $p$ também pode apontar para tudo aquilo que $q$ pode apontar.
10. Se $p$ é um ponteiro e $a$ uma variável não ponteiro, e $p = a$, então:
    - Flow-Sensitive - $p$ aponta para um local desconhecido.
    - Flow-Insensitive - $p$ também pode apontar para um local desconhecido.

O que se pode observar é que ao empregar *Flow-Insensitive alias analysis*, os conjuntos de *alias* sempre irão aumentar, isto é, uma vez que uma informação tenha sido inserida, ela nunca mais será removida, ao passo que em *Flow-Sensitive alias analysis* isso não é verdade, pois uma instrução da forma $p = \&a$, faz com que toda a informação prévia relativa a $p$ seja descartada e o que passa a valer é o fato de que "$p$ aponta para $a$".

Para computar *Flow-Insensitive alias analysis* o Xingó simplesmente percorre todas as instruções do programa uma única vez e constrói um conjunto de relações de *alias*, com base em *trans*, que lembram um grafo, como mostrado na Figura 4.4. Esse modelo torna a análise praticamente linear no tamanho do programa (quantidade de instruções). As operações mais caras são $*p = q$ e $p = *q$ onde $p$ e $q$ são ponteiros. Tais instruções são marcadas pelo compilador e todos os conjuntos de *alias* são acertados depois que o programa é completamente percorrido.

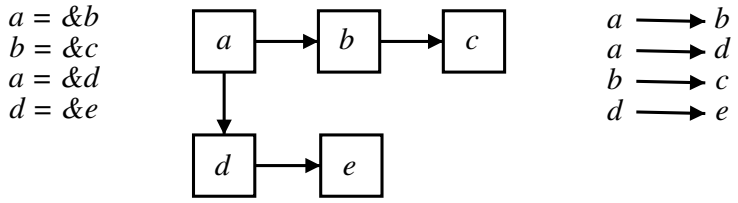


Figura 4.4: *Flow-Insensitive alias analysis*

Para computar *Flow-Sensitive alias analysis* o Xingó deve estabelecer um sistema de *data-flow equations* e resolver o mesmo. O conjunto de *data-flow equations* utilizado pelo Xingó, que relaciona *in*, *out* e *trans* é:

$$\begin{aligned} out[B] &= trans_B(in[B]) \\ in[B] &= \cup out[P] \end{aligned} \tag{4.3}$$

Como se observa, esse sistema de equações leva a uma solução iterativa do problema, ou seja, elas são aplicadas a todos os blocos básicos até que a solução do problema se estabilize. Como este problema é *forward*, entende-se por estabilização da solução o fato de repetir o cálculo de tais equações até que nenhuma mudança ocorra em nenhum dos conjuntos *out* de qualquer um dos blocos básicos.

Os dois métodos de *alias analysis* têm vantagens e desvantagens, dependendo dos objetivos desejados. A Tabela 4.1 mostra uma comparação entre os resultados obtidos ao se utilizar os dois métodos para um mesmo programa.

A tabela apresenta as relações de *alias* para cada linha do programa. Na primeira linha a relação <*p,r> na coluna *Flow-Sensitive* significa que *p e $r$ correspondem à mesma posição de memória, isto é, *p é um *alias* de r, o mesmo valendo para as demais relações mostradas.

| void func() | Flow-Sensitive | Flow-Insensitive |
|---|---|---|
| { | | |
| p = &r; | {<*p,r>} | {<*p,r>,<*q,s>,<*q,r>,<*q,t>} |
| if (...) | | |
| q = p; | {<*p,r>,<*q,r>} | {<*p,r>,<*q,s>,<*q,r>,<*q,t>} |
| else | | |
| q = &s; | {<*p,r>,<*q,s>} | {<*p,r>,<*q,s>,<*q,r>,<*q,t>} |
| ... | {<*p,r>,<*q,s>,<*q,r>} | {<*p,r>,<*q,s>,<*q,r>,<*q,t>} |
| q = &t; | {<*p,r>,<*q,t>} | {<*p,r>,<*q,s>,<*q,r>,<*q,t>} |
| } | | |

Tabela 4.1: Comparação entre *Flow-Sensitive* e *Flow-Insesitive alias analysis*

A análise *Flow-Insensitive* faz com que todas as relações de *alias* coletadas no programa sejam válidas em todos os pontos do mesmo.

### 4.1.2 Reaching Definitions

Uma *definição* de uma variável $x$ é uma instrução que atribui, ou pode atribuir um valor para $x$. Podemos dividir as definições em dois grupos: ambíguas e não-ambíguas. Definições não-ambíguas são aquelas que definitivamente atribuem um valor a $x$, como por exemplo, uma instrução da forma $x = 5$. Uma definição ambígua é tal que não fornece a certeza de atribuir ou não um valor para a variável $x$, como por exemplo uma instrução $*q = 5$, onde $q$ é uma variável ponteiro e não se sabe exatamente se $q$ aponta ou não para $x$.

A partir disso, diz-se que uma definição $d$ alcança um ponto $p$ se existe um caminho de $d$ até $p$, tal que $d$ não seja "morta" ao longo do caminho. Intuitivamente se a definição $d$ de uma variável $a$ alcança um ponto $p$, então $d$ é o ponto onde o valor de $a$ usado em $p$ foi definido pela última vez. Uma definição da variável $a$ é morta entre dois pontos se houver outra definição da mesma ao longo do caminho. De forma a manter a relação conservativa, somente definições não-ambíguas de uma variável $a$ matam outras definições de $a$, o que leva a concluir que um determinado uso da variável $a$ pode ser alcançado por mais de uma definição da mesma [5].

Com o intuito de descobrir quais definições em particular conseguem alcançar um ponto qualquer do programa, utiliza-se a análise de fluxo de dados chamada *Reaching Definitions* [5, 6, 26], a qual irá determinar justamente tal informação. *Reaching definitions* é um problema *forward*, isto é, segue o sentido de execução do programa. As informações fornecidas por esta análise são extremamente utilizadas durante o processo de otimização,

como por exemplo em *Dead Code Elimination* que elimina todas as definições do programa que não alcançam nenhum uso em particular.

Nesta análise define-se $in[B]$ como sendo o conjunto de definições que estão vivas ao início do bloco básico $B$ e $out[B]$ como sendo o conjunto de definições que estão vivas na saída de $B$. Adicionalmente a estes dois conjuntos, são utilizados outros dois: $gen[B]$ e $kill[B]$. O conjunto $gen[B]$ corresponde às definições que são geradas pelo bloco básico $B$. Diz-se que uma definição $d$ de uma variável $a$ é gerada por um bloco $B$ se ela ocorre em $B$ e permanece viva até a saída do mesmo, ou seja, não existe outra definição não-ambígua de $a$ após $d$. Uma definição ambígua de $a$ não mata $d$, pois como não se sabe se $a$ foi ou não de fato definida, o mais seguro a se fazer é não matar $d$. O conjunto $kill[B]$ corresponde a todas as definições do programa que são mortas por $B$.

No Xingó o cálculo de *Reaching Definitions* é implementado pela classe *ReachDefClass*. Quando um objeto desta classe é criado, sua primeira providência é inicializar os conjuntos *gen* e *kill* para cada bloco básico do CFG. A Figura 4.5 mostra o pseudo-código do algoritmo empregado pelo Xingó para inicializar tais conjuntos.

```
para cada bloco básico B faça
    gen[B] ← ∅
    kill[B] ← ∅
    para cada instrução i ∈ B faça
        se i é uma definição então
            v ← variável definida por i
            g2 ← InstOrd(i)
            k2 ← todas as definições da variável v
            k2 ← k2 − g2
            gen[B] ← gen[B] − k2
            gen[B] ← gen[B] ∪ g2
            kill[B] ← kill[B] − g2
            kill[B] ← kill[B] ∪ k2
        fim
    fim
fim
```

Figura 4.5: Inicialização dos conjuntos *gen* e *kill*

A função *InstOrd* presente no algoritmo está implementada na classe *CFGraphInfo*. Tal função retorna o número que uma determinada instrução recebeu ao ser *catalogada* em preparação para a DFA. Também no algoritmo, a linha "$k2 \leftarrow$ todas as definições da variável $v$", emprega a informação computada por *CFGraphInfo*.

Uma vez que os conjuntos *gen* e *kill* tenham sido inicializados para todos os blocos

básicos, o Xingó está pronto para computar *Reaching Definitions*. A Figura 4.6 mostra o pseudo-código do algoritmo utilizado pelo Xingó para computar *Reaching Definitions*.

Esse código faz uso da ordenação quase-topógica dos blocos básicos. A ordem resultante é consultada utilizando-se a função *DFSOrderNode*, a qual recebe como parâmetro o i-ésimo nó que se deseja consultar. O algoritmo também utiliza um sistema de *work-list* para saber quais blocos básicos precisam ter o conjunto *out* recomputado.

```
para i ← 1 até o total de blocos básicos faça
        insira i ao final de work-list
fim
enquanto work-list ≠ vazio faça
        i ← primeiro elemento de work-list
        B ← DFSOrderNode(i)
        in[B] ← ∪_{p∈pred[B]} out[p]
        oldout ← out[B]
        out[B] ← gen[B] ∪ (in[B] − kill[B])
        se out[B] ≠ oldout então
                para cada sucessor s de B faça
                        insira s em work-list
                fim
        fim
fim
```

Figura 4.6: Algoritmo para computar *Reaching Definitions* no Xingó

Uma vez que o Xingó tenha computado os conjuntos *in* e *out*, ele propaga a informação coletada para dentro dos blocos básicos, isto é, ele determina para cada instrução do programa quais são as definições que a atingem. Outra informação computada a partir de *Reaching Definitions* são *Du-Chains* (cadeias definição-uso) e *UdChains* (cadeias uso-definição). *Du-Chains* determinam para uma definição $d$ em particular de uma variável $a$, quais são os usos de $a$ alcançados por $d$. *Ud-Chains* são justamente o contrário, ou seja, para um dado uso $u$ em particular de uma variável $a$, elas representam quais são as definições $d$ de $a$ que alcançam o uso $u$.

### 4.1.3 Liveness Analysis

Dizemos que uma determinada variável está *viva* se o seu valor puder ser utilizado no futuro. Se em um determinado ponto do programa uma variável nunca mais tiver o seu valor utilizado, dizemos que esta variável está *morta* a partir deste ponto.

A Figura 4.7 ilustra este conceito. A análise é feita seguindo-se o sentido inverso ao da execução do programa, desta forma este é um problema *backward*. Por exemplo, a variável $b$ está viva no bloco B4, pois é usada no mesmo. Ela também é usada no bloco B3, e também está viva neste. O bloco B2 atribui um valor a $b$, significando que seu conteúdo não será mais necessário a partir deste ponto. Desta forma a variável $b$ está viva na entrada dos blocos básicos B4 e B3. A variável $a$ por sua vez, está viva na entrada do bloco básico B2 e na entrada do bloco B5. A definição de $a$ em B4, faz com que o valor de $a$ não seja mais necessário a partir desse ponto. A variável $c$ é a única que está viva em todo o programa, pois mesmo sendo definida no bloco B3, seu valor é utilizado no bloco B6.

O objetivo de *Liveness Analysis* [5, 6, 26] é descobrir quais variáveis estão vivas em que pontos do programa. Essa análise é extremamente útil, por exemplo, para se realizar a alocação de registradores.

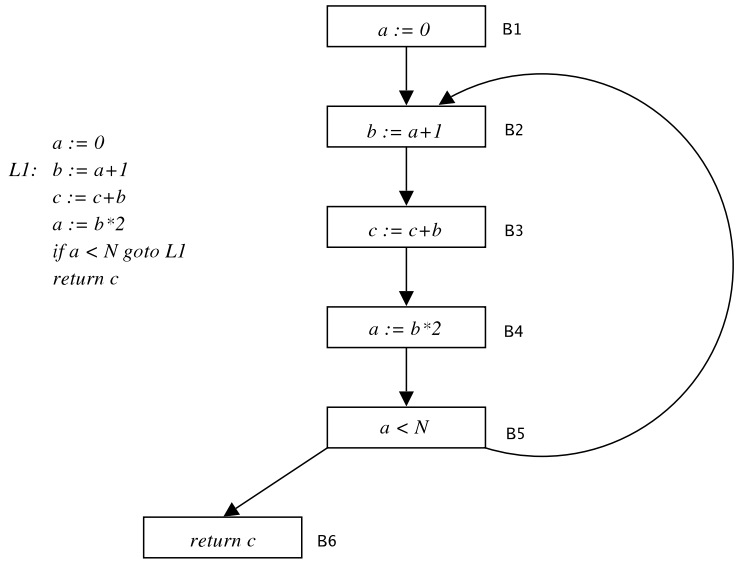


Figura 4.7: CFG de um programa

Nesta análise define-se $in[B]$ como sendo o conjunto de variáveis vivas no início do bloco básico $B$ e $out[B]$ como o conjunto de variáveis vivas na saída de $B$. Os conjuntos *gen* e *kill* para este problema são chamados de *use* e *def* respectivamente, pois estes nomes refletem com maior clareza o que de fato acontece. O conjunto *use[B]* pode ser entendido como o conjunto de variáveis que são utilizadas por $B$ antes que qualquer definição das mesmas ocorra em $B$. O conjunto *def[B]* corresponde ao conjunto de variáveis que são definidas em $B$ antes que qualquer uso das mesmas ocorra.

O Xingó implementa *Liveness Analysis* através da classe *LivenessClass*. A primeira providência de um objeto desta classe é inicializar os conjuntos *use* e *def* para cada bloco

básico que compõe o CFG. A Figura 4.8 mostra o pseudo-código do algoritmo utilizado pelo Xingó para inicializar estes conjuntos.

**para** cada bloco básico $B$ **faça**
    $use[B] \leftarrow \varnothing$
    $def[B] \leftarrow \varnothing$
    **para** cada instrução $i$ de baixo para cima $\in B$ **faça**
        $u2 \leftarrow$ variáveis utilizadas por $i \cup$ variáveis potencialmente utilizadas por $i$
        $d2 \leftarrow$ variável definida por $i$
        $use[B] \leftarrow use[B] - d2$
        $use[B] \leftarrow use[B] \cup u2$
        $def[B] \leftarrow def[B] - u2$
        $def[B] \leftarrow def[B] \cup d2$
    **fim**
**fim**

Figura 4.8: Inicialização dos conjuntos *use* e *def*

**para** $i \leftarrow 1$ até o total de blocos básicos **faça**
    insira $i$ ao final de *work-list*
**fim**
**enquanto** *work-list* $\neq$ vazio **faça**
    $i \leftarrow$ primeiro elemento de *work-list*
    $B \leftarrow$ RDFSOrderNode($i$)
    $out[B] \leftarrow \cup_{s \in suc[B]}\ in[s]$
    $oldin \leftarrow in[B]$
    $in[B] \leftarrow use[B] \cup (out[B] - def[B])$
    **se** $in[B] \neq oldin$ **então**
        **para** cada predecessor $p$ de $B$ **faça**
            insira $p$ em *work-list*
        **fim**
    **fim**
**fim**

Figura 4.9: Algoritmo para computar *Liveness* no Xingó

Uma vez que o Xingó tenha inicializado os conjuntos *use* e *def*, ele está pronto para realizar *Liveness Analysis*. A Figura 4.9 mostra o pseudo-código do algoritmo utilizado pelo Xingó para realizar esta análise.

A função *RDFSOrderNode* presente no algoritmo é utilizada de forma a percorrer os nós do CFG seguindo o sentido inverso ao de execução do programa, pois pretende-se que cada conjunto *in* de um determinado bloco $B$ seja computado antes que ele seja

necessário para computar *out* nos predecessores de $B$. Além de seguir o ordenamento quase-topológico (de forma inversa), o algoritmo também emprega uma *work-list* para saber quais conjuntos *in* sofreram alteração, e desta forma recomputar os predecessores dos respectivos blocos.

Depois que os conjuntos *in* e *out* estão computados para todos os blocos do CFG, o Xingó propaga a informação coletada para todos os blocos básicos, isto é, determina quais as variáveis que estão vivas antes de cada instrução do programa.

### 4.1.4 Available Expressions

Uma expressão $x + y$ está *disponível* em um ponto $p$ do programa se todos os caminhos partindo do nó inicial do programa (não necessariamente livres de ciclos) até $p$ avaliam $x + y$ e após a última avaliação de $x + y$ antes de $p$, não existe nenhuma definição de $x$ ou de $y$. A partir desta definição, diz-se que um bloco $B$ *mata* uma expressão $x + y$ se ele atribuir ou puder atribuir um valor a $x$ ou a $y$. Deve-se observar que o simples fato de poder atribuir um valor a uma variável (definição ambígua) também faz com que uma expressão não esteja mais disponível a partir de tal definição, pois neste problema de análise de fluxo de dados, a solução conservativa a ser tomada, considera que o número de expressões disponíveis é menor do que o valor real [5].

A simples definição do conceito de expressões disponíveis leva a concluir que este é um problema *forward* e desta forma segue o sentido de execução do programa. O objetivo desta análise é descobrir quais expressões estão disponíveis em um ponto qualquer do programa. Desta forma define-se $in[B]$ como sendo o conjunto de expressões que estão disponíveis na entrada do bloco básico $B$ e $out[B]$ como o conjunto de expressões que estão disponíveis na saída desse mesmo bloco. A realização de tal análise é a base da otimização *common subexpressions elimination* [5].

Assim como nas outras análises, esta também é dividida em um fase local e outra global. A fase local corresponde a inicialização dos conjuntos *gen* e *kill*, que aqui serão chamados de *e_gen* e *e_kill*. Um bloco básico $B$ gera uma expressão $e$ se ele avalia $e$ e não define (nem mesmo potencialmente) qualquer um dos operandos de $e$. Um bloco mata uma expressão $e$ se ele puder definir (mesmo ambiguamente) qualquer um dos operandos de $e$.

O Xingó implementa esta análise de fluxo de dados através da classe *AvailExpClass*. Tal como as demais classes que compõem a DFA, a primeira providência de um objeto da classe *AvailExpClass* é inicializar os conjuntos que correspondem a fase local da análise. Os conjuntos *e_gen* e *e_kill* são criados aplicando-se as regras a seguir para cada instrução $i$ de um bloco básico do programa:

- Adicione a expressão gerada por $i$ ao conjunto *e_gen*

- Remova de *e_kill* a expressão gerada por $i$
- Remova de *e_gen* todas as expressões cujos operandos $i$ defina ou possa definir
- Adicione a *e_kill* todas as expressões cujos operandos $i$ defina ou possa definir

Estas regras devem ser aplicadas necessariamente na ordem em que são apresentadas, caso contrário levarão a uma inicialização errônea dos conjuntos *e_gen* e *e_kill*. A Figura 4.10 mostra o pseudo-código do algoritmo empregado pelo Xingó para inicializar os conjuntos *e_gen* e *e_kill*.

```
para cada bloco básico B faça
    e_gen ← ∅
    e_kill ← ∅
    para cada instrução i ∈ B faça
        se i é uma expressão então
            g2 ← ExpressionId(i)
            v ← variável definida por i
            k2 ← VarUsedExp(v)
            para cada variável vp que i potencialmente possa definir faça
                k2 ← k2 ∪ VarUsedExp(vp)
            fim
            e_gen[B] ← e_gen[B] ∪ g2
            e_kill[B] ← e_kill[B] − g2
            e_gen[B] ← e_gen[B] − k2
            e_kill[B] ← e_kill[B] ∪ k2
        fim
    fim
fim
```

Figura 4.10: Inicialização dos conjuntos *e_gen* e *e_kill*

A função *ExpressionId* presente no código, retorna o número identificador da expressão a que a instrução passada como parâmetro corresponde. A função *VarUsedExp* recebe como parâmero uma variável e retorna quais expressões utilizam esta variável como um de seus operandos. Ambas as funções *ExpressionId* e *VarUsedExp* estão implementadas na classe base da DFA, a classe *CFGraphInfo*.

Depois que a fase local da análise está pronta, o Xingó parte para a fase global. Este problema de análise de fluxo de dados, ao contrário dos anteriores, utiliza um operador de confluência diferente. Nos problemas até agora apresentados, o operador de confluência era a união ($\cup$), mas neste problema o operador de confluência é a intersecção ($\cap$). Isto

```
in[Header] ← ∅
out[Header] ← e_gen[Header]
para todo bloco B ≠ Header faça
        in[B] ← ∅
        out[B] ← U − e_kill[B]
fim
para i ← 1 até o total de blocos básicos faça
        insira i ao final de work-list
fim
enquanto work-list ≠ vazio faça
        i ← primeiro elemento de work-list
        B ← DFSOrderNode(i)
        in[B] ← ∩_{p∈pred[B]} out[p]
        oldout ← out[B]
        out[B] ← e_gen[B] ∪ (in[B] − e_kill[B])
        se out[B] ≠ oldout então
                para cada sucessor s de B faça
                        insira s em work-list
                fim
        fim
fim
```

Figura 4.11: Algoritmo para computar *Available Expressions* no Xingó

decorre da definição de expressão disponível, pois ela precisa ser avaliada em *todos* os caminhos a partir do nó inicial.

A Figura 4.11 mostra o pseudo-código do algoritmo utilizado pelo Xingó para computar *Available Expressions*. Observa-se que este algoritmo foi inicializado de forma diferente dos demais. O bloco *Header* teve o seu conjunto *in* inicializado com $\varnothing$, pois como não possui nenhum predecessor, obviamente não haverá nenhuma expressão disponível em sua entrada. Já o seu conjunto *out* foi inicializado com $e\_gen[Header]$, o qual nunca irá se alterar e desta forma *Header* será removido da *work-list* na primeira iteração. Os demais blocos básicos tiveram o seu conjunto *out* inicializados com $U - e\_kill$, onde $U$ corresponde ao conjunto universo, que contém todas as expressões do CFG. Esta é uma estimativa inicial muito grande que será refinada no decorrer do algoritmo, pois o operador de confluência $\cap$ faz com que o tamanho do conjunto nunca aumente, somente diminua.

Depois que a análise está completa, o Xingó novamente propaga a informação coletada para todos os blocos do CFG, isto é, ele determina quais expressões estão disponíveis antes de cada uma das instruções presentes em cada um dos blocos básicos.

### 4.1.5 Incremental Data Flow Analysis

Cada vez que uma otimização é executada e realiza uma transformação no programa, a informação sobre DFA se torna inválida e desta forma é preciso recomputar toda a informação antes de se poder realizar uma nova otimização. Mas recomputar todas as informações da DFA sempre que uma otimização modificar o programa fará com que o tempo de compilação seja demasiado longo.

Para evitar recomputar toda a DFA, o Xingó utiliza uma técnica chamada *Incremental Data Flow Analysis* [10] ou IDFA que ao invés de recomputar toda a informação desde o início, recomputa somente o que foi modificado, realizando um *patch* na informação disponível anteriormente de modo que ela continue válida.

Em alguns casos a informação proveniente da IDFA pode diferir da que seria obtida caso a DFA fosse recomputada desde o princípio. Desta forma a IDFA deve ser tal que a informação por ela gerada, caso não seja exata, ao menos seja conservativa, isto é, não permita que nenhuma otimização realize uma transformação indevida. Métodos para se realizar IDFA podem ser encontrados em [25] e [10].

O Xingó utiliza um método iterativo, onde a idéia básica é reiniciar o processo de iteração das *data-flow equations*. Cada otimização que é executada marca os blocos básicos por ela modificados. A partir desta informação, as classes responsáveis pela DFA reinserem os blocos modificados em uma *work-list* e o processo iterativo de resolução dos sistemas de *data-flow equations* é reiniciado.

Se esta técnica não for aplicada da forma correta, o resultado obtido vai diferir muito do que seria obtido pela aplicação do método exaustivo (recomputar tudo desde o início), ou poderá até mesmo estar incorreto. É necessário que a iteração seja reiniciada de um estado considerado *seguro*. Se o atual estado da DFA é um estado seguro, então basta reiniciar o processo iterativo, caso contrário parte da solução (senão toda ela) terá que ser reinicializada para um estado considerado seguro. Por estado seguro entende-se um ponto a partir do qual a DFA pode ter o processo iterativo reiniciado e produzir um resultado correto e conservativo.

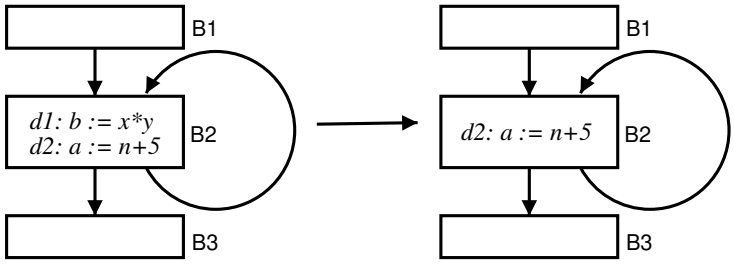


Figura 4.12: Aplicação de *Dead Code Elimination*

A Figura 4.12 mostra o que pode acontecer caso a técnica de IDFA não seja aplicada

corretamente. No grafo da esquerda o bloco B2 possui duas definições $d1$ e $d2$ respectivamente. Se aplicarmos a análise *Reaching Definitions*, teremos que:

$$in[B2] = \{d1, d2\}$$
$$out[B2] = \{d1, d2\}$$

Supondo que a otimização *Dead Code Elimination* descobriu que a definição $d1$ pode ser eliminada do programa, pois não atinge nenhum uso em particular, obteremos o grafo da direita. A aplicação de *Reaching Definitions* em tal grafo irá resultar em:

$$in[B2] = \{d2\}$$
$$out[B2] = \{d2\}$$

Se a IDFA fosse aplicada ao grafo da direita considerando-se a informação previamente computada, o bloco B2 seria inserido na *work-list*. Na primeira iteração, o algoritmo irá atualizar o conteúdo de $in[B2]$ fazendo a união dos conjuntos *out* de todos os seus predecessores. Os predecessores do bloco B2 são B1 e o próprio B2. Supondo que $out[B1] = \varnothing$, então o novo $in[B2]$ será $\{d1, d2\}$. O conjunto $kill[B2]$ claramente não contém $d1$, desta forma ao se realizar $out[B2] = gen[B2] \cup (in[B2] - kill[B2])$, o resultado será $out[B2] = \{d1, d2\}$, que é igual ao valor anterior de $out[B2]$, o que fará com que B2 seja removido da *work-list* e o seguinte resultado seja armazenado:

$$in[B2] = \{d1, d2\}$$
$$out[B2] = \{d1, d2\}$$

Tal resultado é incorreto e possivelmente poderia causar problemas se o grafo fosse mais complexo. Neste caso específico, onde uma instrução foi eliminada, o estado inicial seguro para se reinicializar a DFA seria recomputar tudo desde o início ou tornar os conjuntos *out* de todos os blocos igual a $\varnothing$.

Cada tipo de modificação introduzida no programa deve ser analisada para determinar que estado da DFA pode ser considerado seguro para as mudanças realizadas. Idealmente, a complexidade de um algoritmo incremental $I$ para um problema de análise de fluxo de dados $P$ deveria depender somente do tamanho de $\Delta P$, mas isso raramente pode ser alcançado. Uma abordagem mais realista diz que a complexidade será significativamente menor do que recomputar toda a solução desde o início [25].

## 4.2 Otimização de Código

Uma vez que a *Data Flow Analysis* tenha sido computada, o Xingó está pronto para aplicar as otimizações ao CFG.

O Xingó possui as seguintes otimizações: *Dead Code Elimination*, *Copy Propagation*, *GCSE*, *Code Motion*, *Strength Reduction*, *Constant Propagation*, *Pointer Optimization* e *PeepHole Optimization*. Essas otimizações são executadas em um *loop* até que mais nenhuma transformação ocorra. Elas estão implementadas de forma a não depender do resultado gerado por outras otimizações, o que as tornam independentes umas das outras.

Uma questão importante é a ordem em que as otimizações são executadas, pois a sua escolha é crucial para se obter compiladores otimizantes agressivos [26]. Não existe uma ordem específica que irá produzir os melhores resultados possíveis. É possível criar exemplos para mostrar que nenhuma ordem é ótima para todos os programas [12]. Em geral se utiliza uma seqüência fixa de aplicação das otimizações, ou um conjunto de *flags* é utilizado para controlar o processo de otimização.

O compilador Xingó utiliza uma ordem fixa para aplicar as otimizações. Tal ordem é derivada da seqüência proposta em [26]. O Xingó aplica as otimizações na seguinte seqüência:

1. Pointer Optimization
2. Copy Propagation
3. Constant Propagation
4. Dead Code Elimination
5. GCSE
6. PeepHole Optimization
7. Code Motion
8. Dead Code Elimination
9. Strength Reduction

O Xingó contém uma classe chamada *Optimizer* que controla o laço que contém as otimizações. Essa classe é responsável por invocar cada uma das otimizações, verificar quais blocos básicos foram modificados e reiniciar a execução da *Data Flow Analysis*, para que a mesma atualize as informações necessárias.

As otimizações do Xingó são todas globais, isto é, atuam em todo o CFG, mas são apenas intra-procedurais. Na presença de chamadas de função, abordagens conservativas são tomadas de modo a evitar transformações indevidas no programa.

Todas as otimizações que o Xingó possui são independentes de máquina. As otimizações independentes de máquina são transformações no programa que melhoram o código

do mesmo sem considerar qualquer propriedade da máquina alvo. As melhorias podem estar relacionadas ao tamanho do código gerado ou a velocidade do mesmo, ou ambas as coisas.

As transformações realizadas por um compilador otimizante devem possuir diversas propriedades. Em primeiro lugar, a transformação deve preservar o significado do programa nunca alterando o que ele computa, ou seja, nunca mudar o resultado de uma determinada entrada ou causar um erro que não existia. Em segundo lugar, uma transformação deve aumentar a velocidade do programa de maneira mensurável, embora algumas vezes isso implique em aumentar o tamanho do código. Por último, o esforço de se implementar uma otimização deve valer a pena, pois de nada adianta implementar uma transformação trabalhosa que aumenta o tempo de compilação, se esta produz resultados que não se fazem sentir no programa final.

As otimizações independentes de máquina implementadas pelo Xingó são executadas ao nível da representação intermediária, isto é, na XIR. Isto se deve ao fato de que as operações necessárias para implementar as construções de alto nível se tornam explícitas neste tipo de representação. Como a XIR é independente da máquina alvo, o otimizador do Xingó não sofre mudanças se o gerador de código for substituído por outro, caso se deseje que o compilador gere código para outra arquitetura.

Todas as otimizações implementadas pelo Xingó têm o objetivo de melhorar o desempenho do programa, mesmo que indiretamente. As próximas seções farão uma breve descrição de como o Xingó implementa cada uma das suas otimizações.

### 4.2.1 Constant Propagation

O objetivo desta transformação de código é substituir a utilização de variáveis por valores constantes sempre que possível. O trecho de código abaixo ilustra tal situação:

$$\begin{aligned} &\text{d1: } t \leftarrow 5 \\ &\text{d2: } f \leftarrow g \oplus t \end{aligned}$$

Claramente a variável $t$ na definição d2 pode ser substituída pela constante 5, resultando em $f \leftarrow g \oplus 5$. A aplicação de outras otimizações fatalmente removerá a definição d1, otimizando ainda mais o código.

Para executar este tipo de transformação, o Xingó faz uso da informação coletada pela análise de fluxo de dados *Reaching Definitions*. O algoritmo a seguir mostra o pseudocódigo do algoritmo empregado pelo Xingó para realizar *Constant Propagation*.

***Algoritmo 4.3:*** Constant Propagation
*Entrada*:
Todas as instruções $i$ da forma $v \leftarrow C$, onde $v$ é uma variável e $C$ é uma

```
    constante;
    DuChain e UdChain computadas
para cada definição i recebida como entrada faça
    para cada uso u que a instrução i alcança faça
        se i for a única definição que alcança o uso de v em u então
            substituir v por C em u
        fim
    fim
fim
```

■

O algoritmo recebe como entrada todas as atribuições de valores constantes para variáveis. A seguir, para cada definição $i$ recebida o algoritmo consulta *DuChains* e descobre quais são os usos da variável $v$ definida pela instrução. Para cada uso $u$ de $v$ que $i$ alcança, o algoritmo verifica, consultando *UdChains*, se $i$ é a única definição que o alcança. Em caso afirmativo a variável $v$ no uso $u$ é substituída pela constante $C$ que a instrução $i$ atribuiu a $v$.

### 4.2.2 Dead Code Elimination

O objetivo desta transformação de código é diminuir o espaço que o programa ocupa, bem como aumentar a velocidade do mesmo, na medida em que ela vai eliminar instruções que não causam nenhum efeito sobre o resultado final do programa. As instruções a seguir mostram três definições de duas variáveis diferentes:

$$\text{d1: } a \leftarrow i \oplus j$$
$$\text{d2: } b \leftarrow k \oplus l$$
$$\text{d3: } a \leftarrow u \oplus v$$

Pode-se observar que existem duas definições da variável $a$ no trecho apresentado. A definição da variável $a$ em d1 não é utilizada em nenhum momento, nem por d2 e nem por d3; e a variável $a$ é novamente definida em d3, fazendo com que o valor atribuído a $a$ em d1 nunca passe por d3, desta forma a definição d1 está computando um valor que nunca será utilizado por nenhuma instrução.

Nestas condições d1 recebe a definição de código morto, pois realiza uma computação desnecessária, e pode portanto ser eliminada do programa, resultando no código a seguir:

$$\text{d2: } b \leftarrow k \oplus l$$
$$\text{d3: } a \leftarrow u \oplus v$$

O pseudo-código do algoritmo utilizado pelo Xingó para eliminar código morto é mostrado a seguir:

***Algoritmo 4.4:*** Dead Code Elimination
*Entrada*:
Informação de *DuChains*
**para** cada definição $d$ do programa **faça**
**se** a variável $v$ definida em $d$ não alcançar nenhum uso **então**
remover $d$ do programa
**fim**
**fim** ■

O algoritmo é bem simples, para cada definição $d$ do programa, o algoritmo verifica com o auxílio de *DuChains* se a variável definida em $d$ alcança algum uso. Se nenhum uso for alcançado por $d$ então ela é considerada como código morto e portanto pode ser removida do programa, causando assim uma melhora no espaço ocupado pelo mesmo, bem como em seu desempenho.

### 4.2.3 Pointer Optimization

Esta transformação de código tem como objetivo melhorar o desempenho do programa diminuindo o acesso indireto às variáveis por meio de ponteiros. O trecho de código a seguir ilustra uma situação onde uma variável é acessada indiretamente.

$$\text{d1: } p = \&i$$
$$\text{d2: } j = *p$$

Em d1, a variável $p$ recebe o endereço da variável $i$, passando então a apontar para ela. Em d2, o valor de $i$ é utilizado indiretamente através de $*p$. Claramente $p$ aponta para $i$ em d2. Desta forma o trecho de código poderia ser mudado para:

$$\text{d1: } p = \&i$$
$$\text{d2: } j = i$$

fazendo com que a instrução em d1 se torne código morto, caso não alcance mais nenhum uso de $p$. Este tipo de transformação é realizada pelo Xingó com o auxílio de *alias analysis*. Deve-se observar que só é seguro realizar tal alteração se, e somente se a variável ponteiro apontar para uma única variável no ponto sob análise e não existir a possibilidade dela apontar para um lugar desconhecido ou para NULL.

***Algoritmo 4.5:*** Pointer Optimization
*Entrada*:
Informação de *alias analysis*
**para** cada instrução $i$ do programa **faça**
**se** $i$ é da forma $*p \leftarrow x$ ou $x \leftarrow *p$ **então**
**se**, e somente se $p$ aponta para uma única variável $v$ **então**
substitua $*p$ por $v$
**fim**
**fim**
**fim** ■

O pseudo-código apresentado, mostra em linhas gerais o método utilizado pelo Xingó para realizar esta transformação de código. Primeiro o algoritmo procura uma instrução que realize uma escrita ou leitura na memória, em seguida ele verifica se a variável ponteiro aponta para uma única variável e se não existe a possibilidade do ponteiro ser NULL ou apontar para um lugar desconhecido. Caso seja seguro, o algoritmo irá substituir a referência à memória pela variável para qual o ponteiro aponta. Esta transformação de código pode não causar uma melhoria no tamanho do programa, todavia resulta em uma melhoria no seu tempo de execução.

### 4.2.4 Copy Propagation

Esta transformação de código por si só não causa nenhuma melhora no desempenho ou no espaço ocupado pelo programa. Seu objetivo é otimizar o uso de variáveis presentes em instruções de cópia. Uma instrução de cópia é uma atribuição da forma $a \leftarrow b$. *Copy Propagation* tenta transformar os usos de $a$ em usos de $b$ sempre que possível, o que poderá fazer com que a instrução de cópia se torne código morto e seja eliminida por *dead code elimination*.

A Figura 4.13 mostra o funcionamento desta transformação de código. Em (a) observa-se que o bloco básico B1 contém uma instrução de cópia ($b = a$). Ainda em (a), os blocos básicos B2 e B3 fazem uso da variável $b$. A variável $a$ não foi definida entre a instrução de cópia em B1 e os usos de $b$ em B2 e B3. Desta forma os usos de $b$ podem ser substituídos por usos de $a$, resultando no CFG apresentado na Figura 4.13 (b).

Se não existir nenhum outro uso de $b$ em nenhum outro ponto do programa, então a instrução de cópia em B1 se transformará em código morto e poderá ser eliminada.

Embora esta transformação pareça simples e semelhante a *constant propagation*, duas condições devem ser obedecidas. Em instruções de cópia $c$ da forma $b \leftarrow a$, substitui-se o uso de $b$ pelo uso de $a$ em todas as instruções $s$ que satisfaçam as seguintes condições:

1. A instrução de cópia $c$ é a única definição que alcança o uso de $b$ em $s$

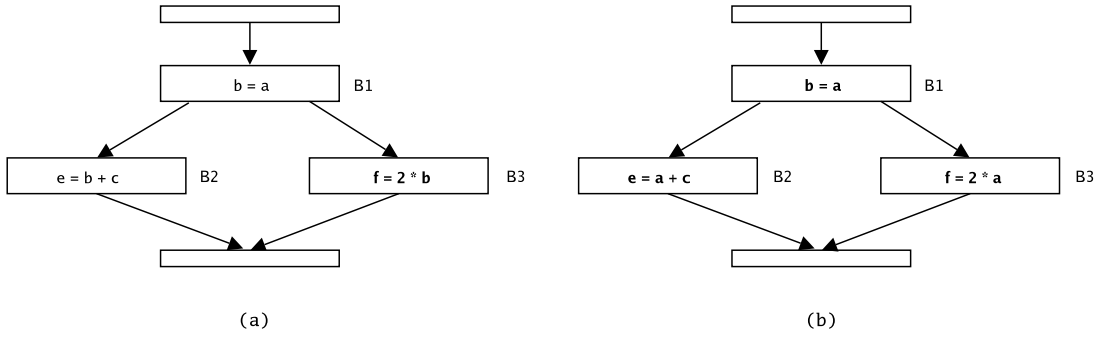


Figura 4.13: Exemplo de *Copy Propagation*

2. Em todos os caminhos de $c$ até $s$ e que podem passar por $s$ diversas vezes, mas não uma segunda vez por $c$, não deve haver nenhuma definição da variável $a$.

A condição 1 pode ser verificada através de *UdChains*, mas a condição 2 acaba levando a um problema de *data flow analysis*, cujo objetivo é determinar quais cópias estão vivas em que pontos do programa.

Para resolver este novo problema de DFA o Xingó inicialmente irá catalogar todas as cópias do programa, atribuindo a cada uma delas um número distinto. O segundo passo é inicializar os conjunto *c_gen* e *c_kill* para cada bloco básico, onde *c_gen* corresponde as cópias que são geradas por um bloco básico e *c_kill* corresponde as cópias que são mortas por um bloco básico.

A Figura 4.14 mostra o pseudo-código do algoritmo empregado pelo Xingó para inicializar os conjuntos *c_gen* e *c_kill*.

A função *CopyId* presente no algoritmo retorna o número a que uma cópia está associado. A função *VarUsedCopy* retorna todas as cópias cuja variável origem é igual a variável passada como parâmetro. Potenciais definições de variáveis também matam as cópias que por acaso as usem ou definam. Tal estratégia é conservativa, na medida em que um número menor de cópias acaba sendo considerado como vivo em um determinado ponto do programa, evitando que uma substituição indevida possa ocorrer.

Para descobrir quais cópias estão vivas na entrada de cada bloco básico, o Xingó resolve o seguinte sistema de equações:

$$\begin{aligned} in[B] &= \cap_{p \in predB}\ out[P] \\ out[B] &= c\_gen[B] \cup (in[B] - c\_kill[B]) \end{aligned}$$

Uma vez determinados os conjuntos $in[B]$, o Xingó está pronto para realizar as devidas alterações. O pseudo-código a seguir mostra o algoritmo utilizado pelo Xingó para realizar *copy propagation*.

```
para cada bloco básico B faça
    c_gen ← ∅
    c_kill ← ∅
    para cada instrução i ∈ B faça
        se i é uma instrução de cópia então
            g2 ← CopyId(i)
        fim
        se i é uma definição então
            v ← variável definida por i
            k2 ← VarUsedCopy(v)
            para cada variável vp que i potencialmente possa definir faça
                k2 ← k2 ∪ VarUsedCopy(vp)
            fim
            c_gen[B] ← c_gen[B] ∪ g2
            c_kill[B] ← c_kill[B] − g2
            c_gen[B] ← c_gen[B] − k2
            c_kill[B] ← c_kill[B] ∪ k2
        fim
    fim
fim
```

Figura 4.14: Inicialização dos conjuntos *c_gen* e *c_kill*

***Algoritmo 4.6:*** Copy Propagation

```
para cada bloco básico B faça
    availCopy ← in[B]
    para cada instrução i ∈ B faça
        para cada cópia c ∈ availCopy faça
            se a variável v definida em c é utilizada em i então
                substitua em i a variável v pela variável usada em c
            fim
        fim
        v ← variável definida por i
        availCopy ← availCopy - VarUsedCopy(v)
        para cada variável vp que i potencialmente possa definir faça
            availCopy ← availCopy - VarUsedCopy(vp)
        fim
        se i é uma instrução de cópia então
            availCopy ← availCopy ∪ CopyId(i)
            u ← variável usada em i
```

fazer VarUsedCopy reconhecer que $u$ é utilizada por $i$
**fim**
**fim**
**fim** ■

Para cada bloco básico do CFG o algoritmo obtém as cópias que estão vivas na entrada do mesmo e depois percorre todas as instruções do bloco. Para cada instrução ele verifica se algum de seus operandos pode ser substituído por alguma variável que seja o operando fonte de alguma cópia que esteja viva no conjunto *availCopy*. Em caso afirmativo, a devida substituição é realizada.

Depois que todas as tentativas de substituição foram realizadas, o algoritmo parte agora para a atualização do conjunto *availCopy*.

O algoritmo descobre quais as variáveis que a instrução $i$ define ou pode definir e remove do conjunto *availCopy* todas as cópias que envolvam tais variáveis (usando ou definindo). A seguir o algoritmo verifica se a instrução $i$ é uma cópia. Em caso afirmativo, acrescenta $i$ ao conjunto *availCopy* e atualiza o conjunto de informações consultadas por *VarUsedCopy*, de forma que conste que $i$ utiliza a variável $u$. Esse procedimento é necessário, pois $i$ pode ter sido modificada durante as tentativas de se realizar *copy propagation*.

### 4.2.5 Global Common Subexpression Elimination

Esta transformação de código tem como objetivo aumentar a velocidade de execução do programa, pois tenta eliminar a repetição de determinados tipos de computação reaproveitando resultados previamente calculados. Embora possa aumentar a velocidade de execução de um programa, essa transformação pode dificultar a alocação de registradores, pois ela pode fazer com que algumas variáveis tenham uma *live range* muito longa, isto é, permaneçam vivas em um caminho muito longo no programa.

A Figura 4.15 apresenta um CFG que é a motivação para esta transformação. Em (a) o bloco básico B1 realiza a computação da expressão $4 * i$ duas vezes, sendo que esta computação é novamente realizada no bloco básico B2. No caminho em que $4 * i$ é avaliada em B1 até B2, a variável $i$ não é definida nenhuma vez, desta forma $4 * i$ sempre irá produzir o mesmo resultado e é desta forma denominada *common subexpression*.

A Figura 4.15 (b) mostra como eliminar as diversas computações de $4*i$. Inicialmente a primeira ocorrência da expressão é avaliada e atribuída a uma nova variável. A partir deste ponto, cada nova avaliação da expressão que é alcançada pela nova variável é substituída por um uso da mesma. Em (b), introduziu-se em B1 uma nova instrução $t1 = 4 * i$, e $u = 4 * i$ foi substituída por $u = t1$, o mesmo ocorrendo com $j = 4 * i$ que foi substituída

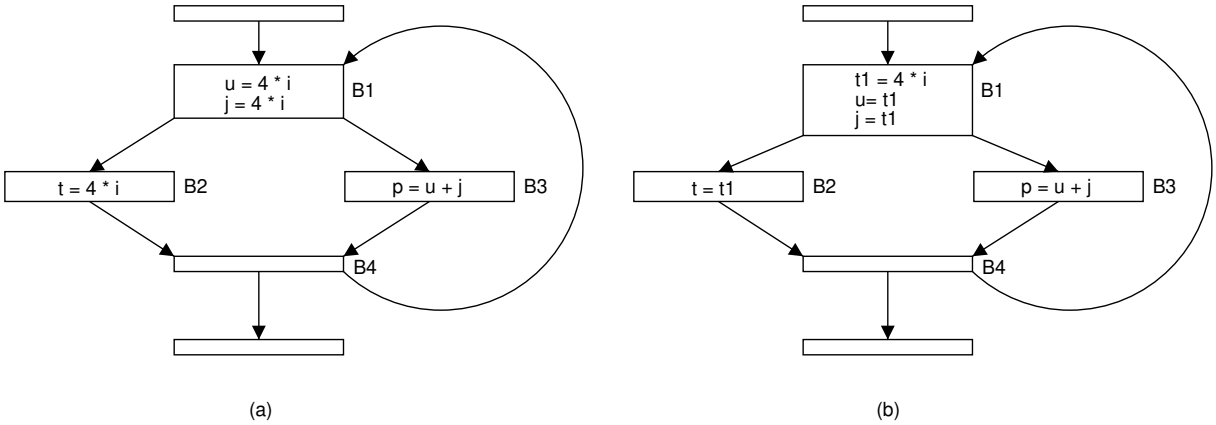


Figura 4.15: Exemplo de *Common Subexpression Elimination*

por $j = t1$. Como a expressão $4 * i$ também estava disponível em $t = 4 * i$ no bloco B2, esta instrução também foi transformada, tornando-se $t = t1$.

Para realizar tais transformações em um CFG, o Xingó divide esta otimização em duas fases, uma local e outra global. A fase local elimina as subexpressões comuns que ocorrem em um mesmo bloco básico e a fase global considera todos os blocos básicos do CFG no momento de eliminar tais expressões. Essa divisão é feita para que a fase global encontre apenas uma ocorrência da expressão a ser otimizada disponível em cada bloco básico, quando tiver que percorrer os predecessores de um dado bloco.

O pseudo-código a seguir mostra o algoritmo empregado pelo Xingó para realizar a fase local da eliminação das subexpressões. A função *AvailExps* retorna quais são as expressões que estão disponíveis na instrução passada como parâmetro, enquanto que a função *ExpressionId* retorna o número com que uma expressão está associada. O algoritmo percorre todas as instruções até encontrar uma expressão. Se a expressão estiver em *exps*, ele verifica se existe alguma variável $t$ associada com a expressão $e$ encontrada. Em caso afirmativo, ele substitui $e$ por $t$, significando que houve uma substituição em algum ponto anterior do bloco; se não houver variável associada a $e$, ele cria uma nova variável $t$, substitui $e$ por $t$ e associa ambos.

***Algoritmo 4.7:*** LocalCSE
*Entrada*:
*Available Expressions Analysis*
**para** cada bloco básico $B$ **faça**
exps $\leftarrow \varnothing$
**para** cada instrução $i \in B$ **faça**
exps $\leftarrow$ exps $\cap$ AvailExps($i$)
**se** $i$ é uma expressão **então**

```
            se a expressão e em i ∈ exps então
                se existe variável t associada a e então
                    substitua a expressão e por t na instrução i
                senão
                    crie uma nova variável t
                    faça t receber e
                    substitua a expressão e por t na instrução i
                    associe t com a expressão e
                fim
            senão
                exps ← exps ∪ ExpressionId(i)
            fim
        fim
    fim
```

■

***Algoritmo 4.8:*** GlobalCSE

```
Entrada:
    Available Expressions Analysis
para cada bloco básico B faça
    exps ← in[B]
    para cada instrução i ∈ B faça
        exps ← exps ∩ AvailExps(i)
        se i é uma expressão então
            se a expressão e em i ∈ exps e i ∈ in[B]então
                encontre todas as expressões e vindas dos predecessores de B
                crie uma nova variável t
                faça t receber e antes de cada expressão encontrada
                substitua e por t em cada expressão encontrada
                substitua e por t na instrução i
            fim
        fim
    fim
fim
```

■

Se por acaso $e$ não pertence a *exps*, então o algoritmo simplesmente adiciona tal expressão a esse conjunto. O algoritmo GlobalCSE mostra o pseudo-código empregado pelo

Xingó para realizar a eliminação das subexpressões de forma global. Para cada bloco básico do programa, o algoritmo verifica quais as expressões que estão vivas na entrada do bloco e percorre todas as instruções até encontrar uma expressão que esteja disponível na entrada do bloco básico. Caso tal expressão seja encontrada, ele percorre todos os predecessores do bloco básico até encontrar a expressão (ou expressões) equivalente, criando então uma nova variável $t$ e atribui-lhe $e$, a seguir substitui $e$ por $t$ nas expressões encontradas.

### 4.2.6 PeepHole Optimization

Este tipo de transformação de código se caracteriza por analisar pequenos trechos de código e realizar mudanças pequenas e simples que em geral podem acabar melhorando o desempenho do programa ou criar oportunidades para que transformações mais complexas sejam realizadas.

No Xingó este tipo de transformação tem a função de eliminar variáveis que não são mais empregadas e tentar fazer com que uma variável que é usada somente uma vez ou poucas vezes deixe de ser utilizada.

Para eliminar as variáveis que não são mais utilizadas, o Xingó simplesmente verifica todas as variáveis relativas a um CFG e verifica quais delas não são usadas nem definidas por nenhuma instrução. Esta verificação é feita com as funções *VarDefUses*, *VarPotUses*, *VarDefDefs* e *VarPotDefs* as quais estão implementadas na classe *CFGraphInfo*.

A função *VarDefUses* retorna quais as instruções que utilizam de fato uma determinada variável, enquanto que a função *VarPotUses* retorna quais as instruções podem potencialmente utilizar a variável, isto é, por meio de uma referência indireta através de um ponteiro.

A função *VarDefDefs* retorna quais as instruções que definem de fato a variável (definição não-ambígua), enquanto que a função *VarPotDefs* retorna quais as instruções que podem pontencialmente definir a variável, isto é, por meio de uma referência indireta através de um ponteiro.

Se a união dos conjuntos retornados pelas quatro funções for vazia, isto significa que a variável não é utilizada nem definida em nenhuma instrução e pode portanto ser eliminada do CFG. Somente variáveis locais e temporárias são eliminadas, variáveis globais e parâmetros não são considerados.

A outra transformação considerada se refere a tentar fazer com que uma variável temporária fique sem uso de modo que possa ser eliminada. A Figura 4.16 mostra a primeira situação considerada e a transformação realizada.

O Xingó percorre o CFG procurando seqüências de instruções como a apresentada. Quando uma seqüência como esta é encontrada, ele verifica se a variável $t$ é um temporário.

$$
\begin{array}{l}
t \leftarrow b \oplus c \\
a \leftarrow t \\
\quad \Downarrow \\
a \leftarrow b \oplus c \\
t \leftarrow t
\end{array}
$$

Figura 4.16: Eliminação do uso de temporários

Em caso afirmativo ele verifica se a definição de $t$ só atinge um único uso, o qual está na instrução seguinte. Se essa condição for verdadeira, então ele realiza a transformação apresentada na Figura 4.16, criando assim uma instrução de cópia que é código morto e será eliminada por *dead code elimination*. Se a variável $t$ não ocorrer em nenhum outro ponto do programa, ela será eliminada na próxima vez em que *peephole* for executada.

Caso a definição da variável $t$ alcance mais de um uso, o Xingó realiza outra transformação de código, a qual é mostrada na Figura 4.17.

$$
\begin{array}{l}
t \leftarrow b \oplus c \\
a \leftarrow t \\
\quad \Downarrow \\
a \leftarrow b \oplus c \\
t \leftarrow a
\end{array}
$$

Figura 4.17: Alteração no uso de temporários

Embora essa transformação não faça com que a instrução de cópia se torne código morto de imediato, isso pode vir a ocorrer quando for realizado *copy propagation* e os usos de $t$ possam possivelmente ser substituídos por usos de $a$. No caso de $a$ também ser um temporário essa transformação não é realizada.

### 4.2.7 Code Motion

Esta transformação de código tem como objetivo aumentar o desempenho do programa, pois ela verifica todos os *loops* existentes e move para fora dos mesmos todas as computações que produzem sempre o mesmo resultado em cada iteração. Esse tipo de computação recebe a denominação de cômputo laço-invariante. A Figura 4.18 mostra um exemplo onde uma computação laço-invariante é detectada em um *loop* e movida para fora do mesmo.

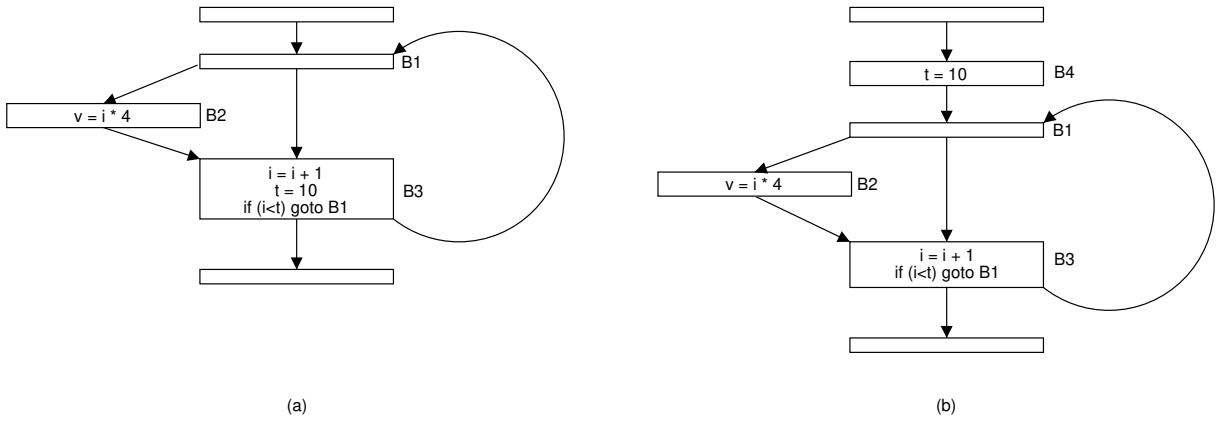


Figura 4.18: Exemplo de *Code Motion*

Na Figura 4.18 (a) pode-se observar no bloco básico B3 que a instrução $t = 10$ é laço invariante, pois em cada iteração do laço a variável $t$ sempre recebe o mesmo valor, isto é, 10. Desta forma ela pode ser movida para fora do mesmo. A Figura 4.18 (b) mostra o CFG depois da transformação realizada. Um novo bloco básico (B4) é criado para receber todas as computações laço-invariante que forem movidas para fora do *loop* e recebe a denominação de *Pre-Header*. O *Pre-Header* deve ser inserido no CFG antes do bloco de entrada do *loop*.

Para determinar quais são as instruções de um *loop* que são laço-invariante, o Xingó utiliza o algoritmo apresentado a seguir:

***Algoritmo 4.9:*** Detecção de Cômputos Laço-Invariante
*Entrada*: Um laço $L$ consistindo em um conjunto de blocos básicos.
*Saída*: As instruções em $L$ que são laço-invariantes.

Método:

1. Marque como "invariante" todas as instruções cujos operandos são todos constantes ou possuem todas as definições incidentes fora de $L$.
2. Repita o passo (3) até que nenhuma instrução seja marcada como laço-invariante.
3. Marque como "invariante" todas as instruções não previamente marcadas, cujos operandos são todos constantes, possuem todas as definições incidentes fora de $L$, ou que tenham exatamente uma definição incidente, e esta definição esteja marcada como laço-invariante.

■

Uma vez que as computações laço-invariante tenham sido identificadas, o Xingó está pronto para movê-las para o *Pre-Header* do *loop*. Para que uma computação laço-invariante possa ser movida para fora do *loop* ela tem que atender a quatro quesitos:

1. O bloco básico contendo a sentença laço-invariante deve dominar todos os nós de saída do *loop*, onde um nó de saída é aquele que possui um sucessor que não pertence ao *loop*.

2. Se uma sentença laço-invariante $i$ define uma variável $v$, não deve existir no *loop* outra instrução que também defina $v$.

3. Se uma sentença laço-invariante $i$ define uma variável $v$, então todos os usos de $v$ no *loop* somente podem ser alcançados por $i$.

4. Se uma sentença laço-invariante $i$ define uma variável $v$, então o bloco básico contendo $i$ deve dominar todos os blocos básicos do *loop* que contêm instruções fazendo uso de $v$.

Para verificar cada uma das quatro condições, o Xingó utiliza basicamente informação sobre dominadores e informação de *reaching definitions*. Uma vez que uma computação laço-invariante tenha passado em todos os quesitos, o Xingó a move para o *Pre-Header* do *loop*.

### 4.2.8 Strength Reduction

Em muitas arquiteturas a multiplicação é uma operação mais cara do que a adição, por isso é mais vantajoso substituir multiplicações por adições sempre que possível. Muitos *loops* são indexados por variáveis, as quais são utilizadas para controlar a quantidade de vezes que o mesmo é executado, e não raro tais variáveis, chamadas de variáveis de indução, são usadas em alguma computação interna ao *loop* sob a forma de uma multiplicação. Como a variável de indução é em geral incrementada ou decrementada por uma constante, seus valores formam uma progressão aritmética e desta forma pode ser possível transformar as multiplicações, que utilizam a variável de indução do *loop* como um operando, em operações de adição. Tal transformação é chamada de *strength reduction*.

As variáveis de indução são divididas em dois conjuntos. O primeiro conjunto compreende as variáveis que são incrementadas ou decrementadas por uma constante. Uma variável que pertença a este conjunto recebe a denominação de variável de indução básica. O segundo conjunto compreende as variáveis definidas no *loop*, cujo valor é uma função linear de uma variável de indução básica. Elas são chamadas de variáveis de indução.

O primeiro passo para se realizar *strength reduction* é detectar as variáveis de indução que estão presentes em um *loop*. O pseudo-código a seguir mostra o algoritmo básico utilizado pelo Xingó para encontrá-las:

***Algoritmo 4.10:*** Detecção de Variáveis de Indução
*Entrada*: Um laço $L$ consistindo em um conjunto de blocos básicos
*Saída*: Um conjunto de variáveis $j$ e uma tripla da forma ($i$,$d$,$c$) associada a cada variável de indução, onde $i$ é uma variável de indução básica e $c$ e $d$ são constantes, tal que o valor de $j$ é dado por $j \leftarrow d * i + c$.

```
para cada bloco básico B em L faça
    para cada instrução i em B faça
        se i é da forma a ← a ± c, onde c é constante, então
            crie a tripla (a,1,0) e a associe a a
            marque a como variável de indução básica
        fim
    fim
fim
para cada bloco básico B em L faça
    para cada instrução i em B faça
        se i é da forma k ← b * j ± e, onde b e e são constantes então
            se j é uma variável de indução básica então
                crie a tripla (j,b,e) e a associe a k
            senão
                sendo (a,d,c) a tripla associada a j, então:
                    (1) não deve haver definição de a entre a última
                        definição de j e a definição de k.
                    (2) nenhuma definição de j fora de L atinge k.
                se as condições (1) e (2) forem verdadeiras então
                    crie a tripla (a,b * d,b * c + e) e a associe a k.
                fim
            fim
        fim
    fim
fim
```

■

O algoritmo procura em primeiro lugar as variáveis de indução básicas e associa a cada uma delas a tripla ($a$,1,0), onde $a$ é a variável de indução básica. Uma vez que todas as variáveis de indução básicas tenham sido encontradas, o algoritmo vai agora procurar

as variáveis que são uma função linear de alguma outra variável de indução. As triplas associadas a cada uma das variáveis de indução são utilizadas para se saber quais são os coeficientes da função linear que dão o valor de uma variável de indução em função de uma variável de indução básica. Desta forma cada variável de indução está associada a uma variável de indução básica. Tal tripla será utilizada para tornar as variáveis de indução independentes das variáveis de indução básicas.

Uma vez que todas as variáveis de indução tenham sido encontradas, o Xingó parte agora para a transformação do código. O pseudo-código a seguir mostra o algoritmo do Xingó para realizar *strength reduction*.

***Algoritmo 4.11:*** Strength Reduction

*Entrada*: Um laço $L$ consistindo em um conjunto de blocos básicos
e as variáveis de indução
**para** cada variável de indução não-básica $j$ **faça**
    crie uma nova variável $s$
    associe $s$ a $j$
    substitua cada definição de $j$ em $L$ por $j \leftarrow s$
**fim**
**se** $L$ não tiver *Pre-Header* **então**
    crie um *Pre-Header* para $L$ e insira-o no CFG
**fim**
**para** cada variável de indução básica $a$ **faça**
    depois de cada definição de $a$ da forma $a \leftarrow a \pm n$ em $L$ **faça**
        **para** cada tripla da forma ($a$,$d$,$c$) que não esteja associada a $a$ **faça**
            $j \leftarrow$ variável associada a tripla
            $s \leftarrow$ variável associada a $j$
            crie uma nova instrução da forma $s \leftarrow s + d * n$
            adicione ao *Pre-Header* de $L$ a instrução $s \leftarrow d * i + c$
        **fim**
    **fim**
**fim**

■

Em um primeiro momento, o algoritmo cria uma nova variável $s$ para cada variável de indução não básica $j$. Essa nova variável será utilizada para tornar $j$ independente de sua variável de indução básica associada. Cada definição de $j$ no laço $L$ é transformada em uma atribuição da forma $j \leftarrow s$, tornando assim $j$ independente da variável de indução básica a qual esteja associada.

Em um segundo momento, o algoritmo verifica todas as variáveis de indução básicas $a$ e insere depois de cada definição da mesma instruções da forma $s \leftarrow s + d * n$, para cada variável de indução $j$ não-básica que tenha o seu valor dependente de $a$. Também para cada variável $s$ associada a uma variável $j$, o algoritmo insere a inicialização do valor de $s$ antes do início da execução do *loop*. A inicialização de $s$ é uma instrução da forma $s \leftarrow s + d * n$, a qual é adicionada ao fim do *Pre-Header* de $L$.

Com isso, o único uso de cada variável de indução básica será provavelmente apenas controlar o número de vezes que $L$ será executado.

# Capítulo 5

# Resultados Experimentais

No momento em que esta dissertação foi escrita (março de 2004), o Xingó possuía um total de oito otimizações de código implementadas e a geração do código C a partir da XIR estava funcionando para uma quantidade razoável de programas.

Os testes de geração do código C foram feitos utilizando-se basicamente três *benchmarks*: Mediabench, SPEC e NullStone.

O *benchmark* NullStone (www.nullstone.com) realiza uma série de testes com o objetivo de avaliar o desempenho e a corretude do compilador. O NullStone é reconhecido como o melhor *benchmark* de compiladores de produção do mundo e foi adquirido com o apoio do projeto FAPESP 00/15083-9. A geração de código C foi bem sucedida em 6581 de um total de 6611 testes realizados pelo NullStone. Nestes testes nenhuma otimização foi aplicada aos programas de entrada. Com a aplicação de otimização, a geração de código C foi bem sucedida em 6497 de um total de 6611 testes realizados. Por geração de código bem sucedida, entende-se a correta compilação e execução do código C gerado pelo Xingó.

Os *benchmarks* Mediabench e SPEC foram utilizados em conjunto, com o objetivo de avaliar a corretude do código C gerado pelo Xingó. No presente momento a geração de código C foi bem sucedida para os programas adpcm, epic, g721, bzip2, gzip, mcf, twolf. Tais programas foram testados sem a aplicação de otimizações. No presente momento esses programas ainda não foram avaliados com a aplicação de otimizações. Outros programas pertencentes a estes dois *benchmarks* ainda estão sendo avaliados. Um problema enfrentado na utilização do Mediabench e do SPEC é que nem todos os programas são aceitos pelo LCC, *front-end* do Xingó. Uma vez que um programa não passe pelo LCC, o Xingó não tem como compilá-lo.

O compilador Xingó está em constante desenvolvimento, pois tem como objetivo de longo prazo alcançar um desempenho igual ou superior ao compilador GCC, no que se refere a otimização de código, bem como no número de programas aceitos. Para isto, as otimizações atualmente implementadas estão sendo comparadas com as equivalentes no

GCC, para a realização dos devidos ajustes; também estão sendo introduzidas modificações no LCC de modo que ele reconheça como corretos uma gama maior de programas de entrada.

Visando melhor expor os experimentos realizados, apresenta-se a seguir todos os passos da otimização e geração do código C pelo Xingó para um programa exemplo. O programa exemplo é apresentado na Figura 5.1. Tal programa corresponde ao código fonte do algoritmo de ordenação QuickSort. O mesmo foi escolhido por ser pequeno e por permitir demonstrar as diversas otimizações do Xingó.

Depois que o arquivo é pré-processado e analisado pelo LCC, ele é convertido para a XIR. Após a conversão o Xingó constrói o CFG para cada função do programa, tornando possível a geração de código C.

A Figura 5.2 mostra o código C gerado pelo Xingó para a função *partition*, que faz parte do QuickSort. Essa função será utilizada para ilustrar todo o processo de otimização do Xingó, pois contém uma diversidade maior de estruturas em relação às demais funções. A Figura 5.2 mostra o código gerado sem a aplicação de nenhuma otimização. Este código apresenta uma grande quantidade de variáveis temporárias e uma grande quantidade de instruções, as quais empregam diversos passos para realizar computações simples. A falta de qualidade do programa da Figura 5.2 é resultante da conversão dos DAGs do LCC para a XIR.

O processo de otimização da função *partition* ocorreu em 18 etapas, que são apresentadas a seguir. As diversas transformações realizadas pelas otimizações podem ser visualizadas nas Figuras 5.3 até 5.20, permitindo assim a inspeção de cada alteração realizada no código C.

Os programas das Figuras 5.2 e 5.20 foram compilados utilizando-se o GCC e os seus tempos foram comparados de modo a se observar os efeitos da aplicação das otimizações. O programa da Figura 5.2 levou aproximadamente 30 segundos para executar 10000000 de vezes em um computador Pentium Celeron 1.2 GHz com 256MB de memória RAM. O programa da Figura 5.20 levou aproximadamente 17 segundos para executar 10000000 de vezes no mesmo computador. A aplicação das otimizações resultou em um *speep-up* de aproximadamente 43% no tempo de execução do programa.

As seções seguintes apresentam agora cada etapa empregada para otimizar o programa exemplo.

## 5.1 Etapa 1: Pointer Optimization

A primeira otimização realizada pelo Xingó é *pointer optimization*. Esta otimização tentará eliminar referências indiretas à variáveis. A Figura 5.3 mostra o código C gerado pelo Xingó após esta otimização. As instruções marcadas com o sinal ▶ foram alteradas

em relação à Figura 5.2.

A primeira instrução do bloco básico B1 da Figura 5.2, é $\_t2 = \&a$. Essa instrução faz com que $\_t2$ aponte somente para $a$, não podendo apontar para nenhuma outra posição de memória. A variável $\_t2$ é utilizada na instrução seguinte: $\_t3 = *((int**)\_t2)$. Nesta instrução o valor de $a$ está sendo lido indiretamente através da variável $\_t2$, mas $\_t2$ aponta somente para $a$, então tal instrução é transformada em $\_t3 = ((void*)a)$, como pode ser visto no bloco básico B1 da Figura 5.3.

Situação semelhante ocorre para as demais instruções da Figura 5.2 que lidam com ponteiros. Assim a instrução $\_t10 = *((int**)\_t9)$ transformou-se em $\_t10 = ((void*)a)$, pois $\_t9$ também aponta somente para $a$. Todas as alterações podem ser observadas na Figura 5.3.

## 5.2 Etapa 2: Copy Propagation

A próxima otimização aplicada pelo Xingó é *copy propagation*, que tem o seu resultado mostrado na Figura 5.4. Novamente o sinal ▶ indica quais instruções foram alteradas em relação a figura anterior (5.3).

Esta otimização de código vai localizar todas as instruções de cópia e tentar fazer com que a variável fonte seja utilizada no lugar da variável destino nas instruções subseqüentes.

No bloco básico B1 da Figura 5.3, a instrução de cópia $\_t3 = ((void*)a)$ foi encontrada, fazendo com que a instrução $a = ((void*)\_t3)$ se transformasse em $a = ((void*)a)$ na Figura 5.4. Ainda no bloco básico B1 da Figura 5.3, a instrução de cópia $\_t10 = ((void*)a)$ fez com que a instrução $\_t11 = \_t8 + ((void*)\_t10)$ se transformasse em $\_t11 = ((void*)a) + \_t8$ na Figura 5.4.

A instrução de cópia $i = \_t27$ presente no bloco básico B4 da Figura 5.3 fez com que a variável $\_t27$ fosse propagada para diversos blocos básicos, afetando instruções nos blocos B5, B8 e B12 da Figura 5.4, a qual apresenta todas as alterações causadas por *copy propagation*.

## 5.3 Etapa 3: Constant Propagation

Esta otimização vai tentar substituir usos de variáveis por constantes sempre que possível. O primeiro passo é localizar as instruções de cópia onde o operando fonte é uma constante. No bloco básico B1 da Figura 5.4 a instrução $\_t7 = 2$ é a primeira localizada, fazendo com que o uso de $\_t7$ em $\_t8 = p << \_t7$ seja substituído pela constante 2, transformando essa instrução em $\_t8 = p << 2$, como pode ser visto na Figura 5.5.

Essa otimização é bem simples e seus efeitos se restringiram a nível de bloco básico para o exemplo apresentado. Todas as alterações podem ser observadas na Figura 5.5.

## 5.4 Etapa 4: Dead Code Elimination

Depois da realização de *pointer optimization*, *copy propagation* e *constant propagation* é bem provável que algumas instruções se tornem sem efeito, ou seja, realizam computações que não serão utilizadas em nenhum ponto do programa e portanto podem ser eliminadas. A transformação responsável por eliminar esse tipo de computação é a otimização *dead code elimination*.

A Figura 5.6 mostra o código C após a realização de *dead code elimination*. Observa-se que diversas instruções presentes na Figura 5.5 foram eliminadas. O bloco básico B1 teve o seu número de instruções reduzido de 21 para 9, e o bloco básico B12 de 33 para 13; ou seja, cada um desses blocos teve mais da metade de suas instruções removidas. O programa como um todo passou de 93 para 44 instruções, uma otimização de espaço maior que 50%

Todas as definições de variáveis que não alcançavam nenhum uso, mesmo que potencial, foram eliminadas por esta transformação de código. A instrução $\_t2 = \&a$ presente no bloco básico B1 da Figura 5.5 ilustra esse fato. Nenhuma instrução do programa utiliza a variável $\_t2$ como um operando, bem como não existe nenhum ponteiro que possa apontar para $\_t2$ em algum ponto do programa, desta forma essa instrução está computando um valor que nunca será utilizado e foi removida na Figura 5.6.

## 5.5 Etapa 5: Common Subexpression Elimination

Esta etapa do processo de otimização procura localizar e eliminar as subexpressões comuns que por ventura possam existir no programa. Como dito no Capítulo 4, esta transformação de código é realizada em duas fases, uma local a nível de bloco básico e outra global em todo o CFG.

A Figura 5.6 apresenta as subexpressões comuns $\_t42 << 2$ e $\_t27 << 2$. A fase local vai otimizar primeiro o bloco básico B12, que é o único que contém mais de uma subexpressão disponível internamente. As instruções $\_t66 = \_t27 << 2$ e $\_t85 = \_t27 << 2$ são subexpressões comuns e a variável $\_t27$ não é definida entre elas. A otimização criou então uma nova variável, $\_t5$ e criou a instrução $\_t5 = \_t27 << 2$ e transformou $\_t66 = \_t27 << 2$ em $\_t66 = \_t5$, e $\_t85 = \_t5$. Essas alterações podem ser visualizadas na Figura 5.7.

Ainda no bloco básico B12 da Figura 5.6, as instruções $\_t76 = \_t42 << 2$ e $\_t94 =$

$\_t42 << 2$ também são subexpressões comuns e $\_t42$ não é definida entre a avaliação das mesmas. A fase local de *common subexpression elimination* criou uma nova variável $\_t13$ e a instrução $\_t13 = \_t42 << 2$, substituindo $\_t76 = \_t42 << 2$ por $\_t76 = \_t13$, e $\_t94 = \_t42 << 2$ por $\_t94 = \_t13$.

A fase global vai descobrir que a nova instrução $\_t13 = \_t42 << 2$ no bloco básico B12 e a instrução $\_t47 = \_t42 << 2$ no bloco básico B7 são subexpressões comuns e $\_t42$ não é definida entre ambas. Desta forma uma nova variável $\_t14$ é criada. A instrução $\_t14 = \_t42 << 2$ é criada e inserida em B7, a instrução $\_t47 = \_t42 << 2$ é transformada em $\_t47 = \_t14$ e a instrução $\_t13 = \_t42 << 2$ em B12 é transformada em $\_t13 = \_t14$. A Figura 5.7 mostra o código C gerado depois que as fases local e global de *common subexpression elimination* são realizadas no programa.

## 5.6 Etapa 6: PeepHole Optimization

Esta otimização basicamente tenta eliminar variáveis que não são utilizadas e tenta fazer com que outras variáveis fiquem sem uso. A Figura 5.8 mostra o código C depois da realização desta otimização. A diminuição na quantidade de variáveis locais é evidente. O total passou de 78 (Figura 5.7) para 43 (Figura 5.8) variáveis locais.

As variáveis removidas não estavam presentes em nenhuma instrução, seja como operando de alguma operação, seja como variável destino de alguma atribuição. Referências por meio de ponteiros também devem ser consideradas, e no caso destas variáveis, nenhum ponteiro apontava ou podia apontar para as mesmas.

Em um segundo momento, *peephole optimization* vai tentar fazer com que outras variáveis fiquem sem uso para posteriormente poderem ser removidas. Para isso um grupo de duas instruções é analisado por vez. No bloco básico B1 da Figura 5.7 as instruções $\_t17 = p - 1$ e $i = \_t17$ foram selecionadas para otimização. Pelo fato da variável $\_t17$ só ser usada em uma única instrução e não ter usos potenciais, *peephole optimization* transformou estas duas instruções em $i = p - 1$ e $\_t17 = \_t17$ respectivamente, como mostra a Figura 5.8. Transformação semelhante ocorre com as instruções $\_t22 = r + 1$ e $j = \_t22$ no bloco básico B1. Essa transformação além de fazer com que uma variável fique sem outros usos no programa, cria uma instrução que é código morto, o qual será removido por *dead code elimination* posteriormente.

No bloco básico B4 da Figura 5.7 as instruções $\_t27 = i+1$ e $i = \_t27$ foram selecionadas para otimização. Ao contrário do caso anterior, a variável $\_t27$ possui diversos usos ao longo do programa (blocos B5, B8 e B12), desta forma uma transformação diferente é realizada, transformando estas instruções em $i = i+1$ e $\_t27 = i$ respectivamente. Embora $\_t27$ ainda contenha diversos usos ao longo do programa, a realização de *copy propagation* pode substituir os mesmos pela variável $i$ tornando $\_t27$ candidata a eliminação. Uma

transformação semelhante a esta é realizada no bloco básico B6 com as instruções $\_t42 = j - 1$ e $j = \_t42$.

## 5.7 Etapa 7: Dead Code Elimination

A próxima otimização aplicada pelo Xingó é *code motion*, que não realizou nenhuma modificação no código e por isso o código C resultante é o mesmo que o anterior. A otimização seguinte é *dead code elimination*, que novamente vai eliminar as instruções que realizam computações que não alcançam nenhum uso definido ou potencial.

O programa otimizado é apresentado na Figura 5.9, onde somente duas instruções ($\_t17 = \_t17$ e _t22=_t22) foram removidas do bloco básico B1 e o restante do programa permaneceu inalterado.

## 5.8 Etapa 8: Copy Propagation

A próxima otimização aplicada pelo Xingó é *strength reduction*, que não realizou nenhuma modificação nas instruções do programa, mas inseriu um *Pre-Header* para cada laço do mesmo. O código C para esta otimização não é mostrado, pois a inserção de cada *Pre-Header* pode ser visualizada na Figura 5.10.

Depois que *strength reduction* é aplicada, toda a seqüência de otimizações foi realizada. O Xingó verifica se alguma delas modificou o programa e em caso afirmativo todo o ciclo é reiniciado. Neste ponto, com exceção de *code motion*, todas as outras otimizações realizaram transformações no programa, por isso todo o ciclo é reiniciado.

A primeira otimização é novamente *pointer optimization*, mas ela não realiza nenhuma alteração no código do programa, por isso o código C após a sua realização também não será mostrado. A próxima otimização é *copy propagation* e o resultado de sua aplicação é mostrado na Figura 5.10.

A instrução de cópia $\_t27 = i$ presente no bloco básico B4 na Figura 5.9 (bloco básico B6 na Figura 5.10), faz com que os usos de $\_t27$ ao longo do programa, que são alcançados pela instrução de cópia, sejam substituídos por usos de $i$, o que possivelmente tornará a variável $\_t27$ sem nenhum uso, fazendo com que a mesma seja candidata a eliminação por *peephole optimization*, justificando assim a transformação realizada na etapa 6 do processo de otimização.

A instrução de cópia $\_t3 = ((void*)a)$ no bloco básico B1, fará com que os usos de $\_t3$ alcançados pela cópia, sejam substituídos por usos de $a$ (quando possível), tornado possivelmente $\_t3$ também candidata a eliminação por *peephole optimization*.

Todas as alterações realizadas por *copy propagation* podem ser visualizadas na Figura 5.10.

## 5.9 Etapa 9: Dead Code Elimination

A próxima otimização aplicada pelo Xingó é *constant propagation*, que não causa nenhuma alteração no código, e por isso não é mostrado o código C depois de sua aplicação. O Xingó parte então para a realização da próxima otimização, que é *dead code elimination*.

Nesta nova execução, serão removidas instruções dos blocos básicos B1, B6, B9, B10 e B15 que se tornaram sem efeito depois da realização de *copy propagation*. O programa resultante é apresentado na Figura 5.11.

## 5.10 Etapa 10: Common Subexpression Elimination

A aplicação das outras otimizações fez com que novas subexpressões comuns aparecessem, permitindo assim a aplicação desta otimização.

A fase local vai agir sobre o bloco básico B15. Em um primeiro momento as instruções $\_t69 = ((void*)a) + \_t5$ e $\_t88 = ((void*)a) + \_t5$ da Figura 5.11 são selecionadas para otimização. Uma nova variável chamada $\_t9$ é criada e a instrução $\_t9 = ((void*)a) + \_t5)$ é adicionada ao bloco básico. A instrução $\_t69 = ((void*)a) + \_t5$ se torna $\_t69 = ((void*)\_t9)$ e a instrução $\_t88 = ((void*)a) + \_t5$ se torna $\_t88 = ((void*)\_t9)$. As alterações podem ser visualizadas na Figura 5.12.

Transformação semelhante ocorre com as instruções $\_t79 = ((void*)a) + \_t14$ e $\_t97 = ((void*)a) + \_t14$. Depois da fase local, a fase global é executada e neste exemplo em particular a subexpressão $((void*)a) + \_t14)$ presente no bloco B10 não foi otimizada com a mesma expressão do bloco B15, em virtude de alterações na informação de *data-flow analysis*, causadas pelas otimizações da fase local.

## 5.11 Etapa 11: PeepHole Optimization

Esta nova execução de *peephole optimization* simplesmente irá remover as variáveis que se tornaram sem uso devido a aplicação de outras otimizações. Nenhuma instrução é alterada neste ponto do processo de otimização.

A Figura 5.13 mostra o código C gerado após a aplicação desta otimização. O total de variáveis locais passou de 45 (Figura 5.12) para 26.

## 5.12 Etapa 12: Copy Propagation

As demais otimizações em seqüência (*code motion*, *dead code elimination* e *strength reduction*) não causam nenhuma alteração no programa, por isso o código C para as mesmas não será mostrado.

Neste ponto o Xingó verifica se alguma otimização modificou o programa, e pelo fato do programa ter sido alterado, uma nova iteração é realizada, fazendo com que todas as otimizações sejam executadas novamente.

A primeira otimização a ser executada é *pointer optimization*, que novamente não realiza nenhuma alteração no código do programa. A próxima otimização é *copy propagation*.

As únicas alterações causadas por esta otimização se limitam ao bloco básico B15. A instrução de cópia $\_t69 = ((void*)\_t9)$ faz com que a instrução subseqüente $\_t70 = *((int*)\_t69)$ (Figura 5.13) seja transformada em $\_t70 = *((int*)\_t9)$ (Figura 5.14).

No mesmo bloco básico, a instrução de cópia $\_t79 = ((void*)\_t10)$ faz com que a instrução subseqüente $\_t80 = *((int*)\_t79)$ (Figura 5.13) seja transformada em $\_t80 = *((int*)\_t10)$ (Figura 5.14).

A Figura 5.14 mostra o programa depois que as duas transformações são realizadas.

## 5.13 Etapa 13: Dead Code Elimination

A próxima otimização após *copy propagation* é *constant propagation*, mas neste ponto do processo de otimização, nenhuma alteração é causada por esta transformação de código.

A otimização seguinte é *dead code elimination*, que novamente irá remover instruções que não causam nenhum efeito no programa. As alterações se restringem ao bloco básico B15. As instruções $\_t69 = ((void*)\_t9)$ e $\_t79 = ((void*)\_t10)$ (Figura 5.14) são as únicas a serem removidas. O programa já otimizado é apresentado na Figura 5.15.

## 5.14 Etapa 14: Common Subexpression Elimination

Esta terceira execução de *common subexpression elimination* vai novamente tentar eliminar as subexpressões comuns. Na última execução de tal otimização, a subexpressão comum $((void*)a) + \_t14$ presente nos blocos básicos B10 e B15 (Figura 5.15) não foi detectada, mas nesta nova tentativa tal expressão é eliminada.

A variável $\_t3$ é criada e a instrução $\_t3 = ((void*)a)+\_t14$ é adicionada ao bloco básico B10. A instrução $\_t50 = ((void*)a)+\_t14$ é transformada em $\_t50 = ((void*)\_t3)$. No bloco básico B15, a instrução $\_t10 = ((void*)a) + \_t14$ é transformada em $\_t10 = ((void*)\_t3)$.

Como nenhuma outra subexpressão comum é detectada no programa, a otimização chega ao seu fim. O código final resultante da transformação é mostrado na Figura 5.16.

## 5.15 Etapa 15: PeepHole Optimization

Novamente a transformação de código *peephole optimization*, vai simplesmente remover variáveis locais que não são mais utilizadas.

A Figura 5.17 mostra o código final. Neste ponto o total de variáveis locais passou de 28 (Figura 5.16) para 20.

## 5.16 Etapa 16: Copy Propagation

As próximas otimizações são na seqüencia *code motion*, *dead code elimination* e *strength reduction*, as quais não causam nenhuma alteração no código do programa.

Novamente mais uma iteração é concluída e o Xingó verifica se ocorreu alguma alteração no código do programa. Como o programa foi alterado nesta iteração do *loop*, toda a seqüência de otimizações é realizada novamente. A primeira é *pointer optimization*, que não realiza nenhuma alteração no programa, desde que foi executada pela primeira vez.

A otimização seguinte é *copy propagation*, que realiza alterações nos blocos B7, B10 e B15. Em B7 a instrução de cópia $\_t32 = \_t2$ faz com que a instrução subseqüente $\_t35 = ((void*)a + \_t32$ (Figura 5.17) se transforme em $\_t35 = ((void*)a + \_t2$. No bloco básico B10 a instrução de cópia $\_t50 = ((void*)\_t3)$ faz com que a instrução subseqüente $\_t51 = *((int*)\_t50)$ (Figura 5.17) se transforme em $\_t51 = *((int*)\_t3)$. No bloco básico B15, a instrução de cópia $\_t5 = \_t2$ faz com que a instrução $\_t9 = ((void*)a) + \_t5$ (Figura 5.17) se transforme em $\_t9 = ((void*)a) + \_t2$. Ainda neste bloco, a instrução de cópia $\_t10 = ((void*)\_t3)$ faz com que as instruções $\_t80 = *((int*)\_t10)$ e $\_t97 = ((void*)\_t10)$ (ambas na Figura 5.17) se transformem em $\_t80 = *((int*)\_t3)$ e $\_t97 = ((void*)\_t3)$ respectivamente.

O código final resultante desta aplicação de *copy propagation* é mostrado na Figura 5.18.

## 5.17 Etapa 17: Dead Code Elimination

Na seqüência a *copy propagation*, a otimização *constant propagation* é aplicada, mas não surte qualquer efeito no programa, desta forma a próxima otimização é invocada.

A transformação aplicada agora é *dead code elimination* que simplesmente vai remover as instruções de cópia dos blocos básicos B7, B10 e B15, pois a última aplicação de *copy propagation* fez com que as mesmas se tornassem instruções desnecessárias.

Desta forma a instrução $\_t32 = \_t2$ é removida de B7, a instrução $\_t50 = ((void*)\_t3)$ é removida de B10 e a instrução $\_t5 = \_t2$ é removida de B15. O código final é apresentado na Figura 5.19.

## 5.18 Etapa 18: PeepHole Optimization

Finalmente, a otimização seguinte a *dead code elimination* é a transformação de código *common subexpression elimination*, que nesta etapa do processo de otimização não causa nenhum efeito no programa.

A próxima otimização a ser aplicada é *peephole optimization*, que novamente elimina variáveis locais que não são mais utilizadas. A Figura 5.20 mostra o código depois da aplicação desta otimização. O total de variáveis passou de 20 para 16.

As otimizações seguintes a esta são executadas, mas nenhuma delas causa qualquer alteração no programa. Como nesta iteração o programa foi alterado, o Xingó executa novamente todas as otimizações, mas nesta nova rodada de execução, nenhuma alteração é realizada no programa e o processo de otimização chega ao seu fim. Desta forma o programa da Figura 5.20 representa o resultado final do processo de otimização iniciado no programa da Figura 5.2.

Este capítulo apresentou todo o processo de otimização, bem como a geração do código C a cada transformação realizada, para uma função *partition* do programa exemplo. O código fonte original é mostrado na Figura 5.1 e a Figura 5.20 mostra o código C depois de aplicadas todas as otimizações. Como citado anteriormente, as otimizações resultaram em um *speed-up* de 43% no tempo de execução do programa.

```
#include<stdio.h>
#include<stdlib.h>

int vetor[ ] = {10, 6, 7, 2, 8, 5, 4, 1, 3, 9};

int partition(int a[ ], int p, int r)
{
        int x = a[p];
        int i = p-1;
        int j = r+1;
        int aux;

        while (1)
        {
                do
                {
                        i++;
                }while (a[i]<x);
                do
                {
                        j--;
                }while (a[j]>x);
                if (i>=j)
                        return j;
                aux = a[i];
                a[i] = a[j];
                a[j] = aux;
        }
}

void quicksort(int a[ ], int p, int r)
{
        int q;
        if (p<r)
        {
                q = partition(a, p, r);
                quicksort(a,p,q);
                quicksort(a,q+1,r);
        }
}

void main()
{
        int i;
        quicksort(vetor, 0, 9);
        for(i=0;i<10;i++)
        {
                printf("%d ",vetor[i]);
        }
        printf("\n");
}
```

Figura 5.1: Programa exemplo: QuickSort

```
int partition(int* a,int p,int r)
{
        int _t30;
        int _t31;
        int _t32;
        int _t25;
        int _t17;
        int _t20;
        void* _t50;
        void* _t49;
        void* _t48;
        int _t47;
        void* _t9;
        void* _t10;
        void* _t11;
        int _t12;
        int _t40;
        int _t41;
        int _t42;
        void* _t33;
        void* _t34;
        void* _t35;
        int _t36;
        void* _t68;
        void* _t67;
        int _t66;
        int _t65;
        int _t26;
        int _t27;
        int _t60;
        int _t59;
        int _t53;
        int _t51;
        int _t46;
        int _t45;
        int _t80;
        void* _t79;
        void* _t78;
        void* _t77;
        int _t38;
        int _t70;
        void* _t69;
        int _t63;
        int _t98;
        void* _t97;
        int _t57;
        int _t55;
        int _t92;
        int _t90;
        int _t84;
        int _t82;
        int _t76;
        int _t75;
        int _t73;
        int _l1;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t2 = &a;
        _t3 = *((int**)_t2);
        _t4 = &a;
        *((int**)_t4) = _t3;
        _t6 = p;
        _t7 = 2;
        _t8 = _t6<<_t7;
        _t9 = &a;
        _t10 = *((int**)_t9);
        _t11 = _t8 + ((void*) _t10 );
        _t12 = *((int*)_t11);
        x = _t12;
        _t15 = p;
        _t16 = 1;
        _t17 = _t15 - _t16;
        i = _t17;
        _t20 = r;
        _t21 = 1;
        _t22 = _t20 + _t21;
        j = _t22;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        _t25 = i;
        _t26 = 1;
        _t27 = _t25 + _t26;
        i = _t27;
        goto B5;
B5:
        _t30 = i;
        _t31 = 2;
        _t32 = _t30<<_t31;
        _t33 = &a;
        _t34 = *((int**)_t33);
        _t35 = _t32 + ((void*) _t34 );
        _t36 = *((int*)_t35);
        _t38 = x;
        if( _t36<_t38 ) goto B4;
B6:
        _t40 = j;
        _t41 = 1;
        _t42 = _t40 - _t41;
        j = _t42;
        goto B7;
B7:
        _t45 = j;
        _t46 = 2;
        _t47 = _t45<<_t46;
        _t48 = &a;
        _t49 = *((int**)_t48);
        _t50 = _t47 + ((void*) _t49 );
        _t51 = *((int*)_t50);
        _t53 = x;
        if( _t51>_t53 ) goto B6;
B8:
        _t55 = i;
        _t57 = j;
        if( _t55<_t57 ) goto B12;
B9:
        _t59 = j;
        return _t59;
        goto B10;
B10:
        goto T;
T:
        goto END;
B12:
        _t60 = 2;
        _l1 = _t60;
        _t63 = i;
        _t65 = _l1;
        _t66 = _t63<<_t65;
        _t67 = &a;
        _t68 = *((int**)_t67);
        _t69 = _t66 + ((void*) _t68 );
        _t70 = *((int*)_t69);
        aux = _t70;
        _t73 = j;
        _t75 = _l1;
        _t76 = _t73<<_t75;
        _t77 = &a;
        _t78 = *((int**)_t77);
        _t79 = _t76 + ((void*) _t78 );
        _t80 = *((int*)_t79);
        _t82 = i;
        _t84 = _l1;
        _t85 = _t82<<_t84;
        _t86 = &a;
        _t87 = *((int**)_t86);
        _t88 = _t85 + ((void*) _t87 );
        *((int*)_t88) = _t80;
        _t90 = aux;
        _t92 = j;
        _t93 = 2;
        _t94 = _t92<<_t93;
        _t95 = &a;
        _t96 = *((int**)_t95);
        _t97 = _t94 + ((void*) _t96 );
        *((int*)_t97) = _t90;
        goto B2;
END:
        ;
}
```

Figura 5.2: Código C gerado para a função *partition*

```
int partition(int* a,int p,int r)
{
        int _t30;
        int _t31;
        int _t32;
        int _t25;
        int _t17;
        int _t20;
        void* _t50;
        void* _t49;
        void* _t48;
        int _t47;
        void* _t9;
        void* _t10;
        void* _t11;
        int _t12;
        int _t40;
        int _t41;
        int _t42;
        void* _t33;
        void* _t34;
        void* _t35;
        int _t36;
        void* _t68;
        void* _t67;
        int _t66;
        int _t65;
        int _t26;
        int _t27;
        int _t60;
        int _t59;
        int _t53;
        int _t51;
        int _t46;
        int _t45;
        int _t80;
        void* _t79;
        void* _t78;
        void* _t77;
        int _t38;
        int _t70;
        void* _t69;
        int _t63;
        int _t98;
        void* _t97;
        int _t57;
        int _t55;
        int _t92;
        int _t90;
        int _t84;
        int _t76;
        int _t75;
        int _t73;
        int _l1;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t2 = &a;
        ▶_t3 = ((void*) a );
        _t4 = &a;
        ▶a = ((void*) _t3 );
        _t6 = p;
        _t7 = 2;
        _t8 = _t6 << _t7;
        _t9 = &a;
        ▶_t10 = ((void*) a );
        _t11 = _t8 + ((void*) _t10 );
        _t12 = *((int*)_t11);
        x = _t12;
        _t15 = p;
        _t16 = 1;
        _t17 = _t15 - _t16;
        i = _t17;
        _t20 = r;
        _t21 = 1;
        _t22 = _t20 + _t21;
        j = _t22;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        _t25 = i;
        _t26 = 1;
        _t27 = _t25 + _t26;
        i = _t27;
        goto B5;
B5:
        _t30 = i;
        _t31 = 2;
        _t32 = _t30 << _t31;
        _t33 = &a;
        ▶_t34 = ((void*) a );
        _t35 = _t32 + ((void*) _t34 );
        _t36 = *((int*)_t35);
        _t38 = x;
        if( _t36<_t38 ) goto B4;
B6:
        _t40 = j;
        _t41 = 1;
        _t42 = _t40 - _t41;
        j = _t42;
        goto B7;
B7:
        _t45 = j;
        _t46 = 2;
        _t47 = _t45 << _t46;
        _t48 = &a;
        ▶_t49 = ((void*) a );
        _t50 = _t47 + ((void*) _t49 );
        _t51 = *((int*)_t50);
        _t53 = x;
        if( _t51>_t53 ) goto B6;
B8:
        _t55 = i;
        _t57 = j;
        if( _t55<_t57 ) goto B12;
B9:
        _t59 = j;
        return _t59;
        goto B10;
B10:
        goto T;
T:
        goto END;
B12:
        _t60 = 2;
        _l1 = _t60;
        _t63 = i;
        _t65 = _l1;
        _t66 = _t63 << _t65;
        _t67 = &a;
        ▶_t68 = ((void*) a );
        _t69 = _t66 + ((void*) _t68 );
        _t70 = *((int*)_t69);
        aux = _t70;
        _t73 = j;
        _t75 = _l1;
        _t76 = _t73 << _t75;
        _t77 = &a;
        ▶_t78 = ((void*) a );
        _t79 = _t76 + ((void*) _t78 );
        _t80 = *((int*)_t79);
        _t82 = i;
        _t84 = _l1;
        _t85 = _t82 << _t84;
        _t86 = &a;
        ▶_t87 = ((void*) a );
        _t88 = _t85 + ((void*) _t87 );
        *((int*)_t88) = _t80;
        _t90 = aux;
        _t92 = j;
        _t93 = 2;
        _t94 = _t92 << _t93;
        _t95 = &a;
        ▶_t96 = ((void*) a );
        _t97 = _t94 + ((void*) _t96 );
        *((int*)_t97) = _t90;
        goto B2;
END:
        ;
}
```

Figura 5.3: Código C após a realização de *Pointer Optimization*

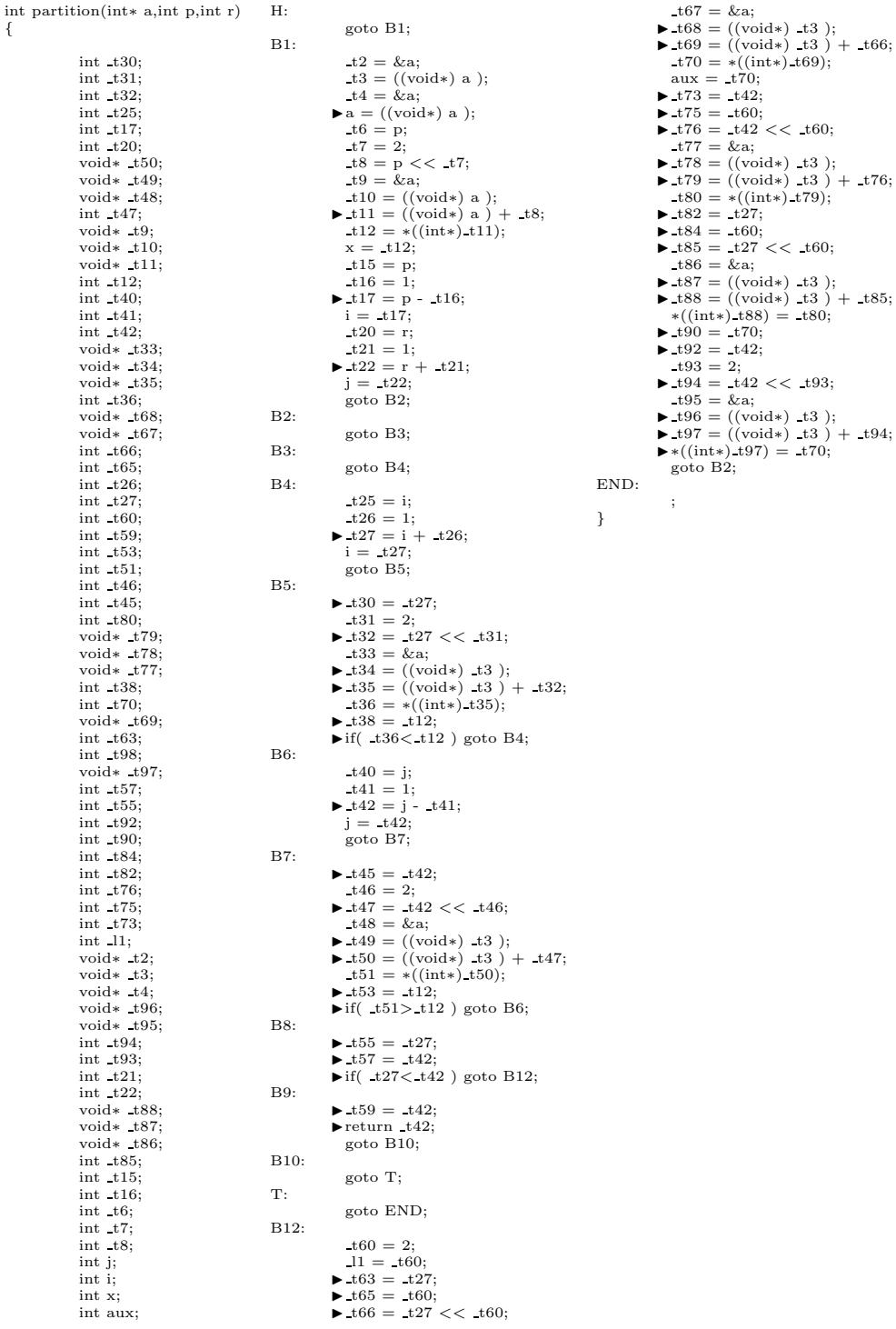


Figura 5.4: Código C após a realização de *Copy Propagation*

```
int partition(int* a,int p,int r)
{
        int _t30;
        int _t31;
        int _t32;
        int _t25;
        int _t17;
        int _t20;
        void* _t50;
        void* _t49;
        void* _t48;
        int _t47;
        void* _t9;
        void* _t10;
        void* _t11;
        int _t12;
        int _t40;
        int _t41;
        int _t42;
        void* _t33;
        void* _t34;
        void* _t35;
        int _t36;
        void* _t68;
        void* _t67;
        int _t66;
        int _t65;
        int _t26;
        int _t27;
        int _t60;
        int _t59;
        int _t53;
        int _t51;
        int _t46;
        int _t45;
        int _t80;
        void* _t79;
        void* _t78;
        void* _t77;
        int _t38;
        int _t70;
        void* _t69;
        int _t63;
        int _t98;
        void* _t97;
        int _t57;
        int _t55;
        int _t92;
        int _t90;
        int _t84;
        int _t82;
        int _t76;
        int _t75;
        int _t73;
        int _l1;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t2 = &a;
        _t3 = ((void*) a );
        _t4 = &a;
        a = ((void*) a );
        _t6 = p;
        _t7 = 2;
      ▶_t8 = p << 2;
        _t9 = &a;
        _t10 = ((void*) a );
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        x = _t12;
        _t15 = p;
        _t16 = 1;
      ▶_t17 = p - 1;
        i = _t17;
        _t20 = r;
        _t21 = 1;
      ▶_t22 = r + 1;
        j = _t22;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        _t25 = i;
        _t26 = 1;
      ▶_t27 = i + 1;
        i = _t27;
        goto B5;
B5:
        _t30 = _t27;
        _t31 = 2;
      ▶_t32 = _t27 << 2;
        _t33 = &a;
        _t34 = ((void*) _t3 );
        _t35 = ((void*) _t3 ) + _t32;
        _t36 = *((int*)_t35);
        _t38 = _t12;
        if( _t36<_t12 ) goto B4;
B6:
        _t40 = j;
        _t41 = 1;
      ▶_t42 = j - 1;
        j = _t42;
        goto B7;
B7:
        _t45 = _t42;
        _t46 = 2;
      ▶_t47 = _t42 << 2;
        _t48 = &a;
        _t49 = ((void*) _t3 );
        _t50 = ((void*) _t3 ) + _t47;
        _t51 = *((int*)_t50);
        _t53 = _t12;
        if( _t51>_t12 ) goto B6;
B8:
        _t55 = _t27;
        _t57 = _t42;
        if( _t27<_t42 ) goto B12;
B9:
        _t59 = _t42;
        return _t42;
        goto B10;
B10:
        goto T;
T:
        goto END;
B12:
        _t60 = 2;
      ▶_l1 = 2;
        _t63 = _t27;
      ▶_t65 = 2;
      ▶_t66 = _t27 << 2;
        _t67 = &a;
        _t68 = ((void*) _t3 );
        _t69 = ((void*) _t3 ) + _t66;
        _t70 = *((int*)_t69);
        aux = _t70;
        _t73 = _t42;
      ▶_t75 = 2;
      ▶_t76 = _t42 << 2;
        _t77 = &a;
        _t78 = ((void*) _t3 );
        _t79 = ((void*) _t3 ) + _t76;
        _t80 = *((int*)_t79);
        _t82 = _t27;
      ▶_t84 = 2;
      ▶_t85 = _t27 << 2;
        _t86 = &a;
        _t87 = ((void*) _t3 );
        _t88 = ((void*) _t3 ) + _t85;
        *((int*)_t88) = _t80;
        _t90 = _t70;
        _t92 = _t42;
        _t93 = 2;
      ▶_t94 = _t42 << 2;
        _t95 = &a;
        _t96 = ((void*) _t3 );
        _t97 = ((void*) _t3 ) + _t94;
        *((int*)_t97) = _t70;
        goto B2;
END:
        ;
}
```

Figura 5.5: Código C após a realização de *Constant Propagation*

```
int partition(int* a,int p,int r)
{
        int _t30;
        int _t31;
        int _t32;
        int _t25;
        int _t17;
        int _t20;
        void* _t50;
        void* _t49;
        void* _t48;
        int _t47;
        void* _t9;
        void* _t10;
        void* _t11;
        int _t12;
        int _t40;
        int _t41;
        int _t42;
        void* _t33;
        void* _t34;
        void* _t35;
        int _t36;
        void* _t68;
        void* _t67;
        int _t66;
        int _t65;
        int _t26;
        int _t27;
        int _t60;
        int _t59;
        int _t53;
        int _t51;
        int _t46;
        int _t45;
        int _t80;
        void* _t79;
        void* _t78;
        void* _t77;
        int _t38;
        int _t70;
        void* _t69;
        int _t63;
        int _t98;
        void* _t97;
        int _t57;
        int _t55;
        int _t92;
        int _t90;
        int _t84;
        int _t82;
        int _t76;
        int _t75;
        int _t73;
        int _l1;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t3 = ((void*) a );
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        _t17 = p - 1;
        i = _t17;
        _t22 = r + 1;
        j = _t22;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        _t27 = i + 1;
        i = _t27;
        goto B5;
B5:
        _t32 = _t27<<2;
        _t35 = ((void*) _t3 ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B4;
B6:
        _t42 = j - 1;
        j = _t42;
        goto B7;
B7:
        _t47 = _t42<<2;
        _t50 = ((void*) _t3 ) + _t47;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B6;
B8:
        if( _t27<_t42 ) goto B12;
B9:
        return _t42;
        goto B10;
B10:
        goto T;
T:
        goto END;
B12:
        _t66 = _t27<<2;
        _t69 = ((void*) _t3 ) + _t66;
        _t70 = *((int*)_t69);
        _t76 = _t42<<2;
        _t79 = ((void*) _t3 ) + _t76;
        _t80 = *((int*)_t79);
        _t85 = _t27<<2;
        _t88 = ((void*) _t3 ) + _t85;
        *((int*)_t88) = _t80;
        _t94 = _t42<<2;
        _t97 = ((void*) _t3 ) + _t94;
        *((int*)_t97) = _t70;
        goto B2;
END:
        ;
}
```

Figura 5.6: Código C após a realização de *Dead Code Elimination*

```
int partition(int* a,int p,int r)
{
        int _t30;
        int _t31;
        int _t32;
        int _t25;
        int _t17;
        int _t20;
        void* _t50;
        void* _t49;
        void* _t48;
        int _t47;
        void* _t9;
        void* _t10;
        void* _t11;
        int _t12;
        int _t40;
        int _t41;
        int _t42;
        void* _t33;
        void* _t34;
        void* _t35;
        int _t36;
        void* _t68;
        void* _t67;
        int _t66;
        int _t65;
        int _t26;
        int _t27;
        int _t5;
        int _t60;
        int _t59;
        int _t53;
        int _t51;
        int _t46;
        int _t45;
        int _t80;
        void* _t79;
        void* _t78;
        void* _t77;
        int _t38;
        int _t70;
        void* _t69;
        int _t14;
        int _t13;
        int _t63;
        int _t98;
        void* _t97;
        int _t57;
        int _t55;
        int _t92;
        int _t90;
        int _t84;
        int _t82;
        int _t76;
        int _t75;
        int _t73;
        int _l1;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t3 = ((void*) a );
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        _t17 = p - 1;
        i = _t17;
        _t22 = r + 1;
        j = _t22;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        _t27 = i + 1;
        i = _t27;
        goto B5;
B5:
        _t32 = _t27<<2;
        _t35 = ((void*) _t3 ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B4;
B6:
        _t42 = j - 1;
        j = _t42;
        goto B7;
B7:
        ▶_t14 = _t42<<2;
        ▶_t47 = _t14;
          _t50 = ((void*) _t3 ) + _t47;
          _t51 = *((int*)_t50);
          if( _t51>_t12 ) goto B6;
B8:
          if( _t27<_t42 ) goto B12;
B9:
          return _t42;
          goto B10;
B10:
          goto T;
T:
          goto END;
B12:
        ▶_t5 = _t27<<2;
        ▶_t66 = _t5;
          _t69 = ((void*) _t3 ) + _t66;
          _t70 = *((int*)_t69);
        ▶_t13 = _t14;
        ▶_t76 = _t13;
          _t79 = ((void*) _t3 ) + _t76;
          _t80 = *((int*)_t79);
        ▶_t85 = _t5;
          _t88 = ((void*) _t3 ) + _t85;
          *((int*)_t88) = _t80;
        ▶_t94 = _t13;
          _t97 = ((void*) _t3 ) + _t94;
          *((int*)_t97) = _t70;
          goto B2;
END:
          ;
}
```

Figura 5.7: Código C após a realização de *Common Subexpression Elimination*

```
int partition(int* a,int p,int r)
{
        int _t32;
        int _t17;
        void* _t50;
        int _t47;
        void* _t11;
        int _t12;
        int _t42;
        void* _t35;
        int _t36;
        int _t66;
        int _t27;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        int _t13;
        void* _t97;
        int _t76;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
```

```
H:
        goto B1;
B1:
        _t3 = ((void*) a );
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
      ▶ i = p - 1;
      ▶ _t17 = _t17;
      ▶ j = r + 1;
      ▶ _t22 = _t22;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
      ▶ i = i + 1;
      ▶ _t27 = i;
        goto B5;
B5:
        _t32 = _t27<<2;
        _t35 = ((void*) _t3 ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B4;
B6:
      ▶ j = j - 1;
      ▶ _t42 = j;
        goto B7;
B7:
        _t14 = _t42<<2;
        _t47 = _t14;
        _t50 = ((void*) _t3 ) + _t47;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B6;
B8:
        if( _t27<_t42 ) goto B12;
B9:
        return _t42;
        goto B10;
B10:
        goto T;
T:
        goto END;
B12:
        _t5 = _t27<<2;
        _t66 = _t5;
        _t69 = ((void*) _t3 ) + _t66;
        _t70 = *((int*)_t69);
        _t13 = _t14;
        _t76 = _t13;
        _t79 = ((void*) _t3 ) + _t76;
        _t80 = *((int*)_t79);
        _t85 = _t5;
        _t88 = ((void*) _t3 ) + _t85;
        *((int*)_t88) = _t80;
        _t94 = _t13;
        _t97 = ((void*) _t3 ) + _t94;
        *((int*)_t97) = _t70;
        goto B2;
END:
        ;
}
```

Figura 5.8: Código C após a realização de *PeepHole Optimization*

```
int partition(int* a,int p,int r)
{
        int _t32;
        int _t17;
        void* _t50;
        int _t47;
        void* _t11;
        int _t12;
        int _t42;
        void* _t35;
        int _t36;
        int _t66;
        int _t27;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        int _t13;
        void* _t97;
        int _t76;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t3 = ((void*) a );
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        i = i + 1;
        _t27 = i;
        goto B5;
B5:
        _t32 = _t27<<2;
        _t35 = ((void*) _t3 ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B4;
B6:
        j = j - 1;
        _t42 = j;
        goto B7;
B7:
        _t14 = _t42<<2;
        _t47 = _t14;
        _t50 = ((void*) _t3 ) + _t47;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B6;
B8:
        if( _t27<_t42 ) goto B12;
B9:
        return _t42;
        goto B10;
B10:
        goto T;
T:
        goto END;
B12:
        _t5 = _t27<<2;
        _t66 = _t5;
        _t69 = ((void*) _t3 ) + _t66;
        _t70 = *((int*)_t69);
        _t13 = _t14;
        _t76 = _t13;
        _t79 = ((void*) _t3 ) + _t76;
        _t80 = *((int*)_t79);
        _t85 = _t5;
        _t88 = ((void*) _t3 ) + _t85;
        *((int*)_t88) = _t80;
        _t94 = _t13;
        _t97 = ((void*) _t3 ) + _t94;
        *((int*)_t97) = _t70;
        goto B2;
END:
        ;
}
```

Figura 5.9: Código C após a realização de *Dead Code Elimination*

```
int partition(int* a,int p,int r)
{
        int _t32;
        int _t17;
        void* _t50;
        int _t47;
        void* _t11;
        int _t12;
        int _t42;
        void* _t35;
        int _t36;
        int _t66;
        int _t27;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        int _t13;
        void* _t97;
        int _t76;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t3 = ((void*) a );
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        _t27 = i;
        goto B7;
B7:
        ▶_t32 = i<<2;
        ▶_t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        _t42 = j;
        goto B10;
B10:
        ▶_t14 = j<<2;
        _t47 = _t14;
        ▶_t50 = ((void*) a ) + _t14;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        ▶if( i<j ) goto B15;
B12:
        ▶return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        ▶_t5 = i<<2;
        _t66 = _t5;
        ▶_t69 = ((void*) a ) + _t5;
        _t70 = *((int*)_t69);
        _t13 = _t14;
        ▶_t76 = _t14;
        ▶_t79 = ((void*) a ) + _t14;
        _t80 = *((int*)_t79);
        _t85 = _t5;
        ▶_t88 = ((void*) a ) + _t5;
        *((int*)_t88) = _t80;
        ▶_t94 = _t14;
        ▶_t97 = ((void*) a ) + _t14;
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.10: Código C após a realização de *Copy Propagation*

```
int partition(int* a,int p,int r)
{
        int _t32;
        int _t17;
        void* _t50;
        int _t47;
        void* _t11;
        int _t12;
        int _t42;
        void* _t35;
        int _t36;
        int _t66;
        int _t27;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        int _t13;
        void* _t97;
        int _t76;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t32 = i<<2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t50 = ((void*) a ) + _t14;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = i<<2;
        _t69 = ((void*) a ) + _t5;
        _t70 = *((int*)_t69);
        _t79 = ((void*) a ) + _t14;
        _t80 = *((int*)_t79);
        _t88 = ((void*) a ) + _t5;
        *((int*)_t88) = _t80;
        _t97 = ((void*) a ) + _t14;
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.11: Código C após a realização de *Dead Code Elimination*

```
int partition(int* a,int p,int r)
{
        int _t32;
        int _t17;
        void* _t50;
        int _t47;
        void* _t11;
        int _t12;
        int _t42;
        void* _t35;
        int _t36;
        int _t66;
        int _t27;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        int _t13;
        void* _t97;
        int _t76;
        void* _t2;
        void* _t3;
        void* _t4;
        void* _t10;
        void* _t9;
        void* _t96;
        void* _t95;
        int _t94;
        int _t93;
        int _t21;
        int _t22;
        void* _t88;
        void* _t87;
        void* _t86;
        int _t85;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t32 = i<<2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t50 = ((void*) a ) + _t14;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = i<<2;
      ▶ _t9 = ((void*) a ) + _t5;
      ▶ _t69 = ((void*) _t9 );
        _t70 = *((int*)_t69);
      ▶ _t10 = ((void*) a ) + _t14;
      ▶ _t79 = ((void*) _t10 );
        _t80 = *((int*)_t79);
      ▶ _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
      ▶ _t97 = ((void*) _t10 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.12: Código C após a realização de *Common Subexpression Elimination*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        void* _t97;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t32 = i<<2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t50 = ((void*) a ) + _t14;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = i<<2;
        _t9 = ((void*) a ) + _t5;
        _t69 = ((void*) _t9 );
        _t70 = *((int*)_t69);
        _t10 = ((void*) a ) + _t14;
        _t79 = ((void*) _t10 );
        _t80 = *((int*)_t79);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t10 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.13: Código C após a realização de *PeepHole Optimization*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        void* _t97;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t32 = i<<2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t50 = ((void*) a ) + _t14;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = i<<2;
        _t9 = ((void*) a ) + _t5;
        _t69 = ((void*) _t9 );
      ▶ _t70 = *((int*)_t9);
        _t10 = ((void*) a ) + _t14;
        _t79 = ((void*) _t10 );
      ▶ _t80 = *((int*)_t10);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t10 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.14: Código C após a realização de *Copy Propagation*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        int _t70;
        void* _t69;
        int _t14;
        void* _t97;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t32 = i<<2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t50 = ((void*) a ) + _t14;
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = i<<2;
        _t9 = ((void*) a ) + _t5;
        _t70 = *((int*)_t9);
        _t10 = ((void*) a ) + _t14;
        _t80 = *((int*)_t10);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t10 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.15: Código C após a realização de *Dead Code Elimination*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t79;
        void* _t3;
        int _t70;
        void* _t69;
        int _t14;
        void* _t97;
        int _t2;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t15;
        int _t16;
        int _t6;
        int _t7;
        int _t8;
        int j;
        int i;
        int x;
        int aux;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
      ▶ _t2 = i<<2;
      ▶ _t32 = _t2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
      ▶ _t3 = ((void*) a ) + _t14;
      ▶ _t50 = ((void*) _t3 );
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
      ▶ _t5 = _t2;
        _t9 = ((void*) a ) + _t5;
        _t70 = *((int*)_t9);
      ▶ _t10 = ((void*) _t3 );
        _t80 = *((int*)_t10);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t10 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.16: Código C após a realização de *Common Subexpression Elimination*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t3;
        int _t70;
        int _t14;
        void* _t97;
        int _t2;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t8;
        int j;
        int i;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t2 = i<<2;
        _t32 = _t2;
        _t35 = ((void*) a ) + _t32;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t3 = ((void*) a ) + _t14;
        _t50 = ((void*) _t3 );
        _t51 = *((int*)_t50);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = _t2;
        _t9 = ((void*) a ) + _t5;
        _t70 = *((int*)_t9);
        _t10 = ((void*) _t3 );
        _t80 = *((int*)_t10);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t10 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.17: Código C após a realização de *PeepHole Optimization*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t3;
        int _t70;
        int _t14;
        void* _t97;
        int _t2;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t8;
        int j;
        int i;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t2 = i<<2;
        _t32 = _t2;
      ▶ _t35 = ((void*) a ) + _t2;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t3 = ((void*) a ) + _t14;
        _t50 = ((void*) _t3 );
      ▶ _t51 = *((int*)_t3);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t5 = _t2;
      ▶ _t9 = ((void*) a ) + _t2;
        _t70 = *((int*)_t9);
        _t10 = ((void*) _t3 );
      ▶ _t80 = *((int*)_t3);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
      ▶ _t97 = ((void*) _t3 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.18: Código C após a realização de *Copy Propagation*

```
int partition(int* a,int p,int r)
{
        int _t32;
        void* _t50;
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t5;
        int _t51;
        int _t80;
        void* _t3;
        int _t70;
        int _t14;
        void* _t97;
        int _t2;
        void* _t10;
        void* _t9;
        void* _t88;
        int _t8;
        int j;
        int i;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t2 = i<<2;
        _t35 = ((void*) a ) + _t2;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t3 = ((void*) a ) + _t14;
        _t51 = *((int*)_t3);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t9 = ((void*) a ) + _t2;
        _t70 = *((int*)_t9);
        _t80 = *((int*)_t3);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t3 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.19: Código C após a realização de *Dead Code Elimination*

```
int partition(int* a,int p,int r)
{
        void* _t11;
        int _t12;
        void* _t35;
        int _t36;
        int _t51;
        int _t80;
        void* _t3;
        int _t70;
        int _t14;
        void* _t97;
        int _t2;
        void* _t9;
        void* _t88;
        int _t8;
        int j;
        int i;
H:
        goto B1;
B1:
        _t8 = p<<2;
        _t11 = ((void*) a ) + _t8;
        _t12 = *((int*)_t11);
        i = p - 1;
        j = r + 1;
        goto B2;
B2:
        goto B3;
B3:
        goto B4;
B4:
        goto B5;
B5:
        goto B6;
B6:
        i = i + 1;
        goto B7;
B7:
        _t2 = i<<2;
        _t35 = ((void*) a ) + _t2;
        _t36 = *((int*)_t35);
        if( _t36<_t12 ) goto B6;
B8:
        goto B9;
B9:
        j = j - 1;
        goto B10;
B10:
        _t14 = j<<2;
        _t3 = ((void*) a ) + _t14;
        _t51 = *((int*)_t3);
        if( _t51>_t12 ) goto B9;
B11:
        if( i<j ) goto B15;
B12:
        return j;
        goto B13;
B13:
        goto T;
T:
        goto END;
B15:
        _t9 = ((void*) a ) + _t2;
        _t70 = *((int*)_t9);
        _t80 = *((int*)_t3);
        _t88 = ((void*) _t9 );
        *((int*)_t88) = _t80;
        _t97 = ((void*) _t3 );
        *((int*)_t97) = _t70;
        goto B3;
END:
        ;
}
```

Figura 5.20: Código C após a realização de *PeepHole Optimization*

# Capítulo 6

# Conclusão e Trabalhos Futuros

Devido ao aumento da complexidade dos novos processadores, compiladores otimizantes de alto desempenho são necessários para se poder conduzir experimentos nas áreas de otimização e geração de código.

Visando a atender essa necessidade, a contribuição desta tese de mestrado, assim acredita-se, é a apresentação do compilador Xingó, o qual é um compilador otimizante com a característica de poder gerar código C a partir de sua representação intermediária, a XIR.

O compilador Xingó está sendo desenvolvido como um projeto do Laboratório de Sistemas de Computação (LSC) do Instituto de Computação da Unicamp.

No momento em que esta tese foi escrita, o Xingó possuía um total de oito otimizações de código implementadas e a geração de código C a partir da XIR estava funcionando para uma quantidade razoável de programas.

Os testes de geração de código C foram feitos utilizando-se basicamente três *benchmarks*: Mediabench, SPEC e NullStone.

O *benchmark* NullStone realiza uma série de testes com o objetivo de avaliar o desempenho e a corretude do compilador. A geração de código C foi bem sucedida em 6581 de um total de 6611 testes realizados pelo NullStone. Nestes testes nenhuma otimização foi aplicada aos programas de entrada. Com a aplicação de otimização, a geração de código C foi bem sucedida em 6497 de um total de 6611 testes realizados. Por geração de código bem sucedida, entende-se a correta compilação e execução do código C gerado pelo Xingó.

Os *benchmarks* Mediabench e SPEC são utilizados em conjunto, com o objetivo de avaliar a corretude do código C gerado pelo Xingó. No presente momento a geração de código C era bem sucedida para os programas adpcm, epic, g721, bzip2, gzip, mcf, twolf; sem a aplicação de nenhuma otimização. A geração de código C otimizado para tais programas ainda não foi testada. Outros programas pertencentes a estes dois *benchmarks* não puderam ser testados, pois os mesmos não passaram pelo LCC, o qual está sendo

modificado para suplantar tal problema, pois se um programa não é aceito pelo LCC, o Xingó não tem como realizar a sua compilação.

O Xingó tem o objetivo de alcançar um desempenho igual ou superior ao compilador GCC, no tocante a otimização de código e no número de programas aceitos. Para atingir tal meta as otimizações do Xingó estão sendo constantemente comparadas com as equivalentes no GCC, para a realização dos devidos ajustes e modificações.

## 6.1 Trabalhos Futuros

Entre os trabalhos futuros a serem realizados no Xingó, estão incluídos a implementação de novas otimizações, como *code hoisting*, *control-flow optimization* entre outras, pois um conjunto de apenas oito transformações de código não é o suficiente para tornar um compilador otimizante competitivo com o GCC e outros compiladores comerciais.

A realização de testes de geração de código C utilizando outros *benchmarks* como entrada também integra a lista de trabalhos futuros, pois assim ter-se-á a garantia da corretude de tal processo para o maior número possível de programas. Os *benchmarks* cogitados são o MiBench e o c-torture (utilizado pelo GCC).

E por último, a implementação de novas classes e métodos que integram a *post-pass data-flow analysis* também está prevista entre os trabalhos futuros a serem realizados, pois tal módulo tem o objetivo de tornar mais simples a implementação de otimizações dependentes de máquina.

# Referências Bibliográficas


[1] *GNU Compiler Collection*. `http://gcc.gnu.org`. Acessado em 19/03/2004.

[2] *NULLSTONE - Automated Compiler Perfomance Analysis*. `http://www.nullstone.com`. Acessado em 19/03/2004.

[3] Alfred V. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. On Prog. Lang. and Sys.*, 11(4):491–516, 1989.

[4] Alfred V. Aho and S. Johnson. Code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1976.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[6] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, United Kingdom, 1998.

[7] Andrew W. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. Technical report, `http://www.cs.virginia.edu/zephyr`, University of Virginia, 1998. Acessado em 19/03/2004.

[8] Preston Briggs. Register allocation via graph coloring. Technical Report CRPC-TR92218, Ctr. for Research on Parallel Computation, Rice Univ., April 1992.

[9] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-Insensitive Interprocedural Alias Analysis in the Pressence of Pointers. In David Gelertner, Alexandru Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, 892*. Springer-Verlag, 1995.

[10] Martin D. Carroll and Barbara G. Ryder. Incremental data flow analysis via dominator and attribute updates. *15th ACM Symposium on Principles of Programming Languages*, pages 274–284, 1988.

[11] Chaitin, Gregory, March Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register Allocations Via Coloring. In *Computer Languages*, volume 6, pages 47–57, 1981.

[12] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.

[13] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–117, 1998.

[14] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[15] C. Fraser, D. Hanson, and T. Proebsting. Engineering a simple, efficient code generator. *ACM Letters of Programming Languages and Systems*, 1(3):213–226, 1992.

[16] Christopher W. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, Menlo Park, California, 1995.

[17] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[18] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[19] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, Bugnion E., and M. Lam Maimizing. Mutiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[20] Marc Shapiro II and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[21] Guy L. Steele Jr. and Jon L. While. How to print floating-point numbers accurately. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 112–126, 1990.

[22] S. Jung and Y. Paek. The very portable optimizer for digital signal processors. *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, 2001.

[23] Gerry Kane and Joe Heinrich. *Mips Risc Architecture*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[24] R. Leupers. LANCE: A C compiler platform for embedded DSPs. *Embedded Systems/Embedded Intelligence*, Feb. 2001.

[25] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. *ACM Symp. on Principles on Programming Languages*, pages 184–196, 1990.

[26] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.

[27] Erik Ruf. Context-insensitive alias analysis. Technical report, Microsoft Corporation - Advanced Technology Division, 1995.

[28] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[29] A. Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors*. PhD thesis, Princeton University Department of EE, May 15 1998.

[30] A. Sudarsanam, S. Malik, S. Tjiang, and S. Liao. Optimization of embedded DSP programs using post-pass data-flow analysis. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1997.

[31] S. W. K. Tjang. An Olive twig. Technical report, Synopsys Inc., 1993.

[32] Jyh-Shiarn Yur, Barbara G. Ryder, and William Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *International Conference on Software Engineering*, pages 442–451, 1999.