Análise dos caminhos de execução de programas para a paralelização automática de códigos binários para a plataforma intel x86

André Mantini Eberle

André Mantini Eberle

Análise dos caminhos de execução de programas para a paralelização automática de códigos binários para a plataforma intel x86

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Rodrigo Fernandes de Mello

USP – São Carlos
Dezembro de 2015

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Eberle, André Mantini
E16a Análise dos caminhos de execução de programas para a paralelização automática de códigos binários para a plataforma intel x86 / André Mantini Eberle; orientador Rodrigo Fernandes de Mello . -- São Carlos, 2015.
70 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2015.

1. Paralelização automática. 2. Análise de dependências . 3. Código Binário. 4. Plataforma Intel x86. I. , Rodrigo Fernandes de Mello, orient. II. Título.

André Mantini Eberle

Analysis of the execution paths of programs to perform automatic parallelization of binary codes on the platform intel x86

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação - ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Rodrigo Fernandes de Mello

USP – São Carlos
December 2015

Resumo

Aplicações têm tradicionalmente utilizado o paradigma de programação sequencial. Com a recente expansão da computação paralela, em particular os processadores multinúcleo e ambientes distribuídos, esse paradigma tornou-se um obstáculo para a utilização dos recursos disponíveis nesses sistemas, uma vez que a maior parte das aplicações tornam-se restrita à execução sobre um único núcleo de processamento. Nesse sentido, este trabalho de mestrado introduz uma abordagem para paralelizar programas sequenciais de forma automática e transparente, diretamente sobre o código-binário, de forma a melhor utilizar os recursos disponíveis em computadores multinúcleo. A abordagem consiste na desmontagem (*disassembly*) de aplicações *Intel x86* e sua posterior tradução para uma linguagem intermediária. Em seguida, são produzidos grafos de fluxo e dependências, os quais são utilizados como base para o particionamento das aplicações em unidades paralelas. Por fim, a aplicação é remontada (*assembly*) e traduzida novamente para a arquitetura original. Essa abordagem permite a paralelização de aplicações sem a necessidade de esforço suplementar por parte de desenvolvedores e usuários.

Palavras-chave: paralelização automática, análise de dependências, código-binário, plataforma Intel x86.

Abstract

Traditionally, computer programs have been developed using the sequential programming paradigm. With the advent of parallel computing systems, such as multi-core processors and distributed environments, the sequential paradigm became a barrier to the utilization of the available resources, since the program is restricted to a single processing unit. To address this issue, we introduce a transparent automatic parallelization methodology using a binary rewriter. The steps involved in our approach are: the disassembly of an *Intel x86* application, transforming it into an intermediary language; analysis of this intermediary code to obtain flow and dependency graphs; partitioning of the application into parallel units, using the obtained graphs and posterior reassembly of the application, writing it back to the original *Intel x86* architecture. By transforming the compiled application software, we aim at obtaining a program which can explore the parallel resources, with no extra effort required either from users or developers.

Keywords: automatic parallelization, dependency analysis, binary rewriter, binary code, Intel x86 platform.

x

Lista de Figuras

Lista de Tabelas

Lista de Siglas

RNG	*Gerador de números aleatórios (Random number generator)*
RAW	*Dependência do tipo leitura após escrita (Read After Write)*
WAR	*Dependência do tipo escrita após leitura (Write After Read)*
WAW	*Dependência do tipo escrita após escrita (Write After Write)*
PGHPF	*The Portland Group Compiler Technology*
POS	*Reticulado de ordem parcial (Partial order set)*

Sumário

CAPÍTULO

1

Introdução

A necessidade por processamento de alto desempenho e a redução de custo de recursos computacionais motivaram a construção de ambientes *scale-up* e *scale-out* (MICHAEL *et al.*, 2007). Esses ambientes permitem explorar o poder computacional de várias máquinas e componentes de maneira integrada. A tentativa de utilizar recursos disponíveis, no entanto, tem como obstáculo a natureza sequencial dos paradigmas mais convencionais (imperativo e orientado a objetos) da computação. Nesse contexto, aplicações tradicionais (principalmente o legado) acabam por explorar pouco o potencial desses ambientes. Para contornar esses problemas, esforços têm sido realizados para paralelizar essas aplicações de forma a permitir a utilização desses recursos.

As primeiras pesquisas realizadas na área de paralelização automática de programas datam do final da década de 1980 e início da década de 1990. Esses trabalhos focaram, principalmente, na paralelização de código-fonte para as linguagens C e Fortran (GILDER; KRISHNAMOORTHY, 1994). Em resumo, eles realizavam análises das dependências entre instruções e de fluxo em estruturas bem conhecidas dos programas, como laços, repetições, recursões em códigos-fonte, com objetivo de produzir técnicas para a paralelização de regiões dos programas que utilizavam esse tipo de estrutura. Ainda nesse período, outros trabalhos buscaram analisar as dependências de instruções a fim de compreender trechos de código-fonte que poderiam ser paralelizados (GRIEBL; COLLARD, 1995). Na década de 1990, pesquisadores como Zima *et al.* (1993) buscaram por soluções de paralelização automática de código-fonte para sistemas de memória compartilhada distribuída. Após utilizar máquinas de busca, i.e., IEEExplore, ACM Digital Library e Google Scholar, observou-se um período de 1993 ao início de década de 2000 em que poucos trabalhos

foram desenvolvidos em termos de paralelização de código. Em contrapartida, destacam-se, nesse período, estudos e propostas de arcabouços para a escrita de códigos paralelos tanto para sistemas *scale-up* quanto *scale-out* (MICHAEL *et al.*, 2007).

No início de década de 2000, trabalhos concentraram-se na paralelização automática de programas seja explorando características do código-fonte ou do binário (BASTOUL, 2003; DEPARTMENT *et al.*, 2003). Automatizando a paralelização, usuários sem conhecimentos do paradigma de programação concorrente (GHEZZI; JAZAYERI, 1997) teriam acesso aos recursos disponíveis tanto em computadores de uso pessoal, os quais já contavam com múltiplos núcleos ou processadores, quanto em ambientes distribuídos, tais como aglomerados (*clusters*) de computadores, grades (*grids*) computacionais, etc. Esses trabalhos focaram na paralelização sobre o código-fonte, principalmente sobre a linguagem *C*, utilizando modelos poliedrais (BASTOUL, 2003). O modelo poliedral é uma abstração que pode ser aplicada sobre iterações que ocorrem em laços de repetição, quando esses laços possuem limites, ou seja, as condições de término definidas por um contador. Nesse modelo, o domínio das iterações pode ser representado por um conjunto de inequações lineares, as quais podem ser modeladas por um poliedro. Mais especificamente, esse poliedro forma um conjunto convexo de pontos em um reticulado (*lattice*) (LI; PINGALI, 1992). Utilizando essa técnica, esses trabalhos focaram em transformar essas repetições de forma a criar blocos de execução paralelos. Apesar dos avanços significativos dessa abordagem, o foco em estruturas fechadas de repetição apenas permite a paralelização em casos específicos.

À partir de 2008, observou-se novo interesse pela área, com diversos trabalhos publicados. Bondhugula (BONDHUGULA, 2011) propõe uma técnica para otimizar a comunicação entre blocos de execução paralelos, obtidos à partir da paralelização de uma aplicação sequencial. Como a comunicação é limitada pela dependência entre diferentes instruções, e por sua vez, entre diferentes blocos, essa abordagem descreve o problema da distribuição dos blocos frente ao grau de paralelismo obtido. Howard *et al.* (HOWARD; RYAN; COLLINS, 2011) apresentam uma abordagem utilizando algoritmos evolutivos para a paralelização automática. Utilizando Programação Genética, esse trabalho descreve operações sobre blocos sequenciais de instruções, buscando formas de refatoração que levam ao paralelismo. A abordagem explora possíveis maneiras de separar as instruções em um dado número de processadores, de forma a não violar a dependência entre elas. (KOVACEVIC *et al.*, 2013) apresentaram um arcabouço completo para a paralelização automática de aplicações. Nesse trabalho, os passos para obter o paralelismo são delineados, mas não são explicados em detalhes. Nessa abordagem, apenas dependências de acesso aos registradores do processador são consideradas, não se levando em consideração os acessos à memória.

Essas diversas frentes de pesquisa buscaram por diferentes maneiras de prover paralelismo para aplicações projetadas e desenvolvidas para execução sequencial. Contudo, observa-se a necessidade de ferramentas que permitam traduzir códigos binários sequen-

ciais em versões paralelas, a fim de utilizar os recursos atualmente disponíveis em ambientes *scale-up* e *scale-out*. Isso se dá principalmente no que se refere a aplicações legadas (feitas para arquiteturas específicas), usualmente disponíveis apenas na forma de código binário. Esta é a lacuna considerada por este trabalho de mestrado, a qual motivou a definição da hipótese e do objetivo. Antes de apresentá-los, o problema abordado por este trabalho é melhor descrito na seção seguinte.

1.1 Problema de paralelização automática de código binário

O paradigma mais tradicional de desenvolvimento de *software* é voltado para a programação sequencial de aplicações. Nessa abordagem, as aplicações são definidas por conjuntos de instruções, que executam em uma sequência pré-definida. Quando um algoritmo é descrito em termos de uma sequência de instruções, a lógica inerente ao processo implica em dependências entre as partes envolvidas na execução. Dessa maneira, uma instrução, de maneira geral, não pode ser executada fora de ordem, caso contrário os dados podem entrar em um estado de inconsistência, comprometendo a lógica envolvida, i.e., a semântica da aplicação.

Programas construídos dessa maneira executam tão rápido quanto o processador permite, ou seja, a restrição de velocidade está na capacidade de execução de instruções por unidades de tempo concedidas pelo processador. Durante muitos anos, processadores sofreram uma rápida evolução em termos dessa capacidade. No entanto, em meados dos anos 2000, os fabricantes encontraram limitações físicas que comprometeram essa evolução, limitando, por enquanto, a expansão direta de velocidade de cada unidade de processamento. O principal obstáculo é a temperatura que o componente atinge quando o *clock* do processador é aumentado. Para contornar esse problema, o foco da construção desses componentes mudou, do aumento da frequência do relógio do processador para a inclusão de múltiplas unidades funcionais e núcleos em um único computador. No entanto, a utilização desse tipo de abordagem leva ao fato de que uma aplicação sequencial não pode mais utilizar todo o potencial oferecido pelo computador, pois sua construção a restringe a um único núcleo.

Além de processadores com vários núcleos, outra forte tendência na computação são os ambientes distribuídos. Em aglomerados de computadores (*clusters*), ambientes de grade computacional (*grid computing*) e na chamada computação em nuvem (*cloud computing*), sítios de máquinas são utilizados, em geral grupos de processadores, para o armazenamento de dados e a execução de aplicações. Porém, como no caso de processadores de vários núcleos, aplicações sequenciais não podem utilizar plenamente o potencial desses ambientes, pois sempre estarão restritas a um único núcleo.

Um dos obstáculos, portanto, para o uso da computação paralela e distribuída encontra-se na dificuldade em migrar programas do tradicional paradigma sequencial para uma versão paralela, que permita utilizar o potencial dos novos tipos de ambientes computacionais. A dificuldade em modificar a aplicação para tal fim surge dos problemas causados pela rigidez na ordem de execução das instruções sequencialmente definidas. Como algumas instruções utilizam resultados de computações anteriores, cria-se uma dependência entre elas e, portanto, precisam ser executadas em sequência. Porém, essa dependência não ocorre necessariamente entre todas as instruções de uma mesma aplicação. De fato, se grupos de instruções puderem ser isolados de forma que não exista dependência entre eles, cada grupo poderia ser executado em paralelo sem prejudicar a lógica dos algoritmos envolvidos. Dessa maneira, pode-se projetar uma abordagem para a paralelização automática de aplicações. Para criar uma versão paralela de uma aplicação, deve-se analisá-la, estabelecer as dependências entre suas instruções, e gerar uma nova aplicação, pronta para ser executada em paralelo.

A análise de uma aplicação a fim de se obter paralelismo pode ser dividida em duas abordagens distintas. Ela pode ser feita em alto nível, seja em código-fonte ou outra abstração, ou em baixo nível, quando aplicada diretamente sobre uma linguagem de máquina. Muitos dos trabalhos existentes focam em análises de código-fonte como uma forma de auxiliar o desenvolvedor (BASTOUL, 2003; DEPARTMENT *et al.*, 2003; GRIEBL; COLLARD, 1995). Neste trabalho de mestrado, o foco é dado para a linguagem de máquina, com o objetivo de se obter um paralelizador completo e transparente para o usuário final. Além disso, pode-se dividir a análise propriamente dita em duas fases. Uma fase estática, a qual é feita diretamente sobre as instruções da aplicação, seja em alto ou baixo nível, e uma possível fase dinâmica, em que resultados da execução podem ser analisados para se tomar decisões de paralelização. As duas fases não são mutuamente exclusivas, e podem ser usadas uma em suporte da outra.

O problema da análise de dependências entre instruções é complexo. As dependências, de forma geral, podem ser diretas ou indiretas, ou seja, uma instrução pode depender imediatamente de outra, ou pode depender de uma (ou mais) instrução(ões) intermediária(s). Dessa forma, as dependências podem ser vistas como um grafo direcionado. Na Figura 1.1 é ilustrado um exemplo de código, em linguagem de máquina, onde existem dependências entre instruções. Na Figura 1.2, apresenta-se o grafo de dependências para esse código. As instruções 020 e 010 dependem da instrução 009, pois elas leem a memória na mesma posição onde foi realizada uma escrita. De forma similar, a instrução 007 depende da instrução 002. A dependência entre 009 e 007 ocorre de maneira diferente. Nesse caso, 009 irá sobrescrever o valor que 007 iria ler, e portanto 009 deve ser executada após 007.

A obtenção desse grafo, porém, pode ser muito difícil. Como uma aplicação tende a percorrer caminhos de execução variados, resultados de computações anteriores de-

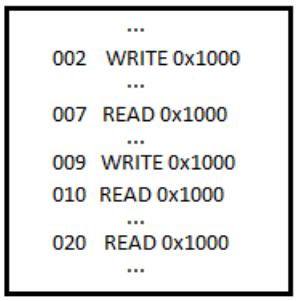

Figura 1.1: Exemplo de código de máquina com dependências.

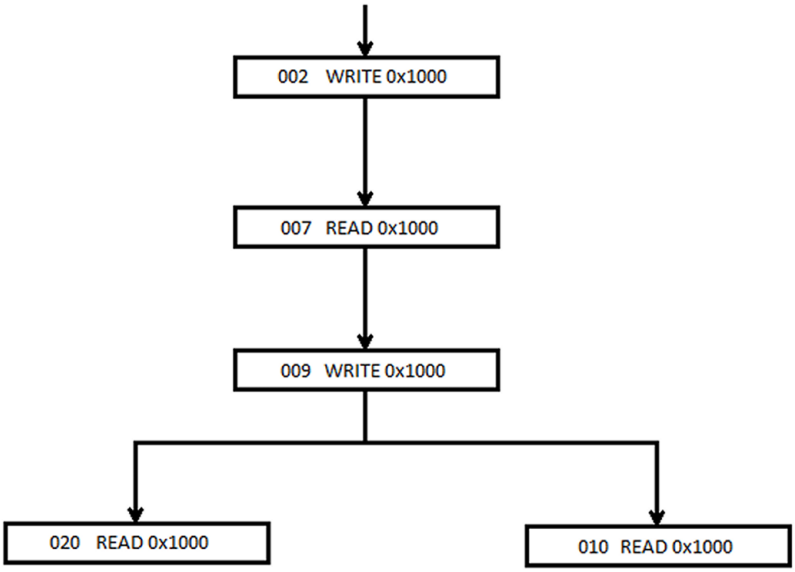

Figura 1.2: Exemplo de grafo de dependências.

terminam dependências entre instruções. Nesses casos, pode-se avaliar as condições necessárias para a ocorrência de dependências, verificando as computações envolvidas e buscando analisar seus possíveis resultados. No entanto, em alguns cenários as combinações possíveis de condições podem ser exponenciais. Portanto, a obtenção de um grafo completo e preciso de dependências envolve um problema exponencialmente difícil para diversas aplicações (GAREY; JOHNSON, 1979).

A complexidade se deve à dificuldade em prever os possíveis estados de uma aplicação após um período de execução. Entradas fornecidas à aplicação, seja pelo usuário, seja por meio de elementos externos (como geradores de números aleatórios), podem produzir estados variados. Dependendo de como o resultado de computações sobre essas entradas é reutilizado, o número possível de combinações de estados pode crescer exponencialmente. Isso ocorre, principalmente, quando há muitos caminhos condicionais, sendo que cada ramo de execução leva a potenciais estados distintos. No Algoritmo 1.1, um exemplo

de dependências complexas é ilustrado. Nesse algoritmo, o valor de i é inicializado de maneira aleatória, sobre o qual incidem várias computações. Em cada bloco condicional, a computação realizada depende do resultado do bloco anterior. Dessa forma, para cada bloco, as possibilidades para o valor de i dobram. Assim, ao término do bloco, o número de possibilidades para i é 2^k, em que k é o número de blocos condicionais entre o valor inicial de i e o momento de leitura de i para acesso ao vetor vec. Se i é uma variável índice, como nesse caso, então ela determina um acesso à memória, como um deslocamento de vec e, portanto, conhecê-la é necessário para entender como a memória é acessada.

Algoritmo 1.1: Exemplo de pseudocódigo. Neste caso, prever o valor da variável i, ao fim do bloco, é exponencialmente difícil (GAREY; JOHNSON, 1979).

```
1: Seja vec um vetor de tamanho arbitrário, inicializado com valores arbitrários
2: i ← Saída de um RNG (gerador de números aleatórios)
3: if i < 10 then
4:     i ← i * 12
5: else
6:     i ← i − 3
7: end if
8: if i < 5 then
9:     i ← i * 15
10: else
11:    i ← i/2
12: end if
13: if i > 100 then
14:    i ← i * 81
15: else
16:    i ← i + 4
17: end if
    {... outros blocos condicionais similares}
18: X ← vec[i]
```

Outra questão importante está relacionada ao acesso à memória. Em laços de repetição, estruturas de dados podem ser acessadas em vários níveis. Por exemplo, uma árvore pode ser percorrida parcial ou integralmente, em que a partir de uma raiz, vários nós são lidos, possivelmente através de referências (ponteiros). Se alguma outra instrução acessar a mesma estrutura em outro ponto do programa, é preciso saber quais partes são acessadas para determinar se há ou não dependências. De forma geral, é muito difícil prever, para uma execução, quando e quais partes da estrutura serão acessadas dentro do laço de repetição. Ainda nesse sentido, a definição dos locais em que ocorrem acessos à memória compreende outro problema.

Esses diversos problemas justificam a proposta de trabalhos voltados para auxiliar desenvolvedores por meio da análise de dependências em aplicações, tanto em código-fonte como sobre código binário. Com base nessa motivação, a seção seguinte apresenta a hipótese e objetivo deste trabalho de mestrado.

1.2 Hipótese e objetivo

A falta de ferramentas automáticas que permitam traduzir código binário de aplicações de propósito geral para versões paralelas motivou a hipótese deste trabalho, a qual é definida como segue:

A análise dos caminhos de execução de um código binário permite o particionamento desse programa em regiões independentes e sua posterior paralelização automática.

Motivado por essa hipótese, o objetivo deste trabalho foi o desenvolvimento de uma metodologia para a paralelização automática envolvendo a desmontagem (*disassemble*), a análise dos caminhos de execução determinísticos, i.e., aqueles que dependem apenas da entrada do programa, bem como os estocásticos, i.e., aqueles relacionados a resultados intermediários e desconhecidos na etapa de análise, a criação de grafos de fluxo de controle e dependências e o particionamento em unidades paralelas para execução em ambientes multinúcleo (*multi-core*). Como contribuição, aplicações legadas podem utilizar os recursos de ambientes paralelos atualmente disponíveis, principalmente em ambientes multinúcleo.

1.3 Organização da monografia

Esta monografia está organizada da seguinte maneira. No Capítulo 2, apresenta-se uma revisão da literatura, contando com os principais trabalhos encontrados, assim como suas limitações. Em seguida, conceitos utilizados durante o projeto são apresentados no Capítulo 3. No Capítulo 4, apresentam-se a abordagem desenvolvida e as atividades realizadas, destacando-se os problemas encontrados. O Capítulo 5 apresenta as conclusões e discussões sobre trabalhos futuros.

CAPÍTULO

2

Trabalhos relacionados

2.1 Considerações iniciais

Este capítulo apresenta alguns estudos primários e secundários (*surveys*), com o intuito de compreender a área de pesquisa na qual este trabalho de mestrado está inserido. Além disso, esses estudos permitiram melhor embasar o desenvolvimento da abordagem de paralelização automática.

2.2 Estudos principais

Banerjee (BANERJEE, 2013) procura fundamentar matematicamente conhecimentos e técnicas existentes para a determinação de dependências entre instruções. Esse trabalho introduz diversos conceitos e notações para acessos de leitura e escrita à memória (particularizado pela atribuição de um valor a uma variável) e para estruturas de repetição. O programa é definido como um conjunto de instruções, sobre o qual é imposta uma relação de ordem R, com o operador menor que (ou seja, $<$). Como a aplicação é executada sequencialmente, R é definida pela ordem em que instruções são processadas. Como a atribuição é a única instrução considerada (note que instruções de laços como *do-while* servem apenas para controle de fluxo, e não são consideradas para dependências), então elementos são apenas acessos à memória.

Seja P um programa e $S \in P$ e $T \in P$ elementos do seu conjunto de instruções, então se define $S < T$ se e somente se S executa antes de T. No entanto, se S ocorre dentro de um laço de repetição, então esse acesso não pode ser considerado como uma

única execução. Em particular, considera-se que S possui instâncias, de acordo com a iteração considerada. Seja L o laço que contém S e I o índice que controla seu número de iterações. Dessa maneira, as instâncias de S são definidas por $S(I)$. Se outro acesso $T \in L$ for considerado, instâncias para $T(J)$ também podem ser representadas. Assim diz-se que $S(I) < T(J)$, se e somente se, a instância I de S executa antes de $T(J)$. As posições léxicas de S e T dentro do laço são utilizadas para definir conjuntos de I e J, em que $S(I_i) < T(I_j)$ ocorre. O trabalho vai além, e define laços aninhados $L_1, L_2,...,L_k$. Nesse caso, um acesso S pode ocorrer em um ou mais L_i, e suas instâncias são representadas por $S(I_1,I_2,...,I_m)$, em que m denota o número de estruturas que S participa.

Sobre um laço de repetição bem definido, com a relação de ordem sobre as instâncias dos acessos conhecida, Banerjee define a existência de uma dependência entre $S' = S(I_1,I_2,...,I_m)$ e $T' = T(J_1,J_2,...,J_n)$ (definida por $S'\sigma T'$) se $S' < T'$, S' e T' acessam uma mesma região de memória M e um dos acessos é uma escrita. Além disso, a dependência é classificada em um de três tipos:

- *flow-dependent* (dependente do fluxo), em que S' escreve em M e T' realiza uma leitura;
- *anti-dependent* (antidependente), se S' lê M e, posteriormente, T' escreve nessa região;
- *output-dependent* (dependente por saída), se S' e T' escrevem em M.

Além dos tipos de dependência, Banerjee introduz também os conceitos de nível e distância, definidos pelo número de iterações que separam uma execução de S e T necessárias para a ocorrência de uma dependência. Para decidir se existe ou não uma dependência em um laço, ele é definido em termos de funções dos seus acessos à memória. Em particular, as variáveis envolvidas são expressas em um sistema de equações lineares, sujeitas a algumas restrições (também lineares). O trabalho demonstra que, se uma solução inteira existe para esse sistema, então uma dependência existe entre os acessos envolvidos, e as instâncias específicas podem ser derivadas da solução. Posteriormente, apresenta-se abordagens para resolver esses sistemas em casos específicos. Banerjee contribui para a conceituação de um elemento crítico para a paralelização automática, que é a análise de dependência, além de trazer avanços no tratamento de laços de repetição. No entanto, uma limitação importante dessa abordagem é desconsiderar o caminho de execução, no caso de saltos condicionais, que podem inviabilizar a relação de ordem R proposta de acordo com o comportamento da aplicação. Além disso, esse trabalho pressupõe a existência de índices bem definidos em laços, algo que nem sempre ocorre na prática.

2.3 Estudos secundários

Nesta seção, alguns estudos secundários (*surveys*) são abordados, assim como lacunas conhecidas na área de paralelização automática.

Em 2005, DiPasquale *et al.* (DIPASQUALE; WAY; GEHLOT, 2005) publicaram um estudo comparativo com várias técnicas para paralelização automática. A primeira técnica abordada testou a análise sobre variáveis escalares e vetores. Quanto a variáveis escalares, laços foram analisados de forma a tentar transformar os índices da iteração (quando presentes) em equações lineares. Essa abordagem permitiu paralelizar o laço resolvendo tais equações. No entanto, essa técnica é limitada a laços em programas que funcionam de maneira muito específica. A técnica de análise sobre vetores envolve segregação (ver Seção 3.10.4) de dados em vetores para cada instância de execução. O vetor é analisado e, quando possível, reduzido a uma equação que representa seus valores no tempo. A partir dessa equação é possível determinar se há dependências para esse vetor, dentro de um laço, por exemplo. Se não existem dependências, então o laço pode ser paralelizado. Uma desvantagem dessas técnicas é que elas são restritas a cenários muito específicos o que, em uma aplicação real, pode ser muito raro. Além disso, essas técnicas não abordam ponteiros e outros tipos de funcionalidades clássicas.

A próxima técnica abordada é denominada análise comutativa. A ideia é que se algumas operações, como uma iteração de um laço, apresentam certas equivalências, então a ordem de execução dessas iterações não influencia no resultado final. Essas equivalências são baseadas na propriedade matemática de comutação entre certas operações. Em particular, essa técnica é aplicada sobre objetos em linguagens orientadas a objetos.

Em sequência, aborda-se a técnica de paralelização em alto nível, a qual propõe uma linguagem intermediária para paralelização. Essa linguagem, implementada em uma biblioteca, é referenciada por chamadas dentro da aplicação. Em particular, estruturas típicas das linguagens C e FORTRAN, como vetores, são automaticamente preparadas para serem paralelizadas quando acessadas em laços de repetição. Além disso, outras otimizações são feitas para explorar automaticamente o paralelismo nos procedimentos do programa. A grande limitação dessa técnica está em sua restrição a poucas linguagens e a necessidade de refatorar o código-fonte para utilizá-la.

Em outro trabalho, (FRUMKIN *et al.*, 1998) apresentam comparações entre ferramentas de paralelização automática disponíveis na literatura. A primeira ferramenta abordada é denominada *CAPTools* (IEROTHEOU *et al.*, 1996). Essa ferramenta aceita código FORTRAN sequencial e converte chamadas entre regiões independentes em termos de passagem de mensagens, de forma similar a bibliotecas como MPI e PVM. Nessa ferramenta, uma interface gráfica é fornecida ao usuário para que ele decida sobre o nível de paralelismo e de análise entre outros parâmetros.

A próxima ferramenta é o compilador *PGHPF*. Também voltado para a linguagem FOR-

TRAN, essa ferramenta visa automaticamente paralelizar laços de repetição, criando blocos de aplicação a serem distribuídos entre processadores de uma ou mais máquinas. Esse compilador permite que o desenvolvedor decida sobre o tamanho dos dados a serem separados entre esses blocos, de forma a controlar a quantidade de comunicação realizada, e também a quantidade de dados a serem segregados.

A terceira ferramenta considerada é composta por diretivas de compilação para uma arquitetura específica, em particular os processadores *SGI Origin2000*. Essas diretivas são inseridas no código-fonte, dentro de laços de repetição, para indicar variáveis que não possuam dependências entre si. Essa inserção pode ser feita pelo desenvolvedor, ou por ferramentas disponíveis para essa arquitetura.

As ferramentas consideradas são comparadas no trabalho de Frumkin *et al.* (FRUMKIN *et al.*, 1998), utilizando *benchmarks* de forma a conhecer os ganhos obtidos com o paralelismo. Uma conclusão importante dos resultados é que todas as técnicas superam o desempenho de execuções sequenciais. Porém, o aumento no desempenho depende de ajustes e calibragens feitos pelo desenvolvedor ao empregar todas as técnicas consideradas. Essa restrição denota uma limitação importante dessas ferramentas, já que a necessidade de intervenção do usuário reduz o grau de automação do processo.

2.4 Considerações finais

Este capítulo apresentou trabalhos relacionados à área de paralelização de aplicações, com técnicas automáticas e manuais. Algumas fundamentações matemáticas foram introduzidas, assim como descrições de diversas abordagens para o problema. Esses estudos contribuíram para este trabalho de mestrado no sentido de propiciar um melhor entendimento das etapas envolvidas na paralelização, além de contribuir com ideias para atacar o problema de análise de dependências e particionamento de programas.

Diferentemente das abordagens apresentadas neste capítulo, este trabalho realiza o processo de paralelização de maneira automática, diretamente sobre o código binário. Além disso, a análise de dependências é feita sobre o código como um todo, e não apenas em estruturas particulares. Dessa maneira, a necessidade de intervenção por parte de desenvolvedores e usuários sobre o processo é minimizada.

CAPÍTULO

3

Conceitos sobre a paralelização de aplicações

3.1 Considerações iniciais

Este capítulo apresenta o conceito de paralelização de um programa e as complicações decorrentes desse processo. Também são apresentadas abordagens existentes para realizar essa operação, e outros conceitos pertinentes, além de uma taxonomia. Introduz-se também alguns conceitos da teoria de ordem, pertinentes ao problema tratado.

3.2 Paralelização de aplicações

A paralelização de uma aplicação consiste em transformar um programa sequencial em uma versão que pode ser executada em paralelo, sem comprometer sua semântica (MESSINA; WILLIAMS; FOX, 1994). A versão transformada é formada por um conjunto de unidades funcionais que, durante uma eventual execução, podem ser distribuídas para vários processadores. Essas unidades podem ser blocos de instruções da aplicação original ou versões com semântica equivalente, porém aptas a executar em paralelo.

Uma versão transformada, quando executada em um dado ambiente, deve produzir resultados idênticos ao da aplicação original, não apenas ao final do processo, mas também em eventuais computações intermediárias. Essa característica é denominada consistência da transformação, e precisa ser preservada para uma paralelização bem sucedida.

Para paralelizar a aplicação, várias abordagens são possíveis. O processo pode ser

efetuado tanto sobre código-fonte como sobre aplicações compiladas em linguagem de máquina. A transformação pode ser manual, por meio de análise humana, bem como automática com técnicas algorítmicas. As abordagens podem focar em componentes específicos do programa, ou o todo. Essas possibilidades são descritas em mais detalhes nas próximas seções.

3.3 Análise de instruções

A fase inicial do processo de paralelização consiste em interpretar a semântica do programa, de forma a compreender suas computações. Para tanto, deve-se considerar como a aplicação está apresentada para o processo. Se ela está disponível como código-fonte ou se está compilada. Esses dois casos são detalhados na próxima seção, e suas vantagens e desvantagens são ponderadas.

Após a conclusão da interpretação, uma abstração pode ser gerada, e a presença ou não do código-fonte não possuirá impacto nos processos seguintes de paralelização e particionamento das instruções.

3.3.1 Paralelização sobre código-fonte

A análise de código-fonte consiste em observar a linguagem do programa e buscar maneiras de paralelizar as instruções. Uma abordagem muito comum consiste em o próprio desenvolvedor repensar trechos de código e criar alternativas paralelas. Esses casos são explícitos e, em geral, as unidades paralelas estarão bem descritas no código transformado. Outra maneira consiste em algoritmos que buscam estruturas específicas dentro do código, e quando possível, as transformam em versões que podem ser executadas em paralelo.

A abordagem em código-fonte apresenta uma vantagem importante quando a paralelização é manual, já que esse código possui abstração mais alta, mais fácil de ser compreendido por seres humanos. No entanto, essa abordagem possui complicações para processos automáticos, já que exige maior interpretação das abstrações e da linguagem.

Uma limitação importante deste caso é uma questão prática: muitas vezes, o código-fonte não está disponível. Nesse caso, a paralelização seria inviável. Outra limitação é a restrição de técnicas automáticas para uma linguagem específica.

3.3.2 Paralelização sobre linguagem de máquina

Aplicações compiladas são descritas por uma linguagem de baixo nível, para uma máquina específica. Em geral, essas arquiteturas possuem instruções de baixa abstração. Como consequência, técnicas automáticas precisam de menos instrumentos para interpretar esses cenários, já que não precisam fazer análises léxicas e sintáticas. No entanto,

essas linguagens são, em geral, menos intuitivas para leitura humana e apresentam maior dificuldade para a paralelização manual.

Uma vantagem importante deste tipo de processo é a capacidade de transformar qualquer aplicação de uma dada arquitetura em uma versão paralela, independentemente da disponibilidade do código-fonte. Para linguagens interpretadas, no entanto, o próprio código-fonte pode ser visto como binário. para esses casos, a possibilidade de realizar a paralelização depende da capacidade da ferramenta em tratar o código final.

3.4 Paralelização manual e automática

A atuação do desenvolvedor, ou de um terceiro sobre o código-fonte para transformar uma região sequencial em blocos paralelos caracteriza a paralelização manual (HUANG; STEFFAN, 2011). Nesse cenário, diversas técnicas podem ser empregadas, desde métodos em que estruturas específicas são localizadas e transformadas até casos *ad-hoc*, em que a intuição e conhecimento prévio são aplicados. Enquanto resultados excelentes podem ser obtidos, especialmente quando o domínio do programa é bem compreendido, a abordagem manual apresenta uma limitação evidente: o processo é artesanal, sendo restrito a aplicações específicas, além de necessitar da disponibilidade do código-fonte e conhecimento do desenvolvedor. Para aplicações compiladas a dificuldade aumenta, dado que a falta de abstração torna a interpretação mais complicada.

A paralelização automática, em contrapartida, consiste em utilizar algoritmos para varrer instruções e encontrar regiões que possam ser paralelizadas. Esse processo tipicamente busca por blocos com estruturas pré-definidas a serem modificadas, ou trechos sem dependências entre si para serem particionados. Essa abordagem possui uma vantagem importante, já que não depende da intervenção do desenvolvedor.

Uma outra possibilidade são técnicas híbridas ou semi-automáticas, onde algumas etapas são automatizadas, com intervenção do desenvolvedor nas demais fases.

3.5 Foco da análise e particionamento

A transformação pode ser feita sobre alguns possíveis cenários: entre blocos ou regiões distintas do programa; entre iterações de estruturas de repetição; e, finalmente, em cenários híbridos, em que ocorrem ambos casos.

3.5.1 Paralelização entre blocos

Define-se por bloco básico um trecho sequencial de instruções da aplicação. Nessa abordagem, trechos da aplicação original são extraídos e analisados, de forma a encontrar um conjunto em que os trechos não possuam dependências entre si.

Uma possível paralelização busca o particionamento de blocos básicos independentes, ou seja, regiões em que os resultados das computações envolvidas não possuem interdependência. Nesse caso, a ordem de execução das regiões não interfere no resultado final. Se casos como esses puderem ser detectados, então o particionamento é trivial: cada região é inserida em uma unidade funcional, podendo eventualmente ser executada em paralelo.

Uma outra possível situação consiste em regiões com instruções misturadas. Nesse caso, algumas instruções de um bloco básico dependem de outras em uma outra região, mas não todas. Por exemplo, seja b_1 um bloco básico com n instruções, numeradas de $I_1, \ldots, I_n$, e b_2 um bloco básico com m instruções, $J_1, \ldots, J_m$. Considere que $I_{(i+2)}$ dependa de I_i e $J_{(j+2)}$ de J_j, e que J_1 dependa de $I_{(n-1)}$ e J_2 de I_n. Dessa maneira, existe uma dependência entre b_2 e b_1, porém apenas entre partes distintas dos blocos. Em particular, o subconjunto $J_j \in J$ para todo j par depende de $I_i \in I$ para todo i par. O mesmo vale para os subconjuntos com i e j ímpares. Para esse cenário, as instruções que possuem dependências entre si podem ser extraídas dos blocos básicos em que se encontram, formando conjuntos separados. Nesse caso em particular, conjuntos seriam formados por instruções I_i e J_j para i e j separados por paridade. O particionamento então é feito transformando esses grupos em unidades funcionais, que podem ser executadas em paralelo.

3.5.2 Paralelização entre iterações

Um caso específico e importante na paralelização trata de estruturas de repetição. Essas regiões de código executam uma ou mais vezes, com uma característica bem definida: as instruções executadas são sempre as mesmas, variando apenas os dados sendo trabalhados, e eventuais caminhos de execução. É importante notar que as funções internas do laço podem variar entre iterações. Por exemplo, um caminho condicional pode chamar procedimentos diferentes dependendo dos valores na memória. Em outros casos, como polimorfismo dinâmico e estático, mensagens distintas são traduzidas para métodos distintos. Nesse caso, a análise precisa considerar todos os caminhos possíveis.

Para efeito da análise de dependências, pode-se considerar a estrutura como um bloco básico, que é executado diversas vezes em sequência. Com essa abordagem, o processo pode ser realizado de maneira similar à paralelização entre blocos, com uma ressalva: o número de iterações nem sempre é conhecido. Essa característica leva a complicações nas decisões de análise. Uma possibilidade é considerar o número infinito, e analisar todas as possíveis dependências para esse caso. Quando tal análise é possível, as técnicas de particionamento em blocos podem ser aplicadas. Dessa maneira, instruções que dependam de outras em iterações seguintes são extraídas para formar grupos paralelizáveis.

Uma característica muito importante em estruturas de repetição é que o número de

iterações pode ser muito alto. De fato, para algumas aplicações, o tempo gasto iterando em laços pode ser muito maior do que o tempo gasto em blocos sem repetição. Como consequência, é desejável para a paralelização que as iterações sejam o mais independentes entre si possível, de maneira a formarem unidades paralelas. Quando esse cenário é possível, o grau de paralelismo (definido em 3.8) obtido pode ser muito alto, com um ganho importante para o programa. No entanto, nem sempre essa independência existe. Para contornar essas restrições, algumas técnicas podem ser aplicadas, com o intuito de remover dependências e permitir esse tipo de particionamento. Algumas dessas técnicas são apresentadas no Capítulo 4, junto com uma discussão sobre suas aplicações.

Além dessa abordagem existem outras maneiras de tratar a análise e o particionamento de laços de repetição, como análise de índices e separação de acessos à memória. No Capítulo 2, apresenta-se uma abordagem que considera os acessos no laço como um sistema de equações lineares. Apresenta-se também uma discussão sobre como resolver o particionamento, encontrando soluções para o sistema.

3.5.3 Paralelização híbrida

O particionamento de regiões contendo trechos com e sem repetição pode ser feito de maneira híbrida, com conceitos dos dois tipos de paralelização. Quando laços de repetição são abordados como blocos executados repetidamente de maneira sequencial, eles podem ser analisados junto com outros blocos sem repetição. Dessa maneira, grupos de instruções podem ser extraídos de todas as regiões, de acordo com as dependências existentes entre elas, e particionados em unidades funcionais.

3.5.4 Unidades de particionamento

O particionamento consiste na criação de unidades funcionais, cada uma contendo blocos de instruções prontos para serem executados. O resultado do processo cria unidades que não possuem dependências entre si, o que permite sua execução em paralelo. No decorrer da análise e particionamento, unidades geradas em algumas regiões de código podem, no entanto, depender de unidades geradas em outras fases do processo de paralelização. Essa característica permite que enquanto algumas regiões da aplicação apresentem algum grau de paralelismo, outras podem não ter tal característica. Também é possível o surgimento de gargalos, dado que algumas regiões da aplicação apresentem pouco ou nenhum grau de paralelismo.

O tamanho de cada unidade pode variar de acordo com a abordagem utilizada. Por questões práticas, como o tempo para enviar cada unidade para um processador ou outras eventuais restrições, pode ser desejável criar unidades maiores ou menores. Essa decisão depende da estratégia escolhida, e possui implicações no tamanho dos blocos a serem analisados e unidades mínimas a serem geradas. O tamanho consiste, em geral, no

número de instruções dentro da unidade. Caso haja laços de repetição dentro da unidade, o número de iterações estimado pode ser utilizado para definir o tamanho.

3.6 Escopo da análise e paralelização

O particionamento e análise são feitos sobre regiões de instruções bem definidas da aplicação, escolhidas de acordo com a abordagem utilizada. A escolha possui implicações importantes sobre o processo e, em geral, decorre de um compromisso entre as questões que devem ser consideradas: o desempenho do processo e o grau de paralelismo obtido. É importante notar que cada região pode ser subdividida em um ou mais blocos para a análise. Para efeito de simplificação, esse tipo de região será referenciada como uma super-região (HANK *et al.*, 1993).

Quando uma super-região é definida, ela torna-se uma caixa-preta dentro do processo de análise para outras super-regiões. Em particular, todas as dependências internas são expressas em função de suas entradas e saídas. Uma super-região pode conter um desvio para uma ou mais super-regiões, inclusive a si mesma. A análise de dependências e particionamento são feitos sobre todas as suas instruções. Uma super-região muito grande pode conter muitos caminhos de execução e, portanto, muitas dependências. Dessa maneira, a análise tenderá a ser demorada. Em contrapartida, mais instruções podem ser comparadas individualmente, aumentando as possibilidades de posterior particionamento.

Uma outra característica importante, mas menos evidente, é o número de combinações entre casos de dependências. Em particular, alguns caminhos de execução podem permitir um número muito alto de possibilidades para o resultado de computações. Como exemplo, caminhos condicionais em certas configurações podem criar cenários com um número exponencial de resultados possíveis. Quando ocorre essa explosão combinatória, uma abordagem comum é assumir todos os valores possíveis para essas computações. Esses casos podem criar dependências muito abrangentes, inviabilizando o particionamento. Super-regiões muito grandes possuem tendência a apresentar um número maior desses casos, podendo reduzir o grau de paralelismo obtido.

3.7 Preservação de consistência

Para que a semântica da versão transformada da aplicação seja equivalente à original, algumas propriedades devem ser observadas. Em particular, as dependências entre instruções devem ser mantidas. Quando esse caso ocorre, a ordem original de execução dessas instruções deve ser preservada, caso contrário, o resultado das computações será diferente, e a aplicação perderá consistência e terá seu resultado comprometido. Mais informações sobre a natureza das dependências são apresentadas na Seção 3.10.

A consistência é uma métrica crítica para o processo de paralelização. A não observância dessa característica implica em uma eventual distorção do programa original, o que não é aceitável para o processo. Dessa maneira, análises e métodos que garantam a preservação da consistência são um ponto central da paralelização.

3.8 Grau de paralelismo

Uma métrica relevante sobre aplicações transformadas é o seu grau de paralelismo (GOSDEN, 1966), que é o número de unidades paralelas possíveis em uma execução. Esse valor pode variar dentro de um mesmo programa, dependendo da região considerada. Em particular, alguns blocos podem possuir alto grau de paralelização, enquanto outras regiões podem ter um grau muito pequeno ou até mesmo inexistente.

Embora, idealmente, seja desejável o maior grau de paralelismo possível, para alguns casos um paralelismo muito alto pode ser desvantajoso. Por exemplo, o ganho de desempenho para uma única instrução paralela pode não compensar a sobrecarga de desempenho (*overhead*) para distribuí-la. Quanto exatamente é esse impacto varia de arquitetura para arquitetura, e deve ser considerado no momento de encapsular as instruções em unidades paralelas. As variações sobre essa quantidade dependem principalmente do número de dependências existentes e que possam ser removidas entre as regiões consideradas. Para alguns casos, as instruções podem possuir muitas interdependências, impossibilitando sua separação. Em outros casos, total separação é possível, permitindo uma alta paralelização.

3.9 Teoria de ordem – grafos de ordem parcial

Um aspecto importante da paralelização é a geração de grafos como subprodutos das análises envolvidas. Em particular, os grafos de fluxo, que denotam a ordem de execução da aplicação, e de dependências, que definem quais instruções podem ser executadas em paralelo com outras, e quais devem ser executadas sequencialmente. Como a ordem é importante para esse último caso, esse grafo pode ser tratado como um grafo de ordem parcial (WARD, 1954). Além disso, o particionamento envolve técnicas sobre grafos bem estabelecidas em teoria de ordem. Por esse motivo, esta seção introduz alguns conceitos e técnicas dessa teoria.

Seja U um conjunto, e seja (E,F) um par de elementos $E,F \in U$. Seja AU o conjunto de todos os pares (E,F) possíveis em U. Define-se a relação $\oplus$ sobre U como um relacionamento entre dois de seus elementos. Denomina-se $\oplus(E,F)$ a relação $\oplus$ sobre um par (E,F). $\oplus$ pode ser definida sobre todos os pares em AU, ou apenas sobre um subconjunto $SAU \in AU$ de pares. Se $\oplus$ for definida sobre todo AU ela é dita uma relação total, caso contrário $\oplus$ é uma relação parcial. $\oplus$ é definida como uma relação de ordem

quando para $\oplus(E,F)$, $\oplus$ definir uma direção, denotada por $E \oplus F$. Nesse caso, diz-se que E relaciona F. Além disso, $\oplus$ deve obedecer às seguintes propriedades:

- $\forall E \in U$, $\oplus(E,E)$ é definido (reflexiva);
- $\forall E,F,G \in U$, se $E \oplus F$ e $F \oplus G$, então $E \oplus G$ (transitiva);
- $\forall(E,F)$ se $E \oplus F$ existe, então $F \oplus E$ não pode ocorrer (anti-simétrica).

Seja G um grafo direcionado definido sobre um conjunto U e uma relação de ordem $\oplus$. Cada vértice $V \in G$ é definido como um elemento $E \in U$. Uma aresta $A(V,T)$ entre os vértices $V(G)$ e $T(G)$ existe se e somente se $\oplus(E,F)$ é definido, em que E é o elemento do vértice V e F o de T. A direção da aresta depende da ordem do relacionamento, portanto se $E \oplus F$, a aresta sairá de V em direção a T e vice-versa. Como consequência das características de $\oplus$, o grafo possui algumas propriedades: se $A(V,T)$ e $A(T,X)$ existem, então $A(V,X)$ existe; se $A(V,T)$ existe então $A(T,V)$ não existe e $A(V,V)$ existe. Um grafo de dependências pode ser interpretado também dessa maneira. Se os vértices forem instruções, então as arestas definem a ordem de execução entre elas. Se $\oplus$ não estiver definido entre duas instruções (ou elementos), então não há dependência entre eles.

Sobre grafos de ordem parcial define-se o conceito de anticadeia (COMTET, 1974) (*antichain*), que consiste em um conjunto de vértices sobre os quais $\oplus$ não está definido (ou seja, entre os quais não há arestas). A cardinalidade da maior anticadeia existente no grafo define a sua largura W. É interessante observar que o maior grau de paralelismo possível em uma região da aplicação é o valor W de seu grafo de dependências. Todas as anticadeias nesse grafo consistem em instruções sem dependências entre si. Se as anticadeias puderem ser determinadas, então o particionamento consiste em separar seus elementos. Em particular, é desejável que dentro desses grupos $\oplus$ esteja definida como uma relação de ordem total. Os algoritmos conhecidos para a obtenção de anticadeias possuem complexidade exponencial (JOHNSTON, 1976), uma limitação que precisa ser contornada para sua aplicação no particionamento de aplicações.

3.10 Taxonomia

Nesta seção, são apresentadas taxonomias para a paralelização de aplicações. Primeiramente é detalhada a taxonomia de Bernstein (BERNSTEIN, 1966) sobre dependências entre instruções, em seguida uma taxonomia sobre paralelismo implícito e explícito. Em seguida, o conceito de super-regiões é detalhado, o qual é definido sobre grafos de fluxo de execução de programas. Posteriormente, o conceito de segregação, o qual produz cópias idênticas de trechos de memória de um programa a fim de evitar problemas de consistência. Finalmente, apresenta-se o conceito de execução especulativa, o qual faz a tentativa

de executar instruções futuras a fim de aumentar o desempenho de aplicações com o custo de, eventualmente, desfazer tais operações para manter consistência.

3.10.1 Taxonomia para dependências entre instruções

A dependência entre duas instruções ocorre quando a semântica da aplicação resulta da ordem de execução entre elas, ou seja, quando uma execução fora de ordem pode comprometer a consistência do algoritmo. Uma maneira de classificar esse tipo de dependência foi proposta por Bernstein (BERNSTEIN, 1966). Esse tipo de dependência refere-se ao acesso a variáveis, ou seja, a dados, e pode ser dividido em três categorias: *read-after-write* (RAW) (leitura após escrita), *write-after-read* (WAR) (escrita após leitura) e *write-after-write* (WAW) (escrita após escrita).

Dependências do tipo RAW são provenientes de leituras em variáveis (registros ou posições de memória) feitas após uma escrita nessa mesma variável. Como o resultado esperado da leitura é o que foi escrito anteriormente, não é possível executar consistentemente essa leitura antes da escrita.

As dependências WAR são produzidas por escritas após a leitura de uma variável. Nesse caso, a escrita sobrescreve o valor, o que comprometeria o resultado da leitura caso essas instruções executassem fora de ordem. Porém, é possível evitar esses problemas de consistência com algumas estratégias, como salvar o valor em outro lugar, ou alterar a posição de escrita (e subsequentes acessos), caso a semântica permita. O último tipo, WAW, refere-se à dependência da ordem entre escritas. Essencialmente, para manter a consistência, a última escrita deve prevalecer ao final da execução dessas instruções. Em paralelização, quando esse caso é identificado, todas as escritas exceto a última podem ser descartadas, sem prejudicar a semântica da aplicação.

Na Figura 3.1, é apresentado um exemplo de uma dependência do tipo RAW. Para esse caso, suponha que não ocorra nenhuma modificação no registador **edx** entre as instruções 2 e 4 de um programa em linguagem de montagem para processadores da família de arquiteturas x86. Dessa forma, na instrução 2 escreve-se o valor 5 na região de memória apontada por **edx**. Na instrução 4, essa região de memória é lida. Nesse tipo de dependência, espera-se que na instrução 4, o valor 5 seja retornado. A Figura 3.2 mostra um exemplo da dependência do tipo WAR. Novamente, assume-se que *edx* não é modificado entre as instruções 2 e 4. Nesse caso, a instrução 2 precisa ser executada antes da 4. Se a instrução 4 for executada primeiro, o valor contido na região de memória apontada por *edx* seria sobrescrito, e uma execução posterior a instrução 2 poderia retornar um valor inválido. Um exemplo de dependência do tipo WAW é mostrado na Figura 3.3. Nesse caso, o valor na região apontada por *edx* é alterado na instrução 2 e na 4. Uma posterior leitura nessa região, no entanto, precisa mostrar o valor escrito na instrução 4, ou seja, 8 nesse exemplo. Observe que para esse tipo de dependência, as instruções 2 e 4 podem

ser executadas fora de ordem, desde que seja garantido que o valor escrito pela instrução 4 predomine para uma próxima leitura.

```
1 func:
2       mov [edx],5
3       ...
4       mov eax,[edx]
```

Figura 3.1: Dependência do tipo RAW.

```
1 func:
2       mov eax,[edx]
3       ...
4       mov [edx],5
```

Figura 3.2: Dependência do tipo WAR.

```
1 func:
2       mov [edx],5
3       ...
4       mov [edx],8
```

Figura 3.3: Dependência do tipo WAW.

3.10.2 Paralelismo transparente e não transparente

Uma outra taxonomia em termos do tipo de paralelismo obtido em aplicações distingue o paralelismo transparente do não transparente (YARDIMCI, 2008). O paralelismo transparente é aquele em que um usuário tem a ilusão de que a aplicação executou de forma sequencial, quando de fato ela executou em paralelo. Nesse caso, a paralelização da aplicação é transparente para o usuário final ou para o desenvolvedor do software. A paralelização pode ocorrer tanto em tempo de compilação como durante a execução, por meio de módulos do sistema operacional ou empregando alguma outra abstração intermediária.

O paralelismo é dito não transparente quando ele é evidente para o usuário ou para o desenvolvedor. Nesse cenário, o desenvolvedor pode criar o paralelismo de maneira direta, por meio de funcionalidades do sistema operacional ou da plataforma alvo da aplicação, ou utilizando ferramentas que buscam alterar o código-fonte para introduzir regiões paralelizáveis. Para o usuário, o paralelismo fica explícito quando a aplicação é distribuída entre processadores de forma manual, ou automática mas com informações claras sobre a localidade dos processos.

Este trabalho tem como objetivo gerar uma ferramenta para paralelização transparente, de forma a suprimir o trabalho extra necessário por parte de desenvolvedores e usuários para atingir o paralelismo. Ao automatizar o processo de distribuição, a ferramenta visa facilitar o processo, aumentando a abrangência de programas sendo paralelizados.

3.10.3 Super-blocos

(HANK *et al.*, 1993) introduzem o conceito de super-blocos (*superblocks*), os quais são definidos por blocos básicos de instruções que não possuem entradas oriundas de outros blocos básicos. Uma entrada é definida como um salto condicional, ou uma consequência natural da execução em direção ao bloco básico. Na Figura 3.4, observa-se um bloco de instruções com um salto entrando entre as instruções *Inst 2* e *Inst 3*, vindo de uma outra região da aplicação. Na Figura 3.5, um grafo de fluxo de execução de uma aplicação é apresentado, no qual os vértices representam blocos de instruções, e as arestas caminhos possíveis de execução. Em particular, os vértices marcados em cinza representam o bloco observado na Figura 3.4. Esse bloco é particionado em dois vértices, pois há mais de um caminho para atingir a segunda região, após a instrução *Inst 2*. Nesse caso, a entrada representada na Figura 3.4 torna-se a aresta representada em tracejado no grafo.

Para ser considerado um super-bloco, os blocos não podem conter entradas (exceto alguma entrada em seu início), mas podem conter saídas, ou seja, saltos para fora do bloco. No grafo de fluxo, um conjunto de vértices é considerado um super-bloco quando, para cada vértice, no máximo uma aresta entra, independentemente do número de arestas saindo. Na Figura 3.6, tem-se um exemplo de um grafo de fluxo com uma super-bloco definido pelos vértices marcados em cinza.

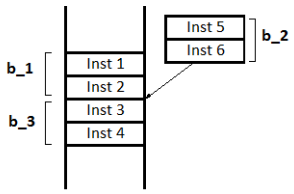

Figura 3.4: Exemplo de bloco com entrada lateral (Adaptado de (HANK *et al.*, 1993)).

3.10.4 Segregação

DiPasquale *et al.* (DIPASQUALE; WAY; GEHLOT, 2005) introduzem a ideia de segregação de dados. Essa técnica, aplicada à paralelização, tem como objetivo particionar os dados acessados pela aplicação e separá-los em regiões distintas. Esse particionamento pode envolver a clonagem dos dados ou apenas o particionamento de um vetor. O objetivo dessa

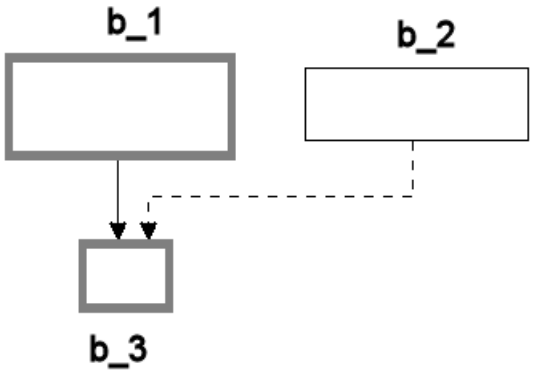

Figura 3.5: Exemplo de bloco com entrada lateral em um grafo de fluxo de execução.

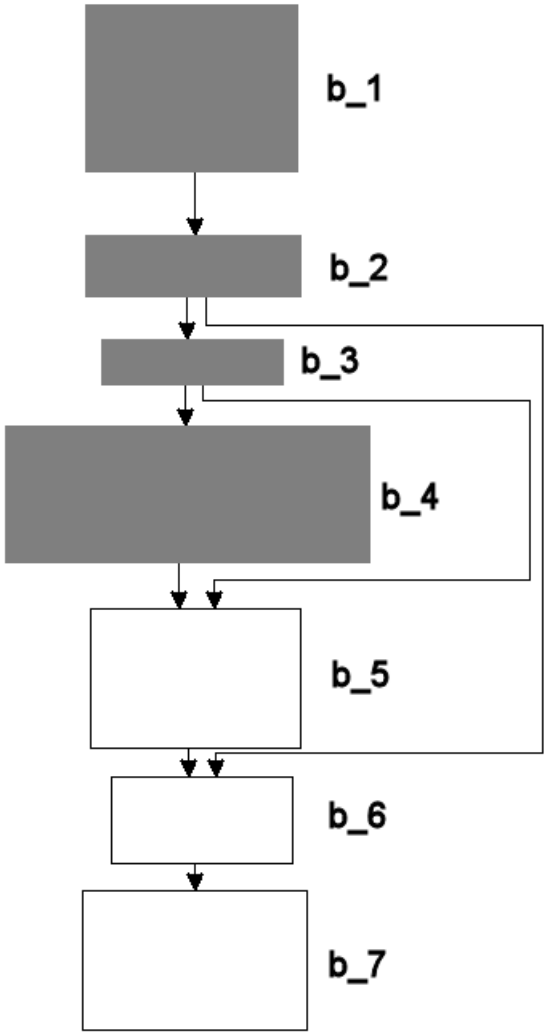

Figura 3.6: Exemplo de super-bloco em um grafo de fluxo de execução.

abordagem é criar imagens de dados, de forma que aplicações paralelizadas possam trabalhar sobre elas de maneira independente. De acordo com a semântica da aplicação, a segregação não só é possível, como também elimina alguns tipos de dependências que

poderiam comprometer o paralelismo.

3.10.5 Execução especulativa

É possível que, para uma dada aplicação, duas regiões distintas de código possuam ou não dependência entre si, de acordo com parâmetros da aplicação. Esses parâmetros são tipicamente desconhecidos em tempo de compilação, como por exemplo, entradas de usuários, saídas de *RNG* (geradores de números aleatórios), etc. Nesse caso, uma abordagem possível é a execução especulativa. Ela consiste em executar essas regiões em paralelo e, caso os parâmetros fornecidos caracterizem a dependência, a execução da região dependente é descartada. Caso não ocorra a dependência, a execução é preservada. Em geral, os testes que determinam se ocorreu ou não uma dependência são baseados em informações fornecidas por uma análise anterior. Outro fator importante para essas execuções é que nem sempre elas podem ser retroagidas, ou seja, descartadas sem perdas. Dessa maneira, a aplicação dessas técnicas depende do caso considerado. Nos Algoritmos 3.1 e 3.2, apresenta-se um exemplo de dependência entre blocos que ocorre apenas com parâmetros específicos. Em particular, o Bloco 2 depende do Bloco 1 quando o valor X, provido pelo usuário, é $X < 10$. Nesse caso, o Bloco 2 lê uma região de memória escrita no Bloco 1. Dessa maneira, os dois blocos poderiam ser executados em paralelo, de maneira especulativa. Porém, caso seja detectado que $X < 10$, a execução do Bloco 2 precisaria ser descartada e reiniciada.

Algoritmo 3.1: Exemplo de execução especulativa – Bloco 1.

```
1: X = Entrada do Usuário
2: if X < 10 then
3:     Escrever X na posição de memória 0x100
4: else
5:     Escrever X na posição de memória 0x200
6: end if
```

Algoritmo 3.2: Exemplo de execução especulativa – Bloco 2.

```
1: Y = Leitura da posição de memória 0x100
```

3.11 Considerações finais

Este capítulo apresentou conceitos em paralelização automática e em teoria de ordem, pertinentes ao problema abordado. Apresentou-se também as opções de abordagens feitas neste trabalho. Em particular, por paralelização implícita, automática e sobre o binário compilado. Essa escolha visa minimizar o esforço necessário para realizar a paralelização,

permitindo que usuários com pouco ou nenhum domínio do problema possam se beneficiar de paralelismo em suas aplicações.

Código 3.1: Exemplo de instruções da arquitetura Intel x86.

```
main:
    push ebp                    55
    xor eax, eax                31 C0
    mov ebp, esp                89 E5
    push edi                    57
    cmp edx, 0FFFFFF57          83 FA 57
```

CAPÍTULO

4

Abordagem para a paralelização automática de aplicações

4.1 Objetivo

O objetivo deste trabalho foi o de construir um conjunto de ferramentas que receba como entrada uma aplicação compilada para a arquitetura Intel x86, e fosse capaz de retornar uma aplicação equivalente com o máximo possível de estruturas paralelizadas automaticamente. A aplicação de saída deve preservar completamente a semântica da aplicação original, sendo equivalente em todas as entradas e saídas. Nesse sentido, o conjunto de ferramentas buscou analisar as dependências de instruções, detectar pontos passíveis de paralelização, e reconstruir a aplicação com as novas instruções obtidas. Para atingir o objetivo final, foram estabelecidas metas específicas, usando como base os estudos feitos anteriormente:

- Construir um desmontador para uma aplicação Intel x86 para transformação em uma linguagem intermediária;
- Construir um analisador capaz de gerar o grafo de fluxo de controle da aplicação;
- Construir um analisador capaz de gerar o grafo de dependências da aplicação;
- Construir um módulo que, automaticamente, elimine dependências quando aplicável;
- Construir um módulo que, com base no grafo final de dependências, fosse capaz de particionar o programa em unidades paralelas.

4.2 Metodologia empregada

Para atingir o objetivo proposto, as seguintes atividades foram planejadas e executadas: i) estudo da arquitetura Intel x86 (BENNETT, 2011); ii) implementação de um desmontador; iii) análise de fluxo de execução; iv) análise estática de dependências; v) particionamento dos grafos de dependência para obtenção de paralelismo; iv) remontagem de uma versão paralela da aplicação.

Iniciou-se com o estudo da arquitetura Intel x86, instruções e organizações típicas de aplicações. O objetivo dessa fase foi o de compreender as instruções da arquitetura, como é realizado o *encoding* em *bytes* e quais são as variações possíveis de acesso a registradores e memória. Além disso, estudou-se estruturas típicas de montagem da aplicação, de seu fluxo de execução, de estabelecimento e chamada de funções, para melhor compreensão das aplicações sendo abordadas. O foco desse estudo foram os principais compiladores em linguagem C disponíveis, para delinear as primeiras estratégias de paralelização. No entanto, deve-se ressaltar que o foco do trabalho não se restringe apenas à paralelização de aplicações criadas por esses compiladores. Esse estudo serve como um ponto de partida para uma generalização para qualquer aplicação escrita para a plataforma x86.

Na etapa seguinte, projetou-se um desmontador (*disassembler*) para aplicações Intel x86 para, em seguida, obter uma abstração da aplicação. Essa abstração deve apresentar os acessos tanto à memória como a registradores, explicando se esses são relativos à escrita ou leitura. Deve-se também obter os pontos em que ocorrem saltos para outras regiões do programa, condicionais ou não, determinando sob quais circunstâncias eles ocorrem. Quando possível, funções devem ser delimitadas, determinando seu ponto de entrada e seus pontos de retorno. Nesse caso, todas as chamadas para cada função devem ser identificadas. Nota-se que, nesse momento, nem todas as chamadas podem ser identificadas, em particular nos casos em que as chamadas são dinâmicas. Esses casos são abordados nas próximas fases.

Como terceira etapa analisou-se o fluxo de execução das aplicações. Após a abstração ser obtida, o fluxo de execução deve ser obtido na forma de um grafo direcionado, o qual serve como base para análises subsequentes. Esse grafo consiste na sequência de instruções, em que cada vértice representa um bloco sequencial, e as arestas correspondem aos possíveis caminhos, especialmente em caso de desvios (*branching*) e laços de repetição.

A seguir, foi conduzida uma análise estática sobre as dependências entre instruções. As dependências devem ser modeladas, também, na forma de um grafo, de acordo com o comportamento observado das instruções. A ideia é analisar cada instrução, tentando-se obter os valores esperados em registradores/memória para aquele ponto da execução. Essa análise é estática, ou seja, leva apenas em consideração as instruções tal como obtidas na fase de desmontagem, sem que a aplicação seja executada. Essa análise deve ser feita de maneira a obter o grafo mais preciso possível, porém, entende-se que em uma

análise estática há limitações do que pode ser extraído. Em particular, é impossível saber o resultado de leituras de entrada e saída do sistema, como valores providos por usuários, conteúdo de arquivos, etc. Caminhos condicionais também dependem de condições, cujos valores, em geral, não estão disponíveis nessa etapa.

Em seguida, foram realizados particionamentos e modificações sobre o grafo de dependência entre instruções. Técnicas precisam ser aplicadas sobre tais grafos de forma a isolar grupos de vértices que podem ser executados em paralelo. Em particular, utilizou-se métodos de separação de grafos em anticadeias, descritos nas próximas seções. Ainda nesse sentido, buscou-se identificar pontos em que modificações permitiram aumentar o paralelismo do programa em questão.

Como etapa seguinte, buscou-se montar uma versão paralelizada da aplicação sob análise. Os particionamentos obtidos na fase anterior precisaram de uma nova análise, especialmente para identificar a viabilidade de paralelização. Não há interesse em isolar blocos muito pequenos, com uma única instrução, por exemplo. Após essa seleção, a aplicação final é obtida. A ideia foi a de gerar um documento de informações sobre a análise para paralelização, identificando pontos em que ela pode ocorrer. Além disso, insumos devem ser gerados para uma paralelização dinâmica, em especial um estudo sobre como obter e empregar informações que só podem ser obtidas dinamicamente, para servir de base para um sistema operacional, ou outra plataforma, realizar a distribuição dos componentes paralelos.

4.3 Estudo da arquitetura Intel x86

O primeiro passo para abordar o problema compreendeu um estudo detalhado da arquitetura Intel x86 (INTEL, 2010), adotada neste trabalho. Esse estudo visou a melhor compreensão do formato de instruções e tipos de acesso à memória e a registradores. Para isso, foram utilizados documentos fornecidos pelo fabricante de processadores, assim como referências disponíveis na literatura. Além disso, um estudo adicional foi realizado sobre aplicações existentes, considerando tanto código-fonte conhecido como indisponível. Essas aplicações sofreram o processo de desmontagem para posterior estudo.

Dessa maneira, foi possível conhecer estruturas típicas de montagem de códigos binário. Nesse sentido, buscou-se conhecer como são organizadas as funções, como elas são chamadas, como podem ser delimitadas, além de entender como são tipicamente montadas estruturas como laços, blocos condicionais, entre outras. Observou-se, nesse processo, que a arquitetura Intel x86 possui algumas peculiaridades que são relevantes para este trabalho. Principalmente, o fato das instruções não possuírem tamanho fixo, tornando impossível localizar uma instrução no programa sem conhecer as instruções anteriores. Essa restrição criou uma dificuldade para a realização da análise, no sentido em que o tratamento de uma região específica da aplicação depende do conhecimento das anteriores.

O entendimento da arquitetura Intel x86 também objetivou compreender e modelar o fluxo de execução de um programa. Por execução, entende-se a sequência natural das instruções, uma vez que o programa tenha sido iniciado no processador. Como existe a possibilidade de desvios (*branches*) condicionais, esse fluxo pode, em determinados pontos, tomar diferentes direções. Além disso, o fluxo pode retornar a pontos já executados. Esses casos compõem, em uma visão de maior abstração, o que se conhece por laços e blocos condicionais. Dessa maneira, esse fluxo pode, se alguns cuidados forem tomados, ser representado por grafos. Nesse caso, os vértices consistem de blocos de execução que não podem ser interrompidos (nota-se que interrupção é usada de maneira não estrita nesse contexto, pois apenas considera a interrupção em uma linha de execução, sem considerar interrupções do processador, escalonamento, etc.), ou seja, não possuem nenhum tipo de desvio condicional, com possível exceção de sua última instrução. Essa exceção é permitida pois o término desse bloco pode ser delimitado por um desvio desse tipo. Nota-se, nesse caso, que desvios não condicionais são permitidos dentro desses blocos, já que determinam um sentido de execução obrigatório. Essa particularidade cria uma dificuldade para a construção do analisador, a qual é abordada a seguir.

Os acessos à memória e registradores na arquitetura Intel x86 são feitos de maneiras variadas. Em geral, o código da instrução pode tanto determinar o tipo de acesso, quanto representar instruções genéricas, em que o tipo de acesso é provido em parâmetros posteriores. Dada a necessidade dessa arquitetura em apresentar compatibilidade reversa com sistemas antigos, de 8-bits e 16-bits, e também com os sistemas alvo da arquitetura, de 32-bits, os parâmetros contabilizam o tamanho do acesso (nota-se que arquiteturas de 64-bits não foram abordadas até o momento, mas o estudo pode ser facilmente generalizado para tais sistemas). Em caso de acesso à memória, observa-se que o alvo pode ser um endereço fixo, com os tamanhos citados, ou um endereço contido em um registrador, obtido de maneira dinâmica, e são classificados como acessos diretos e indiretos, respectivamente. Observa-se também que esse endereço pode ser provido por funções de alocação de memória (específicas para cada plataforma, mas que podem ser vistas como funções genéricas de alocação), podem ser fixos, em geral providos pelo compilador, ou relativos, em que um índice é adicionado formando o endereço de acesso. Essa característica torna o universo de instruções possíveis e sua codificação demasiadamente complexos.

No Código 4.1, um exemplo de instruções x86 com seus respectivos mnemônicos é apresentado. À direita, observa-se os *bytes*, em notação hexadecimal, da instrução, com seu código e, quando aplicável, seus argumentos. Na coluna da esquerda, estão os mnemônicos associados. Como exemplo, toma-se a primeira instrução, *push ebp*, que determina que o conteúdo do registrador *ebp* seja inserido no topo da pilha de memória. Essa instrução é expressa por um único *byte*, 55, em que 50 representa a instrução *push* e 05 o registrador alvo. No entanto, a segunda instrução *xor eax, eax*, necessita de dois *bytes* para ser representada. Essa instrução, que realiza uma operação lógica *XOR* entre o re-

gistrador *eax* e ele mesmo, efetivamente zerando o registrador, é representada pelo *byte* 31, que denota um ou-exclusivo (*XOR*), e o *byte* $C0$ denota os dois registradores utilizados, além do tipo de acesso feito. Em particular, $C0$ torna-se 11000000 em notação binária. Os dois primeiros bits, quando estão setados (11), definem que o acesso é feito em dois registradores. Dessa forma, os demais 6 bits passam a codificar os registradores. Como *eax* é denotado por três bits zeros 000, a operação é feita entre o registrador *eax* e ele mesmo. Observa-se, nesses casos, que o tamanho de cada instrução varia de acordo com a operação.

No Código 4.2, são apresentadas as mesmas instruções dispostas na memória. Os *bytes* são dispostos em um vetor sem nenhum alinhamento, na sequência em que se encontram. Suponha agora que se deseja encontrar a posição da instrução *push edi* (a quarta no Código 4.1), descrita pelo *byte* 57, no vetor. Nota-se que essa tarefa é impossível se as instruções imediatamente anteriores não forem analisadas, pois, não só existem dois *bytes* com o mesmo valor no vetor, com significados distintos, como também é impossível localizar a posição exata da instrução sem conhecer o tamanho das anteriores. De forma similar, observando um *byte* aleatório no vetor, é impossível saber que instrução ele descreve. Isso pode ser observado com o próprio *byte* 57, que em sua primeira ocorrência descreve apenas uma instrução (*push edi*), e na segunda é parte de uma outra (*cmp edx, 0FFFFFF57*), como valor.

Código 4.1: Exemplo de instruções da arquitetura Intel x86.

```
main :
    push ebp                    55
    xor eax, eax                31 C0
    mov ebp, esp                89 E5
    push edi                    57
    cmp edx, 0FFFFFF57          83 FA 57
```

Código 4.2: Exemplo de instruções da arquitetura Intel x86 em memória.

```
55 31 C0 89 E5 57 83 FA 57
```

A análise dos *bits* de uma instrução também apresenta níveis variados de complexidade. No Código 4.3, encontra-se um outro exemplo das instruções x86. Todas as instruções apresentadas são variações de uma mesma instrução, no caso *mov*, representada pelo *byte* $8B$. Porém, para cada instrução, o argumento determina os tipos diferentes de acesso, por registador e por memória. Na arquitetura Intel x86, nesse tipo de instrução, o segundo *byte*, ou primeiro operando, é interpretado de acordo com seus dois bits mais significativos. Se o valor desses bits for 11, então o acesso é feito em dois registradores, definidos nos bits seguintes. Se o valor for 10, então o acesso é de uma região da memória para um

registrador. A base da memória é um registrador, definido nos bits seguintes. Para 10, um segundo operando é necessário para determinar um deslocamento à partir da base, com tamanho fixo de 4 *bytes*. Se o valor for 01, então o mesmo ocorre, porém o deslocamento é dado por um operando de apenas 1 *byte*. Se o valor dos bits mais significativos for 00, então não há deslocamento. No entanto, há exceções para essas regras. Para algumas combinações específicas há deslocamento mesmo com valor 00 e, em alguns casos, multiplica-se um valor ao deslocamento, para mover-se em um vetor de tamanho arbitrário, por exemplo.

Código 4.3: Instruções da arquitetura Intel x86 – tipos de acesso.

```
func:
    mov eax,[edi+010]          8B 45 10
    mov eax,eax                8B C0
    mov eax,ebx                8B C3
    mov eax,[edi+010]          8B 85 10 00 00 00
```

4.4 Desmontador e grafo de fluxo de controle

Com as informações obtidas no estudo da arquitetura Intel x86, procurou-se modelar uma ferramenta para uma análise inicial do código, de forma a criar uma abstração para a análise de dependências. Para melhor utilizar as informações das aplicações, buscou-se, primeiro, planificar as informações de acesso à memória e registradores, de forma a eliminar a complexidade dessas instruções. Criou-se, portanto, uma abstração simples, em que cada acesso é descrito por poucos atributos: i) se o acesso é uma leitura ou escrita (em caso de ambos ele é dividido em vários acessos); ii) em que o acesso é realizado, se em memória ou registradores; iii) caso seja em memória se o endereço é fixo ou se está em um registrador e em qual; iv) em caso de registradores, qual deles é acessado; v) em caso de memória, também se observa se o acesso é menor que 32-bits. Com essa abordagem, não é mais necessário conhecer qual instrução exatamente é utilizada para o acesso. Também omite-se o tamanho do acesso para o caso de registradores, já que para fins de paralelização isso é irrelevante, e os acessos são todos assumidos como 32-bits. Para realizar esse processo, no entanto, um desmontador é necessário, de forma a processar a semântica das instruções e criar a abstração.

Para realizar o processo de desmontagem de uma aplicação compilada para uma arquitetura da família Intel x86, alguns obstáculos foram observados (o foco deste trabalho foi dado para arquivos binários no formato Microsoft *PE-32*, mas pode ser estendido a outros formatos bem estabelecidos como *ELF*). Como já mencionado, o tamanho das instruções não é fixo, o que vincula a análise de uma instrução à análise das anteriores, pois é impos-

sível saber onde ela começa sem esse processo. Além disso, pontos de início de trechos, tais como funções, precisam ser conhecidos pelos dois motivos descritos a seguir. O primeiro é evitar a análise repetida de regiões. O segundo motivo é que funções podem não ter outras imediatamente anteriores, de forma que a análise seria inviável, pois não poderiam ser diretamente localizadas. Em decorrência dessas limitações, percebeu-se que um desmontador correto não poderia ser realizado separadamente da análise de fluxo, ou seja, da criação do grafo de execução. Portanto, decidiu-se modelar e construir o desmontador em conjunto com essa análise em uma primeira fase.

O desmontador foi projetado para operar em várias etapas. Na primeira etapa, as instruções devem ser lidas individualmente, e de acordo com seu significado, uma ação é tomada. O primeiro significado extraído trata da construção do grafo. Para instruções de desvio condicional (por desvio entende-se um salto simples, sem chamadas de função, as quais são tratadas separadamente), entende-se que um vértice do grafo chegou ao fim, e que novos vértices devem ser criados ou vértices já visitados recebem novas arestas. Nesse último caso, se o vértice já foi analisado, entende-se que um laço foi encontrado, e o vértice recebe uma marcação com essa informação. Para um desvio não condicional, não existe o fim do vértice, porém é necessário verificar se o alvo já foi visitado. Observa-se, nesse caso, que essa nova aresta poderia não ser conhecida anteriormente, quando o vértice foi visitado. Dessa maneira, o vértice pode ser dividido em dois. Além disso, após o desvio não condicional, outras instruções podem existir e, durante a análise, é necessário decidir se ela segue o salto não condicional ou a ordem das instruções. Decisões incorretas podem omitir partes da aplicação, gerando análises incompletas.

Para abordar essas complicações da arquitetura Intel x86, algumas regras simples de análise foram utilizadas: i) a análise não segue os saltos, que são empurrados em uma pilha para análise posterior; ii) grava-se o ponto de entrada e saída de cada vértice, de forma que acessos no meio desses são identificados e tratados de maneira apropriada; iii) finalmente, acessos a locais já visitados são identificados, de forma a não ocorrer acessos repetidos. Em caso de instruções de chamadas de função, a nova função é armazenada em uma pilha de funções, e analisada após o término da função atual (entende-se que todo o código da aplicação pertence a alguma função). A análise é iniciada pelo ponto de entrada da aplicação, recuperado do arquivo binário correspondente. Dessa maneira, apenas funções que podem ser atingidas à partir do ponto de entrada são analisadas, e eventuais instruções inatingíveis são omitidas. Porém, nessa fase, acessos dinâmicos a funções não são resolvidos e, portanto, pode-se dizer que a análise ainda é incompleta. O próximo item obtido nessa etapa é o tamanho das instruções. Para contornar o problema de localização de instruções, um mapa de espalhamento (*hash*) é criado, associando o endereço da instrução a seu tamanho. Esse mapa é feito para auxiliar em fases futuras.

Após o término da primeira fase, obtém-se uma lista com todas as funções da aplicação, um mapa com o tamanho das instruções, e um grafo direcionado, em que os vértices

representam blocos de execução e as arestas os caminhos possíveis. Além disso, o grafo traz informações extras como pontos de entrada e saída de estruturas de repetição.

4.5 Linguagem intermediária

Essa etapa apresentou diversos problemas decorrentes da complexidade inerente da arquitetura Intel x86. Esse cenário criou uma dificuldade importante para o processo de desmontagem. Em particular, as otimizações presentes nesses sistemas apresentam situações nas quais diversas instruções com tamanhos e codificações diferentes possuem a mesma funcionalidade. Além disso, a arquitetura possui macro-instruções, que realizam várias operações em uma única instrução. Esse cenário torna a implementação das análises demasiadamente complexa. Com o intuito de simplificar esse processo, considerou-se uma abordagem de análise em alto nível, onde a ferramenta trabalharia sobre uma abstração e não sobre a aplicação diretamente. Outro fator interessante desse caminho seria o aumento da independência da ferramenta em relação à arquitetura. Se uma abstração eficiente pudesse ser criada, então a paralelização poderia ser efetuada sobre qualquer linguagem que pudesse ser transformada adequadamente. Motivado por esses fatores, definiu-se uma nova forma de abstração representada por uma linguagem intermediária.
Para a criação da linguagem intermediária buscou-se atingir alguns critérios:

- Semântica para representação total das funcionalidade da arquitetura Intel x86. Como o foco deste trabalho são essas aplicações, é desejável ter a cobertura completa;
- Simplicidade. A ideia aqui é representar as funcionalidades com um conjunto mínimo de instruções;
- Micro-instruções. As instruções devem ser mínimas, ou seja, realizam uma única função simples. Nesse caso, seria uma função aritmética ou uma atribuição/leitura de um registrador/memória;
- Inversível para a arquitetura Intel x86. Qualquer instância dessa linguagem deve ser traduzível para uma aplicação Intel x86 equivalente.

Com esse foco, confeccionou-se a linguagem intermediária, sobre a qual as análises de fluxo, dependências e posterior particionamento são realizados. Idealmente, a equivalência deveria existir para outras linguagens existentes, mas como essas não estão no escopo deste trabalho, esses casos não foram abordados. No Apêndice A está disponível a especificação técnica e alguns exemplos da linguagem. Uma observação importante é que o conjunto possível de funcionalidades está restrito a alguns grupos específicos, em

particular: aritméticos; acessos a memória/registradores; funções de fluxo, como saltos e chamadas.

A introdução da nova linguagem exigiu um remodelamento do desmontador, de forma a incluir uma nova etapa de tradução. Após a leitura da aplicação de entrada, cada instrução agora é transformada em uma ou mais instruções equivalentes na linguagem intermediária. As ferramentas de análise também foram modificadas, para aceitar as instruções simplificadas, ao invés de instruções Intel x86. Uma vantagem importante desse processo foi permitir a mudança de foco do trabalho sobre a arquitetura Intel x86 para o estudo das técnicas de análise e paralelização propriamente ditas. Outra consequência interessante é a possibilidade, futuramente, da escrita de tradutores de outras linguagens para a linguagem intermediária, permitindo que outras aplicações se beneficiem das técnicas de paralelização.

4.6 Análise de dependências

Com a fase de desmontagem e criação da linguagem intermediária concluída, buscou-se os primeiros entendimentos sobre como realizar a paralelização, mais especificamente, em como obter blocos independentes. Para tanto, determinou-se que o primeiro foco deveria ser dado à análise de dependências, ou seja, obter quais acessos são feitos e como eles dependem entre si.

As dependências entre as instruções podem ser classificadas de algumas maneiras. A primeira delas diferencia a dependência como direta ou indireta. Nesse caso, se uma instrução *C* depende de uma instrução *B*, e *B* depende de uma instrução *A*, diz-se que *C* possui uma dependência direta de *B*, *B* depende diretamente de *A*, mas *C* possui uma dependência indireta de *A*. Uma sucessão de dependências indiretas forma uma cadeia ordenada, que, a princípio, determina a sequência de execução das instruções contidas. É importante notar, no entanto, que em alguns casos as dependências podem ser removidas ou alteradas, e que essas cadeias não são a forma definitiva obrigatória para os blocos paralelizados. Uma segunda classificação para as dependências é definida por Bernstein (BERNSTEIN, 1966), que são descritas em detalhes no Capítulo 2. Para alguns desses tipos, modificações podem ser suficientes para suprimir a dependência.

Para abordar o problema da análise de dependência, decidiu-se criar um novo grafo direcionado, separado do grafo de fluxo de controle, em que os vértices representam instruções, ou blocos de instruções, e as arestas dependências entre elas. Esse grafo foi idealizado para a fase seguinte, em que particionamentos são feitos para obter blocos independentes entre si. O grande obstáculo para esse processo é conhecer quais endereços de memória são acessados, pois o acesso pode ser feito a um endereço apontado por um registrador. A princípio, não se conhece o valor do registrador, o que impossibilita conhecer quais regiões são acessadas por uma instrução. Sem esse conhecimento, não é possível determinar se

existe uma dependência, já que em pontos distintos da execução, registradores diferentes podem apontar para a mesma região de memória. Na Figura 4.1, um exemplo de um grafo direcionado de fluxo de controle é apresentado. Os vértices correspondem a blocos de instruções, e as arestas os caminhos possíveis. As arestas sólidas representam caminhos incondicionais ou obrigatórios. As arestas tracejadas representam caminhos após saltos condicionais, enquanto as pontilhadas correspondem a caminhos após instruções de salto condicional, quando o salto não é realizado. A aresta mais espessa representa um laço de repetição (ou um ciclo no grafo).

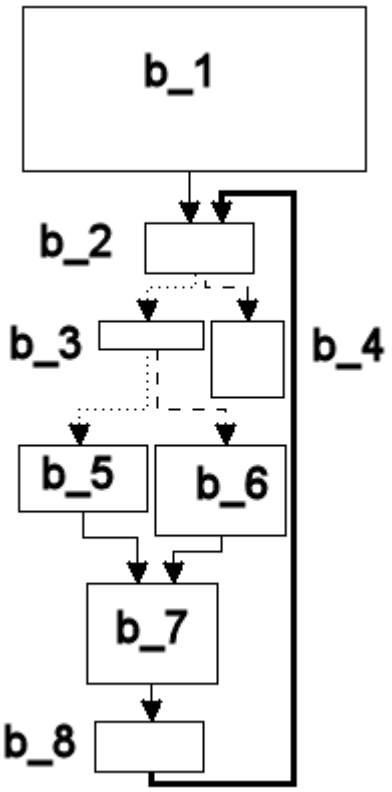

Figura 4.1: Exemplo de grafo de fluxo de execução.

Ainda nesse sentido, entradas desconhecidas podem determinar a existência ou não de dependências, pois esses valores podem, por exemplo, determinar um índice em um vetor, que pode gerar dependências com outras instruções. Essas limitações dificultam, ou até mesmo impossibilitam, a criação de um grafo de dependências completo e totalmente preciso. No entanto, percebeu-se que um grafo de menor precisão, mas consistente, pode ser suficiente para atingir algum grau de paralelismo, desde que algumas restrições sejam observadas. Por precisão, nesse contexto, entende-se que apenas dependências reais sejam representadas. Se dependências falsas forem representadas, entende-se que a precisão do grafo diminui. Por consistência entende-se que todas as dependências devem ser representadas. Se algumas dependências estiverem faltando, o grafo é dito inconsistente.

É importante observar que a precisão pode ser flexibilizada, ou seja, pode-se observar vários graus de precisão, como alto ou baixo. A consistência, no entanto, é completa, e o grafo pode ser consistente ou não. A motivação dessas definições é que um grafo impreciso pode apresentar baixo ou nenhum paralelismo, o que não prejudica a semântica da aplicação. Um grafo inconsistente, porém, prejudica a semântica da aplicação, o que não é de interesse deste trabalho. Dessa forma, pode-se obter um grafo consistente, por exemplo, assumindo-se que todas as instruções dependem entre si, o que levaria também ao grafo mais impreciso. Assim, pode-se considerar, em casos de valores desconhecidos, todos os valores possíveis, criando-se grafos de menor precisão que, como pressupõem todas as situações, não perdem a consistência. No entanto, assumir valores desconhecidos para qualquer caso, especialmente registradores que apontam para a memória, geraria grafos muito imprecisos, prejudicando o grau de paralelismo obtido ao final.

Nesse sentido, decidiu-se investigar abordagens para tentar resolver valores possíveis para esses registradores, em particular, resolver qualquer valor possível de maneira estática. Ainda nesse contexto, percebeu-se que seria interessante gerar insumos para que uma eventual análise dinâmica, ou em tempo real, pudesse determinar valores desconhecidos com base em entradas fornecidas durante a execução. Para abordar esse problema, decidiu-se usar um método de geração de expressões matemáticas para as instruções. A ideia foi iniciar a análise ao fim de uma função, tentando gerar expressões possíveis para a saída. Por saída, nesse caso, entende-se o valor final de todos os registradores, e também de todos os endereços de memória acessados no decorrer da função. Observa-se que as expressões podem ser descritas tanto em função de valores escalares constantes providos diretamente em instruções, como em função de valores de entrada, valores obtidos à partir da memória ou valores desconhecidos, fornecidos por usuários/agentes externos. Para realizar esse processo, a análise é iniciada na última instrução da função (ou últimas, caso exista múltiplos retornos), e retrocede em direção ao seu início.

É importante lembrar que, apesar de ser possível múltiplos finais para a função, existe um único começo, em que a análise se encerra. Para cada instrução, um símbolo é gerado, designando o valor de um registrador ou um endereço de memória naquele instante. Esse símbolo serve de entrada para instruções analisadas anteriormente, de forma a transformar o significado do valor presente na variável. Dessa forma, em um dado momento da análise, o valor final de registradores e endereços é representado em função desses símbolos, que por sua vez também são representados por outros símbolos, e assim sucessivamente. Ao término da análise, os últimos símbolos serão valores de entrada, seja por argumentos da função, seja por valores existentes na memória ou valores globais.

Em caso de múltiplos vértices, a ordem da análise é importante. Todos os vértices paralelos devem ser analisados antes do vértice "pai". Esse processo permite que vários possíveis valores sejam obtidos para cada expressão. Dessa forma, ao chegar ao vértice "pai", símbolos para um registrador, por exemplo, podem conter várias expressões, cada

uma oriunda de um caminho distinto da função. Observa-se que, em caso de muitos caminhos, algumas expressões podem obter um número muito grande de possibilidades. Em particular, se para cada dois vértices dois valores distintos puderem ser gerados para uma expressão, o número final de valores será 2^n em que n é o número de pares de vértices. Nesses casos, a expansão exponencial dos valores deve ser identificada, e excluída, ou seja, assume-se um valor desconhecido para aquela parte da expressão. De forma semelhante, entradas externas ao programa também devem ser pressupostas como desconhecidas, o que pode incluir saídas de funções externas, como bibliotecas. Em caso de estruturas de repetição, uma nova semântica é adicionada. Inspirada em expressões regulares, operadores de repetição são inseridos.

Uma vez analisada uma função, a relação entre entradas e saídas pode ser usada diretamente nos casos em que essa função é chamada, ou seja, como parte da análise. Assim, quando chamadas de funções são encontradas durante a geração de expressões, os símbolos são os produzidos por essa subfunção. Nesse caso, portanto, se a subfunção ainda não foi analisada, a função atual é empurrada em uma pilha, e a nova é analisada. Em uma fase posterior, esses valores podem ser gerados para bibliotecas inteiras, e usados como valores previamente fornecidos em análises de aplicações.

As Figuras 4.2 e 4.3 apresentam um exemplo de uma aplicação desmontada. Esse exemplo foi obtido através da aplicação *IDA Starter Edition* (HEX-RAYS..., 2013), e ilustra um grafo de fluxo de execução, em que os vértices contêm o código desmontado de uma aplicação. Para efeito de ilustração, na Figura 4.4 é apresentado o código-fonte original, escrito em linguagem *C*. Essa ferramenta auxiliou na validação do aplicativo construído neste trabalho, principalmente no entendimento dos grafos de execução, e na codificação de instruções na arquitetura Intel x86.

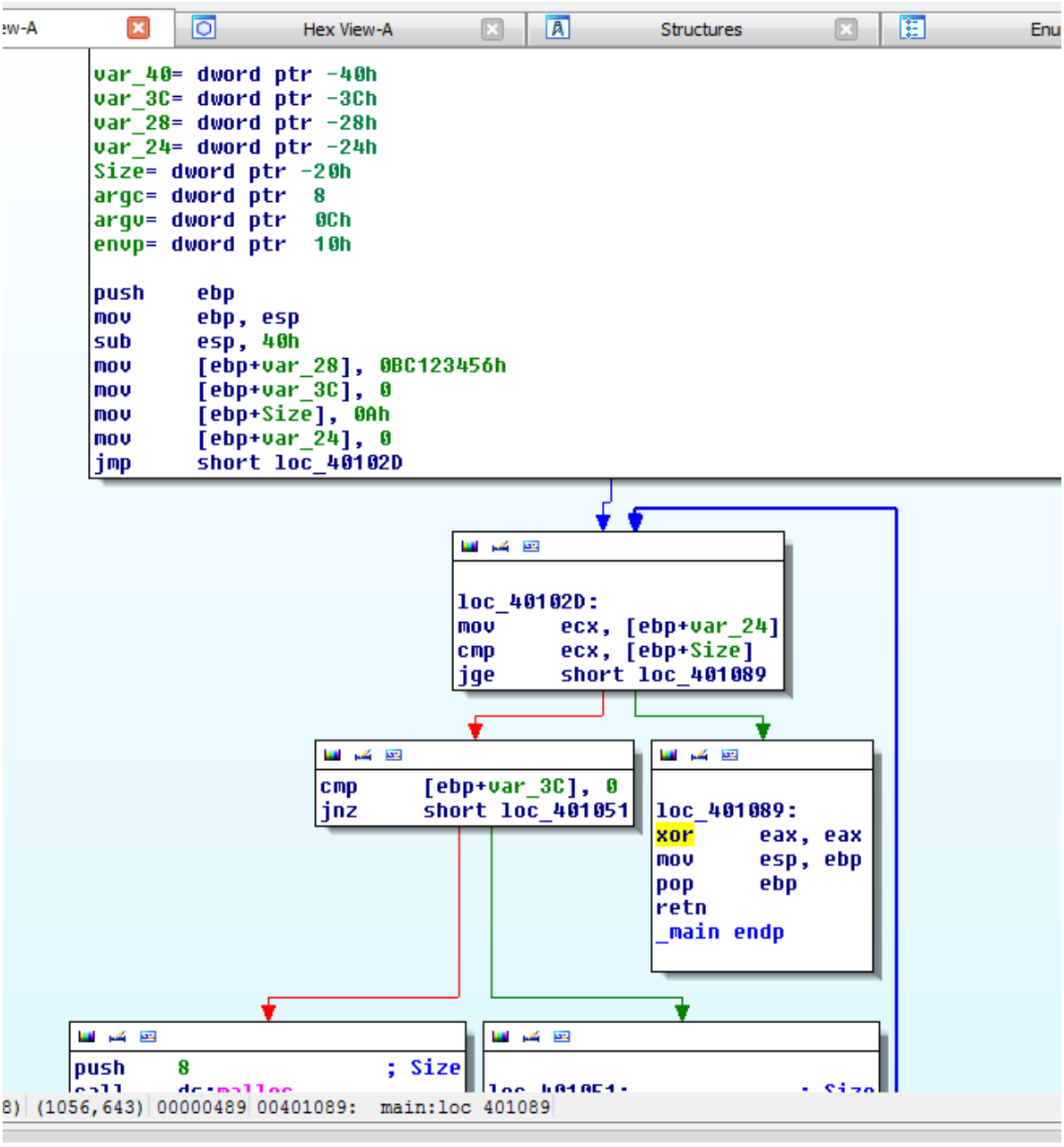

Figura 4.2: Exemplo de aplicação desmontada pela ferramenta *IDA Starter Edition* (Parte 1).

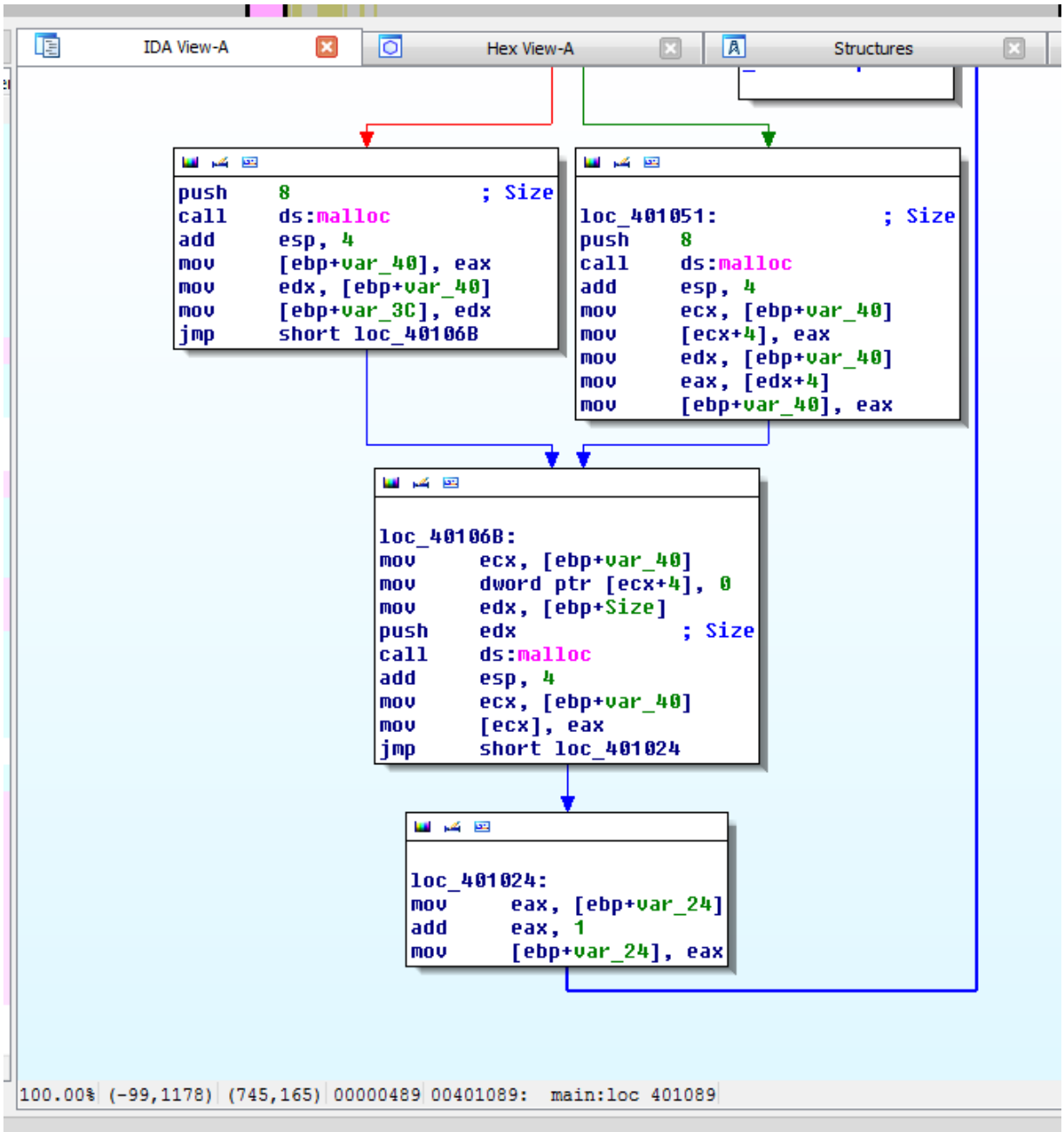

Figura 4.3: Exemplo de aplicação desmontada pela ferramenta *IDA Starter Edition* (Parte 2).

```
int xx = 0xBC123456;
int * a[7];
int i;
int n;
double s1, s2;

link_list * head = 0;
link_list * t;

n = 10;
for (i=0; i < n ; i++) {
   if (!head) {
      head = t = malloc(sizeof(link_list));
   }
   else {
      t->next = malloc(sizeof(link_list));
      t = t->next;
   }
   t->next = 0;
   t->data = malloc(n);

}
```

Figura 4.4: Exemplo de algoritmo em linguagem *C* para uma aplicação.

A obtenção das expressões representativas de um bloco foi feita utilizando um processo de substituição de símbolos. O processo é iniciado ao final do bloco, analisando as instruções em direção ao seu início. A ordem de análise já é conhecida nessa fase, tendo sido obtida na análise de fluxo. Dessa maneira, instruções de saltos não são relevantes e podem ser ignoradas. Instruções de chamada de função, no entanto, possuem semântica relevante, e serão abordadas mais adiante. Assim, para o primeiro momento, apenas instruções aritméticas e de acesso à memória são consideradas. Uma característica importante dessas instruções é que elas podem ser representadas por expressões matemáticas bem definidas. O Código 4.4 apresenta um exemplo dessa representação, na qual à esquerda instruções da arquitetura Intel x86 aritméticas e de acesso são convertidas em expressões matemáticas equivalentes, apresentadas na coluna da direita. A análise procede obtendo expressões desse tipo, e classificando cada elemento como um símbolo. Em cada iteração, os símbolos obtidos são substituídos pelos novos símbolos obtidos na instrução anterior. Como exemplo, suponha a sequência de instruções apresentada na Figura 4.5. A análise inicia na linha 4, e a primeira expressão, denominada e_1 é obtida: `ebx = ebx XOR ebx`. À partir da linha 3 obtém-se a expressão e_2: `eax = [ebx]`. Na linha 2, e_3: `ebx = ebx + 4`. Nesse momento, existe um novo valor para ebx. Como existem outras expressões anteriores com dependência à esquerda de ebx, o processo retorna e realiza as substituições necessárias. Assim, e_1 torna-se `ebx = (ebx+4) XOR (ebx+4)` e e_2 `eax = [ebx+4]`.

Finalmente, na linha 1 obtém-se e_4: `ebx = 0x50000`. Novamente as substituições são realizadas, e obtém-se e_3 `ebx = 0x50000+4`, e_2 `eax = [0x50000+4]` e e_1 `ebx = (0x50000+4) XOR (0x50000+4)`. Finalmente, para cada elemento as expressões mais antigas possuem prioridade. Por exemplo, e_4 e e_1 são valores para `ebx` e e_1 é tomado como o valor predominante para `ebx`. Essa prioridade representa o conteúdo final desse elemento, ao fim do bloco. A Figura 4.6 contém as expressões finais da análise. Além desse processo, outras informações importantes para a análise são geradas. Em particular, para cada expressão obtida, armazenam-se quais registradores são acessados, e se ocorre uma leitura ou escrita. Determinam-se também, com base nos símbolos substituídos, quais regiões de memória foram acessadas. O Código 4.5 apresenta um exemplo, baseado na análise anterior. A primeira coluna apresenta o código original, a segunda as expressões obtidas. O Código 4.6 apresenta mais uma coluna, descrevendo quais acessos foram realizados em cada linha, e se foi uma escrita ou uma leitura. Essas informações serão úteis ao término dessa fase, para a montagem do grafo de dependências.

Código 4.4: Instruções da Arquitetura Intel x86 – Representações matemáticas.

```
func:
    add eax,0xC0          eax = eax+192
    sub eax,ebx           eax = eax−ebx
    imul 4                eax = eax∗4
    mov ebx,[ecx]         ebx = [ecx]
```

```
1 mov ebx,0x50000
2 add ebx,4
3 mov eax,[ebx]
4 xor ebx,ebx
```

Figura 4.5: Exemplo de instruções para a análise de dependências.

```
1 ebx = (0x5000+4) XOR (0x5000+4)
2 eax = [0x5000+4]
```

Figura 4.6: Exemplo de expressões finais para a análise de dependências.

Código 4.5: Exemplo de resultado de uma análise de dependências.

```
func:
    1 mov ebx,0x50000          ebx = 0x50000
    2 add ebx,4                ebx = 0x50000+4
    3 mov eax,[ebx]            eax = [0x50000+4]
    4 xor ebx,ebx              ebx = (0x50000+4) XOR (0x50000+4)
```

Código 4.6: Exemplo de resultado de uma análise de dependências com acessos a registradores ou memória. **W** denota uma escrita, e **R** uma leitura.

```
func:
    1 mov ebx,0x50000          W(ebx)
    2 add ebx,4                W(ebx), R(ebx)
    3 mov eax,[ebx]            W(eax), R(0x50004), R(ebx)
    4 xor ebx,ebx              W(ebx), R(ebx)
```

A abordagem de análise apenas de acessos e operações aritméticas, no entanto, não resolve o problema de dependência entre procedimentos, em particular, no caso onde ocorre a chamada de uma outra função. Para resolver esses cenários, o bloco chamado é tratado como um grupo de expressões, definidas sobre suas entradas e saídas. Em particular, toma-se como entrada todos os registradores e valores na memória antes do procedimento ser chamado, e como saída esses valores após o seu término. As expressões geradas consistem nos valores que sofreram alterações. Dessa forma, elas são utilizadas como símbolos para a análise do bloco atual. Uma limitação importante dessa técnica é que se torna impossível determinar a dependência de uma instrução do bloco sendo analisado, com outras dentro do procedimento. Nesse caso, é possível saber apenas se existe uma dependência com o procedimento como um todo.

Para a análise completa de uma aplicação, porém, é necessário determinar quais blocos serão analisados individualmente, e também determinar como abordar laços de repetição e caminhos condicionais.

Caminhos condicionais constituem fluxos de execução em que o programa escolhe entre vários caminhos, de acordo com alguma condição. Durante a fase estática é, em geral, impossível conhecer qual caminho será tomado. Dessa maneira, para obter-se as dependências corretas, é necessário considerar todas as opções. Como consequência, regiões com trechos condicionais poderão conter expressões compostas de várias possibilidades para seus valores. Um exemplo desse caso é apresentado na Figura 4.7. Nesse caso, dependendo do valor de ebx, a aplicação saltará da linha 3 para a linha 6, ou continuará adiante. Na prática, esse bloco forma uma estrutura *se-senão-então* onde será executada ou a linha 4 ou a linha 6. Dessa maneira, ao final o registrador eax terá ou o endereço

$0x10000$ ou $0x20000$. Para representar esse caso, introduz-se o operador **'ou'** às expressões, representado pelo símbolo $|$. Na Figura 4.8, apresenta-se o resultado da análise desse bloco. Para as fases posteriores da análise que utilizam os resultados desse bloco, todos os valores possíveis para cada elemento são considerados. É importante notar que, para uma eventual fase dinâmica de paralelização, é desejável armazenar as condições de entrada do bloco condicional. Nesse caso, seria possível determinar qual caminho foi tomado, permitindo melhores decisões de paralelização.

```
1 mov eax,0
2 test ebx
3 jc 6
4 mov eax,0x10000
5 jmp 7
6 mox eax,0x20000
7 mov ebx,[eax]
```

Figura 4.7: Exemplo de bloco condicional.

```
ebx = [0x10000|0x20000]
eax = 0x10000|0x20000
```

Figura 4.8: Exemplo de expressões para um bloco condicional.

4.7 Análise de dependências em laços

Por fim, laços de repetição devem ser tratados. Uma característica importante desses casos é que eles podem ser representados como um mesmo bloco executado várias vezes em sequência. Dessa forma, as dependências internas são sempre iguais para cada iteração. Se esse bloco for interpretado como um procedimento, pode-se considerar o laço como sendo várias execuções da mesma função (é importante lembrar que todos os caminhos condicionais são considerados), com cada iteração possuindo expressões de dependências constantes. Com essa abordagem, no entanto, dependências de instruções específicas entre iterações não são explicitadas, já que estão implícitas nos resultados de cada análise.

Para ilustrar esse cenário, considere o código apresentado na Figura 4.9. Na linha 2 uma atribuição é feita dentro de uma região de memória. Em 3, o registrador ecx é incrementado de 1. Nas linhas 4 e 5, ecx é comparado com 20, se for menor a execução retorna para a linha 2. Dessa maneira, por 20 iterações, a memória é acessada pelo índice contido em ecx. Em outras palavras, esse código representa um laço sobre um índice i em ecx, que varia de 0 a 20. Para efeito de análise, esse código pode ser interpretado como uma sequência de execuções da parte interna do laço. A Figura 4.10 apresenta o código equivalente. É

importante notar que nem sempre é possível conhecer, durante a fase estática, o número total de iterações. No entanto, é possível assumir um número infinito de repetições. Nesse caso, assume-se que todas as posições possíveis são acessadas.

Para o exemplo atual, todos os valores de $eax + i$, para todo i, seriam considerados como escritos. No entanto, se os limites eax forem conhecidos (no caso de alocações de memória, por exemplo), i pode ser limitado por esse valor. Uma vez que o laço tenha sido quebrado em várias repetições, a análise de dependências prossegue de maneira similar à feita sobre blocos sem repetição. Uma característica interessante desse tipo de laço, porém, é que a existência de índices de acesso cria dependências entre todas as iterações. Um exemplo desse caso é apresentado mais adiante, assim como uma discussão sobre como contornar essa limitação.

```
1 mov ecx,0
2 mov [eax+ecx],[ebx+ecx]
3 add ecx,1
4 test ecx, 20
5 jl 2
```

Figura 4.9: Exemplo de laço de repetição.

```
1 mov ecx,0
2 mov [eax+ecx],[ebx+ecx]
3 add ecx,1
4 mov [eax+ecx],[ebx+ecx]
5 add ecx,1
...
40 mov [eax+ecx],[ebx+ecx]
41 add ecx,1
```

Figura 4.10: Exemplo de laço de repetição desenrolado em execuções sequenciais.

O resultado da análise de dependências é um grafo direcionado, em que cada vértice representa uma instrução e cada aresta, uma dependência. Para simplificar o processo de particionamento posterior, arestas transitivas redundantes são removidas, ou seja, se uma instrução A depende de uma instrução B, e B por sua vez depende de uma instrução C, a aresta entre A e C é removida. Essa remoção é trivial, bastando seguir os caminhos entre os vértices no grafo e verificando se há caminho repetido. Nesse caso, é importante notar que o grafo final retém a propriedade transitiva, as arestas são removidas apenas para simplificação. Os grafos de dependências são gerados para cada bloco separadamente, e representam apenas as dependências internas.

Para se efetuar a análise, decidiu-se por interpretar cada procedimento ou função como sendo uma super-região de análise (ver Seção 3.6). Essa escolha deve-se a alguns fatores:

- A arquitetura Intel x86 possui entradas e saídas de procedimento bem definidas, sendo fácil a separação;

- Procedimentos costumam encapsular funções bem definidas e autocontidas o que apresenta boa capacidade de paralelização.

Com os blocos da aplicação definidos, a análise é efetuada e as dependências obtidas são analisadas para a criação do grafo. Suas relações são mapeadas por dois tipos distintos: pelos acessos à memória e pelos acessos a registradores (tanto escrita como leitura). A sequência dos acessos é organizada de acordo com as dependências de Bernstein. Dessa forma, as escritas e leituras são ordenadas para os mesmos registradores e endereços de memória. Se um acesso de leitura é feito após um acesso de escrita (RAW), então existe uma dependência. De forma similar, se uma escrita é feita após uma leitura (WAR), existe também uma dependência que, porém, pode ser removida caso o valor original seja copiado. Finalmente, se ocorre uma escrita após escrita (WAW), a dependência existe, mas pode ser mitigada desde que o valor final prevaleça na execução. É interessante armazenar nesse momento o tipo exato de dependência, para permitir posteriormente a aplicação de técnicas de eliminação.
Como exemplo, considere novamente o código e a análise contidos nos Códigos 4.5 e 4.6. A primeira coluna apresenta um bloco de instruções, a segunda do primeiro Código as expressões matemáticas equivalentes. A coluna da direita do segundo código apresenta os acessos contidos em cada passo. Após a análise, determina-se que a instrução 4 possui um acesso de escrita e leitura em ebx. Buscando o próximo acesso nesse registrador, obtém-se uma leitura na instrução 3. Dessa maneira, observa-se uma escrita após a leitura e, portanto, uma dependência. Note que seria possível, posteriormente, salvar o valor em ebx antes da instrução 3, permitindo que 4 execute em paralelo. No entanto, sem essa alteração, a dependência persiste. De forma similar, observa-se uma outra escrita em ebx, na instrução 2. Assim, uma dependência RAW existe entre 2 e 3. Como uma leitura ocorre em 2 e outra escrita em 1, o mesmo tipo de dependência existe entre 1 e 2. A Figura 4.11 apresenta o grafo obtido ao final da análise, com as arestas de transição removidas. Observe que esse código em particular possui baixo potencial para paralelização, já que apresenta total dependência entre as instruções. Outro exemplo de análise é apresentado na Figura 4.12. Nesse caso, a instrução b_29 não possui dependência com 26 e b_33, e poderia executar em paralelo.
Um caso interessante são os índices de repetição em laços. Considere novamente o exemplo contido na Figura 4.9. As Figuras 4.7 e 4.9 apresentam a análise desse bloco. Observe que a instrução 3 depende de 2, pois ocorre uma leitura em ecx em 3 e uma escrita em 2, formando um RAW. Da mesma forma, 4 depende de 3, por um WAR. De maneira similar, essas dependências ocorrem para cada iteração, sempre em ecx. Como esse registrador é o índice, no entanto, uma técnica simples de eliminação pode ser aplicada: todos os

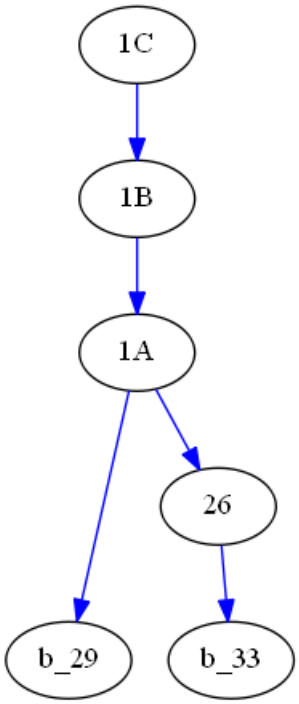

Figura 4.11: Exemplo de grafo de dependências após a análise I.

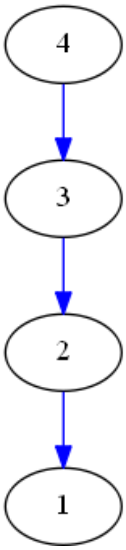

Figura 4.12: Exemplo de grafo de dependências após a análise II.

valores de ecx podem ser pré-calculados, e aplicados para cada caso. Essa abordagem decorre das expressões matemáticas obtidas nas atribuições. Observe que elas evidenciam a falta de dependência de ecx. A nova análise é apresentada no Código 4.9. As dependências existentes entre 2 e 3 e entre as demais iterações desapareceram, uma vez que não há mais acesso a ecx. No entanto, dependências ainda existem entre as instruções 3 e 5, 5 e 7, etc. Como essas dependências são WAW, no entanto, apenas o valor final importa. Assim, atribuindo-se o valor final de ecx na última iteração, todas as dependências entre iterações desaparecem. Dessa maneira, é possível paralelizar o laço por completo, sem perda de consistência. Essa abordagem pode ser estendida para qualquer caso em que as expressões obtidas demonstrarem que uma dependência de um registrador ou posição de memória não existe, apesar de evidenciado nas instruções. É importante notar, no entanto, que para a aplicação dessa técnica, a iteração não pode ser tratada como um procedimento. Para contornar essa limitação, a ferramenta passou a abordar o laço como uma sequência de réplicas de seu código.

Código 4.7: Exemplo de resultado da análise de um laço.

```
func:
    mov ecx,0                          ecx = 0
    mov [eax+ecx],[ebx+ecx]            [eax+0] = [ebx+0]
    add ecx,1                          ecx = 0+1
    mov [eax+ecx],[ebx+ecx]            [eax+0+1] = [ebx+0+1]
    add ecx,1                          ecx = 0+1+1
    ...                                ...
    mov [eax+ecx],[ebx+ecx]            [eax+0+1+1...+1] = [ebx+0+1+1...+1]
    add ecx,1                          ecx = 0+1+1...+1+1
```

Código 4.8: Exemplo de resultado da análise de um laço com acessos a registradores ou memória. **W** denota uma escrita, e **R** uma leitura.

```
func:
    mov ecx,0                          W(ecx)
    mov [eax+ecx],[ebx+ecx]            R(ebx,ecx,[ebx],eax); W([eax])
    add ecx,1                          W(ecx);
    mov [eax+ecx],[ebx+ecx]            R(ebx,ecx,[ebx+1],eax); W([eax+1])
    add ecx,1                          W(ecx);
    ...                                ...
    mov [eax+ecx],[ebx+ecx]            R(ebx,ecx,[ebx+1+1...+1],eax);
                                       W([eax+1+1...+1])
    add ecx,1                          W(ecx);
```

Código 4.9: Exemplo de resultado da análise de um laço.

```
func:
    mov ecx,0                          ecx = 0
    mov [eax+0],[ebx+0]                [eax+0] = [ebx+0]
    add ecx,1                          ecx = 0+1
    mov [eax+1],[ebx+1]                [eax+0+1] = [ebx+0+1]
    add ecx,1                          ecx = 0+1+1
    ...                                ...
    mov [eax+(1+1+...+1)],             [eax+0+1+1...+1] = [ebx+0+1+1...+1]
        [ebx+(1+1+...+1)]
    add ecx,1                          ecx = 0+1+1...+1+1
```

4.8 Particionamento

Uma vez que o grafo de dependências tenha sido obtido, o próximo passo consiste no particionamento da aplicação em unidades paralelas. O processo consiste em agrupar vértices do grafo em unidades disjuntas. Em particular, se o vértice A não pode ser atingido pelo vértice B por nenhum caminho do grafo, eles são elegíveis para a separação. Se esse é o caso, então algumas análises sobre o grafo são possíveis: a largura W do grafo consiste no maior grau de paralelismo possível para aplicação. Como o grau depende da região considerada, então essa medida pode ser aplicada sobre o subgrafo de dependência da região desejada. Esses conceitos são bem estabelecidos dentro de teoria de ordem, em particular, em grafos de ordem parcial (ver Seção 3.9). A primeira tentativa para o particionamento consistiu em tentar encontrar as anticadeias do grafo para construir os grupos de instruções necessários para as unidades funcionais. O algoritmo empregado é uma abordagem clássica da literatura (JOHNSTON, 1976) e utiliza uma técnica de varredura de todos os elementos, buscando todos os vértices que não possuem caminhos entre si. A ideia é para todo vértice V no grafo de dependências G, verificar se existe um caminho $C(V,U)$, para outro vértice U. Se não existir, U é incluído no conjunto. Para cada nova inserção, deve-se verificar se existe um caminho para todos os vértices já incluídos. Cada vez que o conjunto de vértices é verificado, o último elemento incluído é removido. Dessa maneira, todas as anticadeias podem ser obtidas. No entanto, esse algoritmo possui uma limitação importante: sua complexidade é exponencial e o tempo de execução é muito alto. Essa característica tornou essa abordagem inviável, motivando a escolha de um método diferente.

Após novos estudos, decidiu-se por uma outra abordagem: percorrer o grafo à partir dos vértices iniciais, e usar os nós com um grau maior que 2 como pontos de referência para

a obtenção dos grupos de instruções. A ideia é que se um vértice possui grau 1 ou 2, então ele possui apenas uma ou nenhuma dependência direta. De maneira similar, ele depende apenas de um ou de nenhum vértice. Se o grau é maior do que 2, então existem mais dependências. Nesse caso, evidencia-se a existência de uma separação em linhas distintas de dependências. Conjuntos que contenham elementos de cada uma dessas linhas formam anticadeias. A Figura 4.13 apresenta um exemplo desse processo em um grafo de dependências. Os vértices 4 e 44 apresentam grau 3 (o grau do vértice é a soma de todas as arestas entrando e saindo). Para o vértice 4, isso significa que ele depende diretamente de dois outros: A e 5, e cada um desses inicia uma linha de vértices, cada uma terminando no vértice 44 que, por sua vez, precisa ser executado antes de F e 45. As duas linhas A, F e $5, 6, 7, 8, 9, 3F, 40, 41, 42, 43$ podem ser particionadas em grupos separados, e executadas em paralelo. Com esse processo, o grau de paralelismo obtido passa a depender exclusivamente do grafo de dependências. Dessa maneira, para obter-se melhorias, o foco deve voltar-se para a eliminação de dependências e reestruturação do grafo. A implementação dessa técnica de particionamento na ferramenta encontra-se em fase inicial.

4.9 Ferramenta e implementação

O código-fonte da ferramenta criada durante este trabalho foi disponibilizado em um repositório *GIT*, localizado em (FERRAMENTA..., 2015). O seu uso pode ser feito em dois modos: i) desmontagem de uma aplicação (binário) Intel x86 32 bits (por enquanto com suporte apenas a arquivos *PE-32*, o formato *ELF* ainda encontra-se em desenvolvimento), e remontagem na linguagem intermediária; ii) modo de análise de fluxo e dependências e particionamento da aplicação.
A ferramenta ainda encontra-se instável e com algumas restrições:

- A implementação da análise de dependências ainda possui alguns erros, e não está funcional para todas as aplicações. Em particular, algumas aplicações estão gerando exceções quando a análise é aplicada;
- O particionamento encontra-se incompleto, e ainda não explora o máximo paralelismo teórico previsto neste trabalho;
- A remontagem em linguagem intermediária não está funcional;
- A tradução da linguagem intermediária de volta para a arquitetura Intel x86 encontra-se em fase preliminar.

A ferramenta é capaz de desmontar alguns tipos de aplicação e gerar os grafos de fluxo e dependências.

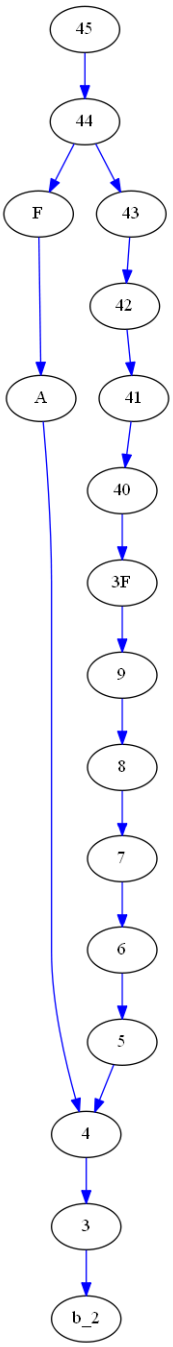

Figura 4.13: Exemplo de grafo de dependências e técnica de particionamento.

4.10 Experimentos

Para o estágio atual da ferramenta, alguns experimentos preliminares foram planejados e executados, com o objetivo de validar as técnicas escolhidas e sua implementação até o momento. Algumas aplicações comuns foram escolhidas para conduzir esses experimentos:

- Programa I - efetua a multiplicação entre duas matrizes;

- Programa II - efetua a transposição de uma matriz;
- Programa III - efetua a soma entre duas matrizes;
- Programa IV - efetua a multiplicação entre um escalar e uma matriz.

Os experimentos foram conduzidos de acordo com os seguintes passos:

- i) Para cada programa, definiu-se um tamanho para suas entradas;
- ii) Em seguida, um número N foi definido, que determina um número de execuções;
- iii) A versão sequencial do programa foi executada N vezes, e o tempo total anotado para cada execução. Por fim, o tempo médio foi calculado;
- iv) O programa foi submetido à ferramenta de paralelização deste trabalho;
- v) Como a ferramenta está incompleta, os passos finais foram efetuados manualmente. Após a obtenção do grafo de dependências, particionou-se manualmente o programa, e foi realizada a transformação de volta para a arquitetura Intel x86;
- vi) As unidades paralelas foram encapsuladas em *threads*, executadas sobre a plataforma *Microsoft Windows*;
- vi) A versão paralelizada foi executada N vezes, e o tempo médio anotado.

Para reduzir o impacto do tempo de criação de *threads* no experimento, elas foram criadas antes do experimento, e mantidas em estado de suspensão. Após o início da contagem do tempo, as unidades de execução foram atribuídas para cada *thread*, seguindo um modelo produtor-consumidor. Para cada execução de um programa, os resultados das computações obtidos a partir da versão paralela foram comparados aos produzidos pela versão sequencial, para garantir que a semântica do programa não foi violada.
A fase manual dos experimentos seguiu cuidadosamente os métodos discutidos neste capítulo, de forma a validar as técnicas estudadas. Dessa maneira, os resultados podem ser obtidos também de maneira automática, quando a implementação da ferramenta estiver completa.
Para cada programa, escolheu-se matrizes de tamanho 256×256. Os valores de N foram escolhidos de forma a criar tempos razoavelmente grandes (maior do que 1 segundo), para melhorar a precisão das medidas. Os valores estão a seguir:

- Programa I - $N = 32$;
- Programa II - $N = 8192$;
- Programa III - $N = 4096$;

- Programa IV - $N = 8192$.

Os experimentos consideraram uma máquina com um processador Intel i7-4820K 3.7GHz com 16 GB de RAM, com oito núcleos virtuais. As aplicações foram compiladas com o compilador Microsoft C/C++ com otimização de código desabilitada, e executadas em ambiente Microsoft Windows 7. A Tabela 4.1 apresenta os resultados e os tempos de execução e ganhos de desempenho obtidos, com os tempos em milissegundos.

Tabela 4.1: Resultados dos experimentos, com os tempos em milissegundos.

Programa	Versão sequencial		Versão paralela		*Speed-up*
	Tempo total	Tempo médio	Tempo total	Tempo médio	
I	2953	92.28	382	11.93	7.73
II	1965	0.23	266	0.03	7.38
III	2137	0.52	285	0.06	7.49
IV	1695	0.2	241	0.029	7.03

Os resultados demonstram um *speed-up* considerável para os programas considerados, com ganho expressivo para os núcleos utilizados. No entanto, essas aplicações são simples, e experimentos com algoritmos mais complexos ainda estão pendentes.
A próxima seção apresenta um exemplo de paralelização, em particular utilizando o programa II discutido

4.11 Exemplo de paralelização

Esta seção apresenta o passo a passo da paralelização para um programa de transposição de matrizes. Para esse exemplo utilizou-se o compilador Microsoft C/C++, sem otimização de código. O Código 4.10 contém a versão da aplicação em linguagem C, em que *m1* contem a matriz de entrada, *m* o número de linhas, *n* o número de colunas e *out a matriz de saída*. O Código 4.11 contém sua versão compilada para Intel x86 em *assembly*, apresentada em mnemônicos, na qual a primeira coluna contém o endereço e a segunda a instrução. A paralelização é dividida nas seguintes etapas:

- Desmontagem e análise das instruções;
- Tradução para uma linguagem intermediária;
- Criação do grafo de fluxo de controle;
- Criação do grafo de dependências;
- Particionamento;
- Remontagem em binário.

Além dessas etapas, para fins de execução deve-se montar o ambiente necessário para executar cada unidade paralela na arquitetura alvo.

A desmontagem é feita sobre o binário final, traduzindo cada operação para o equivalente na linguagem intermediária. O resultado desse processo para a aplicação considerada é apresentado no Código 4.12. A primeira coluna apresenta o endereço original da instrução na aplicação. A segunda coluna contém o endereço traduzido na nova representação, e a terceira coluna apresenta a instrução. É importante observar, nesse momento, que o laço mais interno ($L1$), contido nas linhas $3-5$ do Código 4.10 está contido entre os endereços 13 e 24 da linguagem intermediária. Essa observação é importante pois esse será o laço candidato para a paralelização.

Após a desmontagem o grafo de fluxo de controle e de dependências são obtidos. A Figura 4.14 apresenta o grafo de dependências de $L1$. Os demais laços e regiões não são apresentados neste exemplo pois apresentam pouca ou nenhuma capacidade de paralelização. As expressões geradas para $L1$ estão contidas em ... O grafo apresenta alta linearidade e, portanto, baixo potencial para paralelização das regiões internas. A análise das expressões, no entanto, mostra uma característica relevante: todas os valores de saída dependem apenas de valores na pilha, em particular $[ebp+8]$, $[ebp+14]$, $[ebp-4]$ e $[ebp-8]$. Se o laço for considerado como uma execução sequencial desse trecho, observa-se que o único valor escrito em cada iteração é o valor para $[ebp+8]$, como observado na expressão 3. Dessa maneira, na segunda iteração essa expressão será $[ebp-8] = (1+(1+[ebp-8]))$, na terceira iteração $[ebp-8] = (1+(1+(1+[ebp-8])))$ e assim sucessivamente. Analisando a função como um todo observa-se que o valor inicial de $[ebp-8]$ é 0, no endereço 11 da linguagem intermediária. Como consequência, a expressão final para o laço é reduzida para $[ebp-8] = (1+(1+(...+0)))$, em que o valor final depende do número de iterações. Esse comportamento evidencia um índice, uma vez que o valor é constante para cada iteração. Utilizando a técnica de eliminação de índice, é possível fixar esse valor para a execução de cada trecho, eliminando-se a dependência. Assim, como as expressões não indicam mais nenhuma dependência, o laço pode ser particionado em unidades paralelas. É importante observar que, para esse exemplo, as iterações do laço foram consideradas como procedimentos separados.

Uma vez que as unidades foram identificadas, o código precisa ser alterado para ser executado na plataforma alvo. Para este exemplo, considerou-se *threads* Microsoft Windows, executando no sistema operacional Microsoft Windows 7. Buscou-se uma região livre da pilha para o armazenamento das constantes para os índices e os demais valores de entrada. Além disso, antes do trecho considerado, incluiu-se instruções para mover o valor das constantes dessa região para a posição do índice em $[ebp+8]$ e demais posições. Os índices precisam ser providos por uma função externa. Para a execução, a pilha deve ser preenchida com os valores de entrada, o que também é provido por funções externas. Como consequência, para este exemplo, a versão remontada da aplicação está pronta

para ser paralelizada, mas o processo em si ainda depende de elementos na plataforma de execução.

Código 4.10: Exemplo de aplicação para transposição de matrizes em linguagem C.

```
1 void mattrans(int ** m1, int m, int n, int ** out) {
2         int i,j;
3         for (i = 0; i < m; i++) {
4                 for( j = 0 ; j < n ; j++ ) {
5                         out[i][j] = m1[j][i];
6                 }
7         }
8 }
```

Código 4.11: Exemplo de aplicação para transposição de matrizes em linguagem de máquina.

```
0C0                 push    ebp
0C1                 mov     ebp, esp
0C3                 sub     esp, 8
0C6                 push    esi
0C7                 mov     [ebp+var_4], 0
0CE                 jmp     short loc_4010D9
0D0                 mov     eax, [ebp+var_4]
0D3                 add     eax, 1
0D6                 mov     [ebp+var_4], eax
0D9                 mov     ecx, [ebp+var_4]
0DC                 cmp     ecx, [ebp+arg_4]
0DF                 jge     short loc_40111D
0E1                 mov     [ebp+var_8], 0
0E8                 jmp     short loc_4010F3
0EA                 mov     edx, [ebp+var_8]
0ED                 add     edx, 1
0F0                 mov     [ebp+var_8], edx
0F3                 mov     eax, [ebp+var_8]
0F6                 cmp     eax, [ebp+arg_8]
0F9                 jge     short loc_40111B
0FB                 mov     ecx, [ebp+var_8]
0FE                 mov     edx, [ebp+arg_0]
101                 mov     eax, [edx+ecx*4]
104                 mov     ecx, [ebp+var_4]
```

```
107                 mov     edx, [ebp+arg_C]
10A                 mov     ecx, [edx+ecx*4]
10D                 mov     edx, [ebp+var_8]
110                 mov     esi, [ebp+var_4]
113                 mov     eax, [eax+esi*4]
116                 mov     [ecx+edx*4], eax
119                 jmp     short loc_4010EA
11B                 jmp     short loc_4010D0
11D                 pop     esi
11E                 mov     esp, ebp
120                 pop     ebp
121                 retn
```

Código 4.12: Exemplo de aplicação para transposição de matrizes em linguagem intermediária.

```
0 0: EP 0
0 1: NOP
C0 2: MOV REGREF [ESP+0], REGISTER EBP
C0 3: SUB REGISTER ESP, IMMEDIATE 4
C1 4: MOV REGISTER EBP, REGISTER ESP
C3 5: SUB REGISTER ESP, IMMEDIATE 8
C6 6: MOV REGREF [ESP+0], REGISTER ESI
C6 7: SUB REGISTER ESP, IMMEDIATE 4
C7 8: MOV REGREF [EBP-4], IMMEDIATE 0
CE 9: JMP D
D0 A: MOV REGISTER EAX, REGREF [EBP-4]
D3 B: ADD REGISTER EAX, IMMEDIATE 1
D6 C: MOV REGREF [EBP-4], REGISTER EAX
D9 D: MOV REGISTER ECX, REGREF [EBP-4]
DC E: ACCESS REGREF [EBP+C]
DC F: ACCESS REGISTER ECX
DF 10: JC 26
E1 11: MOV REGREF [EBP-8], IMMEDIATE 0
E8 12: JMP 16
EA 13: MOV REGISTER EDX, REGREF [EBP-8]
ED 14: ADD REGISTER EDX, IMMEDIATE 1
F0 15: MOV REGREF [EBP-8], REGISTER EDX
F3 16: MOV REGISTER EAX, REGREF [EBP-8]
F6 17: ACCESS REGREF [EBP+10]
```

```
F6 18: ACCESS REGISTER EAX
F9 19: JC 25
FB 1A: MOV REGISTER ECX, REGREF [EBP-8]
FE 1B: MOV REGISTER EDX, REGREF [EBP+8]
101 1C: MOV REGISTER EAX, REGREF [(EDX+0)+ECX*4+0]
104 1D: MOV REGISTER ECX, REGREF [EBP-4]
107 1E: MOV REGISTER EDX, REGREF [EBP+14]
10A 1F: MOV REGISTER ECX, REGREF [(EDX+0)+ECX*4+0]
10D 20: MOV REGISTER EDX, REGREF [EBP-8]
110 21: MOV REGISTER ESI, REGREF [EBP-4]
113 22: MOV REGISTER EAX, REGREF [(EAX+0)+ESI*4+0]
116 23: MOV REGREF [(ECX+0)+EDX*4+0], REGISTER EAX
119 24: JMP 13
11B 25: JMP A
11D 26: MOV REGISTER ESI, REGREF [ESP+0]
11D 27: ADD REGISTER ESP, IMMEDIATE 4
11E 28: MOV REGISTER ESP, REGISTER EBP
120 29: MOV REGISTER EBP, REGREF [ESP+0]
120 2A: ADD REGISTER ESP, IMMEDIATE 4
121 2B: RET
```

Código 4.13: Expressões geradas para o exemplo.

```
1 edx=(1+[ebp-8])
2 eax=[[ebp-4]*4+[ebp+14]]
3 [ebp-8]=(1+[ebp-8])
4 ecx=[ebp-8]
5 [[ebp-8]*4+[[ebp-4]*4+[ebp+14]]]=[[ebp-8]*4+[[ebp-4]*4+[ebp↩
    +8]]]
```

4.12 Considerações finais

Este capítulo apresentou os objetivos e as atividades realizadas, destacando, principalmente, conclusões oriundas de estudos da arquitetura Intel x86 e o trabalho realizado para a análise e particionamento de aplicações. Apresentou-se também a ferramenta construída, que ainda encontra-se em estado de desenvolvimento, com as funções de análise de dependências e particionamento incompletas. Por fim, foram apresentados alguns experimentos preliminares, sobre aplicações de uso comum, que validam as técnicas abordadas.

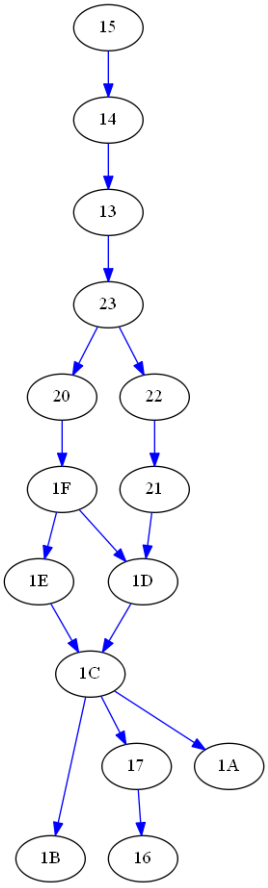

Figura 4.14: Exemplo de grafo de dependências sobre laço.

CAPÍTULO

5

Conclusão e trabalhos futuros

Motivado pelas dificuldades em paralelizar aplicações sequenciais, em especial compiladas e sem código-fonte disponível, e também pela crescente tendência de crescimento de computação paralela e distribuída, este trabalho envolveu o estudo e desenvolvimento de uma metodologia para, automaticamente, analisar uma aplicação sequencial nativa na arquitetura Intel x86, e criar uma versão pronta para ser executada em paralelo. Além disso, este trabalho apresentou uma implementação parcial na forma de uma ferramenta capaz de criar uma versão em linguagem intermediária do programa considerado, realizar a análise do grafo de fluxo de aplicação, e a análise de dependências entre suas instruções. Ela também conta com alguns procedimentos de eliminação de dependências e particionamento parcial do código em unidades funcionais paralelizáveis.
Como trabalhos futuros, inclui-se a conclusão e estabilização da ferramenta, destacando-se:

- Conclusão do módulo de particionamento, que se encontra em desenvolvimento;
- Estabilização da análise de dependências, que apresenta alguns erros para certas aplicações, em particular quando o fluxo de execução é muito complexo;
- Criação do módulo de transformação da linguagem intermediária para a arquitetura alvo Intel x86;
- Inclusão de novas técnicas de eliminação de dependências.

Além disso, abre-se a possibilidade para o estabelecimento de técnicas de paralelização em tempo de execução, com análise dinâmica. A saída da análise estática pode ser uti-

lizada nesse processo para, entre outras coisas, determinar o caminho seguido em estruturas condicionais e/ou de repetição, tomando decisões mais precisas e melhorando o paralelismo. Outro fator importante é o encapsulamento das unidades funcionais em programas para a execução em plataformas específicas. Por exemplo, encapsulamento em *threads* ou outros elementos paralelizáveis. Após a resolução desses pontos, deve-se ainda realizar experimentos com aplicações de maior porte.

Acredita-se que as técnicas abordadas, e a abordagem proposta neste trabalho venham a servir como ponto de partida para trabalhos futuros na mesma linha, contribuindo para a criação de uma ferramenta definitiva de recompilação de aplicações Intel x86 para melhor utilizarem ambientes paralelos e distribuídos.

Referências

BANERJEE, U. *Dependence Analysis*. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 1475770588, 9781475770582.

BASTOUL, C. *Efficient code generation for automatic parallelization and optimization (long version*. [S.l.], 2003.

BENNETT, K. *Intel Core i7-3960X - Sandy Bridge E Processor Review*. nov. 2011. Disponível em: `hardocp.com/article/2011/11/14/intel_core_i73960x_sandy_bridge_e_processor_review/`. Acesso em: 20 de novembro de 2012.

BERNSTEIN, A. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, v. 15, n. 5, p. 757–763, 1966.

BONDHUGULA, U. Automatic distributed-memory parallelization and code generation using the polyhedral framework. 2011.

COMTET, L. *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. Springer Netherlands, 1974. ISBN 9789027704412. Disponível em: `https://books.google.com.br/books?id=C0HPgWhEssYC`.

DEPARTMENT, P. G.; GASPER, P.; HERBST, C.; MCGOUGH, J.; RICKETT, C.; STUBBENDIECK, G. *Automatic Parallelization of Sequential C Code*. 2003.

DIPASQUALE, N.; WAY, T.; GEHLOT, V. Comparative survey of approaches to automatic parallelization. In: *MASPLAS '05*. [S.l.: s.n.], 2005.

FERRAMENTA para a paralelização de códigos. 2015. Disponível em: `https://github.com/andrem-eberle/parallel1`.

FRUMKIN, M.; HRIBAR, M.; JIN, H.; WAHEED, A.; YAN, J. A comparison of automatic parallelization tools/compilers on the sgi origin 2000. In: *Proceedings of the 1998*

ACM/IEEE conference on Supercomputing (CDROM). Washington, DC, USA: IEEE Computer Society, 1998. (Supercomputing '98), p. 1–22. ISBN 0-89791-984-X. Disponível em: http://dl.acm.org/citation.cfm?id=509058.509119.

GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN 0716710447.

GHEZZI, C.; JAZAYERI, M. *Programming Language Concepts*. 3rd. ed. New York, NY, USA: John Wiley & Sons, Inc., 1997. ISBN 0471104264.

GILDER, M. R.; KRISHNAMOORTHY, M. S. Automatic source-code parallelization using hicor objects. *Int. J. Parallel Program.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 22, n. 3, p. 303–350, jun. 1994. ISSN 0885-7458. Disponível em: http://dx.doi.org/10.1007/BF02577736.

GOSDEN, J. A. Explicit parallel processing description and control in programs for multi- and uni-processor computers. In: *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*. New York, NY, USA: ACM, 1966. (AFIPS '66 (Fall)), p. 651–660. Disponível em: http://doi.acm.org/10.1145/1464291.1464361.

GRIEBL, M.; COLLARD, J.-F. Generation of synchronous code for automatic parallelization of while loops. In: *EURO-PAR '95, Lecture Notes in Computer Science 966*. [S.l.]: Springer-Verlag, 1995. p. 315–326.

HANK, R. E.; MAHLKE, S. A.; BRINGMANN, R. A.; GYLLENHAAL, J. C.; HWU, W. mei W. Superblock formation using static program analysis. In: *in Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-26*. [S.l.: s.n.], 1993. p. 247–255.

HEX-RAYS Home, IDA IDA Professional Disassembler. 2013. Disponível em: https://www.hex-rays.com/products/ida/index.shtml.

HOWARD, D.; RYAN, C.; COLLINS, J. J. Attribute grammar genetic programming algorithm for automatic code parallelization. In: *Proceedings of the 5th international conference on Convergence and hybrid information technology*. Berlin, Heidelberg: Springer-Verlag, 2011. (ICHIT'11), p. 250–257. ISBN 978-3-642-24081-2. Disponível em: http://dl.acm.org/citation.cfm?id=2045005.2045038.

HUANG, D.; STEFFAN, J. G. Programmer-assisted automatic parallelization. In: *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*. Riverton, NJ, USA: IBM Corp., 2011. (CASCON '11), p. 84–98. Disponível em: http://dl.acm.org/citation.cfm?id=2093889.2093899.

IEROTHEOU, C. S.; JOHNSON, S. P.; CROSS, M.; LEGGETT, P. F. Computer aided parallelisation tools (captools) - conceptual overview and performance on the parallelisation of structured mesh codes. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 22, n. 2, p. 163–195, fev. 1996. ISSN 0167-8191. Disponível em: `http://dx.doi.org/10.1016/0167-8191(95)00004-6`.

INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Santa Clara, CA, USA, jun. 2010.

JOHNSTON, H. Cliques of a graph-variations on the bron-kerbosch algorithm. *International Journal of Computer & Information Sciences*, Kluwer Academic Publishers-Plenum Publishers, v. 5, n. 3, p. 209–238, 1976. ISSN 0091-7036. Disponível em: `http://dx.doi.org/10.1007/BF00991836`.

KOVACEVIC; STANOJEVIC; MARINKOVIC; POPOVIC. A solution for automatic parallelizationof sequential assembly code. 2013.

LI, W.; PINGALI, K. A singular loop transformation framework based on non-singular matrices. In: . [S.l.: s.n.], 1992.

MESSINA, P. C.; WILLIAMS, R. D.; FOX, G. C. *Parallel computing works !* San Francisco, CA: Morgan Kaufmann, 1994. (Parallel processing scientific computing). ISBN 1-55860-253-4. Disponível em: `http://opac.inria.fr/record=b1104264`.

MICHAEL, M.; MOREIRA, J.; SHILOACH, D.; WISNIEWSKI, R. Scale-up x scale-out: A case study using Nutch/Lucene. In: SAHOO, R. (Ed.). *International Workshop on System Management Techniques, Processes, and Services, 3., Long Beach, 2007.* **Proceedings...**. Washington: IEEE Computer Society, 2007.

WARD, L. Partially ordered topological spaces. *Proceedings of the American Mathematical Society*, American Mathematical Society, v. 5, n. 1, p. pp. 144–161, 1954. ISSN 00029939. Disponível em: `http://www.jstor.org/stable/2032122`.

YARDIMCI, M. F. E. *Dynamic Parallelization and Vectorization of Binary Executables on Hierarchical Platforms*. 2008.

ZIMA, H. P.; BREZANY, P.; CHAPMAN, B. M.; HULMAN, J. *Automatic parallelization for distributed-memory systems: Experiences and current reseach*. 1993.

APÊNDICE

A

Linguagem intermediária

Este apêndice apresenta a especificação da linguagem intermediária e um exemplo de código Intel x86 transformado. A primeira seção apresenta as instruções contidas, em seguida a sintaxe é apresentada.

A.1 Instruções

A linguagem consiste em um conjunto mínimo necessário de instruções para representar uma aplicação Intel x86 arbitrária. Computações de ponto flutuante utilizam as mesmas instruções que números inteiros. O conjunto é definido a seguir, com uma breve explicação de cada elemento:
Aritméticas e lógicas:

- AND – **E** lógico;
- OR – **OU** lógico;
- NOT – negação lógica;
- XOR – **OU** exclusivo;
- ADD – soma;
- ADC – soma (com *carry*);
- SUB – subtração;

- MUL – multiplicação;
- DIV – divisão;
- IDIV – divisão (resto).

Deslocamentos de bits:

- SBB – subtração (com *carry*);
- ROR – rotação para a direita;
- ROL – rotação para a esquerda;
- RCL – rotação para a esquerda (com *carry*);
- RCR – rotação para a direita (com *carry*);
- SHL – deslocamento para a esquerda;
- SHR – deslocamento para a direita;
- SAL – deslocamento para a esquerda (com *carry*);
- SAR – deslocamento para a direita (com *carry*).

Acessos à memória:

- MOV – escrita em registrador ou memória;
- ACCESS – leitura em um registrador ou memória.

Controle de fluxo:

- JMP – salto para uma posição arbitrária;
- JC – salto condicional;
- RT – retorno de procedimento;
- CALL – chamada de procedimento.

A linguagem aceita todos os registradores da arquitetura Intel x86: **EAX**, **EBX**, **ECX**, **EDX**, **ESP**, **EBP**, **ESI** e **EDI**.

A.2 Sintaxe

Cada instrução da linguagem possui o seguinte formato:
instrução argumento1 argumento2 argumento3
O número de argumentos varia de acordo com a instrução. Cada argumento possui um tipo e um especificador. Os tipos possíveis são quatro:

- IMMEDIATE V – denota um valor numérico V imediato;
- STATIC V – denota um endereço de memória V;
- REGISTER R – denota um registrador R;
- REGREF $[(R + disp2) + r1 * mod + disp]$ – denota um acesso ao conteúdo de um registrador, com modificadores. R é o registrador base, com um deslocamento $disp$. $r1$ é um possível segundo registrador a ser somado ao valor inicial, modificado por mod e $disp$. mod e $disp$ serão 0 caso não haja um segundo registrador. Nesse caso $r1$ é indefinido.

A.3 Exemplo

A Figura A.1 apresenta um exemplo de código de uma aplicação Intel x86, enquanto sua versão transformada para a linguagem intermediária encontra-se na Figura A.2.

```
11 mov eax,[ebp+0xFC]
14 inc eax
17 mov [ebp+0xFC],eax
1A mov ecx, [ebp+0xFC]
1D test [ebp+10],ecx
20 jc 40
26 mov edx,[ebp+0xFC]
29 mov eax,[ebp+0x14]
2C mov ecx,[eax+edx*4]
2F mov edx,[ebp+0xF4]
32 mov [ecx+edx*4],0
39 mov [ebp+0xF4],0
40 jmp 1d
```

Figura A.1: Exemplo de aplicação Intel x86.

```
11 C: MOV REGISTER EAX, REGREF
[EBP+FC]
14 D: ADD REGISTER EAX, IMMEDI-
ATE 1
17 E: MOV REGREF [EBP+FC], RE-
GISTER EAX
1A F: MOV REGISTER ECX, REGREF
[EBP+FC]
1D 10: ACCESS REGREF [EBP+10]
1D 11: ACCESS REGISTER ECX
20 12: JC 40
26 13: MOV REGISTER EDX, RE-
GREF [EBP+FC]
29 14: MOV REGISTER EAX, RE-
GREF [EBP+14]
2C 15: MOV REGISTER ECX, RE-
GREF [(EAX+0)+EDX*4+0]
2F 16: MOV REGISTER EDX, RE-
GREF [EBP+F4]
32 17: MOV REGREF
[(ECX+0)+EDX*4+0], IMMEDIATE
0
39 18: MOV REGREF [EBP+F4], IM-
MEDIATE 0
40 19: JMP 1D
```

Figura A.2: Exemplo de aplicação Intel x86 transformada para a linguagem intermediária.