Caracterização do perfil de utilização de recursos de programas a partir de arquivos executáveis utilizando mineração de dados

Renê de Souza Pinto

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Renê de Souza Pinto

Caracterização do perfil de utilização de recursos de programas a partir de arquivos executáveis utilizando mineração de dados

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Alexandre Cláudio Botazzo Delbem

USP – São Carlos
Novembro de 2017

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

d659c de Souza Pinto, Renê
Caracterização do perfil de utilização de recursos
de programas a partir de arquivos executáveis
utilizando mineração de dados / Renê de Souza Pinto;
orientador Alexandre Cláudio Botazzo Delbem. -- São
Carlos, 2017.
201 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2017.

1. Mineração de dados. 2. Avaliação de desempenho.
3. Modelagem computacional. 4. Perfil de execução.
I. Delbem, Alexandre Cláudio Botazzo, orient. II.
Título.

Renê de Souza Pinto

Characterization of resource utilization profile of executable programs through data mining

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

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Alexandre Cláudio Botazzo Delbem

USP – São Carlos
November 2017

Dedico este trabalho ao meu filho Thomas e à minha esposa Juliana, com carinho.

AGRADECIMENTOS

Aos meus pais Walkmar e Lurdinha, pilares da minha educação e do meu caráter. Gratidão eterna por serem referenciais tão brilhantes em minha vida.

Aos meus irmãos Mirela, Walker e Eneida, pelo apoio e por serem a minha família.

Ao meu orientador prof. Dr. Alexandre Cláudio Botazzo Delbem, pelo imensurável apoio, paciência e conhecimentos transmitidos ao longo deste trabalho.

Ao prof. Dr. Francisco José Monaco, pela coorientação e pela incalculável ajuda antes e durante a execução deste trabalho, como orientador e como amigo de longa data.

Ao meu filho Thomas e a minha esposa Juliana, por serem a razão da minha vida.

Aos amigos e sócios da Omega7 Systems, pelo apoio e ajuda durante todo este trabalho.

A Universidade Paulista e a Universidade de Araraquara por toda confiança profissional depositada em mim ao longo destes anos, e que me permitiu solidificar minha carreira de docência no mundo acadêmico.

Ao meu amigo canino Teodoro, sempre ao meu lado durante longas noites de escrita e trabalho.

A todos os amigos, alunos ou professores, do Laboratório de Sistemas Distribuídos e Programação Concorrente (LaSDPC), pelas preciosas sugestões e críticas, sempre construtivas.

Ao Instituto de Ciências Matemáticas e de Computação (ICMC) e todos seus funcionários, e a Universidade de São Paulo, por sua excelência em ensino e pesquisa e por fazer parte da minha vida há tanto tempo.

E a todos que colaboraram direta ou indiretamente para a realização deste projeto.

*“Ninguém caminha sem aprender a caminhar,
sem aprender a fazer o caminho caminhando,
refazendo e retocando o sonho pelo qual se pôs a caminhar.”*
(Paulo Freire)

RESUMO

SOUZA PINTO, R. **Caracterização do perfil de utilização de recursos de programas a partir de arquivos executáveis utilizando mineração de dados.** 2017. 201 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2017.

Este trabalho apresenta uma metodologia para caracterização do perfil de consumo de recursos demandados por um programa de computador a partir da análise do código binário do arquivo executável. A categorização de processos de acordo com seus perfis de consumo de recursos durante a execução — tais como uso de CPU e memória — é uma informação muito desejada para objetivos de projeto e gerenciamento de sistemas. Técnicas convencionais para este propósito são baseadas em testes de caixa branca (que avaliam o código fonte da aplicação), que tendem a ser de difícil aplicação dado a complexidade das arquiteturas de software além da necessidade de acesso ao código fonte; ou detecção de perfis baseada em dados de execução, que depende da disponibilidade de dados de execução confiáveis e da seleção de características que de fato vão correlacionar o perfil de consumo. A abordagem baseada em mineração de dados proposta neste trabalho evita estas dificuldades uma vez que manipula somente os arquivos binários executáveis. O método combina técnicas provindas da teoria da informação, redes complexas e filogenia para produzir um agrupamento hierárquico de um conjunto de arquivos de programas executáveis que pode ser utilizado para prever potenciais similaridades em termos de consumo de recursos em tempo de execução. As questões de pesquisa deste trabalho investigam se a transformação feita pelo compilador preserva similaridades entre código fonte e binário que podem ser detectadas através de algoritmos de compressão; em caso positivo, verificar se as similaridades encontradas no código binário estão relacionadas com o perfil de execução das aplicações, permitindo inferir o comportamento dos programas a partir da análise do código binário. Este trabalho apresenta a sistematização do método assim como os resultados da aplicação para caracterizar aplicações em termos de consumo de CPU e Entrada/Saída em uma plataforma PC padrão. Diversos experimentos foram executados em um repositório de 80 programas de várias fontes obtendo-se resultados significativos que evidenciam que a similaridade dos perfis de execução obtidas com esta abordagem é consistente com as obtidas experimentalmente por aferição. A aplicação do método também é exemplificado através de casos de estudo que caracterizam o perfil de execução de programas executáveis.

Palavras-chave: Mineração de dados, avaliação de desempenho, modelagem computacional, perfil de execução.

ABSTRACT

SOUZA PINTO, R. **Characterization of resource utilization profile of executable programs through data mining**. 2017. 201 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2017.

This work introduces a methodology for characterizing the runtime resource demands of a computer program from the analysis of its binary executable file. Categorization of processes according to the kind of resources required during execution — such as CPU and memory usage — is a sought-after piece of knowledge for the aims of computer system design and management. Conventional techniques available for this purpose include white-box static source code analysis and profile matching based on historical execution data. The former tends to be challenging in face of complex software architectures and requires access to the source code; the latter is dependent on the availability of reliable past data and on the selection of features yielding effective correlations with resource usage. The alternative data mining approach proposed in this paper avoids those difficulties by manipulating binary executable files. The method combines techniques from information theory, complex networks and phylogenetics to produce a hierarchical clustering of a set of executable files, which can be used to predict potential similarities in terms of runtime resource usage. The work's research questions investigate if the transformation performed by the compiler preserves similarity information across the source and binary code representation such that it can be detected by standard compression algorithms; and if the so identified similarities in the symbolic object encoding are correlated to runtime resource usage to an extent which allows for inferring the program's behavior from the analysis of the binary file. The paper introduces the method's rationales and presents results of its application to characterize CPU and IO usages of benchmark applications executed on a standard PC platform. Essays carried out over a set of 80 executable programs from varying sources yielded numerically significant evidences that the prediction of resource usage similarity obtained by the approach is consistent with experimentally measured runtime profile. The application of the method is also exemplified in a few case studies aimed at characterizing executable programs' runtime profile.

Keywords: Data mining, performance evaluation, computational modeling, runtime profiling.

LISTA DE ILUSTRAÇÕES

Figura 1 – Método <i>Neighbor Joining</i>	36
Figura 2 – Saída gerada com <i>mtool</i> mostrando as aferições da carga <i>dirload_io</i>	49
Figura 3 – Gráfico com o tamanho de cada programa do repositório.	58
Figura 4 – Gráficos com perfil de uso de CPU e E/S dos programas (p_1 à p_{12}).	59
Figura 5 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{13} à p_{24}).	60
Figura 6 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{25} à p_{36}).	61
Figura 7 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{37} à p_{48}).	62
Figura 8 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{49} à p_{60}).	63
Figura 9 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{61} à p_{72}).	64
Figura 10 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{73} à p_{80}).	65
Figura 11 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_1 à p_{12}).	66
Figura 12 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{13} à p_{24}).	67
Figura 13 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{25} à p_{36}).	68
Figura 14 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{37} à p_{48}).	69
Figura 15 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{49} à p_{60}).	70
Figura 16 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{61} à p_{72}).	71
Figura 17 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{73} à p_{80}).	72
Figura 18 – Decaimento do valor numérico de ρ	77
Figura 19 – Árvore com agrupamento alternativo para repositório de programas binários (f'_1)	83
Figura 20 – Árvore com agrupamento alternativo para repositório de códigos fonte (f'_2)	84
Figura 21 – Árvore com agrupamento alternativo para repositório de dados de consumo de CPU (f'_3)	84
Figura 22 – Árvore filogenética referente a todo o repositório de programas binários.	88
Figura 23 – Árvore filogenética referente a todo o repositório de programas binários com destaque para os grupos detectados.	89

Figura 24 – Árvore filogenética referente a todo o repositório com os segmentos de texto dos programas.	90
Figura 25 – Árvore filogenética com informações de consumo de CPU e E/S.	92
Figura 26 – Fluxograma para aplicação da técnica BCC.	97
Figura 27 – Árvore filogenética de quarteto.	98
Figura 28 – Possíveis resultados de árvores filogenéticas de quartetos.	98
Figura 29 – Códigos objeto compilados sem nível de otimização (-O0).	111
Figura 30 – Códigos objeto compilados com nível de otimização -O1.	111
Figura 31 – Códigos objeto compilados com nível de otimização -O2.	112
Figura 32 – Códigos objeto compilados com nível de otimização -O3.	112
Figura 33 – Códigos objeto compilados com nível de otimização -Os.	113
Figura 34 – Segmentos de texto compilados sem nível de otimização (-O0).	114
Figura 35 – Segmentos de texto compilados com nível de otimização -O1.	115
Figura 36 – Segmentos de texto compilados com nível de otimização -O2.	115
Figura 37 – Segmentos de texto compilados com nível de otimização -O3.	116
Figura 38 – Segmentos de texto compilados com nível de otimização -Os.	116
Figura 39 – Incidência dos raios de luz considerada pelo algoritmo Gouraud. <i>Adaptado de (Glium Project, 2016).</i>	118
Figura 40 – Códigos objeto compilados sem nível de otimização (-O0).	120
Figura 41 – Códigos objeto compilados com nível de otimização -O1.	120
Figura 42 – Códigos objeto compilados com nível de otimização -O2.	121
Figura 43 – Códigos objeto compilados com nível de otimização -O3.	121
Figura 44 – Códigos objeto compilados com nível de otimização -Os.	122
Figura 45 – Códigos objeto compilados sem nível de otimização (-O0) e códigos fonte.	123
Figura 46 – Códigos objeto compilados com nível de otimização -O1 e códigos fonte.	123
Figura 47 – Códigos objeto compilados com nível de otimização -O2 e códigos fonte.	124
Figura 48 – Códigos objeto compilados com nível de otimização -O3 e códigos fonte.	124
Figura 49 – Códigos objeto compilados com nível de otimização -Os e códigos fonte.	125
Figura 50 – Gráfico com o tamanho de cada programa do repositório.	132
Figura 51 – Árvore filogenética do repositório de <i>malwares</i> para Android.	133

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Sobel original	103
Código-fonte 2 – Código sem otimização	105
Código-fonte 3 – Código otimizado com <i>loop peeling</i>	105
Código-fonte 4 – Sobel modificado com <i>loop peeling</i> para a primeira coluna	105
Código-fonte 5 – Sobel modificado com <i>loop peeling</i> para a primeira linha e coluna	106
Código-fonte 6 – Código do algoritmo Gouraud	118
Código-fonte 7 – Código do algoritmo <i>bubble sort</i>	119
Código-fonte 8 – Código do algoritmo de Modulação ADPCM	149
Código-fonte 9 – Código do algoritmo de Demodulação ADPCM	151
Código-fonte 10 – Sobel original	153
Código-fonte 11 – Sobel modificado com <i>loop peeling</i> para a primeira coluna	154
Código-fonte 12 – Sobel modificado com <i>loop peeling</i> para a primeira linha e coluna	156
Código-fonte 13 – Código do algoritmo Simpson $\frac{1}{3}$	159
Código-fonte 14 – Código do algoritmo Gouraud	160
Código-fonte 15 – Código do algoritmo <i>Bubble sort</i>	161
Código-fonte 16 – Código do algoritmo <i>Heap sort</i>	162
Código-fonte 17 – Código do algoritmo <i>Radix sort</i>	163
Código-fonte 18 – Código do algoritmo <i>Selection sort</i>	165
Código-fonte 19 – Código do algoritmo <i>Insertion sort</i>	166
Código-fonte 20 – Código assembly do programa <i>Bubble sort</i>	167
Código-fonte 21 – Código assembly do programa <i>Heap sort</i>	168
Código-fonte 22 – Código assembly do programa <i>Insertion sort</i>	173
Código-fonte 23 – Código assembly do programa <i>Radix sort</i>	175
Código-fonte 24 – Código assembly do programa <i>Selection sort</i>	183
Código-fonte 25 – Código assembly do programa Simpson $\frac{1}{3}$	186

LISTA DE TABELAS

Tabela 1 – Programas binários do repositório.	46
Tabela 2 – Parâmetros utilizados na execução dos programas <i>CPU-Intensive</i>	51
Tabela 3 – Parâmetros utilizados na execução dos programas <i>I/O-Intensive</i>	52
Tabela 4 – Eficiência (em %) dos compressores utilizados no DAMICORE.	54
Tabela 5 – Valores de ρ para comparação entre repositórios não relacionados, com grande diferença de conteúdo.	77
Tabela 6 – Robustez do agrupamento para vários aspectos dos programas.	80
Tabela 7 – Correlação entre os diferentes aspectos avaliados (valores das médias de ρ).	82
Tabela 8 – Congruência dos repositórios com agrupamento alternativo.	85
Tabela 9 – Congruência entre o repositório original e versões com os programas compilados com diferentes níveis de otimização (utilizando os compiladores <i>gcc</i> , <i>llvm</i> e <i>icc</i>).	93
Tabela 10 – Congruência entre o compilador <i>gcc</i> e os compiladores <i>icc</i> e <i>llvm</i>	93
Tabela 11 – Congruência entre os compiladores <i>icc</i> e <i>llvm</i>	94
Tabela 12 – Programas de diferentes algoritmos que compõem o repositório.	117
Tabela 13 – Programas utilizados para compor o repositório do classificador.	128
Tabela 14 – Resultados do classificador para os programas <i>CPU-Intensive</i>	128
Tabela 15 – Resultados do classificador para os programas <i>CPU-Intensive</i>	129
Tabela 16 – Categorias dos <i>malwares</i> do repositório.	132

LISTA DE ABREVIATURAS E SIGLAS

ADPCM	<i>Adaptive differential pulse-code modulation</i>
CPU	<i>Central processing unit</i>
CSV	<i>Comma-separated value</i>
DAMICORE	<i>Datamine a Code Repository</i>
DNA	<i>Deoxyribonucleic acid</i>
FDCT	<i>Fast Discrete Cosine Transform</i>
FN	<i>Fast Newman</i>
FPGA	<i>Field-programmable gate array</i>
FSB	<i>Front-side bus</i>
GNU	<i>GNU's not Unix</i>
KVM	<i>Kernel-based Virtual Machine</i>
MIDI	<i>Musical Instrument Digital Interface</i>
NCD	<i>Normalized Compression Distance</i>
NJ	<i>Neighbor joining</i>
PC	<i>Personal computer</i>
QoS	<i>Quality of Service</i>
RAM	<i>Random Access Memory</i>
SATA	<i>Serial ATA</i>
SLA	<i>Service-level agreement</i>
SMS	<i>Short Message Service</i>

SUMÁRIO

I	INTRODUÇÃO	25
1	INTRODUÇÃO	27
1.1	Objetivos	29
1.2	Estrutura da Tese	29
2	REVISÃO	31
2.1	Caracterização de cargas de trabalho	31
2.2	Técnicas de Mineração de Dados em Repositórios	33
2.2.1	<i>DAMICORE</i>	34
2.3	Trabalhos relacionados com uso do DAMICORE	38
II	DESENVOLVIMENTO	41
3	PROJETO EXPERIMENTAL	43
3.1	Plataforma e cargas de trabalho	43
3.2	Seleção dos programas	44
3.3	Perfis de referência	47
3.3.1	<i>Ferramenta mtool</i>	47
3.4	Execução dos experimentos	50
3.4.1	<i>Aplicação do DAMICORE</i>	53
4	OBTENÇÃO DOS PERFIS DE REFERÊNCIA	55
4.1	Aferição das cargas de trabalho	55
4.1.1	<i>Gráficos com aquisições temporais</i>	55
4.1.2	<i>Gráficos com dados agregados</i>	57
5	ÍNDICE DE CONGRUÊNCIA PAR-A-PAR	73
5.1	Uma métrica quantitativa de congruência de agrupamento	73
5.1.1	<i>Índice de congruência par-a-par</i>	74
5.2	A ferramenta matches	78
6	AVALIAÇÃO QUANTITATIVA	79
6.1	Análise dos agrupamentos	79
6.1.1	<i>Avaliação de robustez</i>	79

6.1.2	<i>Correlação de similaridades</i>	81
6.1.3	<i>Estratégia alternativa de agrupamento</i>	82
7	AVALIAÇÃO QUALITATIVA	87
7.1	Filogenia do código binário	87
7.2	Estabilidade do DAMICORE para diferentes compiladores e opções de otimização	92
8	TÉCNICA BCC	95
8.1	Teste de Quartetos	97
III	ESTUDOS DE CASO	101
9	ALGORITMOS COM OTIMIZAÇÕES	103
9.0.1	<i>Análise do algoritmo do filtro Sobel</i>	103
9.0.2	<i>Análise com algoritmos distintos</i>	117
9.0.3	<i>Análise dos Binários versus Código fonte</i>	122
10	CLASSIFICADOR DE CPU INTENSIVE	127
11	CÓDIGOS MALICIOSOS	131
IV	CONCLUSÕES	135
12	CONCLUSÕES	137
12.1	Trabalhos Futuros	138
12.2	Publicações e Prêmio	139
	REFERÊNCIAS	141
APÊNDICE A	CÓDIGOS UTILIZADOS NO REPOSITÓRIO DE TESTES DO CAPÍTULO 9	149
A.1	Algoritmos Modulação/Demodulação ADPCM	149
A.2	Algoritmo Sobel	153
A.3	Algoritmo de método numérico	159
A.4	Algoritmo de sombreamento	160
A.5	Algoritmos de ordenação	161
A.6	Códigos assembly “desmontados” dos programas	167

Parte I

Introdução

INTRODUÇÃO

O conhecimento dos recursos demandados por um programa durante sua execução é uma informação importante que pode ser utilizada para diversos fins, tais como planejamento de alocação de recursos, políticas de controle de admissão, algoritmos de balanceamento de carga, mecanismos de alocação elástica e escalonadores, além de representar um fator chave para o desenvolvimento de estratégias de QoS (Qualidade de Serviço). No contexto de sistemas computacionais, um processo em execução pode representar uma carga de trabalho de uma determinada plataforma (na qual é executado), demandando recursos tais como consumo de CPU (Processamento), Entrada/Saída (E/S), Memória e quaisquer outros recursos providos pela mesma. Logo, a informação do perfil de consumo dos processos pode ser utilizada, por exemplo, na alocação dinâmica de recursos de infraestrutura da plataforma visando otimizações em diversos segmentos, como consumo de energia, Qualidade de Serviço, ou mecanismos auto adaptativos.

Os principais métodos de caracterização do perfil de aplicações estão inseridos basicamente em duas categorias: *identificação e inferência*. A primeira compreende a classe de técnicas de caixa-preta que são baseadas em análises de dados obtidos pelo monitoramento do programa durante sua execução aferindo os parâmetros de interesse no sistema. Entretanto, promover aferições demanda adequada instrumentação de toda plataforma (Sistema Operacional e hardware) para reduzir o impacto sobre os mesmos, assim como a garantia de que potenciais impactos da intervenção instrumental possam ser estimados ou avaliados. A validação dos resultados envolve um correto tratamento estatístico que deve ser considerado durante o planejamento dos experimentos, determinando a quantidade de replicações e garantindo um ambiente de execução controlado para evitar que dados espúrios causem interferência no resultado final. Já as técnicas baseadas em inferência utilizam abordagens alternativas à aferição, tentando inferir o perfil das aplicações por métodos que não necessitem de monitoramento em tempo de execução, por exemplo através da análise do código fonte do programa. A análise estática de código fonte (MUKHERJEE, 2014) (manual ou automática) possui o suporte de métodos formais e ferra-

mentas específicas que permitem a determinação de propriedades funcionais e não funcionais dos programas, assim como validação e verificação. Entretanto, a análise estática para avaliação de performance pode tornar-se difícil e passível de erros, especialmente para programas grandes e complexos. Embora possa ser aplicada em códigos binários(LIU; TAN; CHEN, 2013; BARDIN *et al.*, 2011), por exemplo em bytecode Java, a análise estática é majoritariamente desenvolvida para análises de código fonte, sendo portanto dependente da linguagem de programação adotada, além não considerarem otimizações feitas pelo compilador durante a geração do código final da aplicação. Em muitos casos não há disponibilidade de código fonte da aplicação, inviabilizando a análise estática.

Em face dos problemas intrínsecos da análise estática, outros métodos de inferência podem ser utilizados, tais como métodos baseados em técnicas de reconhecimento de padrões. Na prática essas abordagens resumem-se a detectar características da aplicação analisada comparando-a com diversas aplicações previamente conhecidas, a principal dificuldade entretanto consiste na determinação das métricas de similaridade. Se o comportamento de um programa é conhecido *a priori*, esta informação pode ser explorada. Por exemplo, um programa de processamento de imagens terá comportamento mais similar a outros programas da mesma categoria à um editor de textos. Em um ambiente privado, onde o conjunto de aplicações de um cliente é constante e conhecido previamente, este problema pode ser relativamente fácil de lidar. Entretanto, em ambientes mais amplos, tais como sistemas de computação em nuvem e alto desempenho, torna-se difícil efetuar o reconhecimento de padrões e predição de forma confiável. Diversos trabalhos vêm abordando este tipo de problema propondo técnicas de inferência que recorrerem a informações obtidas anteriormente, como logs de execução, utilizando dados históricos (geralmente disponíveis em logs de execução), por exemplo nome de programa, requisições de usuário e tamanho de arquivo(GIBBONS, 1997; DOWNEY, 1997; SEKAR; MANIATIS, 2011), em ambientes mais controlados, este tipo de dado tende a exibir consistência, permitindo utilização para comparação, como é explorado em trabalhos de alocação dinâmica em infraestruturas de supercomputadores(WARNEKE; KAO, 2011; SU *et al.*, 2013), porém, para aplicações e ambientes mais genéricos, a detecção de similaridade torna-se um problema mais desafiador.

Há uma vasta gama de métodos para detecção de similaridades e identificação de padrões em conjunto de dados que podem ser utilizadas nesse tipo de aplicação. Contudo, fatores como extensão do código e seus tamanhos variados, grande número de códigos no repositório, bem como o desafio em identificar as características relevantes para a sua comparação, tornam tanto um, quanto outro problema, consideravelmente difíceis e reduzem o leque de opções em métodos conhecidos, promissores para esses problemas. Esses mesmos fatores complicantes ocorrem também em alguns outros problemas na área de de processamento de textos e/ou longas sequências. Para isso, novas técnicas têm sido propostas. Dentre as mais recentes podem ser destacadas: *i*) projeções em espaço bidimensional por distância aproximada de Kolmogorov para detecção automática de similaridade entre documentos (TELLES; MINGHIM; PAULOVICH,

2005); ii) abordagem baseada em Redes Complexas para sumarização de textos; iii) técnica de Agrupamento de Múltiplas Instâncias de imagens e textos; iv) Aprendizado de Máquina baseado em *Mismatch Kernels* para classificação de proteínas (LESLIE *et al.*, 2004); v) abordagens que não baseiam-se em extração de características, tais como os métodos Comlearn (CILBRASI, 2003) e DAMICORE. Estas últimas são mais adequadas para aplicação com códigos binários uma vez que o processamento deste tipo de código exige maior conhecimento especializado em relação a códigos fonte e linguagem natural.

A abordagem para mineração de dados em repositórios de códigos denominada DAMICORE (DataMine a Code REpository) (SANCHES; CARDOSO; DELBEM, 2011) tem sido utilizada com sucesso em melhoria de desempenho e eficiência de sistemas computacionais baseados em hardware configurável, para identificação de partes comuns em artefatos de software que podem ser vantajosamente implementadas em hardware. O método DAMICORE baseia-se em técnicas oriundas da Teoria da Informação, Filogenia e Redes Complexas. O processo da técnica envolve a combinação de três métodos e tem como característica importante não assumir nenhuma premissa acerca da entrada, de modo que a técnica é aplicável a qualquer tipo de dado. No trabalho mencionado, o método é utilizado na classificação hierárquica de repositório de códigos fonte, para identificação de *software kernels*.

1.1 Objetivos

Este trabalho tem como objetivo desenvolver um novo método de análise de programas de computador capaz de correlacionar similaridades entre código binário e o perfil de utilização de recursos. A proposta pode ser formulada como o trabalho de i) identificar similaridades em códigos executáveis e ii) encontrar grupos com características comuns que permitam realizar a caracterização do perfil de consumo de recursos do programa a partir do seu código binário.

A questão de pesquisa abordada neste trabalho, portanto, tem como hipóteses que i) o padrão de consumo de recursos de um programa decorre das operações realizadas e sua interação com a plataforma e, portanto, similaridades entre perfis de consumo têm correspondência com similaridades no código executável; e ii) as similaridades nos códigos executáveis podem ser detectadas por métodos agnósticos em relação a semântica dos dados (sem conhecimento a priori das características principais dos códigos), propiciando subsídios para sua correlação com os perfis de consumo. A validação dessas hipóteses pertence ao escopo do trabalho.

1.2 Estrutura da Tese

Esta tese está dividida em quatro partes: Introdução, Desenvolvimento, Estudos de Caso e Conclusões. A primeira parte apresenta a motivação, o contexto do trabalho desenvolvido, objetivos e questões de pesquisa, assim como a revisão bibliográfica dos temas relevantes para o

trabalho. A segunda parte apresenta o desenvolvimento do trabalho: projetos de experimentos, análises dos resultados, índice de congruência para comparação de resultados, análises quantitativas e qualitativas de todos os resultados obtidos e a proposta de uma nova técnica que sistematiza os procedimentos de análise de códigos binários, denominada BCC (*Binary Code Clustering*). A terceira parte apresenta três principais estudos de caso onde a BCC é avaliada. Por fim, a quarta parte apresenta as conclusões do trabalho e referências bibliográficas.

REVISÃO

Este capítulo aborda os principais tópicos envolvidos no contexto deste trabalho, trabalhos relacionados e apresentação e discussão da técnica DAMICORE.

2.1 Caracterização de cargas de trabalho

No contexto dos sistemas computacionais, a predição e caracterização de carga abraça importantes aspectos com vasta aplicação, por exemplo, em sistemas de tempo real (KLUGE *et al.*, 2010), sistemas distribuídos (JU; XU; YANG, 1995),(DOULAMIS *et al.*, 2007), economia de energia (KAWAGUCHI; YACHI, 2013), virtualização (DAVIS *et al.*, 2013), banco de dados (WU *et al.*, 2013) e grids computacionais (LERIDA *et al.*, 2013).

Em relação aos processos, prever quais e como os recursos serão utilizados pode auxiliar diversas abordagens de escalonadores, controle de energia, balanceamento de carga, qualidade de serviço e SLAs¹, controle de armazenamento e base de dados, além de permitir a síntese de carga para utilização em simulações. Tem sido reconhecido, por exemplo, em em aplicações modernas de computação distribuída de larga escala, como em computação em nuvem, uma plataforma inteligente deve ser capaz de perceber as características da carga e provisionar sistemas operacionais especializados para oferecer máximo desempenho(ZHAO *et al.*, 2013).

Para que seja possível a execução de experimentos com diversas alternativas sob condições idênticas, é necessário que a carga de trabalho seja reproduzível, o que geralmente não ocorre em ambientes reais de execução. Neste caso, é preciso que o ambiente real seja observado (aferido, estudado) para que uma carga de trabalho sintética e reproduzível seja desenvolvida, seguindo o comportamento da carga de trabalho real. Este processo é chamado de caracterização de carga (JAIN, 1991).

Em (ZHAO *et al.*, 2013) os autores apresentam um modelo de classificação de carga

¹ Acordo de nível de serviço, do inglês, *Service-level agreement*

especializado em Sistemas Operacionais (SOs) para virtualização. Otimizações no SO são feitas a partir da classificação obtida, permitindo que cada carga seja executada com configurações do sistema personalizadas para seu respectivo perfil. Quatro classes de cargas são diferenciadas: *CPU-Intensive*, *I/O-Intensive*, *Memory-Intensive* e *Network-Intensive*, para os programas que fazem uso intensivo, respectivamente, do processador (CPU), entrada-saída (I/O), memória e rede. O modelo de classificação recebe 21 métricas de cada carga como entrada para executar a classificação. Um monitor é executado no sistema e faz aferições a cada 3 segundos, armazenando-as em um banco de dados, este tempo foi determinado pois segundo os autores resultados estatísticos mostraram que as cargas de trabalho não mudam frequentemente seu comportamento, não sendo necessário a aferição a todo momento. O processo de classificação é repetido a cada 30 minutos, reconfigurando os parâmetros do SO a partir da última classificação obtida.

Segundo (YANG *et al.*, 2013) existe uma característica notável nos padrões de acesso a dados em aplicação de larga escala em data centers: 80% das requisições geralmente acessam somente 20% dos dados, levando ao conceito de *hotspot data*, ou dados cruciais, em uma tradução livre. Os autores propõem um modelo dinâmico para predição de *hotspot data* em cargas de Entrada/Saída, que pode ser ajustado através de estatísticas de acesso I/O do sistema.

(CORNEA; BOURGEOIS, 2012) afirmam que a predição do desempenho de aplicações distribuídas é um constante desafio para a pesquisa, com dificuldade ainda maior quando sistemas heterogêneos estão envolvidos, o que acaba limitando-as ao tipo de aplicação, linguagem de programação ou sistema alvo. Uma ferramenta para predição de desempenho de aplicações distribuídas em sistemas heterogêneos é apresentada, mostrando-se altamente precisa. Entretanto a ferramenta possui suporte somente a aplicações escritas nas linguagens de programação C, C++ e Fortran, uma vez que está fundamentada em três fases: análise estática do código fonte, execução do código instrumentado por um analisador estático e *trace-based simulation*, que consiste em uma simulação efetuada registrando os “traços” de execução do programa ou componente do sistema com o propósito de se fazer a predição do desempenho.

Em (LIU *et al.*, 2013) é proposto o sistema *ACIC: Automatic Cloud I/O Configurator*, que busca automaticamente a melhor configuração de Entrada/Saída, a partir de diversos candidatos, para uma dada carga executando em uma nuvem. O sistema possui como alvo aplicações paralelas de alto desempenho (*High Performance Computing*) e utiliza um modelo de predição caixa preta com aprendizado de máquina, que explora diversas variáveis de configuração de Entrada/Saída do sistema em busca da melhor relação custo e performance. O usuário deve fornecer as características de Entrada/Saída de uma carga específica para o preditor, que retornará como saída a melhor configuração do sistema dentre várias disponíveis.

2.2 Técnicas de Mineração de Dados em Repositórios

A área de mineração de dados envolve uma grande diversidade de técnicas. Várias delas podem ser aplicadas a uma grande diversidade de problemas. As de maior desempenho em geral são mais complexas, ou que possuem propriedades que se ajustam melhor a certo problema, ou às quais foi utilizado conhecimento relevante do domínio do problema.

Em outras palavras, é preciso conhecer primeiramente as características fundamentais do problema em questão, podendo assim buscar na literatura pelas metodologias mais avançadas que têm sido utilizadas em problemas com características similares. De posse do problema sintetizado em características principais e das técnicas mais apropriadas para tais características, pode-se propor a investigação de soluções promissoras para o problema.

Os problemas de similaridade de códigos e identificação de padrões necessitam ser bem caracterizados para que seja possível definir um conjunto promissor de técnicas de mineração de dados para se investigar. Aspectos de relevância incluem:

1. Uma alta quantidade de códigos com as mesmas similaridades. Com isso, um grande número de problemas com códigos semelhantes poderá ensejar predições que explorem de forma mais eficiente suas componentes similares. Esse procedimento tem sido chamado de *design-suggestion*;
2. Identificação de aspectos do código que podem influenciar diretamente no desempenho e consumo de recursos. Esse procedimento é conhecido como *mapping suitability*.

Para ambos os casos, quanto maior o número de trechos similares maior será a relação entre custo e benefício da predição. Dessa forma, uma característica importante da técnica de mineração é sua capacidade em lidar com grande número de códigos e com códigos grandes. Em outras palavras, a técnica deve ser escalável.

Outro aspecto importante é a definição do que são as características ou as variáveis dos dados. Por exemplo, em dados de expressão gênica (HASTIE; TIBSHIRANI; FRIEDMAN, 2003), cada variável é em geral o grau de hibridização para cada tipo de gene. Assim, as características são os genes e pode-se utilizar um vetor para armazenar os graus hibridização para cada gene. Em análise de dados por técnicas de espectroscopia pode-se discretizar o espectro de frequências e armazenar as suas intensidade também em um vetor.

Em códigos computacionais estas características não estão necessariamente disponíveis. Não há um espectro ou um conjunto de genes pré-definido. Assim, primeiramente devem ser definidas as subestruturas de códigos a serem procuradas. Tais subestruturas podem não ocorrer exatamente como inicialmente definidas. Por exemplo, uma subestrutura de repetição (um *for* em linguagem C; ou sua transformação em código executável) pode ter variações no seu critério de parada ou no seu corpo que não são possíveis de serem todas previstas. Assim, subestruturas

similares devem ser identificadas como de um mesmo tipo. Em resumo, é preciso utilizar uma técnica de mineração com a qual se possa definir adequadamente um vetor de características (de subestruturas) e que essas não sejam rígidas, isto é, aceite-se em um nível controlável alguma flexibilidade na rotulação das características.

2.2.1 DAMICORE

DAMICORE (DAta Mining of Code REpositories) foi inicialmente introduzido por (SANCHES; CARDOSO; DELBEM, 2011). Trata-se de um arcabouço teórico composto pela combinação de várias técnicas provindas de diferentes campos: Teoria da Informação, Bioinformática e Redes Complexas, que são combinadas em uma cadeia capaz de prover relações hierárquicas entre objetos de dados não estruturados. O processo é efetuado a partir de um repositório de códigos (por exemplo arquivos de imagens, textos, áudio, quaisquer tipos de dados) através de três passos: a) dado uma *métrica de similaridade*, uma *matriz de distâncias* é construída comparando-se cada par de objetos do repositório de códigos; b) a matriz de distâncias é convertida em uma *rede* através da conexão de objetos mais próximos entre si; c) um processo de detecção de comunidades é aplicado para formar grupos de elementos próximos conectados à rede. Originalmente o método DAMICORE seleciona três algoritmos específicos para os propósitos descritos.

A métrica utilizada para a construção da matriz de distâncias é a NCD (*Normalized Compression Distance*), do inglês, Distância de Compressão Normalizada, que é uma aproximação computável da complexidade de Kolmogorov (LI; VITÁNYI, 2013; LUI *et al.*, 2015) e é definida pela equação 2.1.

$$NCD_z(a, b) = \frac{C_z(ab) - \min\{C_z(a), C_z(b)\}}{\max\{C_z(a), C_z(b)\}} \quad (2.1)$$

As variáveis a e b denotam dois objetos do repositório a serem comparados, ab é a concatenação de ambos objetos, e $C_z(x)$ é o tamanho da versão compactada do objeto x , obtida através da aplicação de um algoritmo de compressão Z . A fórmula da NCD pode ser entendida intuitivamente tendo-se em mente que qualquer algoritmo de compressão funciona através da detecção de padrões que se repetem dentro do arquivo a ser compactado, permitindo assim que sequências repetidas sejam representadas (e substituídas) por sequências menores. Logo, a concatenação de dois arquivos muito similares deverá produzir um arquivo compactado menor do que a compactação a partir da concatenação de dois arquivos muito diferentes, uma vez que um bom compressor deve ser capaz de expressar as partes do segundo arquivo como sequências repetidas do primeiro.

Considerando um compressor ideal, um arquivo resultante da concatenação de dois arquivos idênticos deve ser enxergado pelo compressor como duas sequências de dados iguais, ou seja, dado dois arquivos idênticos a e b , e o arquivo ab , resultante da concatenação de ambos,

ab pode ser obtido somente a partir de a (ou b) replicando-se o conteúdo total do arquivo para a geração de ab . Logo, o tamanho do arquivo ab será igual ao tamanho dos arquivos a e b compactados. A partir da equação 2.1, para um compressor ideal e dois arquivos idênticos:

$$C_z(a) = C_z(b) = C_z(ab)$$

$$NCD_z(a, b) = \frac{C_z(a) - C_z(a)}{C_z(a)} = 0$$

Já para dois arquivos totalmente diferentes, em um compressor ideal o tamanho do arquivo concatenado deve ser a soma dos tamanhos dos dois arquivos compactados. Considerando dois arquivos a e b totalmente diferentes, sendo $C_z(a) < C_z(b)$:

$$C_z(a) < C_z(b),$$

$$C_z(ab) = C_z(a) + C_z(b)$$

$$NCD_z(a, b) = \frac{C_z(a) + C_z(b) - C_z(a)}{C_z(b)} = 1$$

Para compressores não ideais, a NCD irá variar de 0 a 1 proporcionalmente ao nível de similaridade encontrado pelo algoritmo. Computando a distância para cada dois objetos de dados do repositório, obtém-se a matriz de distâncias D onde cada elemento D_{ij} corresponde a similaridade entre os objetos i e j de acordo com a NCD.

O segundo passo da técnica consiste na aplicação do algoritmo Neighbor joining (NJ)(FELSENSTEIN; FELENSTEIN, 2004), amplamente utilizado em bioinformática para a construção de árvores filogenéticas. Em biologia, o ramo da filogenia tem como objetivo trabalhar com as relações entre espécies, populações, indivíduos ou genes. Estas relações são descobertas através de métodos de inferência que buscam traços hereditários, tais como sequências de DNA ou estudos morfológicos (baseados na observação de estruturas presentes no corpo das espécies) visando estabelecer uma topologia de relações baseada na classificação, ou em um modelo de processo evolucionário. Os resultados das análises filogenéticas são geralmente apresentados como uma árvore filogenética (ou árvore evolucionária), cujos nós folhas são compostos por Táxon (ou taxa)² que são conectados de acordo com suas similaridades e ancestralidade. As árvores filogenéticas podem ser com raiz (que indica um único acentral original) ou sem raiz, e

² O termo Táxon (plural taxa ou táxons) refere-se a um grupo de uma ou mais populações de um organismo ou organismos, que são considerados pelos taxonomistas como uma unidade taxonômica.

Figura 1 – Método *Neighbor Joining*.

com ou sem ponderamento. Em uma árvore ponderada cada aresta possui um peso que indica a distância de seu ancestral.

Matrizes de distâncias são muito utilizadas em filogenética computacional como um método não paramétrico de distância, ou seja, que não depende de nenhuma distribuição estatística em particular. A partir de uma matriz de distância, diversas árvores filogenéticas podem ser construídas para expressar a filogenia dos elementos da matriz. A principal questão é determinar qual é a melhor árvore para expressar a filogenia em questão. O princípio da **máxima parcimônia** (HALL, 2011) determina que a árvore mais adequada é aquela que requer o menor número de mudanças evolutivas para explicar determinados dados observados. Uma vez que as mudanças entre ancestrais e seus descendentes é mensurada pelo comprimento da aresta, a máxima parcimônia equivale-se a obter a menor soma total das distâncias da árvore. Infelizmente, produzir todas as árvores possíveis a partir de uma matriz de distância para escolher a mais adequada só é possível para um conjunto muito pequeno de dados devido ao crescimento exponencial de acordo com o número de elementos. Assim, os algoritmos filogenéticos utilizam heurísticas ou técnicas mais avançadas para encontrar as árvores mais parcimoniosas dentre o conjunto de árvores possíveis. O algoritmo NJ trabalha neste sentido, utilizando uma abordagem *bottom-up* gulosa para minimizar a soma total dos comprimentos das arestas da árvore. O algoritmo funciona basicamente conectando, a cada iteração, as duas subárvores mais próximas ainda não conectadas (Figura 1). Iniciando pelos dois táxons mais próximos na matriz (A e B), o NJ conecta-os para formar um ancestral comum (X); os nós selecionados são substituídos pelo ancestral criado e uma nova matriz de distâncias é recalculada. O algoritmo procede de forma recursiva até fazer a última junção.

A distância (peso da aresta) entre o nó interno criado (ancestral comum) e cada elemento da nova matriz é calculada de tal maneira que as distâncias entre os nós folhas expressas na matriz original sejam preservadas. A cada iteração, os nós i e j com a menor distância $d(i, j)$ são obtidos a partir da matriz C calculada pela equação 2.2. As distâncias $\hat{d}$ dos nós x e y selecionados para o novo nó z são obtidas pelas equações 2.3 e 2.4, e a distância dos $n - 2$ nós restantes, r_i , para o novo nó é dada pela equação 2.5.

$$C_{ij} = (n-2)d(i, j) - \sum_{k=1}^n d(i, k) - \sum_{k=1}^n d(j, k) \quad (2.2)$$

$$\hat{d}(x, z) = \frac{1}{2}d(x, y) + \frac{1}{2(n-2)} \left[\sum_{k=1}^n d(x, k) - \sum_{k=1}^n d(z, k) \right] \quad (2.3)$$

$$\hat{d}(y, z) = d(x, y) - \hat{d}(y, z) \quad (2.4)$$

$$\hat{d}(r_l, z) = \frac{1}{2} [d(x, r_l) + d(y, r) - d(x, y)] \quad (2.5)$$

O ultimo passo do DAMICORE é a detecção de comunidades na árvore filogenética. Comunidade pode ser definida, de forma mais genérica, como um grupo de nós conectados mais próximos entre si em relação a outros nós na árvore. Na prática, uma comunidade também pode ser vista como um subgrafo com uma grande concentração de arestas contidas em certos grupos de vértices com um número menor de arestas entre estes grupos. Uma abordagem formal para identificar se um grafo possui estruturas de comunidades é através do conceito de modularidade (NEWMAN, 2006). A modularidade pode ser vista como a medida da razão da densidade de arestas de um subgrafo com a densidade esperada se as arestas estivessem conectadas por puro acaso, em um grafo totalmente aleatório (modelo nulo). Logo, um valor alto de modularidade indica que a densidade de arestas do subgrafo é maior que o esperado ao acaso, indicando a formação de uma comunidade. A fórmula de modularidade Q pode ser expressa pela equação 2.6.

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - P_{ij}) \delta(C_i, C_j) \quad (2.6)$$

onde A é a matriz de adjacência que representa o grafo, P_{ij} é a probabilidade dos nós v_i e v_j estarem conectados devido ao acaso (modelo nulo), n é o número de vértice, m é o número de arestas, e $\delta(C_i, C_j)$ é 1 se os nós estão na mesma comunidade, ou 0 caso contrário.

Dentre vários métodos disponíveis para detecção de comunidade em grafos, um amplamente utilizado é o *Fast Newman* (FN) (NEWMAN, 2004) devido sua simplicidade e eficiência. FN é um algoritmo guloso de abordagem *bottom-up* que procura por uma partição no grafo que maximize a modularidade. No início cada nó é considerado uma comunidade, em seguida é feita uma busca por pares de comunidades, que se combinadas, aumentam a modularidade. Depois de testar todas as combinações, o par selecionado é combinado em uma nova comunidade. O algoritmo repete este procedimento até que todo o grafo esteja combinado dentro de uma única comunidade. Este procedimento produz um agrupamento hierárquico, onde as partições com maior modularidade podem ser selecionadas.

A forma pela qual a NCD, NJ e FN são combinados em um processo de mineração de dados representam as principais propriedades do método DAMICORE. A métrica de similaridade entre objetos de dados é computada pela NCD em nível de bytes, fazendo com que o DAMICORE

seja agnóstico em relação a semântica dos dados, sendo capaz de trabalhar com quaisquer tipos de repositórios: textos, imagens, áudios, códigos fonte, programas executáveis, todos são representados como sequências de bytes. Adicionalmente, ao invés de apenas agrupar objetos semelhantes, NJ constrói uma filogenia expondo aspectos comuns que estão relacionados hierarquicamente. Por fim, a aplicação do FN identifica comunidades na filogenia para agrupar objetos relacionados em grupos. O método DAMICORE vem sido aplicado com sucesso em um vasto campo de aplicações, incluindo otimização de hardware(SANCHES; CARDOSO; DELBEM, 2011; SILVA *et al.*, 2015; SILVA *et al.*, 2015), compiladores(MARTINS *et al.*, 2014; MARTINS *et al.*, 2014), ambientes de *e-learning*(MORO *et al.*, 2013; MORO *et al.*, 2014), identificação de fitopatologias a partir de dados de sensores(FERREIRA; MELO; DELBEM, 2010), revisão sistemática de literatura(MARTINS *et al.*, 2016), otimização de sistemas elétricos de potência(MANSOUR *et al.*, 2013) e novas metodologias em bioinformática(M., 2013).

2.3 Trabalhos relacionados com uso do DAMICORE

Alguns trabalhos já fizeram uso da técnica DAMICORE para explorar a detecção de similaridade entre códigos dentro do contexto de consumo de energia, performance e compilação para arquiteturas baseadas em FPGA e sistemas computacionais. Entretanto, a mineração é aplicada a códigos fonte e/ou representações intermediárias dos mesmos. A originalidade deste trabalho encontra-se na utilização do DAMICORE diretamente aplicado ao código binário das aplicações, detectando similaridade que estejam relacionadas ao consumo de recursos das mesmas, sem a necessidade de conhecimento a priori a respeito do perfil de cada aplicação avaliada, não havendo a necessidade de disponibilidade de dados específicos, tais como aferições ou histórico de execução dos programas.

Em (SILVA, 2016) o autor investiga técnicas para otimização do fator desempenho/consumo de energia em arquiteturas multi-núcleos heterogêneas implementadas em FPGA. O autor combina técnicas aplicadas em tempo de compilação (*offline*) através da análise de código-fonte e *profiling* e técnicas aplicadas em tempo de execução (*online*), tais como mapeamento, escalonamento e redimensionamento dinâmico de cache. O DAMICORE foi uma das técnicas utilizada para análise *offline* do código fonte das aplicações.

Em (MARTINS, 2016) o autor desenvolve uma nova abordagem para exploração de sequências de otimizações de compilação aplicadas em códigos fonte de funções de acordo com o processador alvo e tipo de função implementada. O método não requer qualquer estudo prévio sobre o impacto das sequências escolhidas ou treinamento ou modelo que mapeie as relações entre as mesmas. A informação utilizada no processo de seleção de otimizações é extraída a partir de dados de históricos de compilações. O autor utiliza uma variação do método DAMICORE para promover agrupamento de códigos.

Em (SUMOYAMA, 2016) o autor trabalha com sistemas híbridos, compostos por códigos

que podem ser executados em processadores de propósito geral (GPP – *General Purpose Processor*), FPGAs e GPUs (*Graphics Processing Units*). O autor propõe uma técnica para desenvolvimento de um classificador de códigos (*kernels*) para mapeá-los de modo automático em arquiteturas híbridas, focando em GPP e FPGA. O classificador determina o dispositivo mais adequado para execução somente analisando o código fonte, sem a necessidade de execução do mesmo. A técnica proposta é baseada no DAMICORE. O classificador desenvolvido obteve taxas de acerto de 65,67% para códigos sintetizados para FPGA com a ferramenta *LegUP* e 59,19% para *Impulse C*. Em suas conclusões, o autor destaca a necessidade da preparação dos códigos a serem analisados pelo DAMICORE dado a sensibilidade da NCD em relação a conteúdos específicos como comentários, nomes de variáveis, etc, que podem influenciar na qualidade dos resultados. Neste sentido, o uso do código binário (abordagem proposta neste trabalho) elimina a necessidade de manipulações para geração de representações intermediárias, uma vez que dados não relevantes para os programas (como comentários, nomes de variáveis, etc) já são descartados no processo de compilação.

Parte II

Desenvolvimento

PROJETO EXPERIMENTAL

Este capítulo detalha o projeto dos experimentos executados neste trabalho: escolha de plataformas, programas, ferramentas, assim como a execução dos mesmos para obtenção dos resultados.

3.1 Plataforma e cargas de trabalho

A plataforma escolhida para o desenvolvimento dos experimentos é composta por um computador PC com processador de arquitetura Intel[®] 64 (x86 64 bits) executando o Sistema Operacional Linux com kernel versão 3.11, uma vez que tanto hardware quanto software são amplamente utilizados nos mais diversos segmentos. A arquitetura x86 está presente desde máquinas domésticas (Desktops) até data centers de empresas como Google e Facebook (BABCOCK, 2011). O Sistema Operacional Linux possui código-fonte aberto, com mais de 20 anos de ativo desenvolvimento, estando presente em 96,4% dos 500 melhores supercomputadores¹, além de servidores, *clusters* (aglomerados computacionais) de alto desempenho e sistemas de virtualização, como *Kernel-based Virtual Machine* (KVM)² e Xen³.

Em relação a carga de trabalho, neste caso denotada pelo perfil de consumo de recurso dos programas, dois tipos clássicos da literatura são cargas baseadas em *CPU-Intensive* e *I/O-Intensive*, ou em uma tradução livre, cargas intensas em CPU e a Entrada/Saída. Processos *CPU-Intensive* são processos que envolvem muita computação e pouco (ou nenhum) uso de Entrada/Saída (E/S ou I/O). Já os processos *I/O-Intensive* envolvem pouca computação e muita Entrada/Saída (DHAMDHERE, 2006). Diversos trabalhos abordam ou baseiam-se em cargas destes tipos para diversos aspectos, como predição, caracterização ou métrica a ser utilizada em

¹ Segundo estatística gerada online com dados atualizados até Novembro de 2013 no site TOP500 SUPERCOMPUTERS. Disponível em <<http://www.top500.org>>.

² <<http://www.linux-kvm.org>>

³ <<http://www.xenproject.org/>>

avaliação de desempenho ou como fator para escalonadores dinâmicos ((REDDY; BANERJEE, 1989),(WU; KUO, 1999),(HUI; CHANSON, 1999),(TAKEFUSA *et al.*, 2001),(VANMECHELEN; DEPOORTER; BROECKHOVE, 2008),(KIM *et al.*, 2009),(PU *et al.*, 2010),(MAHMOUD *et al.*, 2013)). Há ainda referências no que tange a economia de energia (CHOI; SOMA; PEDRAM, 2005) e cargas de trabalho de banco de dados (CURINO *et al.*, 2011).

Para compor o repositório de programas executáveis diversos softwares foram selecionados de diversas fontes para atuarem como carga de trabalho e uma pequena quantidade foi implementada para compor programas que exibissem determinado comportamento específico. No total, 80 softwares compõem o repositório para mineração. Dentre estes, buscou-se quarenta softwares com características de *CPU-Intensive* e quarenta com características de *I/O-Intensive*. Inicialmente a característica foi determinada através de um ou mais dos seguintes fatores: documentação e literatura disponível, descrição geral do software, breve análise do código-fonte e funcionalidades suportadas. Posteriormente, diversos experimentos foram executados para avaliar e confirmar a característica de cada carga de trabalho.

3.2 Seleção dos programas

A Tabela 1 contém a listagem de todos os programas que compõem o repositório, representados por p_i ⁴.

Para o grupo de programas *CPU-Intensive*, 13 algoritmos clássicos foram selecionados de (MENOTTI, 2010), onde o autor desenvolve uma linguagem para exploração do paralelismo de *loops* em computação reconfigurável, utilizando um conjunto de *benchmarks* para aplicar as técnicas pesquisadas e gerar os resultados. Os *benchmarks* selecionados incluem algoritmos para cálculo da série de Fibonacci, ordenação Bolha, algoritmos de processamento de imagem *Fast Discrete Cosine Transform* (FDCT) e Sobel, modulação de sinal *Adaptive differential pulse-code modulation* (ADPCM), operações vetoriais e de multiplicação de 32bits para processamento de imagem desenvolvidos pela Texas Instruments.

Outros programas selecionados envolvem o clássico *benchmark* Wheatstone (CURNOW; WICHMANN, 1976), geradores de fractais ((MANDELBULBER, 2013) e (MCCARTY, 2013)), gerador de números primos (BERNSTEIN, 2013), compressores de arquivos (HENNING, 2006), processador de árvores rubro-negras (MARTINIAN, 2013) juntamente com outros programas “estressadores” de CPU ((WILSHIRE, 2000) e (BURKARDT, 2013)). Completam o grupo *CPU-Intensive* nove programas implementados para os experimentos, sendo cinco algoritmos de ordenação e quatro algoritmos que executam os métodos numéricos Bisseção, Eliminação de Gauss, Simpson $\frac{1}{3}$ (integração) e Mínimos quadrados (PINTO, 2016).

O site <<http://01.org>> contém um repositório de projetos de código aberto que possuem envolvimento da Intel, desde aqueles em que a empresa atua apenas como colaboradora até

⁴ A partir de agora os programas serão referenciados por p_i de acordo com sua listagem na Tabela 1

aqueles em que é criadora e/ou mantenedora do projeto. Um dos projetos presentes é o **Linux Kernel Performance**⁵, que reúne diversos *benchmarks* para aferição em diversos subsistemas do Sistema Operacional, como rede, memória virtual e Entrada/Saída. Dos *benchmarks* presentes no projeto, foram escolhidos para atuarem como cargas *I/O-Intensive* os seguintes: *ffsb*, *fio*, *disk* (*lmbench*), *sysbench* (*fileio*), *tiotest* (*tiobench*) e *iozone*, com destaque para este último, que é um *benchmark* muito utilizado em trabalhos científicos. Uma busca por “iozone” na base de dados Google Scholar retornou 1520 resultados, registrando mais de 200 citações da ferramenta em referências bibliográficas.

Os *benchmarks* *seqread* e *randread* foram retirados do código da ferramenta *diskbench* (Diskbench, 2013), uma coleção de *benchmarks* para disco, disponível no repositório de códigos fonte Github⁶. Estes dois utilitários efetuam leituras sequenciais (*seqread*) e aleatórias (*randread*) em um dispositivo, não sendo dependente de sistemas de arquivos.

(CILIENDO; KUNIMASA, 2007) listam alguns *benchmarks* para Entrada/Saída e sistemas de arquivos, dentre eles a ferramenta Bonnie (BRAY, 2013), que também foi escolhida para compor as cargas *I/O-Intensive*. Outros *benchmarks* também foram selecionados, (KHLEBNIKOV, 2014), (LINUXINSIGHT, 2014), (Hewlett-Packard, 2013) assim como a ferramenta de “estresse” de disco *iothrash* (Fidelity National Information Services, 2013).

Muitos utilitários distribuídos em ambientes UNIX trabalham diretamente com Entrada/Saída. Alguns utilitários foram escolhidos para compor o repositório, incluindo checadores de consistência de sistema de arquivo (BAUMANN, 2014), e outros (GNU cpio, 2014; BERGLUND, 2014; NORDAL, 2014; DECKER *et al.*, 2014; GNU Freedup, 2014; Debian Project, 2013; SCHILLING, 2014; LOUGHER; LOUGHER, 2013; HELING, 2014; TRIDGELL; MACKERRAS, 2014; TRMAC, 2014).

Os programas *genload_io*, *dirload_io* e *randrw* foram implementados para completar a seleção das cargas. O primeiro escreve uma sequência de blocos em um arquivo criado, já o segundo cria uma sequência de diretórios, efetua escritas em um arquivo e posteriormente remove os diretórios criados. Por fim, *randrw* efetua duas rajadas de escrita de blocos de 158KB em um arquivo, voltando ao início do mesmo antes da segunda rajada.

Todos os programas foram obtidos com seus códigos fonte originais e compilados com o compilador C GNU *gcc* (GCCC, 2014) versão 4.7.3, com a flag de otimização padrão *-O0* (sem otimizações). Os executáveis ligados dinamicamente com a biblioteca C GNU *libc* (GNU, 2015) versão 2.2.5. Todos os testes foram executados na mesma plataforma, descrita na seção 3.4.

⁵ <<https://01.org/lkp/about>>

⁶ <<http://www.github.com>>

Tabela 1 – Programas binários do repositório.

Programas *CPU-intensive* Programas *IO-intensive*

Programa	Programa
p_1) adpcm_code	p_{41}) Bonnie
p_2) adpcm_deco	p_{42}) badblocks
p_3) arithoh	p_{43}) base64
p_4) autocor---	p_{44}) cat
p_5) bisection	p_{45}) cp
p_6) bubble_sor	p_{46}) cpio
p_7) bzip2	p_{47}) dd
p_8) combsort-	p_{48}) dirload_io
p_9) dhry2	p_{49}) disk
p_{10}) dhry2reg	p_{50}) du
p_{11}) dotprod---	p_{51}) duff
p_{12}) double	p_{52}) dupedit
p_{13}) dsp_mul32_	p_{53}) ffsb
p_{14}) fdct-----	p_{54}) find
p_{15}) fft-----	p_{55}) fio
p_{16}) fibonacci-	p_{56}) freedup
p_{17}) float	p_{57}) fsck.fat
p_{18}) fractal---	p_{58}) genisoimage
p_{19}) gauss---	p_{59}) genload_io
p_{20}) genload_wh	p_{60}) io_thrash
p_{21}) goraud----	p_{61}) ioping
p_{22}) gzip	p_{62}) iozone
p_{23}) hanoi	p_{63}) make-many-files
p_{24}) heap-----	p_{64}) md5sum
p_{25}) insertion-	p_{65}) mkisofs
p_{26}) long	p_{66}) mksquashfs
p_{27}) mandelbulber	p_{67}) ncd
p_{28}) max-----	p_{68}) od
p_{29}) mmq	p_{69}) randrw
p_{30}) pop_cnt---	p_{70}) recarray
p_{31}) primegen	p_{71}) rsync
p_{32}) radixsort-	p_{72}) seeker
p_{33}) redblack	p_{73}) shred
p_{34}) register	p_{74}) spew
p_{35}) selection-	p_{75}) split
p_{36}) short	p_{76}) sysbench
p_{37}) simpson13	p_{77}) tac
p_{38}) sobel---	p_{78}) tiotest
p_{39}) summillion	p_{79}) unsquashfs
p_{40}) vecsum---	p_{80}) updatedb

3.3 Perfis de referência

Dentre os aplicativos escolhidos como carga de trabalho, muitos possuem parâmetros que podem (ou devem) ser fornecidos pelo usuário e que impactam na quantidade de carga a ser gerada. Uma etapa preliminar aos experimentos consistiu na execução de cada um dos programas escolhidos para inspecionar o impacto no sistema em cada uma das execuções. Logo, empiricamente parâmetros que geravam diferentes cargas de E/S e CPU foram selecionados.

3.3.1 Ferramenta *mtool*

Para aferir quantitativamente o perfil das cargas de trabalho uma ferramenta denominada *mtool* (*Measure Tool*) foi implementada. A ferramenta funciona de maneira simples, recebendo como argumento via linha de comando o nome e caminho do programa a ser executado seguido dos parâmetros a serem passados para o mesmo. A ferramenta inicia a execução do comando fornecido (através das chamadas ao sistema *fork()* e *execve()*) e logo em seguida entra em um *loop* sendo bloqueada através da chamada ao sistema *sleep()* para que seja acordada apenas a cada 1 segundo para que as aferições sejam feitas. Esta abordagem foi utilizada para que a execução de *mtool* não impactasse de forma significativa a carga geral do sistema.

As aferições são feitas de maneira simples e direta através da leitura de valores contidos no diretório **/proc**, um sistema de arquivos virtual implementado pelo kernel onde cada arquivo (que só existe na memória) contém diversas informações e aferições do sistema. Para cada processo presente no kernel, um diretório é criado possuindo como nome o PID do processo correspondente. Dentre de cada um destes diretórios vários arquivos estão disponíveis contendo diversas informações a respeito do processo, que são fornecidas pelo próprio kernel. A ferramenta *mtool* efetua a leitura dos arquivos **stat**, **statm** e **io** relativos ao processo avaliado. No arquivo **stat** os valores lidos são:

- **utime**: Corresponde a quantidade de tempo que o processo gastou sendo executado em espaço de usuário, ou seja, executando instruções do próprio código e não chamadas ao sistema.
- **stime**: Corresponde a quantidade de tempo que o processo gastou sendo executado em espaço de kernel, ou seja, tempo gasto com chamadas ao sistema.

Os tempos são fornecidos em tiques de relógio do kernel. Na máquina utilizada nos experimentos o relógio do kernel foi configurado com o valor padrão de 100 tiques por segundo. Portanto, dado uma quantidade N de tiques, o tempo em segundos (T_s) é dado por $T_s = \frac{N}{100}$.

A cada aferição, *mtool* escreve em sua saída padrão os tempos (em tiques) gastos em modo usuário e kernel pelo processo avaliado. A taxa de utilização da CPU pode ser calculada

posteriormente através da porcentagem de tempo em que o processo permaneceu em execução durante cada segundo (100 tiques) ou em seu tempo total de execução.

A leitura do arquivo **statm** é feita apenas para aferir o tamanho total do segmento de dados e pilha do processo (incluindo bibliotecas utilizadas). Já em relação ao arquivo **io**, os valores lidos por *mtool* são:

- **read_bytes**: quantidade de bytes lidos do dispositivo (ou seja, que não estava presente em cache)
- **write_bytes**: quantidade de bytes escritos no dispositivo

Outra variável presente no arquivo é **cancelled_write_bytes**, que contém a quantidade de bytes que não foram efetivamente escritos no dispositivo, por exemplo quando é feita uma escrita em um arquivo, porém logo em seguida o arquivo é apagado, fazendo com que os bytes contabilizados em **write_bytes** não sejam escritos de fato no dispositivo.

É importante ressaltar que não é necessário a leitura desta variável, dado que a partição do disco utilizada nos experimentos de Entrada/Saída foi montada com o parâmetro *sync*, garantindo que as operações de escrita fossem efetivadas no disco de maneira síncrona (eliminando possíveis efeitos causados pelo cache de blocos). Todavia, testes foram executados e o valor lido de **cancelled_write_bytes** foi zero em todos os resultados, confirmando que esta variável é dispensável para os experimentos efetuados. A documentação completa do sistema de arquivos */proc* está disponível em ([SEIBOLD et al., 2009](#)).

Para tornar possível a aplicação de métodos estatísticos nos dados obtidos, a ferramenta *mtool* efetua um total de 30 aferições para cada aplicação avaliada. Assim é possível o cálculo de média, desvio padrão, variabilidade, intervalos de confiança, etc dos resultados. Após a execução da última aferição, um sinal SIGKILL é enviado ao processo avaliado para que o mesmo seja finalizado. O processo pai (*mtool*) fica aguardando pela finalização do processo filho (programa avaliado) através da chamada ao sistema *waitpid()* e executa a chamada ao sistema *getrusage()* com o parâmetro *RUSAGE_CHILDREN* para obter mais estatísticas sobre o processo que foi executado. Os seguintes valores são obtidos através desta chamada:

- **ru_maxrss**: quantidade máxima de memória física utilizada pelo processo (segmento de código mais segmento de dados), ou seja, toda memória que não foi para o disco através de swap.
- **ru_minflt**: *soft page faults*, quantidade de falhas de páginas que ocorreram mas que não foi necessário buscar a página de memória no disco (swap), bastando apenas fazer o mapeamento correspondente na tabela de páginas do processo.
- **ru_majflt**: *hard page faults*, quantidade de falhas de páginas que ocorreram e que foi necessário buscar a página em disco.

- **ru_inblock**: quantidade de vezes que o sistema de arquivo efetuou de fato uma operação de leitura de bloco (desconsiderando caches).
- **ru_oublock**: quantidade de vezes que o sistema de arquivo efetuou de fato uma operação de escrita de bloco (desconsiderando caches).
- **ru_nvcsw**: *voluntary context switch*, quantidade de chaveamentos de contexto voluntários, ocorrem quando o processo está aguardando um evento ou requisita um recurso que não está disponível no momento.
- **runivcsw**: *involuntary context switch*, quantidade de chaveamentos de contexto involuntários, ou seja, quando o processo é retirado de execução pelo escalonador.

O resultado de cada aferição é escrito na saída padrão logo depois de adquirido, assim como as estatísticas obtidas com `getrusage()`. A figura 2 ilustra a saída gerada para a carga de trabalho `dirload_io`.

```

1 ./io/dirload_io -s 800000 -n 100000
2 4 2 18712 0 868352
3 2 1 18712 0 876544
4 2 3 18712 0 933888
5 3 1 18712 512 925696
6 2 2 18712 0 950272
7 2 3 18712 512 892928
8 1 4 18712 0 937984
9 2 4 18712 512 917504
10 2 2 18712 0 950272
11 1 5 18712 512 1011712
12 3 3 18712 0 1019904
13 3 4 18712 512 1015808
14 3 4 18712 0 991232
15 3 4 18712 0 966656
16 0 8 18712 512 1019904
17 2 4 18712 0 991232
18 0 9 18712 512 1019904
19 2 6 18712 0 1015808
20 0 11 18712 512 958464
21 0 8 18712 0 925696
22 0 13 18712 512 974848
23 0 13 18712 0 983040
24 0 17 18712 512 929792
25 0 0 18712 0 958464
26 0 28 18712 512 937984
27 0 0 18712 0 868352
28 0 27 18712 0 958464
29 0 0 18712 512 991232
30 0 29 18712 0 950272
31 0 0 18712 512 1007616
32 1528
33 463
34 0
35 302
36 58080
37 23506
38 105
39

```

Figura 2 – Saída gerada com `mtool` mostrando as aferições da carga `dirload_io`.

A primeira linha contém o comando completo (com argumentos) executado. Em seguida, os resultados de cada aferição são dispostos em uma linha contendo cinco valores separados por um espaço simples e organizados na seguinte ordem: tempo de CPU em espaço de usuário, tempo de CPU em espaço de kernel, tamanho da pilha, quantidade de bytes lidos e escritos, respectivamente.

A sete ultimas linhas da saída contém os dados obtidos com *getrusage()*, na seguinte ordem: **ru_maxrss**, **ru_minflt**, **ru_majflt**, **ru_inblock**, **ru_oublock**, **ru_nvcsw** e **runivcsw**.

3.4 Execução dos experimentos

Após a definição das cargas de trabalho e dos parâmetros a serem utilizados todas as aferições foram executadas através da ferramenta *mtool*. Todos os experimentos foram executados em um computador PC Desktop com a seguinte configuração:

- Placa mãe Intel modelo DG31PR, versão AAE58249-306, e BIOS versão PRG3110H.86A.0071.2010.0318.1704
- Um pente de memória RAM de 2GB DDR2 800MHz
- Processador Intel® Core™ 2 Quad CPU, modelo Q8400 de 2,66GHz, dotado de 4 núcleos, cache L2 de 4MB e barramento frontal (*FSB*) de 1333MHz
- Disco rígido Samsung, modelo HD502HI, com capacidade de 500GB, 16MB de cache, velocidade de 5400 RPM e interface Serial ATA (SATA) 3,0Gb/s
- Sistema Operacional GNU/Linux Gentoo, kernel versão 3.11.0

Para cada programa selecionado foram feitas 30 aferições com *mtool*, conforme funcionamento descrito na seção 3.3.1. Entretanto, alguns dos programas possuíam tempo total de execução muito baixo, menor que 2 segundos. Uma alternativa plausível seria executar novamente o programa para cada nova aferição. Entretanto, para evitar influência causada pelos tempos de carregamento e inicialização das cargas nos resultados, optou-se pela implementação de um *loop* no código de cada carga com tempo de execução menor que 30 segundos. Tendo em mente a influência que a adição de um *loop* no código poderia causar nos resultados da aplicação do DAMICORE, testes preliminares foram executados em ambas as versões de códigos e binários, com e sem o *loop*. Os resultados mostraram que a adição de apenas um *loop* não teve influencia nos resultados do DAMICORE, portanto a abordagem adotada foi mantida para a execução dos experimentos. As tabelas 2 e 3 contém cada um dos comandos das cargas utilizadas e os respectivos parâmetros utilizados.

Tabela 2 – Parâmetros utilizados na execução dos programas *CPU-Intensive*.

<i>p</i>	Comando	Parâmetros
1	adpcm_code	
2	adpcm_deco	
3	arithoh	33
4	autcor---	
5	bisection	
6	bubble_sor	
7	bzip2	/media/public/public/VMachines/XP.vdi
8	combsort-	
9	dhry2	33
10	dhry2reg	33
11	dotprod---	
12	double	33
13	dsp_mul32_	
14	fdct---	
15	fft---	
16	fibonacci-	
17	float	33
18	fractal---	
19	gauss---	
20	genload_wh	100000000
21	goraud---	
22	gzip	/media/public/public/VMachines/XP.vdi
23	hanoi	33 5
24	heap---	
25	insertion-	
26	long	33
27	mandelbulber	../aux/test.frac
28	max---	
29	mmq---	
30	pop_cnt---	
31	primegen	10000000
32	radixsort-	
33	redblack	
34	register	33
35	selection-	
36	short	33
37	simpson13	
38	sobel---	
39	summillion	
40	vecsum---	

Tabela 3 – Parâmetros utilizados na execução dos programas *I/O-Intensive*.

<i>p</i>	Comando	Parâmetros
41	Bonnie	-s 300
42	badblocks	/dev/sda4
43	base64	/media/public/public/VMachines/XP.vdi
44	cat	/media/public/public/VMachines/XP.vdi
45	cp	/media/public/public/VMachines/XP.vdi /media/TESTE/XP.vdi
46	cpio	-I /media/public/public/ISOs/winxp.cpio -iv
47	dd	if=/dev/zero of=/media/TESTE/1giga bs=1M count=90
48	dirload_io	-s 800000 -n 100000
49	disk	/dev/sda4
50	du	/
51	duff	-r /
52	dupedit	-depth=100
53	ffsb	../aux/ffsb-conf
54	find	/
55	fio	../aux/fio-job
56	freedup	/
57	fsck.fat	/dev/sda4
58	genisoimage	-o /media/TESTE/tmpfile.iso /media/public/test
59	genload_io	-s 800000 -n 100000 -t2
60	io_thrash	2 2 /media/TESTE/iothrash 250000 5000 10 4096 4096 10000 10 90 0 10
61	ioping	-s 200000k -i0 -P33 -WWW /media/TESTE/teste
62	iozone	-l 2 -u 2 -r 10m -s 100m -F /media/TESTE/t1 /media/TESTE/t2
63	make-many-files	
64	md5sum	/media/public/public/VMachines/XP.vdi
65	mkisofs	-o /media/TESTE/tmpfile.iso /media/public/test
66	mksquashfs	/media/public/test/ /media/TESTE/dir.sqsh
67	ncdu	/
68	od	/media/public/public/VMachines/XP.vdi
69	randrw	800000
70	recarray	/media/TESTE/test.bin
71	rsync	-av /media/public/public/VMachines/ /media/TESTE/rsync/
72	seeker	/dev/sda4
73	shred	-n100 /media/TESTE/shredfile.raw
74	spew	-progress -pattern=numbers -min-buffer-size=64k 1g /media/TESTE/bigfile
75	split	-n4 /media/public/public/VMachines/XP.vdi
76	sysbench	-test=fileio -file-total-size=1G prepare
77	tac	/media/public/public/VMachines/XP.vdi
78	tiotest	-f 950 -t1 -k3 -k2 -d /media/TESTE
79	unsquashfs	-d /media/TESTE/iGO /media/public/public/igo.sqsh
80	updatedb	-o /tmp/updatedb.db

Os arquivos *../aux/test.frac*, *../aux/ffsb-conf* e *../aux/fio-job* são arquivos que contêm parâmetros utilizados pelos programas *mandelbulber*, *ffsb* e *fio* respectivamente, uma vez que estes aplicativos recebem os argumentos e parâmetros de execução através de arquivos. Já o arquivo *XP.vdi* é a imagem de uma máquina virtual, com tamanho total de 30GB e foi utilizado como entrada para os programas *p7*, *p22*, *p43*, *p44*, *p45*, *p64*, *p68*, *p75* e *p77*. Programas que promovem leitura recursiva de diretórios, como o utilitário para cálculo de uso de disco *du*, receberam como parâmetro de entrada a raiz do sistema “/” para gerar carga e garantir a execução durante os 30 segundos necessários para aferição.

A escolha do compressor utilizado nos experimentos foi feita a partir de uma análise com todos os arquivos do repositório. Todos os arquivos foram compactados utilizando os três compressores disponíveis no conjunto de ferramentas DAMICORE: **gzip**, **bzip2** e **ppmd**. A Tabela 4 contém a eficiência calculada para cada um dos compressores e programas do repositório. O compressor que obteve melhor desempenho foi o **ppmd**, logo, foi utilizado em todos os experimentos deste trabalho.

Todos os arquivos gerados nos experimentos de Entrada/Saída foram gravados em uma partição de 26GB montada no diretório */media/TESTE* com o parâmetro *sync*, garantindo que as operações de escrita sejam efetivadas no disco de maneira síncrona, sem efeitos de cache.

Todos os experimentos foram executados em um ambiente padrão, sem login gráfico, apenas com processos *daemon* básicos do sistema, como por exemplo *udev*, *agetty*, threads de kernel, etc. Em todas as execuções a carga total inicial no sistema estava abaixo de 1% e sem atividades de Entrada/Saída. A execução dos processos foi automatizada com o uso de scripts e a cada replicação de uma execução, um comando para limpeza de todos os caches de sistema foi executado (através do arquivo */proc/sys/vm/drop_caches*).

Os dados adquiridos foram armazenados em arquivos e posteriormente processados com o uso de scripts, softwares Octave⁷ e R⁸ para cálculos estatísticos e geração dos gráficos.

3.4.1 Aplicação do DAMICORE

A aplicação do DAMICORE foi feita por um sistema composto por diversos programas e scripts que executam as etapas sequenciais de geração da matriz de distâncias baseadas em NCD, através da suíte de utilitários *CompLearn*⁹, geração da árvore filogenética baseada em NJ, com o uso do software *PHYLIP*¹⁰, e classificação em *clusters* (agrupamentos) baseada em FN. Foi utilizada a implementação descrita em (SANCHES; CARDOSO; DELBEM, 2011). A visualização das árvores foi feita com o software *Figtree*¹¹.

⁷ Disponível em <<http://www.gnu.org/software/octave/>>.

⁸ Disponível em <<http://www.r-project.org>>

⁹ Disponível em <<http://www.complearn.org>>

¹⁰ Disponível em <<http://evolution.genetics.washington.edu/phylip.html>>

¹¹ Disponível em <<http://tree.bio.ed.ac.uk/software/figtree>>

Tabela 4 – Eficiência (em %) dos compressores utilizados no DAMICORE.

Programas <i>CPU-intensive</i>				Programas <i>IO-intensive</i>			
Prog.	gzip	bzip2	ppmd	Prog.	gzip	bzip2	ppmd
p_1)	68,27	66,19	72,06	p_{41})	65,33	66,00	68,32
p_2)	68,75	66,69	72,48	p_{42})	63,42	63,92	66,93
p_3)	68,06	66,38	71,98	p_{43})	66,10	65,29	69,15
p_4)	72,17	70,52	75,67	p_{44})	63,79	64,71	67,64
p_5)	67,52	65,51	71,03	p_{45})	64,91	65,79	68,09
p_6)	72,37	71,04	75,82	p_{46})	63,55	64,53	67,24
p_7)	69,89	69,63	71,65	p_{47})	60,98	60,99	64,39
p_8)	68,21	66,30	71,83	p_{48})	68,71	68,04	72,11
p_9)	67,24	66,08	70,57	p_{49})	63,70	63,71	67,10
p_{10})	66,90	65,82	70,15	p_{50})	63,58	64,26	66,95
p_{11})	72,81	71,25	76,31	p_{51})	69,69	67,74	71,22
p_{12})	67,28	65,55	71,25	p_{52})	61,45	61,46	65,68
p_{13})	70,29	68,59	73,89	p_{53})	65,48	66,20	68,89
p_{14})	76,45	75,20	79,05	p_{54})	63,59	65,13	67,50
p_{15})	75,14	74,56	78,73	p_{55})	71,50	74,03	75,48
p_{16})	72,53	71,25	76,03	p_{56})	65,42	65,53	68,55
p_{17})	67,25	65,64	71,24	p_{57})	61,76	62,31	65,90
p_{18})	69,14	69,72	72,31	p_{58})	68,95	69,40	71,45
p_{19})	73,86	73,01	76,99	p_{59})	69,60	68,76	73,03
p_{20})	65,61	64,31	69,01	p_{60})	62,39	62,08	65,61
p_{21})	72,09	70,75	75,46	p_{61})	62,33	62,45	65,97
p_{22})	61,18	60,72	63,98	p_{62})	63,68	64,21	66,82
p_{23})	66,86	65,32	70,81	p_{63})	68,17	67,01	71,81
p_{24})	68,31	66,22	71,73	p_{64})	65,18	65,32	68,45
p_{25})	69,83	67,60	73,35	p_{65})	63,32	63,68	66,16
p_{26})	67,58	66,15	71,54	p_{66})	61,42	62,71	66,14
p_{27})	70,63	71,13	71,81	p_{67})	60,47	60,81	64,38
p_{28})	72,89	71,31	76,41	p_{68})	65,50	65,31	68,09
p_{29})	66,35	65,01	70,22	p_{69})	67,95	66,58	71,66
p_{30})	72,27	70,60	75,71	p_{70})	69,84	67,94	73,49
p_{31})	68,85	66,46	70,16	p_{71})	60,68	61,83	64,54
p_{32})	68,00	65,58	71,40	p_{72})	73,85	73,22	76,94
p_{33})	70,18	69,71	73,74	p_{73})	61,74	62,60	65,53
p_{34})	67,46	66,04	71,37	p_{74})	67,59	67,78	70,12
p_{35})	69,61	67,92	73,15	p_{75})	63,58	64,20	67,11
p_{36})	67,46	65,99	71,37	p_{76})	61,81	61,91	64,83
p_{37})	65,49	63,41	69,26	p_{77})	64,23	64,74	67,82
p_{38})	70,96	69,42	74,57	p_{78})	66,29	65,91	69,24
p_{39})	65,04	63,56	68,66	p_{79})	69,00	69,34	71,53
p_{40})	72,84	71,64	76,23	p_{80})	64,64	65,85	68,56

OBTENÇÃO DOS PERFIS DE REFERÊNCIA

Este capítulo apresenta as aferições dos programas do repositório para traçar o perfil de referência para consumo de recursos do sistema.

4.1 Aferição das cargas de trabalho

A partir dos dados obtidos das aferições dos programas do repositório, dois tipos de gráficos foram traçados: o primeiro tipo (presente nas figuras 4, 5, 6, 7, 8, 9 e 10) corresponde aos dados temporais, obtidos mediante amostragem periódica em cada aferição, durante a execução do programa; e o segundo tipo (presente nas figuras 11, 12, 13, 14, 15, 16 e 17) corresponde aos valores agregados obtidos pela média observada ao longo de repetidas execuções, tendo os normalizados e intervalos de confiança calculados.

4.1.1 Gráficos com aquisições temporais

Os gráficos com os dados temporais, contém os valores obtidos em cada uma das aferições de cada programa. Foram traçados os valores correspondentes ao uso de CPU em modo usuário (*CPU/user*), uso de CPU em modo kernel (*CPU/kernel*), sendo ambos expressos em porcentagem. Também foram traçados os valores correspondentes a quantidade de dados lidos (*IO/Read*) e escritos (*IO/Write*) no disco, ambos expressos em Kilobytes/segundo (KB/s).

Inspecionando os gráficos referentes aos programas *CPU-Intensive*, que correspondem as figuras 4, 5, 6, 7a, 7b, 7c e 7d, é possível observar o aspecto essencial do perfil de carga: alto uso de CPU (modo usuário ou kernel), todos acima de 80% e a maioria próximos de 100%, e baixo uso de Entrada/Saída (próximo ou igual a zero). Apenas o programa **Primegen** (6g) apresentou comportamento diferente. Apesar de não ter exibido nenhuma atividade de E/S, apresentou atividade intensa de CPU somente em modo kernel, ou seja, durante os 30 segundos de execução o uso de CPU modo kernel manteve-se próximo de 100% e todos os outros recursos

mantiveram-se próximos a zero. Este comportamento pode ser explicado através da inspeção do código e análise de execução do **primegen**. A função do programa é efetuar um algoritmo para cálculo de números primos em sequência e exibir cada número calculado por vez na saída padrão através da função *printf()*, da biblioteca C. A função *printf()*, por sua vez, faz as formatações necessárias e gera a string final, que é escrita na saída padrão através da chamada ao sistema *write()*, que consequentemente será executada em modo kernel. Dado a potência do processador utilizado e o custo do algoritmo implementado, pode-se concluir que as sucessivas chamadas de *write()* tenham consumido a CPU por muito mais tempo do que os cálculos executados para a geração dos números primos, visto que uma chamada ao sistema envolve troca de contexto, ações por parte do kernel para gravação do estado do sistema, ou seja, possui considerável custo computacional. É importante ressaltar que apesar da chamada *write()* ter sido utilizada, a saída padrão não estava redirecionada, tendo como saída o console (terminal), e portanto movimentando dados apenas na memória, e não em um dispositivo de armazenamento.

Já nos gráficos referentes aos programas *I/O-Intensive*, que correspondem as figuras 8, 9, 10, 7e, 7f, 7g, 7h, 7i, 7j, 7k e 7l é possível observar diversos comportamentos de cada programa. Todos executaram atividade de Entrada/Saída, como esperado, trabalhando em diversas taxas de leitura e/ou escrita, que estão relacionadas com o tipo de tarefa executada pela aplicação. As taxas variaram desde 250KB/s até 104MB/s.

Em relação ao uso de CPU, dos 40 programas *I/O-Intensive*, 33 mantiveram a taxa de utilização de CPU abaixo ou próximo de 20%, ficando a maioria próxima de zero. Os programas *p41*, *p43*, *p54*, *p61*, *p64*, *p66* e *p68* apresentaram comportamento diferentes.

- **Bonnie**(*p41* – Figura 7e): apresentou um comportamento oscilatório, iniciando a execução com alta taxa de utilização de CPU (por volta de 80%) até o instante 4s. Entre os instantes 5s e 7s a taxa de utilização de CPU vai a zero e a taxa de E/S atinge pico de 300000KB/s, comportamento que é repetido entre os instantes 14s e 16s. Posteriormente não há mais utilização de E/S e a utilização de CPU (modo usuário) eleva-se a mais de 90% até o instante 25s indo a zero a partir do instante 26s.
- **base64**(*p43* – Figura 7g): apesar de apresentar taxa de E/S da ordem de 72613888KB/s, o programa manteve taxa de utilização de CPU da ordem de 60% durante sua execução.
- **find**(*p54* – Figura 8f): apresentou um comportamento oscilatório no consumo de E/S e em alguns momentos teve picos de utilização de CPU no modo kernel (chegando a 80%).
- **ioping**(*p61* – Figura 9a): apresentou um comportamento notadamente oscilatório no consumo de E/S e CPU em modo kernel, variando entre 0 e 40% e entre 0 e 164245504KB/s, respectivamente.

- **md5sum**(p_{64} – Figura 9d): semelhante ao **base64**, apresentou taxa de E/S da ordem de 72613888KB/s durante sua execução, mas também manteve utilização de CPU por volta de 70%.
- **mksquashfs**(p_{66} – Figura 9f): até o instante 11s o programa apresentou uma taxa de E/S (somente leitura) da ordem de 57933824KB/s e consumo de CPU em 100%, posteriormente o consumo de CPU reduz e mantém-se por volta de 6% enquanto que a taxa de E/S (leitura e escrita) permanecem na ordem de 917504KB/s.
- **od** (p_{68} – Figura 9h): apresentou uma taxa de E/S da ordem de 3932160KB/s com dois picos, porém durante toda sua execução também mantém elevado o consumo de CPU, sempre próximo de 100%.

4.1.2 Gráficos com dados agregados

Para avaliar estatisticamente os dados das aferições, foram calculados a média e o intervalo de confiança (IC) com nível de 95% para os dados de consumo de CPU (modo usuário e kernel) e E/S (leitura e escrita). Gráficos de barras foram traçados para cada um dos programas do repositório. Foram acrescentados os valores **ru_inblock** e **ru_oublock**, chamados de **OP in** e **OP out** nos gráficos, respectivamente. Conforme explicado no capítulo 3, estes valores determinam a quantidade de vezes que o sistema de arquivo efetuou de fato uma operação de leitura/escrita de bloco (desconsiderando caches).

Uma vez que os programas *I/O-Intensive* apresentaram diversas taxas de E/S, muitas destas, centenas ou até milhares de vezes menor ou maior do que as outras, para prover uma forma mais imediata de comparação, todos os valores referentes a E/S (leitura, escrita, OP in e OP out) foram normalizados em 0 a 100%. No caso da leitura/escrita, 100% corresponde ao maior valor obtido para cada um dos dois ($\approx 104MB/s$ e $\approx 300MB/s$, respectivamente). Analogamente, 100% em **OP in** e **OP out** corresponde aos respectivos maiores valores obtidos (6499920 e 6315520). Não há IC para os valores de **OP in** e **OP out** dado que foram adquiridos apenas uma vez (valor total), através da chamada *getrusage()*, durante cada execução.

Os gráficos referentes aos programas *CPU-Intensive* correspondem as figuras 11, 12, 13, 14a, 14b, 14c e 14d. É possível observar os mesmos comportamentos expressos pelos gráficos de dados temporais, com a informação adicional provida pelo IC, que mostra pouca ou nenhuma dispersão entre os gráficos de consumo de CPU.

Os gráficos referentes aos programas *I/O-Intensive* correspondem as figuras 14e, 14f, 14g, 14h, 14i, 14j, 14k, 14l, 15, 16 e 17. Devido a normalização, alguns gráficos ficaram com valores de E/S muito baixos, portanto para melhor visualização das barras, foi utilizada uma escala logarítmica no eixo Y, mesmo assim, alguns gráficos ainda ficaram abaixo de 1%, não possuindo uma barra visível.

Em relação a dispersão, o maior valor pode ser observado no consumo de CPU em modo kernel para o programa **Bonnie**, e na taxa de leitura para o programa **unsquashfs**.

É importante salientar que apesar da normalização, os valores de E/S indicam a porcentagem em relação a referência escolhida, e os gráficos de consumo de CPU indicam a porcentagem com base no tempo. Portanto, são métricas diferentes e que não podem ser diretamente comparadas. Assim, não é possível afirmar que determinado processo consumiu menos CPU do que E/S simplesmente comparando ambas as porcentagens. Todavia, a comparação entre gráficos é perfeitamente possível desde que avaliadas as mesmas métricas. Por exemplo, observando os gráficos referentes aos programas **od**(p_{68} , Figura 15j) e **split**(p_{75} , Figura 17c) podemos afirmar que **od** consumiu mais CPU do que o **split**, uma vez que a média de consumo ficou próxima de 100% com um pequeno intervalo de confiança (baixa dispersão). Já o programa **split** obteve média de consumo de CPU menor que 1% em modo usuário e por volta de 10% em modo kernel, ambos com pequeno IC. Analisando analogamente a taxa de leitura de E/S conclui-se que a carga **split** efetua mais operações de E/S do que a carga **od**. A mesma análise se aplica aos valores **OP in** e **OP out**. Outra informação a ser observada são os tamanhos dos arquivos binários dos programas do repositório, que possuem tamanhos variados, desde 8KB até 2MB. A Figura 3 contém o tamanho de cada programa do repositório.

Figura 4 – Gráficos com perfil de uso de CPU e E/S dos programas (p_1 à p_{12}).

Figura 5 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{13} à p_{24}).

Figura 6 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{25} à p_{36}).

Figura 7 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{37} à p_{48}).

Figura 8 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{49} à p_{60}).

Figura 9 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{61} a p_{72}).

Figura 10 – Gráficos com perfil de uso de CPU e E/S dos programas (p_{73} à p_{80}).

Figura 11 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_1 à p_{12}).

Figura 12 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{13} à p_{24}).

Figura 13 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{25} à p_{36}).

Figura 14 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{37} à p_{48}).

Figura 15 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{49} à p_{60}).

Figura 16 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{61} à p_{72}).

Figura 17 – Gráficos normalizados com perfil de uso de CPU e E/S dos programas (p_{73} à p_{80}).

ÍNDICE DE CONGRUÊNCIA PAR-A-PAR

Este capítulo apresenta um índice de congruência para avaliar a correspondência entre um ou mais repositórios analisados com o DAMICORE.

5.1 Uma métrica quantitativa de congruência de agrupamento

Um algoritmo de agrupamento tem como objetivo agrupar elementos que sejam semelhantes (dado uma métrica de similaridade), ou seja, dois elementos inseridos no mesmo grupo supostamente serão mais semelhantes entre si do que do restante dos elementos do conjunto (repositório). Entretanto, esta semelhança é relativa à todo conjunto de elementos, não havendo garantias de que irá prevalecer caso novos itens sejam adicionados ou removidos do conjunto. Se há congruência no agrupamento de dois elementos, ou seja, ambos são agrupados no mesmo grupo quando o algoritmo é aplicado em diferentes amostragens do conjunto, então o resultado pode ser considerado robusto. Portanto, é possível expressar uma métrica de robustez através da contagem do número de congruências em associações par-a-par de elementos entre diversos experimentos de agrupamento.

Em outro contexto, o número de congruência também pode quantificar a correlação de similaridades encontradas pelas análises de diferentes repositórios de dados. Por exemplo, a partir de um conjunto de programas, diferentes repositórios podem ser formados, por exemplo pelos arquivos em sua forma binária, código fonte, ou até dados de execução (aferências) no sistema. Um algoritmo de agrupamento pode ser aplicado em cada conjunto para investigar similaridades detectadas sob o aspecto de cada análise, permitindo inclusive avaliar quais semelhanças preservam-se entre os diferentes experimentos.

Para quantificar o nível de correspondência dos agrupamentos é importante se determinar uma métrica consistente, capaz de avaliar todas as congruências obtidas por diferentes análises.

Seja G_1 e G_2 dois grupos encontrados por dois experimentos diferentes, por exemplo, grupos formados pela análise de um conjunto de programas binários e grupos formados pela análise do conjunto dos códigos fontes dos programas. Se há alguma correlação entre os dois aspectos, é esperado que muitos programas que sejam agrupados juntos em G_1 , também estejam agrupados juntos em G_2 . O número de congruências em agrupamentos par-a-par contados nos dois experimentos pode representar uma estimativa quantitativa da correspondência entre ambos.

5.1.1 Índice de congruência par-a-par

Diversos métodos para análise de agrupamentos estão disponíveis na literatura, (RAND, 1971; JACCARD, 1912; FOWLKES; MALLOWS, 1983; MEILĀ, 2007), sendo definidos para diferentes propósitos. Com o intuito de fornecer a noção intuitiva de porcentagem de congruência em um método simples de ser calculado, uma possível métrica de congruência, ρ , será aqui definida como *índice de congruência par-a-par*, sendo expressada pela equação 5.1.

$$\rho = \frac{N_e}{N_t} \quad N_t = \max\{N_1, N_2\} \quad (5.1)$$

onde N_e é o número efetivo de congruências encontradas, e $N_t > 0$ é a máxima quantidade de congruências possíveis para o conjunto em análise. Se N_k é o número de associações (par-a-par) em um agrupamento k , então o máximo número de congruência irá ocorrer se todas as associações (agrupamentos) se repetirem no outro conjunto de dados.

Seja j um conjunto de grupos encontrados a partir da análise de um determinado repositório e seja i o i -ésimo grupo de j , se n_{ij} é o número de elementos do grupo, então a máxima combinação de elementos será dada por $\binom{n_{ij}}{2}$ ¹. Somando-se todas as combinações de cada grupo do conjunto obtemos N_j , que representa a máxima congruência possível para o conjunto j . Logo, para dois conjuntos (1, 2) o valor máximo de N_t será $\max\{N_1, N_2\}$, o que pode ser estendido para qualquer quantidade de conjuntos $m, m > 2$, onde $j = 1, 2, \dots, m$.

Como exemplo, considere os dois conjuntos de agrupamentos, G_1 e G_2 , obtidos a partir de duas análises independentes:

$$\begin{array}{ll} G_1 : & G_2 : \\ & K_0^2 = \{a\} \\ K_0^1 = \{a, b, c, d, e, f\} & K_1^2 = \{b, c, d, e, f\} \\ K_1^1 = \{g, h\} & K_2^2 = \{g, h\} \\ K_2^1 = \{i, j\} & K_3^2 = \{i, j\} \end{array} \quad (5.2)$$

¹ Da análise combinatória, o número de combinações (não importando a ordem) $C_{n,k}$ é o coeficiente binomial:

$$C_k^n = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Seja $\{a, b, c, d, e, f, g, h, i, j\}$ um conjunto de programas, G_1 o agrupamento obtido pela análise dos programas em formato binário e G_2 obtido pela análise dos respectivos códigos fonte dos programas. Em G_1 obteve-se três grupos, K_0^1 , K_1^1 e K_2^1 , os grupos K_1^1 e K_2^1 possuem apenas dois elementos, o que caracteriza apenas uma única associação par-a-par em cada: (g, h) e (i, j) . Já o grupo K_0^1 possui seis elementos, portanto permitirá $\binom{6}{2} = 15$ associações par-a-par: $(a, b), (a, c), \dots (a, f), (b, c), (b, d), \dots (e, f)$. Somando todas as associações, têm-se $N_1 = 17$.

Esta análise é aplicada exatamente da mesma forma para G_2 , onde foram identificados quatro grupos, K_0^2 , K_1^2 , K_2^2 e K_3^2 . Os grupos K_2^2 e K_3^2 possuem apenas dois elementos, totalizando apenas duas associações possíveis. Já o grupo K_0^2 possui apenas o elemento a , logo nenhuma associação par-a-par é possível. O grupo K_1^2 possui cinco elementos, totalizando $\binom{5}{2} = 10$ associações possíveis. Portanto, $N_2 = 10 + 1 + 1 = 12$.

Seja U_{G_1} o conjunto que contém todas as associações par-a-par possíveis de G_1 e V_{G_2} o conjunto que contém todas as associações possíveis de G_2 . O número efetivo de congruências entre ambos (N_e) será dado por $N_e = n(U_{G_1} \cap V_{G_2})$, ou seja, a quantidade de elementos comum aos dois conjuntos.

Em resumo, para o exemplo apresentado têm-se:

$$N_e = 12 \quad N_t = \max\{17, 12\} = 17 \quad \rho = \frac{N_e}{N_t} = \frac{12}{17} = 0.7059 \quad (5.3)$$

onde $\rho \approx 0.70$ — o que poderia significar que aproximadamente 70% dos arquivos de código fonte que foram considerados próximos pelo algoritmo de agrupamento também foram considerados similares em sua forma binária. Entretanto, é importante notar que assim como diversos outros índices da literatura, o valor de ρ não está mapeado em uma escala linear de acordo com o número de congruências. No exemplo apresentado, a única diferença entre G_1 e G_2 é a ausência do elemento a no grupo K_1^2 , sendo agrupado sozinho em K_0^2 . Apenas essa diferença é suficiente para reduzir ρ de 100% (congruência total) para 70%. A exclusão de um único elemento de um grupo de tamanho n irá reduzir o número de associações par-a-par possíveis do grupo em $n - 1$. Por exemplo, se o elemento b do grupo K_1^2 também for excluído (ficando em um grupo sozinho, assim como o elemento a), ρ cairá para aproximadamente 47%. Esta informação deverá sempre ser considerada durante a análise do índice ρ obtido em qualquer experimento.

Visto que o estudo e a dedução matemática formal de todas as propriedades numéricas de ρ fugiriam ao escopo deste estudo, é possível verificar o decaimento do seu valor em uma aplicação do DAMICORE através de um experimento executado utilizando um repositório de textos obtidos do conhecido corpus de contos ingleses *The Canterbury tales* (CHAUCER, 1993). Um repositório inicial com 80 textos extraídos do corpus foi criado e a partir deste repositório nove outros repositórios foram gerados automaticamente modificando os arquivos do repositório original a uma taxa constante. Para gerar os repositórios o programa *randbin*² foi implementado.

² Disponível em <https://github.com/rene/randbin>.

O programa basicamente recebe como parâmetros de entrada um arquivo e uma porcentagem de modificação. A partir destas informações o arquivo de entrada será lido e posições aleatórias dentro do mesmo serão sorteadas. Nas posições sorteadas cada byte será modificado por um valor também aleatório. O programa irá sortear um valor até que o mesmo seja diferente do arquivo original, ou seja, garantindo que o valor original será efetivamente modificado. O número de bytes modificados é calculado para respeitar a taxa definida no parâmetro de entrada. Assim, em um arquivo com 100 bytes, uma modificação de 50% irá efetivamente modificar 50 bytes (escolhidos em posições aleatórias) dentro do arquivo.

A partir do repositório de textos original, os nove repositórios foram gerados através do *randbin* com as seguintes taxas de modificação: 1%, 5%, 10%, 15%, 30%, 40%, 60%, 80% e 100%, logo, o segundo repositório contém arquivos 1% modificados em relação ao repositório original, e assim por diante. O DAMICORE foi aplicado em todos os dez repositórios criados (original + gerados) e a congruência entre todos os agrupamentos providos foi calculada. A Figura 18 ilustra o decaimento do índice de congruência: o gráfico apresenta a congruência entre o repositório original e cada um dos repositórios gerados, o primeiro ponto consiste na congruência entre o repositório original consigo mesmo, totalizando uma congruência de índice 1.0 (100%), o segundo ponto representa a congruência entre o repositório original e o modificado em 1%, levando a uma queda da congruência para 0,54 (54%), e assim por diante. O eixo das abcissas contém o nível de modificação de cada repositório.

O gráfico também contém a média da distância NCD (com intervalo de confiança de 95%) entre os arquivos de cada repositório gerado e o original, os intervalos obtidos são expressivamente baixos, mostrando a baixa variabilidade das médias. É possível observar que o segundo repositório, que possui taxa de apenas 1% de modificação, ou seja, não há grande diferença entre os arquivos de ambos os repositórios, a média da distância NCD eleva-se para 0,55 assim como ρ decai para 0,54, o índice de congruência segue decaindo rapidamente, atingindo 28% para o repositório com 15% de modificação e aproximadamente 7% para o repositório com 30% de modificação, mantendo-se na faixa de 4% para os demais repositórios. É importante notar que a partir de um certo nível de diferença entre os arquivos, tanto a distância NCD quanto o índice de congruência tendem a estacionar. Além da queda rápida da congruência para dados com relativa diferença, deve-se considerar o fato de que mesmo para arquivos totalmente distintos ainda haverá grupos providos pelo DAMICORE dado a natureza do algoritmo de agrupamento utilizado. Logo, é natural uma congruência diferente de zero mesmo para dados totalmente distintos. Observando o gráfico e considerando a rápida queda do índice, pode-se inferir que uma congruência acima da linha estacionária representa um valor alto de semelhança, mesmo que o valor absoluto do índice esteja distante de 1.0, por exemplo, para arquivos muito semelhantes (1% modificados) o valor de 0,54 para ρ pode ser considerado um valor alto de congruência.

Um outro exemplo numérico do índice é dado pela Tabela 5, que mostra a congruência obtida para repositórios com dados totalmente não relacionados entre si. Três repositórios foram

Figura 18 – Decaimento do valor numérico de ρ .

utilizados no experimento: o repositório com os 80 programas binários considerados neste trabalho; um repositório com 80 arquivos de música instrumental no formato *Musical Instrument Digital Interface* (MIDI); e o repositório com os 80 textos do corpus Canterbury utilizado no experimento de decaimento do índice descrito nesta seção. O DAMICORE foi aplicado em todos os repositórios e a congruência entre os mesmos calculada.

Tabela 5 – Valores de ρ para comparação entre repositórios não relacionados, com grande diferença de conteúdo.

	Programa	Música	Texto
Programa	1.000	0.067	0.072
Música		1.000	0.076
Texto			1.000

Os resultados contidos na Tabela 5 mostram baixa congruência entre os repositórios: 6,7% entre Programas e Músicas, 7,2% entre Programas e Textos e 7,6% entre Músicas e Textos. Este valores estão em conformidade com a discussão prévia a respeito do decaimento da congruência, não é possível afirmar que há semelhança entre os dados devido a faixa de valores

de congruência obtidos, e neste caso, de fato não há semelhança entre os dados analisados uma vez que tratam-se de três repositórios com dados bem distintos (Programas, Músicas e Textos). Toda esta discussão deve ser levada em consideração durante a análise dos índices de congruência apresentados neste trabalho.

5.2 A ferramenta matches

Para facilitar o cálculo do índice de congruência para os diversos experimentos realizados, foi projetado e implementado o aplicativo matches, um programa desenvolvido em linguagem C que processa as saídas geradas pelo software DAMICORE e produz um relatório com várias informações. O programa matches toma como argumento o nome de um diretório onde espera que todos os arquivos sejam listas do tipo “item-grupo” geradas pela implementação do DAMICORE utilizada. O aplicativo lê todos os arquivos e computa alguns dados, dentre eles:

- uma listagem de todos os itens e os grupos em que são classificados por cada aspecto (no exemplo, código fonte, perfil de consumo de recursos, etc.),
- matriz (quadrada) de congruências de dimensão igual ao número de itens (programas), onde cada elemento é o número de vezes que o par foi agrupado em um mesmo grupo;
- sumário indicando quais pares apresentam uma, duas, três, etc. congruências, e o índice de congruência.

AVALIAÇÃO QUANTITATIVA

A partir do índice de congruência apresentado no Capítulo 5, que define uma métrica quantitativa para avaliar a correspondência entre repositórios distintos analisados com o DAMICORE, os resultados obtidos com o repositório de programas binários também foram avaliados sob outros aspectos juntamente com outros experimentos e repositórios relacionados afim de avaliar a robustez do método. Este capítulo apresenta uma análise quantitativa de todos estes resultados obtidos.

6.1 Análise dos agrupamentos

Dado que o DAMICORE é agnóstico em relação ao tipo de dado dos objetos que compõem o repositório analisado, o mesmo pode ser aplicado nos programas binários assim como em seus códigos fonte, que representam cada programa em uma outra linguagem, neste caso a linguagem de programação (léxica) ao contrário da linguagem de máquina (binária). É possível também analisar os dados dos perfis de consumo de recursos (obtidos pelas aferições dos programas) e comparar como todas estas “representações” dos programas são avaliadas pelo DAMICORE. Se a técnica é robusta, é esperado que semelhanças detectadas no código fonte, por exemplo, também sejam detectadas sob outros aspectos, tal como na representação binária dos programas. Neste sentido, além dos códigos binários, os códigos fonte e outros dados foram analisados pelo DAMICORE e comparados para avaliar a robustez dos agrupamentos resultantes.

6.1.1 Avaliação de robustez

Para avaliar a robustez do agrupamento provido pelo DAMICORE cinco repositórios foram avaliados, cada repositório representando os 80 programas (Tabela 1) sob diferentes aspectos. Cada repositório será denotado por f_i ($i = 1, 2, \dots, 5$), estando todos listados na Tabela 6. O primeiro repositório f_1 consiste no repositório dos 80 programas na forma binária compilados

com o compilador *gcc* e *flag -O0*. O repositório f_2 consiste em 80 arquivos que representam todo o código fonte de cada programa. Os códigos que estavam organizados em mais de um arquivo foram concatenados, fazendo com que todo o código fonte fosse representado por apenas um arquivo. Apenas arquivos de código e cabeçalhos foram considerados, arquivos diversos (documentação, *makefiles*¹) não foram considerados. O repositório f_3 foi composto por 80 arquivos texto no formato CSV² contendo os valores de consumo de recursos (discutidos no Capítulo 4) aferidos para cada programa com todas as aquisições obtidas nos experimentos. Estes arquivos representam os dados exibidos nos gráficos das figuras 4 e 5, por exemplo. O repositório f_4 segue o mesmo padrão do f_3 , porém os arquivos CSV contém apenas as médias dos valores das aferições. Estes dados, por sua vez, estão graficamente representados nos gráficos das figuras 11 e 12, por exemplo. Por fim, o repositório f_5 também segue o mesmo padrão dos repositórios f_4 e f_5 , porém contém apenas os dados de aferição do consumo de CPU.

Tabela 6 – Robustez do agrupamento para vários aspectos dos programas.

Aspecto	ρ (média)	(DP)
f_1) Programas binários	0,85434	0,10487
f_2) Código fonte	0,78419	0,13402
f_3) Dados de consumo (todas aquisições)	0,80845	0,08375
f_4) Médias dos dados de consumo	0,75115	0,09335
f_5) Dados de consumo de CPU extraídos de f_4	0,84123	0,08670

Para avaliar a consistência do agrupamento, o método estatístico *Jackknife* (EFRON; STEIN, 1981) foi aplicado. Este método de reamostragem baseia-se na remoção de uma amostra (ou mais) do conjunto observado, recalculando-se a variável desejada a partir do conjunto restante. A partir das sucessivas reamostragens, é possível calcular diversos parâmetros da variável nos conjuntos observados, tais como média e desvio padrão.

Para cada repositório da Tabela 6 foram gerados 30 repositórios reamostrados aleatoriamente contendo 79 arquivos cada (devido a remoção aleatória de um elemento do repositório). O DAMICORE foi aplicado em cada um dos repositórios gerados, e a congruência par-a-par foi calculada entre todos repositórios gerados. A Tabela 6 contém a média e o desvio padrão do índice ρ obtido para cada f_i .

Todas as congruências calculadas permaneceram acima de 70%, o que indica a robustez da consistência do agrupamento obtido pelo DAMICORE, ou seja, os agrupamentos obtidos nos repositórios reamostrados foram notadamente semelhantes e consistentes entre si. É válido ressaltar que a maior congruência foi obtida de f_1 , que representa os programas binários, acima de f_2 , que representa o código fonte. Esse é um resultado favorável para a análise binária face a análise de códigos fonte. Os repositórios com dados de aferição (f_2 , f_4 and f_5) também

¹ Arquivo com de dependências de código para compilação automática pelo utilitário *make*.

² *Comma-separated value*, do inglês, valores separados por vírgula.

apresentaram alta congruência, lembrando que a análise dos valores numéricos de ρ deve levar em consideração a discussão feita na seção 5.1.1.

6.1.2 Correlação de similaridades

Para investigar a correlação entre os diferentes aspectos abordados (código fonte, binário, dados de consumo) também foi calculado a congruência entre os repositórios reamostrados. A Tabela 7 apresenta as médias de ρ obtidas³. Cada elemento $f_{i,j}$ na tabela representa a congruência média (calculada para os 30 repositórios reamostrados) entre os grupos obtidos para f_i e f_j independentemente. Por exemplo, o elemento $f_{1,2}$ é a congruência obtida entre o repositório de programas binários e o repositório com seus respectivos códigos fontes, ou seja, a congruência quantifica o quanto os grupos gerados para o repositório f_1 também são gerados para o repositório f_2 e vice-versa.

Um resultado notável foi obtido para a congruência entre códigos binários e códigos fonte ($f_{1,2}$), aproximadamente $h = 39\%$, que considerando a escala de ρ (Figura 18), pode ser interpretada como uma congruência significativa. Tal congruência significa que os programas que forem agrupados juntos quando comparados em sua forma binária, tenderão a serem agrupados juntos também quando analisados em formato de código fonte. Nestas circunstâncias, comparar o código binário de programas para detecção de semelhanças não será desvantajoso em relação a utilização dos códigos fonte ou vice versa. Esta informação é de grande valia visto que a mineração de dados também pode ser aplicada ao código fonte (YING *et al.*, 2004; HUCK; MALONY, 2005; MOEYERSOMS *et al.*, 2014) quando o mesmo estiver disponível. Entretanto, a utilização do código binário pode se mostrar menos complexa e passível de erros em cenários onde a implementação dos programas consiste em vários arquivos de código fonte, dificultando a análise neste nível. Por exemplo, os programas cat, cp e dd (contidos no repositório deste trabalho), foram obtidos do pacote de utilitários GNU Coreutils (GNU Coreutils, 2014), que provê diversos programas utilizados em ambientes UNIX. O projeto do código do software é estruturado para que diversos os programas providos utilizem funcionalidades em comum implementadas em diferentes partes do código fonte, de tal forma que o comportamento final de cada programa será dependente de funções e definições implementadas em diferentes arquivos do projeto. Determinar quais partes do código estão “linkadas” com um programa em específico pode não ser trivial, especialmente se os programas fizerem uso extenso de bibliotecas externas, fazendo com que o comportamento esteja mais relacionado ao código das bibliotecas do que ao código do programa em si. Outras dificuldades podem ser encontradas para rastrear as dependências de cada programa dado que muitas vezes sistemas complexos de compilação são utilizados para gerar o binário final, tais como as ferramentas GNU Autoconf, Automake e Libtool (CALCOTE, 2010). Por outro lado, ao compilar um programa de forma estática, o binário

³ Os desvios padrão das médias foram, em média, da ordem de 8% e portanto não foram exibidos na tabela para facilitar a visualização.

Tabela 7 – Correlação entre os diferentes aspectos avaliados (valores das médias de ρ).

	f_1	f_2	f_3	f_4	f_5
f_1	1,00000	0,39234	0,13333	0,14146	0,16290
f_2		1,00000	0,13333	0,15311	0,15385
f_3			1,00000	0,13333	0,28333
f_4				1,00000	0,19005
f_5					1,00000

final irá conter todo o código relevante para o programa (a maioria dos compiladores modernos são capazes até de remover funções não utilizadas pelo programa, por exemplo, códigos antigos não utilizados em softwares muito grandes), mitigando estes problemas relacionado ao código fonte. Outras otimizações também podem ser feitas pelo compilador no código binário e não serem facilmente deduzidas do código fonte.

Outros resultados interessantes da Tabela 7 são as congruências entre o código binário e os arquivos de dados de consumo, denotadas por $f_{1,3}$, $f_{1,4}$ e $f_{1,5}$, todas acima de 12%. Novamente, considerando a escala de ρ , há uma certa correspondência com as similaridades detectas nos arquivos binários. As congruências em relação aos códigos fonte ($f_{2,3}$, $f_{2,4}$ and $f_{2,5}$) também seguem a mesma ordem de magnitude.

6.1.3 Estratégia alternativa de agrupamento

A congruência entre os agrupamentos de arquivos binários e de dados de consumo é uma estimativa indireta de como as similaridades no código executável podem indicar potenciais similaridades nos padrões de consumo de recursos. Essa informação pode ser útil na caracterização dos programas, mesmo que o agrupamento de dados de aferição pode não possuir aplicação prática, é uma observação relevante a ser feita. Em todos os casos deve-se ressaltar que o índice de congruência compara os grupos detectados pelo DAMICORE, sem nenhuma referência de particionamento executado a partir de um conhecimento prévio a respeito dos dados analisados.

O valor numérico de ρ é numericamente afetado pelo mecanismo guloso de detecção de comunidades do DAMICORE, que muitas vezes pode dividir grupos maiores em grupos menores. A aplicação dos algoritmos NJ e FN, respectivamente, determina uma característica da técnica que deverá produzir sempre um certo número de grupos. Mesmo para um repositório composto por arquivos idênticos (e idealmente todos lotados em um único grupo) haverá a produção de mais de um grupo dependendo do tamanho do repositório. Quanto maior o número de grupos detectados, ou de modo geral, quanto maior for a especificidade dos grupos, mais o índice ρ será afetado por eventuais incongruências.

Considere, por exemplo, os dados apresentados na Tabela 7. Nas sucessivas reamostragens ou nos diferentes tipos de dados de cada repositório, o DAMICORE separou os 80 arquivos (referentes a cada aspecto avaliado) em um número de grupos que variou de 12 à 15, aproximada-

mente. Para contrastar este resultado com outras possibilidades de agrupamentos, uma estratégia alternativa de agrupamento foi promovida através da inspeção visual das árvores filogenéticas obtidas nas análises dos repositórios f_1 , f_2 e f_3 . Os agrupamentos encontrados pelo DAMICORE foram desconsiderados e novos grupos foram traçados manualmente considerando-se a formação de cada árvore. As figuras 19, 20 e 21 contêm as árvores filogenéticas e os grupos considerados para os repositórios f_1 (códigos binários), f_2 (códigos fonte) e f_3 (dados de consumo de CPU), respectivamente. Os gráficos próximos a cada grupo representam a média de consumo de CPU dos elementos do grupo (com intervalo de confiança de 95%) e o complemento da taxa, representando um valor de “ociosidade” referente aos programas, ou seja, o tempo em que o processador não executou instruções dos programas do grupo. Este parâmetro foi adicionado para facilitar a comparação visual dos gráficos.

Figura 19 – Árvore com agrupamento alternativo para repositório de programas binários (f'_1)

A Tabela 8 contém as congruências calculadas para os repositórios com os agrupamentos alternativos, sendo denominados f'_1 , f'_2 e f'_3 referente aos repositórios f_1 , f_2 e f_3 , respectivamente. Os valores obtidos são consideravelmente maiores quando comparados com os resultados dos agrupamentos originais (Tabela 7). Por exemplo, a congruência entre os códigos binários e os dados de uso de CPU $f'_{1,5}$ foi superior a 0,25, o que é um valor significativo tendo em mente a

Figura 20 – Árvore com agrupamento alternativo para repositório de códigos fonte (f'_2)

Figura 21 – Árvore com agrupamento alternativo para repositório de dados de consumo de CPU (f'_3)

escala de ρ e consideravelmente acima do valor $f_{1,5} = 0,16$ que foi obtido com o agrupamento original.

Tabela 8 – Congruência dos repositórios com agrupamento alternativo.

	f'_1	f'_2	f'_5
f'_1	1,000000	0,336991	0,255486
f'_2		1,000000	0,204545
f'_5			1,000000

AVALIAÇÃO QUALITATIVA

O Capítulo 6 analisou os agrupamentos providos pelo DAMICORE sob diversos aspectos através de uma análise quantitativa, utilizando o índice par-a-par proposto no Capítulo 5 para quantificar a congruência dos resultados obtidos. Entretanto, as árvores filogenéticas fornecem uma representação gráfica das relações hierárquicas dos dados analisados que não pode ser diretamente extraída somente das informações de agrupamento. Repositórios com volumes de dados extremamente grandes podem gerar árvores extremamente densas e prejudicar (ou anular) a possibilidade de inspeção visual da árvore referente a todo o repositório. Uma alternativa para mitigar este problema é analisar partes do repositório de forma separada (amostrada) ou verificar somente as informações de agrupamento para tentar obter as informações desejadas. Repositórios com volume menor de dados permitem a fácil inspeção gráfica das árvores filogenéticas geradas uma vez que estas serão menores. Este foi um dos fatores considerados durante a organização do repositório de testes deste trabalho, composto por 80 programas.

Este capítulo analisa as árvores filogenéticas providas pelo DAMICORE referentes ao repositório de 80 programas binários, assim como a estabilidade da técnica para códigos compilados com diferentes compiladores e *flags* de otimização.

7.1 Filogenia do código binário

As figuras 22 e 23 contém as árvores filogenéticas referente a todo o repositório dos programas binários compilados com o compilador *gcc* 5.4.0, *flag -O0* (sem otimização). Na Figura 22 os ramos foram coloridos manualmente de acordo com a classificação dos programas: *CPU-Intensive* foram marcados em vermelho e *I/O-Intensive* em azul. Já na Figura 23 os nós folhas foram coloridos manualmente de acordo com o agrupamento detectado (algoritmo FN). Cada cor representa um grupo detectado.

Comparando a árvore da Figura 24 com a árvore dos arquivos originais (Figura 22) é possível observar que não houve mudanças substanciais nos agrupamentos. Observa-se apenas que as distâncias dos principais ramos foram encurtadas, aproximando programas que estavam próximos, como é o caso dos programas *randrw* e *make-many-files*, que juntaram-se no mesmo ramos com os programas de algoritmos de ordenação, e ao programas *bzip2* e *mandelbulber* que aproximaram-se da ramificação da parte inferior da árvore. Já as distâncias entre as duas principais ramificações da árvore foi destacada (aumentada). Este fato pode ser devido a remoção de informações comum a todos os programas que estão presentes nas versões originais, como informações da biblioteca C, etc. Uma vez que esta informação comum é removida, a diferença entre o código restante torna-se acentuada, e acabou sendo destacada na árvore filogenética. Outros experimentos também foram executados com amostras do repositório, obtendo resultados semelhantes. Dado que os resultados não mostraram mudanças significativas no uso apenas do segmento de texto e considerando o objetivo deste trabalho, que é avaliar a aplicação como um todo, os arquivos completos continuaram a ser utilizados nos experimentos com arquivos binários.

A Figura 25 contém a árvore filogenética referente a todo repositório dos programas binários, porém com os gráficos contendo as médias de consumo total de CPU e E/S de cada programa. Os valores referentes a E/S estão normalizados para os maiores valores aferidos, sendo representados em uma escala de 0 a 100%, sendo 100% o maior valor de consumo obtido. Conforme já discutido no início da seção, a parte superior da árvore filogenética é dominada pelos programas com perfil *CPU-Intensive*, ao passo que os programas *I/O-Intensive* estão presentes em sua maioria na parte inferior da árvore. Uma vez que os consumos de CPU e E/S não são mutualmente exclusivos, observa-se entre os programas *I/O-Intensive*, alguns com significativo consumo de CPU, como é o caso dos programas p_{63} , p_{69} e p_{70} , já citados anteriormente. Já os programas presentes mais ao meio da árvore, encontram-se mais espaçados. Próximos aos extremos da árvore, há notadamente subárvores com alta densidade de nós folha. A partir dos fundamentos de inferência filogenética, essas ramificações, ou seja, quando muitos nós folhas compartilham um ancestral comum, indicam um nível maior de similaridade dos objetos conectados na subárvore (internos) em relação aos objetos externos à mesma. Por exemplo, a primeira subárvore na parte superior contém os programas p_1 , p_2 , p_4 , p_6 , p_{11} , p_{13} , p_{16} , p_{21} , p_{30} , p_{28} , p_{38} e p_{30} , que são notadamente *CPU-Intensive*, com nenhum ou muito pouco uso de E/S. Ainda na parte superior há outra subárvore com os programas p_3 , p_{12} , p_{17} , p_{23} , p_{26} , p_{34} e p_{36} , composta por programas com as mesmas características. Entre estes dois grupos estão os programas p_{63} , p_{69} e p_{70} , que fazem uso de ambos: CPU e E/S.

Na parte inferior da árvore há uma subárvore com os elementos p_{22} , p_{42} , p_{45} , p_{46} , p_{64} , p_{47} , p_{44} , p_{50} , p_{54} , p_{58} , p_{65} , p_{68} , p_{71} , p_{73} , p_{75} , p_{76} e p_{77} , que compreende programas com significativo consumo de E/S e também um certo consumo de CPU, com exceção do programa p_{22} (*gzip*), que apresentou maior consumo de CPU (já discutido anteriormente).

Na parte esquerda da árvore estão os programas distribuídos de forma esparsa. Entretanto, ainda é possível observar um padrão de vizinhança, com programas com perfil mais próximos entre si.

Figura 25 – Árvore filogenética com informações de consumo de CPU e E/S.

7.2 Estabilidade do DAMICORE para diferentes compiladores e opções de otimização

Para avaliar o comportamento dos resultados do DAMICORE em versões binárias dos programas compilados com diferentes compiladores e níveis de otimização, 14 repositórios foram criados e analisados com o DAMICORE. Três compiladores amplamente utilizados na comunidade e na indústria foram considerados: *gcc*² versão 5.4.0, *icc*³ versão 15.0.6 e *llvm*⁴ versão 3.8.1.

Cada repositório criado continha os 80 programas do repositório original, compilados com as principais opções (*flags*) de otimização: *-O0*, *-O1*, *-O2*, *-O3* e *-Os*. Estas opções existem

² Disponível em <<https://gcc.gnu.org/>>

³ Disponível em <<https://software.intel.com/en-us/c-compilers>>

⁴ Disponível em <<http://llvm.org>>

nos três compiladores e são, no geral, equivalentes entre si, possuindo como principais objetivos:

- **-O0**: Não faz nenhum tipo de otimização.
- **-O1**: Tenta reduzir o tamanho do código binário e o tempo de execução das instruções sem aumentar significativamente o tempo de compilação e do programa.
- **-O2**: Tenta fazer uma otimização mais agressiva que o nível *-O1* porém com aumento do tempo de compilação.
- **-O3**: Tenta fazer uma otimização ainda mais agressiva que os níveis anteriores, técnicas como vetorização de *loops* e uso de funções *inline* são utilizadas, podendo aumentar o tamanho do código binário do programa.
- **-Os**: Tenta otimizar o tamanho total do código binário do programa, ou seja, segmentos de código e de dados.

O DAMICORE foi aplicado em todos os repositórios criados, obtendo-se as respectivas árvores filogenéticas e os agrupamentos para cada repositório. A congruência par-a-par entre todos os compiladores e *flags* foi calculada. As tabelas 9, 10 e 11 apresentam os resultados de forma dividida: A Tabela 9 contém a congruência de cada compilador com seus diferentes níveis de otimização; A Tabela 10 contém as congruências entre o compilador *gcc* e os compiladores *icc* e *llvm*; e a Tabela 11 contém as congruências entre os compiladores *icc* e *llvm*.

Tabela 9 – Congruência entre o repositório original e versões com os programas compilados com diferentes níveis de otimização (utilizando os compiladores *gcc*, *llvm* e *icc*).

	<i>-O1</i>	<i>-O2</i>	<i>-O3</i>	<i>-Os</i>
<i>gcc</i>	0,602740	0,602740	0,433790	0,523810
<i>llvm</i>	0,518987	0,434599	0,462810	0,430380
<i>icc</i>	0,444444	0,368421	0,355556	0,395556

Tabela 10 – Congruência entre o compilador *gcc* e os compiladores *icc* e *llvm*.

	<i>gcc-O0</i>	<i>gcc-O1</i>	<i>gcc-O2</i>	<i>gcc-O3</i>	<i>gcc-Os</i>
<i>icc-O0</i>	0,50	0,40	0,44	0,38	0,45
<i>icc-O1</i>	0,31	0,33	0,31	0,26	0,30
<i>icc-O2</i>	0,37	0,30	0,34	0,28	0,37
<i>icc-O3</i>	0,38	0,33	0,38	0,32	0,42
<i>icc-Os</i>	0,29	0,33	0,30	0,27	0,29
<i>llvm-O0</i>	0,51	0,41	0,48	0,39	0,46
<i>llvm-O1</i>	0,48	0,39	0,52	0,50	0,48
<i>llvm-O2</i>	0,60	0,58	0,60	0,60	0,59
<i>llvm-O3</i>	0,58	0,50	0,63	0,59	0,56
<i>llvm-Os</i>	0,49	0,43	0,58	0,55	0,51

Tabela 11 – Congruência entre os compiladores *icc* e *llvm*.

	<i>icc-O0</i>	<i>icc-O1</i>	<i>icc-O2</i>	<i>icc-O3</i>	<i>icc-Os</i>
<i>llvm-O0</i>	0,50	0,31	0,37	0,38	0,29
<i>llvm-O1</i>	0,40	0,33	0,30	0,33	0,33
<i>llvm-O2</i>	0,44	0,31	0,34	0,38	0,30
<i>llvm-O3</i>	0,38	0,26	0,28	0,32	0,27
<i>llvm-Os</i>	0,45	0,30	0,37	0,42	0,29

Levando em conta a escala do índice de congruência par-a-par, todas as congruências obtidas são expressivas, seja entre um único compilador e suas diferentes *flags* de otimização (Tabela 9) quanto para diferentes compiladores e níveis de otimização (tabelas 10 e 11). Estes resultados mostram que os agrupamentos promovidos pelo DAMICORE permaneceram consistentes para os diferentes compiladores e *flags* de otimização avaliados, o que era esperado, pois de fato os códigos tendem a ser semelhantes uma vez que devem executar as mesmas instruções que foram programadas na linguagem léxica, a diferença pode ocorrer na forma com a qual cada compilador produz o código binário do programa final, considerando seus níveis de otimização.

TÉCNICA BCC

A partir dos resultados obtidos e discutidos neste trabalho, este capítulo apresenta a proposta de uma nova técnica para avaliação e agrupamento de programas binários, denominada **BCC**.

A técnica **BCC** - *Binary Code Clustering* - é uma metodologia para execução de experimentos de agrupamentos/classificação de códigos binários que sistematiza os procedimentos a serem seguidos para obtenção de resultados consistentes de forma mais prática, facilitando a reprodutibilidade da análise.

A Figura 26 apresenta o fluxograma de toda aplicação da técnica BCC. O fluxograma é composto por condições (enumeradas em Qn , $1 \leq n \leq 7$) a serem avaliadas, e ações (enumeradas em An , $1 \leq n \leq 8$) a serem tomadas levando até a geração dos resultados finais (quadro *FIM*). A técnica deve ser iniciada a partir de um conjunto de programas binários que se deseja analisar, que serão denominados programas incógnitas.

A partir dos programas incógnitas, a primeira condição a ser avaliada ($Q1$) é se existe uma base de programas binários pré-existente. Esta base é necessária se o intuito do usuário é agrupar os programas incógnitas de acordo com classes de programas pré-determinadas (condição falsa para $Q2$). Neste caso, o repositório existente deve conter programas das classes a serem consideradas na avaliação. Se não há nenhuma base pré-existente, então deve-se criar um repositório de programas com os perfis das classes desejadas (ação $A1$). Após criada, os programas devem ser classificados por aferição ou outros métodos necessários para averiguar a classificação dada a cada programa do repositório (ação $A2$). Caso o intuito do usuário seja simplesmente analisar como os programas incógnitas relacionam-se entre si (condição verdadeira para $Q2$), então basta aplicar o DAMICORE (ação $A3$) e seguir com a análise posterior dos resultados obtidos.

Se o usuário já possui uma base de programas pré-existente, então deve-se determinar o intuito da análise (condição $Q3$): os resultados desejados podem tratar-se apenas da classificação

dos programas incógnitas dentro das classes definidas pelos programas do repositório existente; ou o objetivo é obter uma avaliação completa, ou seja, as relações de filogenia e agrupamentos encontrados. Caso apenas a classificação dos programas incógnitas seja necessária, então os testes de quartetos devem ser aplicados a cada programa incógnita (ação A4). O teste de quarteto está descrito em detalhes na seção 8.1, trata-se de uma abordagem que busca averiguar se um objeto de dado x pertence a uma classe U tentando-o classificar com a própria classe e seu complemento $\bar{U} = W$. Para garantir a diferença entre U e W um elemento $\{y \mid y \in U\}$, denominado contraprova, é removido da classe U para formar um novo conjunto de dados $U_c = U - \{y\}$. A partir dos conjuntos U_c e W e dos elementos x (incógnita) e y (contraprova), aplica-se o DAMICORE em um repositório composto pelos quatro objetos, resultando em uma árvore filogenética de quarteto. A partir de alguns testes e resultados com árvores de quarteto, elabora-se um sistema classificador para determinar em qual classe encontra-se o objeto analisado.

Se o usuário deseja obter uma análise completa (filogenia mais agrupamentos) dos programas incógnitas e possui uma base de programas pré-existente então o DAMICORE deve ser aplicado (ação A3). Após a aplicação do DAMICORE a primeira condição a ser avaliada ($Q4$) é se as árvores filogenéticas geradas podem ser visualmente inspecionadas, ou seja, não possuem alto grau de complexidade, densidade, tamanho, etc. Caso a inspeção visual seja possível, então as ações A5 e A6 podem ser executadas, consistindo na análise qualitativa da filogenia e, caso seja de interesse, a análise dos agrupamentos encontrados.

Após obter os resultados da classificação ou da análise completa do DAMICORE, o usuário deve verificar se as informações encontradas são suficientes para o propósito inicial da análise (condição $Q6$). Em caso afirmativo, encerra-se a aplicação da técnica e, se desejado, os programas incógnitas podem ser adicionados a base de dados existente, devendo-se evitar a inserção de dados redundantes. Caso os resultados obtidos não sejam suficientes então deve-se fazer uma amostragem aleatória da base de dados existente e reaplicar o DAMICORE, refazendo todas as ações posteriores com os novos resultados. Caso sucessivas amostragens forem feitas sem melhoria nos resultados gerados (condição verdadeira para $Q7$) então as ações A1 e A2 devem ser executadas, ou seja, uma nova base de dados deve ser organizada.

Dependendo do tamanho do repositório de programas utilizado, a aplicação do DAMICORE poderá gerar árvores extremamente densas e complexas, impossibilitando a análise qualitativa por inspeção visual (condição falsa para $Q4$). Neste caso o usuário deverá analisar apenas os dados dos agrupamentos encontrados. Se as informações dos agrupamentos for suficiente para o propósito inicial da análise, encerra-se a aplicação da técnica e, se desejado, os programas incógnitas podem ser adicionados a base de dados existente, devendo-se evitar a inserção de dados redundantes. Caso as informações dos agrupamentos não sejam suficientes para os objetivos do usuário, então o procedimento de amostragem (A8) deve ser efetuado.

A técnica BCC sintetiza a aplicação dos conceitos discutidos nos capítulos anteriores por meio da formulação de um procedimento sistemático que compreende o método proposto. Os

passos para preparação dos dados, processamento e análise dos resultados, sumarizados dentro do fluxograma da Figura 26, orienta os passos do projeto experimental para guiar o usuário durante toda aplicação da técnica afim de produzir resultados consistentes e relevantes.

Figura 26 – Fluxograma para aplicação da técnica BCC.

8.1 Teste de Quartetos

Um dos pontos críticos que afetam o desempenho de execução do DAMICORE é a construção da matriz de distâncias, de ordem $\mathcal{O}(n^2)$. Para pequenos conjuntos de dados o tempo de execução da técnica é factível, mas para conjuntos maiores torna-se impraticável sem recursos computacionais de alto desempenho. Uma abordagem efetuada por (DELBEM, 2012) e

sintetizada em (CESAR, 2016) busca averiguar se um objeto de dado x pertence a uma classe U tentando-o classificar com a própria classe e seu complemento $\bar{U} = W$, se de fato o elemento x possuir características da classe U então estará próximo da mesma na árvore filogenética e distante de W . Para garantir a diferença entre U e W um elemento $\{y \mid y \in U\}$, denominado contraprova, é removido da classe U para formar um novo conjunto de dados $U_c = U - \{y\}$. A partir dos conjuntos U_c e W e dos elementos x (incógnita) e y (contraprova), aplica-se o DAMICORE em um repositório composto pelos quatro objetos, resultando em uma árvore filogenética de quarteto, como ilustra o exemplo da Figura 27.

Figura 27 – Árvore filogenética de quarteto.

A detecção de comunidades torna-se irrelevante neste caso dado a simplicidade da árvore, composta por apenas quatro nós folhas. É importante notar que a Figura 27 ilustra uma classificação consistente, onde as semelhanças do elemento x com a classe U_c foram mais fortes que as semelhanças da própria contra prova y com U_c , resultando em uma distância menor, ou seja, $d_{xU_c} < d_{yU_c}$. A Figura 28 contém as árvores de quartetos com os possíveis resultados da aplicação do DAMICORE.

(a) A_1 : Classificação Positivo Forte.

(b) A_2 : Classificação Negativo Fraco.

(c) A_3 : Classificação Neutro.

(d) A_4 : Classificação Neutro.

Figura 28 – Possíveis resultados de árvores filogenéticas de quartetos.

A Figura 28a, assim como a Figura 27 denota uma classificação **Positivo Forte**, pois as

características do elemento x possuem semelhança em relação a classe U_c maior que a própria contraprova y , pois $d_{xU_c} < d_{yU_c}$, indicando que o elemento pertence a classe U . Já a Figura 28b representa uma classificação **Negativo Fraco**, pois o elemento de contraprova y permaneceu mais próximo da classe U_c em relação a x , ou seja, $d_{xU_c} > d_{yU_c}$. É importante ressaltar que este resultado não indica necessariamente que o elemento x não pertence a classe U , mas suas características não foram capazes de superar a semelhança da contraprova y em relação a U . A Figura 28c ilustra um resultado onde a distância entre o elemento incógnita x e a contraprova y é menor do que a distância de ambos em relação a classe U_c . Neste caso, os dois elementos são mais próximos entre si do que de suas classes, sendo considerado um resultado **Neutro**. Por fim, a Figura 28d ilustra uma árvore equidistante, onde as distâncias entre os nós folhas são equivalentes, não sendo possível obter uma classificação conclusiva, portanto é considerado também um resultado **Neutro**. A escolha do elemento de contraprova pode ser feita de forma aleatória ou considerando a distância do mesmo em relação a classe U , podendo-se escolher elementos com a menor distância ou distância mediana, por exemplo. Entretanto, estes tipos de escolhas não garantem que a contra prova estará distante da classe complementar. Assim, o processo pode ser repetido sucessivas vezes escolhendo-se diferentes elementos como contraprova para obtenção de resultados consistentes.

Para obter a classificação de um determinado objeto de dado, o processo de análise em quartetos deve ser aplicado para todas as classes de interesse, gerando resultados **Positivo Forte**, **Negativo Fraco** ou **Neutro** para cada classe avaliada juntamente com o objeto em análise. A classificação será determinada de acordo com os seguintes resultados:

1. Se existe um único resultado **Positivo Forte** para uma determinada classe C , então o elemento pertence a classe C ;
2. Se existe um único resultado **Neutro** para uma determinada classe C , e **Negativo Fraco** para todas as outras classes do grupo, então o elemento pertence a classe C por exclusão;
3. Se existe mais de um resultado **Neutro** para diferentes classes, então não é possível classificar o elemento em uma única classe, porém, é possível considerar os resultados neutros como possíveis classes as quais o elemento pertence;
4. Se existem múltiplos resultados **Positivo Forte** ou apenas resultados **Negativo Fraco** então não é possível classificar o elemento;

Um exemplo prático do uso dos testes de quartetos é abordado no Capítulo 10.

Parte III

Estudos de Caso

ALGORITMOS COM OTIMIZAÇÕES

Os programas que compõem o repositório utilizado nos experimentos deste estudo foram sistematicamente e cuidadosamente selecionados para possuírem características notadamente distintas no que tange ao consumo de CPU e Entrada/Saída afim de facilitar a validação dos resultados obtidos pela aplicação da BCC. A técnica mostrou-se estável e foi capaz de agrupar os programas de acordo com seus respectivos perfis de consumo. Um primeiro caso de estudo foi desenvolvido com o intuito de averiguar a capacidade da BCC em agrupar programas sob diferentes aspectos, tais como geração do código binário por diferentes compiladores e opções de otimização, perfis de aplicação semelhantes e análise somente das instruções de máquina executadas pelo programa.

9.0.1 *Análise do algoritmo do filtro Sobel*

Sobel(SOBEL, 2014) é um conhecido filtro para detecção de bordas em imagens extensamente implementado em diversos trabalhos nas mais variadas plataformas, desde FPGAs em sistemas embarcados à GPUs em supercomputadores. A primeira análise desse caso de estudo envolveu a construção de um repositório com três programas que implementam o filtro Sobel utilizando o algoritmo original e duas versões modificadas. O objetivo do teste foi averiguar a capacidade de agrupamento do DAMICORE aplicado a códigos binários muito semelhantes, diferindo-se apenas no compilador utilizado para a geração do código de máquina assim como o uso de diferentes níveis de otimização.

O algoritmo Sobel funciona basicamente através de duas matrizes de convolução que são aplicadas para detecção das mudanças dos pixels na horizontal e na vertical. O Código 1 (em linguagem C) contém a função que implementa o algoritmo original, a função *compute()* é utilizada apenas para efetuar as operações matemáticas com cada elemento das matrizes armazenados nas variáveis $r_{i,j}$ ($r00$, $r01$, etc). As variáveis *num_loads_external* e *num_stores_external* são utilizadas para armazenar a quantidade de leitura e gravação de dados na memória, respectivamente.

Código-fonte 1 – Sobel original

```

1: void originalSobel()
2: {
3:     int i, j;
4:
5:     for(i = 1; i < M-1; i++) {
6:         for(j = 1; j < N-1; j++) {
7:             r00 = imgIn[i-1][j-1];
8:             r10 = imgIn[i][j-1];
9:             r20 = imgIn[i+1][j-1];
10:
11:             r01 = imgIn[i-1][j];
12:             r11 = imgIn[i][j];
13:             r21 = imgIn[i+1][j];
14:
15:             r02 = imgIn[i-1][j+1];
16:             r12 = imgIn[i][j+1];
17:             r22 = imgIn[i+1][j+1];
18:
19:             num_loads_external += 9;
20:             imgOut[i][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
                r22);
21:             num_stores_external += 1;
22:         }
23:     }
24: }

```

Duas outras implementações utilizadas nos testes são derivadas do Código 1, ambas fazem o uso de uma técnica denominada *loop peeling* (BACON; GRAHAM; SHARP, 1994), que consiste em mover o código das primeiras (ou ultimas) iterações de um *loop* para fora do laço. Considere o Código 2, note que a atribuição da variável k será diferente de i apenas na primeira iteração do laço, assim, utilizando *loop peeling* uma otimização pode ser obtida através do Código 3. Esta técnica é útil para garantir um alinhamento de memória em vetores, necessária quando outra técnica denominada *loop vectorization* for aplicada. A “vetorização” de um *loop* consiste em transformar as instruções repetidas pelo laço com um vetor em instruções sequenciais, de tal forma que instruções SIMD (*Single Instruction, Multiple Data*) possam ser aplicadas para fazer uma mesma operação em vários elementos do vetor através da execução de uma única instrução de máquina. Os Códigos 4 e 5 contêm as versões modificadas do Código 1. A primeira modificação (Código 4) aplica *loop peeling* para a primeira iteração do laço que trata as colunas das matrizes, retirando do laço as instruções para a primeira coluna. Já o Código 5 faz

o mesmo procedimento, porém aplicado para as iterações da primeira linha e primeira coluna, respectivamente.

Código-fonte 2 – Código sem otimização

```
1: int k = 20;
2:
3: for (i = 0; i < 30; i++) {
4:   b[i] = (a[i] + a[k]) * 0.5;
5:   k    = i;
6: }
```

Código-fonte 3 – Código otimizado com *loop peeling*

```
1: b[0] = (a[0] + a[20]) * 0.5;
2:
3: for (i = 1; i < 30; i++) {
4:   b[i] = (a[i] + a[i-1]) * 0.5;
5: }
```

Código-fonte 4 – Sobel modificado com *loop peeling* para a primeira coluna

```
1: void reuseColsSobel()
2: {
3:   int i, j;
4:
5:   for(i = 1; i < M-1; i++) {
6:     /* --- peeled j = 1 iteration of the j loop */
7:     r00 = imgIn[i-1][0];
8:     r10 = imgIn[i][0];
9:     r20 = imgIn[i+1][0];
10:
11:     r01 = imgIn[i-1][1];
12:     r11 = imgIn[i][1];
13:     r21 = imgIn[i+1][1];
14:
15:     r02 = imgIn[i-1][2];
16:     r12 = imgIn[i][2];
17:     r22 = imgIn[i+1][2];
18:
19:     num_loads_external += 9;
```

```
20:     imgOut[i][1] = compute(r00,r01,r02,r10,r11,r12,r20,r21,r22)
    ;
21:     num_stores_external += 1;
22:
23:     line0[0] = r00; line0[1] = r01; line0[2] = r02;
24:     line1[0] = r10; line1[1] = r11; line1[2] = r12;
25:     line2[0] = r20; line2[1] = r21; line2[2] = r22;
26:
27:     idx0 = 0;
28:     idx1 = 0;
29:     idx2 = 0;
30:
31:     for (j = 2; j < N-1; j++) {
32:         r00 = line0[idx0+1];
33:         r10 = line1[idx1+1];
34:         r20 = line2[idx2+1];
35:
36:         r01 = line0[idx0+2];
37:         r11 = line1[idx1+2];
38:         r21 = line2[idx2+2];
39:
40:         r02 = imgIn[i-1][j+1];
41:         r12 = imgIn[i][j+1];
42:         r22 = imgIn[i+1][j+1];
43:
44:         num_loads_external += 3;
45:         imgOut[i][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
    r22);
46:         num_stores_external += 1;
47:
48:         line0[idx0+3] = r02;
49:         line1[idx1+3] = r12;
50:         line2[idx2+3] = r22;
51:
52:         idx0++;
53:         idx1++;
54:         idx2++;
55:     }
56: }
57: }
```

Código-fonte 5 – Sobel modificado com *loop peeling* para a primeira linha e coluna

```

1: void reuseRowsColsSobel()
2: {
3:     int i, j;
4:     int *temp;
5:
6:     idx0 = 0;
7:     idx1 = 0;
8:     idx2 = 0;
9:
10:    /* -- peeled first iteration of i = 1 */
11:    /* -- peeled first iteration of j = 1 */
12:    {
13:        r00 = imgIn[0][0];
14:        r10 = imgIn[1][0];
15:        r20 = imgIn[2][0];
16:
17:        r01 = imgIn[0][1];
18:        r11 = imgIn[1][1];
19:        r21 = imgIn[2][1];
20:
21:        r02 = imgIn[0][2];
22:        r12 = imgIn[1][2];
23:        r22 = imgIn[2][2];
24:
25:        num_loads_external += 9;
26:
27:        imgOut[1][1] = compute(r00,r01,r02,r10,r11,r12,r20,r21,r22)
28:        ;
29:
30:        line0[0] = r00; line0[1] = r01; line0[2] = r02;
31:        line1[0] = r10; line1[1] = r11; line1[2] = r12;
32:        line2[0] = r20; line2[1] = r21; line2[2] = r22;
33:
34:        num_stores_external += 1;
35:
36:        idx0++;
37:        idx1++;
38:        idx2++;
39:    }

```

```
40:   for (j = 2; j < N-1; j++) {
41:       r00 = line0[idx0];
42:       r10 = line1[idx1];
43:       r20 = line2[idx2];
44:
45:       r01 = line0[idx0+1];
46:       r11 = line1[idx1+1];
47:       r21 = line2[idx2+1];
48:
49:       r02 = imgIn[0][j+1];
50:       r12 = imgIn[1][j+1];
51:       r22 = imgIn[2][j+1];
52:
53:       num_loads_external += 3;
54:
55:       imgOut[1][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,r22)
56:       ;
57:       line0[idx0+2] = r02; line1[idx1+2] = r12; line2[idx2+2] =
58:       r22;
59:       num_stores_external += 1;
60:
61:       idx0++;
62:       idx1++;
63:       idx2++;
64:   }
65:
66:   for (i = 2; i < M-1; i++) {
67:       /* --- reset pointers and rotate lines */
68:       idx0 = 0;
69:       idx1 = 0;
70:       idx2 = 0;
71:
72:       temp  = line0;
73:       line0 = line1;
74:       line1 = line2;
75:       line2 = temp;
76:
77:       /* --- peeled the first iteration j = 1 */
78:       {
79:           r00 = line0[0];    r01 = line0[1];    r02 = line0[2];
```

```
80:         r10 = line1[0];    r11 = line1[1];    r12 = line1[2];
81:
82:         r20 = imgIn[i+1][0];    r21 = imgIn[i+1][1];    r22 = imgIn
[i+1][2];
83:
84:
85:         num_loads_external += 3;
86:
87:         imgOut[i][1] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
r22);
88:
89:         line2[0] = r20;    line2[1] = r21;    line2[2] = r22;
90:
91:         num_stores_external += 1;
92:
93:         idx0++;
94:         idx1++;
95:         idx2++;
96:     }
97:     for (j = 2; j < N-1; j++) {
98:         r00 = line0[idx0];
99:         r10 = line1[idx1];
100:        r20 = line2[idx2];
101:
102:        r01 = line0[idx0+1];
103:        r11 = line1[idx1+1];
104:        r21 = line2[idx2+1];
105:
106:        r02 = line0[idx0+2];
107:        r12 = line1[idx1+2];
108:        r22 = imgIn[i+1][j+1];
109:
110:        num_loads_external += 1;
111:
112:        imgOut[i][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
r22);
113:
114:        line2[idx2+2] = r22;
115:
116:        num_stores_external += 1;
117:
118:        idx0++;
```

```
119:         idx1++;  
120:         idx2++;  
121:     }  
122: }  
123: }
```

Para restringir o código binário produzido aos principais aspectos do algoritmo implementado pelo mesmo e evitar a geração de informações adicionais que pudessem influenciar na aplicação do DAMICORE, as três variações do Sobel apresentadas foram compiladas apenas no formato objeto, não contendo funções de bibliotecas externas (compilação dinâmica), o código da função *compute()* e nem valores carregados nas matrizes utilizadas, apenas o código mais essencial foi utilizado, como declaração das variáveis, inclusão de arquivos de cabeçalho padrão da biblioteca C e a função que implementa o algoritmo Sobel (mostradas nos Códigos 1, 4 e 5). Os compiladores *gcc*, *icc* e *llvm* foram utilizados em suas versões 5.4.0, 15.0.6 e 3.8.1, respectivamente.

O primeiro teste executado consistiu na organização de repositórios que continham as três versões do Sobel compilados com os três compiladores para uma determinada opção (*flag*) de otimização. As *flags* utilizadas em cada repositório foram *-O0* (sem otimização), *-O1*, *-O2*, *-O3* e *-Os* (otimização para redução de tamanho do binário final). As Figuras 29, 30, 31, 32 e 33 contêm as árvores filogenéticas produzidas pelo DAMICORE. Os nós folhas contêm o nome de cada programa assim como uma numeração que indica o agrupamento detectado na última fase do DAMICORE, porém, uma vez que as árvores são pequenas, esta informação não foi utilizada. Os nomes são compostos pelo compilador utilizado, sendo *gc*, *ic* ou *cl* (para *gcc*, *icc* e *llvm*, respectivamente), seguido pela versão que o programa implementa, sendo *sobel1*, *sobel2* ou *sobel3*, para as versões dos Códigos 1, 4 e 5, respectivamente.

Em todas as árvores os binários ficaram separados em ramos distintos para cada compilador utilizado, produzindo árvores com três ramos principais contendo as três versões dos programas para cada compilador. Na árvore da Figura 29, que contém as versões compiladas sem nenhum nível de otimização aplicado pelo compilador, a semelhança encontrada é a mesma para todos os três compiladores: As versões *sobel1* e *sobel2* ficaram mais próximas no ramo em relação a versão *sobel3*. Nas árvores das Figuras 30, 31, 32 e 33, referentes a todas as outras *flags* de otimização, os resultados foram os mesmos porém com uma diferença em relação a *flag -O0*: as versões *sobel2* e *sobel3* foram as que ficaram mais próximas nos ramos em relação a versão *sobel1*. A coloração dos ramos foi feita de forma manual e utilizada somente para distinguir o compilador utilizado.

Figura 29 – Códigos objeto compilados sem nível de otimização (-O0).

Figura 30 – Códigos objeto compilados com nível de otimização -O1.

Figura 31 – Códigos objeto compilados com nível de otimização -O2.

Figura 32 – Códigos objeto compilados com nível de otimização -O3.

Figura 33 – Códigos objeto compilados com nível de otimização *-Os*.

Mesmo utilizando o código objeto, o arquivo binário pode conter informações adicionais, como nome do compilador utilizado, tabelas de símbolos (nomes de funções, etc) que podem influenciar no agrupamento detectado pelo DAMICORE. Para avaliar a capacidade de agrupamento apenas das instruções binárias dos programas, um segundo teste foi executado com o mesmo protocolo do primeiro teste, com a exceção de que foi utilizado arquivos binários contendo apenas o segmento de texto extraído dos arquivos objeto originais. O segmento de texto contém apenas o código binário que é executado pelo processador, excluindo qualquer outro tipo de informação, inclusive valores de variáveis globais inicializadas, que ficam armazenados em outro segmento (segmento de dados). As Figuras 34, 35, 36, 37 e 38 contém as árvores filogenéticas resultantes. A nomenclatura dos nós folhas é a mesma utilizada no primeiro teste, assim como a coloração dos ramos para distinguir o compilador utilizado. O nome de cada programa também foi colorido manualmente de acordo com a versão implementada pelo mesmo para facilitar a inspeção visual na árvore.

Na árvore da Figura 34, que representa as versões sem nenhuma otimização do compilador, os resultados foram semelhantes ao primeiro teste (Figura 29), onde todos os programas ficaram separados de acordo com o compilador em que foram compilados (mesmo utilizando apenas o segmento de texto). Para o compilador *icc* o resultado também é o mesmo, ficando mais próximas as versões *sobel1* e *sobel2* em relação a *sobel3*. Porém, para os compiladores *gcc* e *clang*, as versões *sobel2* e *sobel3* ficaram mais próximas em relação a versão *sobel1*, resultado que foi obtido com todas as *flags* de otimização utilizadas no primeiro teste.

Figura 34 – Segmentos de texto compilados sem nível de otimização (-O0).

Nas árvores das Figuras 35, 36, 37 e 38, que representam as versões com otimização do compilador, o agrupamento das versões *sobel2* e *sobel3* foi o mesmo obtido no primeiro teste, ficando ambas versões agrupadas e separadas em ramos distintos de acordo com o compilador utilizado. Porém, em todas otimizações utilizadas, os binários referentes a versão *sobel1* ficaram mais próximos nas árvores, não sendo distintos em ramos separados de acordo com o compilador.

Os resultados obtidos mostram que o DAMICORE foi capaz de distinguir até mesmo códigos binários bem semelhantes, ou seja, que implementam o mesmo algoritmo porém de formas pouco diferentes. A utilização dos três compiladores: *gcc*, *icc* e *clang*, que são amplamente utilizados pela indústria e comunidade de desenvolvedores não representou fator de impacto nos resultados, uma vez que os agrupamentos obtidos foram consistentes entre os compiladores. A utilização do segmento de texto, que contém apenas o código binário das instruções de máquina executadas pelo programa mostrou que de fato essa informação está sendo determinística para os resultados do DAMICORE, porém informações adicionais (segmento de dados, segmento de comentários com informações de compilação, etc) também podem colaborar para o agrupamento, fato que pode ser notado nas versões *sobel1*, que quando analisadas apenas em seus segmentos de texto, a semelhança dos códigos binários foi mais forte em relação a eventuais semelhanças geradas pelo compilador, que levassem também a um agrupamento distinto de acordo com o compilador utilizado.

Figura 35 – Segmentos de texto compilados com nível de otimização *-O1*.

Figura 36 – Segmentos de texto compilados com nível de otimização *-O2*.

Figura 37 – Segmentos de texto compilados com nível de otimização *-O3*.

Figura 38 – Segmentos de texto compilados com nível de otimização *-Os*.

9.0.2 Análise com algoritmos distintos

Os experimentos descritos na seção 9.0.1 trataram apenas programas que implementavam o mesmo algoritmo (Sobel) porém de formas pouco diferentes. Um segundo experimento foi executado no mesmo formato dos testes com o algoritmo Sobel, porém foi organizado um novo repositório adicionando-se programas que executam diferentes algoritmos. A Tabela 12 lista todos os programas utilizados. Cinco programas implementam algoritmos de ordenação, dois programas implementam Modulação/Demodulação de sinal ADPCM (*Adaptive Differential Pulse-Code Modulation*), um programa que implementa o método numérico de integração Simpson $\frac{1}{3}$, um programa que implementa o algoritmo de sombreamento de imagens Gouraud, e os três programas Sobel já utilizados anteriormente.

Tabela 12 – Programas de diferentes algoritmos que compõem o repositório.

Programa	Algoritmo implementado
adpcm_code	Modulação de sinal ADPCM
adpcm_deco	Demodulação de sinal ADPCM
sobel1— sobel2— sobel3—	Algoritmos Sobel
simpson13-	Método numérico Simpson $\frac{1}{3}$
goraud—	Algoritmo de sombreamento
bubble_sor	Método de ordenação <i>Bubble sort</i>
heap—	Método de ordenação <i>Heap sort</i>
radixsort-	Método de ordenação <i>Radix sort</i>
selection-	Método de ordenação <i>Selection sort</i>
insertion-	Método de ordenação <i>Insertion sort</i>

Todos os programas foram compilados em código objeto com o compilador *gcc* nas versões sem otimização (*flag -O0*) e com otimização (*flags -O1, -O2, -O3 e -Os*).

As Figuras 40, 41, 42, 43 e 44 contêm as árvores obtidas. As árvores correspondentes as *flags -O0, -O1, -O2 e -Os* possuem grande semelhança: os programas Sobel ficaram bem separados em ramos com um ancestral comum, assim como os programas do algoritmo ADPCM e de ordenação, com exceção para o programa que implementa o método *bubble sort*, que acabou ficando mais próximo do programa que implementa o algoritmo Gouraud. Este fato pode ser explicado pela análise dos códigos fonte. O método *bubble sort* possui apenas dois laços de repetição (*for*) aninhados, percorrendo um vetor de 0 à N (N sendo o tamanho do vetor) promovendo a ordenação de cada elemento. A idéia geral do algoritmo de sombreamento Gouraud considera que se a direção da incidência do raio de luz é perpendicular a superfície de um objeto, então a superfície deve ficar brilhante. Se a direção da luz é paralela à superfície, então a mesma deve ficar escura. A Figura 39 ilustra essa consideração. O cálculo do algoritmo deve ser feito para cada pixel da imagem afim de determinar o brilho do mesmo a partir da informação de incidência do raio de luz, que deverá ser calculado a partir da fonte luminosa no

modelo 3D. O código do algoritmo *Gouraud* utilizado nos experimentos implementa apenas o cerne do algoritmo, utilizando apenas uma laço de repetição *for* para percorrer um vetor de 0 à N (N sendo o tamanho do vetor) aplicando o cálculo do valor final do pixel a partir das informações de cores (r, g, b) (vermelho, verde e azul). Os Códigos 6 e 7 listam as implementações utilizadas de ambos algoritmos: Gouraud e *bubble sort*, respectivamente.

Figura 39 – Incidência dos raios de luz considerada pelo algoritmo Gouraud. Adaptado de (*Glum Project, 2016*).

Código-fonte 6 – Código do algoritmo Gouraud

```

1: #define TP
2: #define N 200
3:
4: int p[N];
5:
6: unsigned int n;
7: unsigned int rd;
8: unsigned int r;
9: unsigned int gd;
10: unsigned int g;
11: unsigned int bd;
12: int b;
13:
14: main() {
15:     unsigned int mask = 0xF800F800, i;
16:
17:     for (i = 0; i < N; i++) {
18:         r += rd;
19:         g += gd;
20:         b += bd;
21:         p[i] = (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
22:     }
23: }
```

Código-fonte 7 – Código do algoritmo *bubble sort*

```
1: #define N 32
2:
3: int ar[N];
4:
5: main() {
6:     int i,j;
7:     int temp;
8:     for (i=0;i<N;i++) {
9:         for (j=0;j<N-1;j++) {
10:            if (ar[j] > ar[j+1]) {
11:                temp = ar[j+1];
12:                ar[j+1] = ar[j];
13:                ar[j] = temp;
14:            }
15:        }
16:    }
17: }
```

Os códigos utilizados apresentam um laço comum (*for*) e operações com vetores. Já os outros códigos de ordenação utilizados são mais complexos. Todos os códigos dos programas do repositório estão disponíveis no Apêndice A.

A árvore referente a *flag -O3* apresentou um agrupamento pouco diferente em relação aos demais: a distância entre os ramos dos programas do algoritmo ADPCM e do ramo com os programas dos algoritmos Gouraud e *Bubble sort* ficou encurtada, aproximando ambos os ramos, porém mantendo o padrão obtido nas outras árvores. Já as otimizações promovidas pelo compilador no código do programa do algoritmo *Radix sort* fez com que ele ficasse mais próximo do programa Simpson $\frac{1}{3}$ do que com os códigos de ordenação. Este último resultado pode ser explicado analisando-se os códigos assembly “desmontados” a partir dos programas objeto compilados¹, todos disponíveis no Apêndice A. No nível de otimização da *flag -O3* tanto para o programa *Radix sort* quanto para o *Simpson* $\frac{1}{3}$ o compilador (*gcc*) acabou optando por utilizar instruções da extensão SSE2², que trabalha com registradores de 128 bits denominados *xmm* (*xmm1*, *xmm2*, *xmm3*, etc), em ambos os códigos há inúmeras instruções com os referidos registradores, o que não aconteceu nos códigos dos outros programas. É importante ressaltar que essa semelhança entre os códigos só acontece na versão binária otimizada (*-O3*), e foi detectada pelo DAMICORE, o que não seria possível se somente os códigos fonte das aplicações fossem analisados.

¹ O processo de *disassembly* (“desmontagem”), foi executado através do utilitário *objdump*, que é componente do pacote de utilitários UNIX *binutils*.

² Sítio SSE2: <<https://www.intel.com/content/www/us/en/support/processors/000005779.html>>

Figura 40 – Códigos objeto compilados sem nível de otimização (-O0).

Figura 41 – Códigos objeto compilados com nível de otimização -O1.

Figura 42 – Códigos objeto compilados com nível de otimização *-O2*.

Figura 43 – Códigos objeto compilados com nível de otimização *-O3*.

Figura 44 – Códigos objeto compilados com nível de otimização -Os.

9.0.3 Análise dos Binários versus Código fonte

O ultimo experimento executado neste caso de estudo avaliou o agrupamento promovido pelo DAMICORE em um repositório contendo os arquivos binários (objeto) compilados com o compilador *gcc*, juntamente com o código fonte de cada programa. As Figuras 45, 46, 47, 48 e 49 contém as árvores filogenéticas resultantes para as *flags* -O0, -O1, -O2, -O3 e -Os, respectivamente. Em todas árvores houve clara distinção entre binários e códigos fonte, que ficaram todos agrupados em ramos bem distintos na árvore. Seguindo o padrão dos experimentos efetuados anteriormente, os ramos foram coloridos manualmente de acordo com o algoritmo implementado pelo programa, os nós folhas foram nomeados de acordo com o número do grupo gerado pelo DAMICORE juntamente com uma letra referente ao tipo de arquivo, sendo **S** para código fonte ou **B** para binário e o nome do arquivo do programa.

Em relação aos arquivos de código fonte, todos os agrupamentos promovidos permaneceram bem semelhantes entre si, sem nenhuma mudança expressiva, a única diferença visual detectada refere-se ao programa do algoritmo Simpson $\frac{1}{3}$ que para a *flag* -O0 permaneceu pouco mais distante do grupo de algoritmos de ordenação, e nas demais *flags* permaneceu pouco mais próximo. Os programas também ficaram agrupados de acordo com a semelhança entre os algoritmos que os mesmos implementam, os códigos dos algoritmos ADPCM ficaram bem próximos em um ramo assim como os programas Sobel e os programas dos algoritmos de ordenação, com exceção dos programas dos algoritmos Goraus e *Bubble sort*, que ficaram mais próximos entre si, resultado devido a semelhança entre os dois códigos, conforme discussão já feita na seção 9.0.2.

Figura 45 – Códigos objeto compilados sem nível de otimização (-O0) e códigos fonte.

Figura 46 – Códigos objeto compilados com nível de otimização *-O1* e códigos fonte.

Figura 47 – Códigos objeto compilados com nível de otimização -O2 e códigos fonte.

Figura 48 – Códigos objeto compilados com nível de otimização -O3 e códigos fonte.

Figura 49 – Códigos objeto compilados com nível de otimização *-Os* e códigos fonte.

Em relação aos arquivos binários, os agrupamentos seguiram o mesmo padrão dos resultados obtidos no teste anterior e discutidos na seção 9.0.2. Os binários que implementavam algoritmos semelhantes foram agrupados juntos no mesmo ramo, salvo as exceções já discutidas: os programas do algoritmo Goraus e *Bubble sort* ficaram mais próximos devido a semelhança de código e para a *flag -O3* o binário *Radix sort* ficou mais próximo ao binário Simpson $\frac{1}{3}$ em relação aos outros programas de ordenação, conforme discutido, ambos foram os únicos códigos em que o compilador utilizou instruções do conjunto da extensão SSE2, fazendo com que ambos os código contivessem inúmeras referências as instruções e registradores da extensão, levando a uma semelhança maior em relação aos códigos compilados com as demais *flags* de otimização.

Através da análise entre binários e códigos fontes pela aplicação da BCC (através do DAMICORE) pode-se notar que as semelhanças destacadas no código fonte são, no geral, mantidas nos códigos binários, o que é esperado dado que o código de máquina é uma “representação” das instruções do programa que foram programadas em linguagem léxica. Entretanto, eventuais otimizações efetuadas pelos compiladores podem levar a mudanças nos códigos binários, que apesar de não alterarem o comportamento do programa em relação ao algoritmo executado pelo mesmo, podem não ser facilmente dedutíveis a partir do código fonte. Neste contexto, o uso da BCC pode revelar estas semelhanças diretamente nos códigos binários sem a necessidade de uma análise especialista de um código Assembly desmontado, ou através da análise do código fonte.

CLASSIFICADOR DE *CPU INTENSIVE*

Em relação a caracterização de arquivos binários executáveis, foi desenvolvido como caso de estudo um classificador de programas binários com perfil de consumo de recursos *CPU-Intensive* através da classificação por árvores de quartetos. O classificador é construído através dos seguintes passos:

1. Concatenam-se vários programas com características *CPU-Intensive* em um único arquivo CPU, representando a classe U_c (sem a contraprova);
2. Concatenam-se vários programas com características *I/O-Intensive* em um único arquivo IO, representando a classe $\bar{U} = W$;
3. Seleciona-se um programa como contraprova (y), *CPU-Intensive* ou *I/O-Intensive*;
4. Adiciona-se o programa incógnito, que em princípio não se sabe ser *CPU-Intensive* ou *I/O-Intensive*;
5. Executa-se o DAMICORE para esse conjunto de quatro arquivos.

A justificativa para a técnica é que, os diversos programas *CPU-Intensive* concatenados no arquivo CPU fazem com que as características que determinam esse comportamento sejam repetidas dentro do arquivo concatenado. O mesmo para os programas *I/O-Intensive*. Se a contraprova for do tipo *CPU-Intensive* e a incógnita for *I/O-Intensive*, é de se esperar que a filogenia resultante apresente o arquivo CPU e a contraprova unidos por uma similaridade, e os dois outros arquivos unidos no outro extremo da árvore. Caso isso não ocorra, deve-se substituir a contraprova por uma amostra *I/O-Intensive* e repetir o experimento, esperando pelo resultado oposto.

A partir do repositório com 80 programas, 40 programas foram selecionados para compor os arquivos concatenados referentes as classe *CPU-Intensive* e *I/O-Intensive*, respectivamente.

Dos 40 programas restantes, dois foram escolhidos aleatoriamente para representarem a contraprova de cada classe e todos os programas *CPU-Intensive* restantes foram aplicados ao classificador. A Tabela 13 lista todos os programas utilizados.

Tabela 13 – Programas utilizados para compor o repositório do classificador.

Classe	Contraprova	Programas concatenados
<i>CPU-Intensive</i>	<i>p₂₉ (mmq)</i>	<i>p₁, p₃, p₅, p₆, p₇, p₈, p₁₂, p₁₅, p₁₆, p₁₇, p₁₉, p₂₀, p₂₆, p₃₁, p₃₂, p₃₃, p₃₄, p₃₇, p₃₉, p₄₀</i>
<i>I/O-Intensive</i>	<i>p₄₇ (dd)</i>	<i>p₄₂, p₄₄, p₄₅, p₄₆, p₄₉, p₅₀, p₅₁, p₅₆, p₅₇, p₅₈, p₆₀, p₆₅, p₆₇, p₇₀, p₇₁, p₇₄, p₇₅, p₇₆, p₇₇, p₈₀</i>

Todos os programas listados na Tabela 15 passaram pelo classificador, que executou o mesmo procedimento para cada caso: primeiro executou-se o DAMICORE utilizando como contraprova o programa da classe *CPU-Intensive* (mmq) e depois repetiu-se o experimento porém considerando como contraprova o programa da classe *I/O-Intensive* (dd). A partir dos resultados respectivos a cada contraprova utilizada emitiu-se a classificação do programa incógnito. A Tabela 14 lista a classificação emitida de acordo com os possíveis resultados e a Tabela 15 contém a classificação final de cada programa avaliado.

Tabela 14 – Resultados do classificador para os programas *CPU-Intensive*.

Res. Contraprova <i>CPU-Intensive</i>	Res. Contraprova <i>I/O-Intensive</i>	Classificação
Positivo Forte	Negativo Fraco	<i>CPU-Intensive</i>
Positivo Forte	Neutro	<i>CPU-Intensive</i>
Neutro	Negativo Fraco	<i>CPU-Intensive</i>
Neutro	Neutro	Inconclusivo
Neutro	Positivo Forte	<i>I/O-Intensive</i>
Negativo Fraco	Neutro	<i>I/O-Intensive</i>
Negativo Fraco	Positivo Forte	<i>I/O-Intensive</i>
Negativo Fraco	Negativo Fraco	Erro
Positivo Forte	Positivo Forte	Erro

Dos 19 programas avaliados, 14 foram classificados como *CPU-Intensive*, 3 resultados foram inconclusivos (sem classificação) e 2 resultados foram *I/O-Intensive*. Se todos os 19 programas forem considerados exclusivamente do perfil *CPU-Intensive*, então o classificador obteve uma taxa de acerto de aproximadamente 74%. Desconsiderando resultados inconclusivos, a taxa sobe para 87%. Nos resultados inconclusivos, de fato o classificador não foi capaz de determinar com segurança a classe do programa analisado. Os dois programas classificados como *I/O-Intensive* foram *gzip* (*p₂₂*) e *mandelbulber* (*p₂₇*). Nos experimentos de aferição de consumo de recursos, ambos apresentaram majoritariamente consumo intenso de CPU durante o período de aferição. Entretanto, analisando as características e o código fonte destes programas, verifica-se que ambos efetuam operações com arquivos. O utilitário *gzip* é um conhecido compressor de arquivos, logo, apesar de consumir processamento executando os algoritmos de

compressão/descompressão, trabalha também com as operações de manipulação de arquivos. Já o programa *mandelbulber* trabalha com fractais e também faz a manipulação de arquivos, tanto leitura quanto gravação. É importante ressaltar que ambos programas também ficaram mais próximos dos arquivos *I/O-Intensive* nas árvores filogenéticas originárias do DAMICORE, as características de ambos em relação aos programas *I/O-Intensive* acabaram sendo mais significativas em relação aos programas *CPU-Intensive*, essas semelhanças foram detectadas pelo classificador.

Tabela 15 – Resultados do classificador para os programas *CPU-Intensive*.

Programa	Resultado
<i>p₂</i>	<i>CPU-Intensive</i>
<i>p₄</i>	<i>CPU-Intensive</i>
<i>p₉</i>	Inconclusivo
<i>p₁₀</i>	Inconclusivo
<i>p₁₁</i>	<i>CPU-Intensive</i>
<i>p₁₃</i>	<i>CPU-Intensive</i>
<i>p₁₄</i>	<i>CPU-Intensive</i>
<i>p₁₈</i>	Inconclusivo
<i>p₂₁</i>	<i>CPU-Intensive</i>
<i>p₂₂</i>	<i>I/O-Intensive</i>
<i>p₂₃</i>	<i>CPU-Intensive</i>
<i>p₂₄</i>	<i>CPU-Intensive</i>
<i>p₂₅</i>	<i>CPU-Intensive</i>
<i>p₂₇</i>	<i>I/O-Intensive</i>
<i>p₂₈</i>	<i>CPU-Intensive</i>
<i>p₃₀</i>	<i>CPU-Intensive</i>
<i>p₃₅</i>	<i>CPU-Intensive</i>
<i>p₃₆</i>	<i>CPU-Intensive</i>
<i>p₃₈</i>	<i>CPU-Intensive</i>

É importante ressaltar que os resultados obtidos pelo classificador poderiam ter sido refinados através da repetição dos testes com sucessivas reamostragens para a escolha de diferentes contraprovas. Entretanto, o objetivo deste estudo de caso é apenas ilustrar a capacidade de identificação de informações de um contexto a partir do código binário. Neste caso, o contexto de perfil de consumo de CPU foi escolhido por simplicidade. A utilização do classificador sem ajuste do parâmetro relativo ao número de reamostragens da contraprova possibilita mostrar que a identificação de contexto é possível e robusta (independe de um ajuste refinado de parâmetro).

CÓDIGOS MALICIOSOS

Apesar deste trabalho avaliar programas binários compilados para arquitetura x86, um terceiro estudo de caso foi efetuado afim de investigar a aplicação da BCC em outros tipos de programas binários a serem considerados em trabalhos futuros. Avaliar não somente o perfil de utilização de recursos dos programas, mas também seus respectivos comportamentos de uma maneira geral (algoritmos implementados, tarefas executadas, etc) é de grande importância para diversos tipos de aplicações, destacando-se entre elas a detecção de códigos maliciosos.

Softwares maliciosos (*malwares*) evoluíram e utilizam técnicas cada vez mais complexas, além de continuarem proliferando-se, especialmente nos aplicativos móveis para a plataforma Android¹, amplamente adotada pela indústria de *smartphones*(SUAREZ-TANGIL *et al.*, 2014). Os objetivos de um *malware* incluem ganho de acesso a um sistema privado, obtenção de informações privadas e/ou sensíveis a determinado contexto, exibição de comerciais não desejados, utilização do dispositivo para execução de tarefas não autorizadas pelo usuário (tais como envio de SPAM), entre outros. Existem diversos trabalhos na literatura que utilizam técnicas como mineração de dados ((SUAREZ-TANGIL *et al.*, 2014),(AMOS; TURNER; WHITE, 2013),(SAHS; KHAN, 2012), (PEIRAVIAN; ZHU, 2013)) e técnicas estatísticas ((PENG *et al.*, 2012),(YERIMA *et al.*, 2013)) para a detecção e classificação de *malwares* dedicados à plataforma Android.

Para este estudo de caso um repositório com 161 *malwares* para Android foi criado a partir de um repositório disponível no site *Free Range Secury Notes*². O repositório original contém diversos aplicativos em seu formato binário na extensão **.apk** para a plataforma Android. O arquivo **.apk** consiste em um arquivo comprimido (de extensão original *.zip*) com os arquivos binários do respectivo aplicativo e um diretório de metadados (*META-INF*), contendo informações diversas tais como certificados, dependências de software, etc. O repositório original

¹ Disponível em <<https://www.android.com/>>

² Disponível em <<http://cgi.cs.indiana.edu/~nhusted/dokuwiki/doku.php?id=datasets>>

provê um documento com uma classificação dos seus respectivos aplicativos feita com base no comportamento principal observado pelos mesmos, sendo definidas quatro categorias, descritas na Tabela 16³ assim como as respectivas quantidades de aplicativos das mesmas.

Tabela 16 – Categorias dos *malwares* do repositório.

Categoria	Qtd.	Comportamento
<i>Information Stealing</i> (INFO)	79	Rouba dados do dispositivo
<i>Privilege Escalation</i> (PRIV)	15	Tenta fazer ataques para obter elevação de privilégios do aplicativo
<i>Premium SMS Transmitting</i> (SMS)	14	Faz o envio de SMS <i>premium</i> , ou seja, mensagens SMS na qual o destinatário é taxado pelo recebimento ao invés do emissor ser taxado pelo envio
<i>Trojan</i> (TROJAM)	53	Efetua o download de alguma carga útil (outros programas, etc)

Figura 50 – Gráfico com o tamanho de cada programa do repositório.

A Figura 50 contém o gráfico com a distribuição dos *malwares* entre os agrupamentos encontrados pela aplicação da BCC. Foram detectados 18 grupos, os grupos 0, 4, 30, 44, 51, 183

³ Os nomes das categorias foram aqui abreviados em INFO, PRIV, SMS e TROJAM.

e 223 contém exclusivamente *malwares* de uma única categoria. Já os outros grupos contém *malwares* de categorias diferentes agrupados no mesmo grupo, porém em todos os casos há predominância de uma categoria no grupo. A Figura 51 contém a árvore filogenética gerada para o repositório. Os ramos estão coloridos de acordo com a categoria dos *malwares* representados nos nós. É possível observar notadamente o agrupamento dos *malwares* da categoria **INFO** (azul) e **TROJAM** (vermelho), ambos em maior quantidade de aplicativos no repositório. Também é possível observar os agrupamentos dos aplicativos da categoria **SMS** (magenta), porém em ramos mais distantes. Já os *malwares* da categoria **PRIV** (verde) ficaram espalhados entre quatros principais ramos da árvore, não obtendo um bom agrupamento visual. Este resultado pode ser explicado pelo fato de que existem diversos tipos diferentes de ataques para elevação de privilégio de um processo, levando a diversos tipos diferentes de códigos, logo, afetando a classificação por semelhança.

Figura 51 – Árvore filogenética do repositório de *malwares* para Android.

Uma possível aplicação evidenciada a partir deste caso de estudo trata-se de um classificador de softwares maliciosos ou vírus. Porém, para obter um detalhamento maior das relações

hierárquicas dos *malwares* do repositório seria necessário um estudo mais aprofundado e específico para cada aplicativo, provavelmente necessitando a disponibilidade do código fonte (indisponível para este repositório). Por se tratar de outra plataforma e arquitetura de software, este estudo fugiria ao escopo deste trabalho. Entretanto, os resultados obtidos evidenciam a capacidade da BCC para prover o agrupamento de códigos binários também de uma arquitetura diferente à Intel x86, abordada neste trabalho.

Parte IV

Conclusões

CONCLUSÕES

O conhecimento do perfil de consumo de recursos de um programa é de interesse para diversas aplicações, incluindo balanceamento de carga, alocação de recursos adaptativa, planejamento de capacidade, entre outras. A quantidade de recursos tais como memória, Entrada/Saída e CPU demandados por um processo são consequência da execução do fluxo de instruções que implementam o algoritmo do programa, ou seja, são intrínsecas ao comportamento do programa, além dos dados de entrada. Neste sentido, a análise do código fonte da aplicação pode ser uma abordagem plausível para inferir o comportamento do programa e sua utilização de recursos através da análise do fluxo de execução do código, seja através de inspeção humana ou por ferramentas automáticas. Para este propósito se faz necessário que o código fonte esteja disponível, o que nem sempre é possível em aplicações comerciais e grandes sistemas de produção. A análise do código fonte também pode ser tornar demasiadamente custosa com o aumento do tamanho e complexidade do código da aplicação, além de serem dependentes de linguagem de programação.

O presente trabalho utiliza o código binário (código de máquina) de programas executáveis ao invés do código fonte para caracterizar o perfil das aplicações especialmente no que tange ao consumo de recursos. Uma vez que o código binário é uma representação do mesmo algoritmo desenvolvido no código fonte, este trabalho tem como objetivo caracterizar o perfil de consumo das aplicações com base somente na análise do código de binário das mesmas. Ao invés de utilizar métodos baseados em análise semântica, ou seja, com técnicas especiais que levam em consideração a semântica dos dados analisados, este trabalho utiliza uma técnica de mineração de dados agnóstica em relação a semântica dos dados (trabalhando em nível de bytes) para identificar padrões nos códigos das aplicações que estejam associados ao consumo de recursos das mesmas, pois as requisições de recursos: Entrada/Saída, CPU, memória, etc, são demandadas através da execuções das instruções de máquina e chamadas ao sistema do Sistema Operacional. O uso destas chamadas e/ou instruções dentro de laços de repetição e outras estruturas de controle também pode levar a padrões que estarão associados ao consumo de recursos.

Este trabalho apresentou uma nova técnica, denominada **BCC** (*Binary Code Clustering*), que sintetiza os procedimentos para análise de códigos binários baseada na técnica DAMICORE. Os resultados experimentais mostraram-se consistentes com a hipótese de pesquisa. Os programas considerados neste trabalho foram analisados sob diversos aspectos, incluindo código fonte, código binário, dados de consumo de recursos, diferentes compiladores e níveis de otimização. Os resultados mostraram que os agrupamentos providos em todos os repositórios avaliados é robusto no sentido de que diferentes amostragens apresentaram agrupamentos consistentes entre si. Além disso, a congruência entre agrupamentos produzidos a partir dos códigos binários dos programas e de seus dados de perfil de consumo é coerente com a hipótese de que similaridades no perfil de consumo de recursos dos programas possuem correspondência nas similaridades detectadas nos códigos binários. Tais conclusões são observadas pela inspeção das árvores filogenéticas obtidas e quantificadas pelo índice de congruência ρ , proposto neste trabalho.

12.1 Trabalhos Futuros

Este trabalho pode ser estendido sob diversos aspectos, tais como:

- Explorar a BCC em outros Sistemas Operacionais
- Investigar a BCC para arquiteturas de hardware (FPGAs, arquiteturas embarcadas, etc)
- Estender a BCC para plataformas paralelas e de alto desempenho
- Analisar o desempenho da BCC para códigos binários em *bytecode*
- Refinar os processos de reamostragem tanto da base de dados quanto das contraprovas utilizadas nos testes de quartetos
- Estender a BCC para lidar com características extraídas de códigos binários para diferentes contextos de aplicações

Um dos casos de estudo apresentados neste trabalho efetuou o agrupamento de *malwares* da plataforma Android, sendo capaz de agrupar os programas maliciosos pelo perfil de ataque ocasionado. Este resultado pode motivar estudos mais aprofundados no sentido de obter novas técnicas, ou melhorar as existentes, para a detecção automática de códigos maliciosos e/ou vírus digitais. A BCC pode representar uma abordagem promissora neste contexto, em que muitas vezes somente o código binário está disponível.

12.2 Publicações e Prêmio

- SOUZA PINTO, R.; DELBEM, A. C. B.; MONACO, F. J. Caracterização do perfil de consumo de recursos de programas binários utilizando a técnica DAMICORE. **XIII Simpósio Brasileiro de Sistemas de Informação**, 2017. Lavras - MG.
- SOUZA PINTO, R.; DELBEM, A. C. B.; MONACO, F. J. Caracterização do perfil de carga a partir de programas binários. **8ª Escola Regional de Alto Desempenho - ERAD-SP**, 2017. São Carlos - SP.
 - Prêmio recebido: Melhor artigo de Pós-Graduação.
- SOUZA PINTO, R.; DELBEM, A. C. B.; MONACO, F. J. Estimation of Runtime Resource Consumption from the Analysis of Executable Program Files. Artigo submetido ao periódico **Applied Soft Computing**. Ed. Elsevier, ISSN: 1568-4946.

REFERÊNCIAS

- AMOS, B.; TURNER, H.; WHITE, J. Applying machine learning classifiers to dynamic android malware detection at scale. In: IEEE. **Wireless communications and mobile computing conference (iwcmc), 2013 9th international**. [S.l.], 2013. p. 1666–1671. Citado na página 131.
- BABCOCK, C. **Will Dell Lead The x86 Data Center Market?** 2011. Disponível em <http://www.informationweek.com/mobile/will-dell-lead-the-x86-data-center-market/d/d-id/1100898?page_number=1>. Acessado em 27 de Dezembro de 2013. Citado na página 43.
- BACON, D. F.; GRAHAM, S. L.; SHARP, O. J. Compiler transformations for high-performance computing. **ACM Computing Surveys (CSUR)**, ACM, v. 26, n. 4, p. 345–420, 1994. Citado na página 104.
- BARDIN, S.; HERRMANN, P.; LEROUX, J.; LY, O.; TABARY, R.; VINCENT, A. The bincoa framework for binary code analysis. In: SPRINGER. **Computer Aided Verification**. [S.l.], 2011. p. 165–170. Citado na página 28.
- BAUMANN, D. **Dosfstools**. 2014. Available at <<http://daniel-baumann.ch/software/dosfstools/>>. Last access on November, 2014. Citado na página 45.
- BERGLUND, C. **Duff**. 2014. Available at <<http://duff.dreda.org/>>. Last access on November, 2014. Citado na página 45.
- BERNSTEIN, D. J. **Primegen**. 2013. Available at <<http://cr.yp.to/primegen.html>>. Last access on December, 2013. Citado na página 44.
- BRAY, T. **Bonnie**. 2013. Available at <<http://code.google.com/p/bonnie-64>>. Last access on December, 2013. Citado na página 45.
- BURKARDT, J. **Sum Million**. 2013. Available at <http://people.sc.fsu.edu/~jburkardt/c_src/sum_million/sum_million.html>. Last access on December, 2013. Citado na página 44.
- CALCOTE, J. **Autotools: A Practitioner’s Guide to GNU Autoconf, Automake, and Libtool**. [S.l.]: No Starch Press, 2010. Citado na página 81.
- CESAR, B. K. M. **Estudo e extensão da metodologia DAMICORE para tarefas de classificação**. Tese (Doutorado) — Universidade de São Paulo, 2016. Citado na página 98.
- CHAUCER, G. **The Canterbury tales**. 1993. Disponível em <<http://www.hti.umich.edu/c/cme/>>. Acessado em Junho de 2017. Citado na página 75.
- CHOI, K.; SOMA, R.; PEDRAM, M. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, IEEE, v. 24, n. 1, p. 18–28, 2005. Citado na página 44.
- CILIBRASI, R. The complearn toolkit software. 2003. Citado na página 29.

- CILIENDO, E.; KUNIMASA, T. **Linux performance and tuning guidelines**. [S.l.]: IBM, International Technical Support Organization, 2007. Citado na página 45.
- CORNEA, B. F.; BOURGEOIS, J. A framework for efficient performance prediction of distributed applications in heterogeneous systems. **The Journal of Supercomputing**, Springer, v. 62, n. 3, p. 1609–1634, 2012. Citado na página 32.
- CURINO, C.; JONES, E. P.; MADDEN, S.; BALAKRISHNAN, H. Workload-aware database monitoring and consolidation. In: ACM. **Proceedings of the 2011 ACM SIGMOD International Conference on Management of data**. [S.l.], 2011. p. 313–324. Citado na página 44.
- CURNOW, H. J.; WICHMANN, B. A. A synthetic benchmark. **The Computer Journal**, Br Computer Soc, v. 19, n. 1, p. 43–49, 1976. Citado na página 44.
- DAVIS, I.; HEMMATI, H.; HOLT, R.; GODFREY, M.; NEUSE, D.; MANKOVSKII, S. Regression-based utilization prediction algorithms: An empirical investigation. 2013. Citado na página 31.
- Debian Project. **cdrkit**. 2013. Available at <<http://www.cdrkit.org/>>. Last access on December, 2013. Citado na página 45.
- DECKER, E. B.; MACKENZIE, D. J.; MEYERING, J.; WOOD, T.; DALLEY, K.; EGGERT, P. **Findutils**. 2014. Available at <<http://www.gnu.org/software/findutils/>>. Last access on November, 2014. Citado na página 45.
- DELBEM, A. C. **Mineração de dados de biofotônica e climáticos para aumento da precisão e da robustez de modelos de predição/deteção do Greening**. [S.l.], 2012. Citado na página 97.
- DHAMDHARE, D. M. **Operating Systems: A Concept-based Approach, 2E**. [S.l.]: Tata McGraw-Hill Education, 2006. Citado na página 43.
- Diskbench. **Simple low-level command-line disk benchmark utilities written in C**. 2013. Disponível em <<https://github.com/screwtop/diskbench>>. Acessado em 27 de Dezembro de 2013. Citado na página 45.
- DOULAMIS, N.; DOULAMIS, A.; LITKE, A.; PANAGAKIS, A.; VARVARIGOU, T.; VARVARIGOS, E. Adjusted fair scheduling and non-linear workload prediction for qos guarantees in grid computing. **Computer Communications**, Elsevier, v. 30, n. 3, p. 499–515, 2007. Citado na página 31.
- DOWNEY, A. B. Predicting queue times on space-sharing parallel computers. In: IEEE. **Parallel Processing Symposium, 1997. Proceedings., 11th International**. [S.l.], 1997. p. 209–218. Citado na página 28.
- EFRON, B.; STEIN, C. The jackknife estimate of variance. **The Annals of Statistics**, JSTOR, p. 586–596, 1981. Citado na página 80.
- FELSENSTEIN, J.; FELENSTEIN, J. **Inferring phylogenies**. [S.l.]: Sinauer Associates Sunderland, 2004. v. 2. Citado na página 35.
- FERREIRA, E. J.; MELO, V. V.; DELBEM, A. C. B. Algoritmos de estimação de distribuição em mineração de dados: Diagnóstico do greening in citrus. In: **II Escola Luso-Brasileira de Computação Evolutiva**. Guimarães, Portugal: [s.n.], 2010. p. 1–1. Citado na página 38.

Fidelity National Information Services. **FIS releases io_thrash a useful utility program for assessing IO performance**. 2013. Available at <http://www.fisglobal.com/products-technologyplatforms-gtm-infobulletins-s004480>. Last access on December, 2013. Citado na página 45.

FOWLKES, E. B.; MALLOWS, C. L. A method for comparing two hierarchical clusterings. **Journal of the American Statistical Association**, v. 78, n. 383, p. 553–569, 1983. Disponível em: <http://www.tandfonline.com/doi/abs/10.1080/01621459.1983.10478008>. Citado na página 74.

GCCC. **GCC: GNU Compiler Collection**. 2014. Available at <https://gcc.gnu.org/>. Last access on September, 2015. Citado na página 45.

GIBBONS, R. A historical application profiler for use by parallel schedulers. In: SPRINGER. **Job Scheduling Strategies for Parallel Processing**. [S.l.], 1997. p. 58–77. Citado na página 28.

Glium Project. **Gouraud Shading**. 2016. Disponível em <https://tomaka.github.io/glium/book/tuto-08-gouraud.html>. Acessado em Junho de 2017. Citado nas páginas 16 e 118.

GNU. **The GNU C Library**. 2015. Available at <http://www.gnu.org/software/libc/>. Last access on September, 2015. Citado na página 45.

GNU Coreutils. **Coreutils - GNU core utilities**. 2014. Available at <https://www.gnu.org/software/coreutils>. Last access on November, 2014. Citado na página 81.

GNU cpio. **Cpio**. 2014. Available at <http://www.gnu.org/software/cpio/>. Last access on November, 2014. Citado na página 45.

GNU Freedup. **Freedup**. 2014. Available at <http://freedup.org/>. Last access on November, 2014. Citado na página 45.

HALL, B. G. **Phylogenetic trees made easy: A how to manual**. [S.l.]: Sinauer, 2011. Citado na página 36.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. H. **The Elements of Statistical Learning**. Springer, 2003. Hardcover. ISBN 0387952845. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0387952845>. Citado na página 33.

HELING, Y. **Ncd**. 2014. Available at <http://dev.yorhel.nl/ncdu/man>. Last access on November, 2014. Citado na página 45.

HENNING, J. L. Spec cpu2006 benchmark descriptions. **ACM SIGARCH Computer Architecture News**, ACM, v. 34, n. 4, p. 1–17, 2006. Citado na página 44.

Hewlett-Packard. **An I/O performance measurement and load generation tool**. 2013. Available at <http://spew.berlios.de>. Last access on December, 2013. Citado na página 45.

HUCK, K. A.; MALONY, A. D. Perfexplorer: A performance data mining framework for large-scale parallel computing. In: IEEE COMPUTER SOCIETY. **Proceedings of the 2005 ACM/IEEE conference on Supercomputing**. [S.l.], 2005. p. 41. Citado na página 81.

HUI, C.-C.; CHANSON, S. T. Improved strategies for dynamic load balancing. **Concurrency, IEEE**, IEEE, v. 7, n. 3, p. 58–67, 1999. Citado na página 44.

- JACCARD, P. The distribution of the flora in the alpine zone. 1. **New phytologist**, Wiley Online Library, v. 11, n. 2, p. 37–50, 1912. Citado na página 74.
- JAIN, R. **The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling**. [S.l.]: New York: John, 1991. Citado na página 31.
- JU, J.; XU, G.; YANG, K. An intelligent dynamic load balancer for workstation clusters. **ACM SIGOPS Operating Systems Review**, ACM, v. 29, n. 1, p. 7–16, 1995. Citado na página 31.
- KAWAGUCHI, S.; YACHI, T. New power consumption prediction procedure using performance counters to increase computer power supply efficiency. In: VDE. **Telecommunications Energy Conference'Smart Power and Efficiency'(INTELEC), Proceedings of 2013 35th International**. [S.l.], 2013. p. 1–6. Citado na página 31.
- KHLEBNIKOV, K. **ioping - simple disk I/O latency measuring tool**. 2014. Available at <<http://squashfs.sourceforge.net/>>. Last access on November, 2014. Citado na página 45.
- KIM, H.; LIM, H.; JEONG, J.; JO, H.; LEE, J. Task-aware virtual machine scheduling for i/o performance. In: ACM. **Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments**. [S.l.], 2009. p. 101–110. Citado na página 44.
- KLUGE, F.; UHRIG, S.; MISCHKE, J.; SATZGER, B.; UNGERER, T. Dynamic workload prediction for soft real-time applications. In: IEEE. **Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on**. [S.l.], 2010. p. 1841–1848. Citado na página 31.
- LERIDA, J. L.; SOLSONA, F.; HERNANDEZ, P.; GINE, F.; HANZICH, M.; CONDE, J. State-based predictions with self-correction on enterprise desktop grid environments. **Journal of Parallel and Distributed Computing**, Elsevier, 2013. Citado na página 31.
- LESLIE, C.; ESKIN, E.; COHEN, A.; WESTON, J.; NOBLE, W. S. Mismatch string kernels for discriminative protein classification. **Bioinformatics**, v. 20, p. 467–476, 2004. Citado na página 29.
- LI, M.; VITÁNYI, P. **An introduction to Kolmogorov complexity and its applications**. [S.l.]: Springer Science & Business Media, 2013. Citado na página 34.
- LINUXINSIGHT. **How fast is your disk?** 2014. Available at <http://www.linuxinsight.com/how_fast_is_your_disk.html>. Last access on November, 2014. Citado na página 45.
- LIU, K.; TAN, H. B. K.; CHEN, X. Binary code analysis. **Computer**, IEEE, n. 8, p. 60–68, 2013. Citado na página 28.
- LIU, M.; JIN, Y.; ZHAI, J.; ZHAI, Y.; SHI, Q.; MA, X.; CHEN, W. Acic: automatic cloud i/o configurator for parallel applications. In: ACM. **Proceedings of the 22nd international symposium on High-performance parallel and distributed computing**. [S.l.], 2013. p. 111–112. Citado na página 32.
- LOUGHER, P.; LOUGHER, R. **Squashfs**. 2013. Available at <<http://squashfs.sourceforge.net/>>. Last access on December, 2013. Citado na página 45.

LUI, L. T.; TERRAZAS, G.; ZENIL, H.; ALEXANDER, C.; KRASNOGOR, N. Complexity measurement based on information theory and kolmogorov complexity. **Artificial life**, MIT Press, 2015. Citado na página 34.

M., M. E. R. **Método automático de determinação de clados utilizando algoritmo de detecção de comunidades**. Dissertação (Mestrado) — Universidade de São Paulo, 2013. Citado na página 38.

MAHMOUD, H. A.; MOON, H. J.; CHI, Y.; HACIGÜMÜŞ, H.; AGRAWAL, D.; EL-ABBADI, A. Cloudoptimizer: multi-tenancy for i/o-bound olap workloads. In: **ACM. Proceedings of the 16th International Conference on Extending Database Technology**. [S.l.], 2013. p. 77–88. Citado na página 44.

MANDELBULBER. **Mandelbulber 3D Fractal Explorer**. 2013. Available at <<http://www.mandelbulber.com>>. Last access on December, 2013. Citado na página 44.

MANSOUR, M. R.; ALBERTO, L. F. C.; RAMOS, R. A.; DELBEM, A. C. Identifying groups of preventive controls for a set of critical contingencies in the context of voltage stability. In: **IEEE. Circuits and Systems (ISCAS), 2013 IEEE International Symposium on**. [S.l.], 2013. p. 453–456. Citado na página 38.

MARTINIAN, E. **Free Source Code**. 2013. Available at <http://web.mit.edu/~{ }emin/www.old/source_code/>. Last access on December, 2013. Citado na página 44.

MARTINS, L. G.; NOBRE, R.; DELBEM, A. C.; MARQUES, E.; CARDOSO, J. M. Exploration of compiler optimization sequences using clustering-based selection. In: **the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems**. Edinburgh: [s.n.], 2014. p. 63. Citado na página 38.

MARTINS, L. G. A. **Exploração de sequências de otimização do compilador baseada em técnicas híbridas de mineração de dados complexos**. Tese (Doutorado) — Universidade de São Paulo, 2016. Citado na página 38.

MARTINS, L. G. A.; NOBRE, R.; CARDOSO, J. a. M. P.; DELBEM, A. C. B.; MARQUES, E. Clustering-based selection for the exploration of compiler optimization sequences. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 13, n. 1, p. 8:1–8:28, mar. 2016. ISSN 1544-3566. Disponível em: <<http://doi.acm.org/10.1145/2883614>>. Citado na página 38.

MARTINS, L. G. A.; NOBRE, R.; DELBEM, A. C. B.; MARQUES, E.; CARDOSO, J. M. P. A clustering-based approach for exploring sequences of compiler optimizations. In: **2014 IEEE Congress on Evolutionary Computation (CEC)**. Beijing: [s.n.], 2014. p. 2436–2443. Doi: 10.1109/CEC.2014.6900634. Citado na página 38.

MCCARTY, J. **Fractal Flame Algorithm in C**. 2013. Available at <<https://github.com/jameswmccarty/Fractal-Flame-Algorithm-in-C>>. Last access on December, 2013. Citado na página 44.

MEILĂ, M. Comparing clusterings—an information based distance. **Journal of Multivariate Analysis**, Elsevier, v. 98, n. 5, p. 873–895, 2007. Citado na página 74.

MENOTTI, R. **LALP: uma linguagem para exploração do paralelismo de loops em computação reconfigurável**. Tese (Doutorado) — Universidade de São Paulo, São Carlos, maio 2010. Citado na página 44.

- MOEYERSOMS, J.; FORTUNY, E. J. de; DEJAEGER, K.; BAESSENS, B.; MARTENS, D. Comprehensible software fault and effort prediction: A data mining approach. **Journal of Systems and Software in review** (cit. on pp. iii, 134, 135), 2014. Citado na página 81.
- MORO, L. F.; RODRIGUEZ, C. L.; ANDRADE, F. R. H.; DELBEM, A. C. B.; ISOTANI, S. Caracterização de alunos em ambientes de ensino online. In: **Workshop de Mineração de Dados em Ambientes Virtuais do Ensino/Aprendizagem, Anais do Congresso Brasileiro de Informática na Educação**. Dourados: [s.n.], 2014. p. 1–10. Citado na página 38.
- MORO, L. F. S.; LOPES, A. M. Z.; DELBEM, A. C. B.; ISOTANI, S. Os desafios para minerar dados educacionais de forma rápida e intuitiva: o caso da damicore e a caracterização de alunos em ambientes de elearning.. in: **Ii workshop de desafios da computação aplicada à educação. In: XXXIII Congresso da Sociedade Brasileira de Computação**. Maceio: [s.n.], 2013. p. 1–10. Citado na página 38.
- MUKHERJEE, S. Static code analysis. In: **Thinking in LINQ**. [S.l.]: Springer, 2014. p. 151–164. Citado na página 27.
- NEWMAN, M. E. Fast algorithm for detecting community structure in networks. **Physical review E, APS**, v. 69, n. 6, p. 066133, 2004. Citado na página 37.
- _____. Modularity and community structure in networks. **Proceedings of the national academy of sciences**, National Acad Sciences, v. 103, n. 23, p. 8577–8582, 2006. Citado na página 37.
- NORDAL, A. **Dupedit**. 2014. Available at <<http://dupedit.com/>>. Last access on November, 2014. Citado na página 45.
- PEIRAVIAN, N.; ZHU, X. Machine learning for android malware detection using permission and api calls. In: IEEE. **Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on**. [S.l.], 2013. p. 300–305. Citado na página 131.
- PENG, H.; GATES, C.; SARMA, B.; LI, N.; QI, Y.; POTHARAJU, R.; NITA-ROTARU, C.; MOLLOY, I. Using probabilistic generative models for ranking risks of android apps. In: ACM. **Proceedings of the 2012 ACM conference on Computer and communications security**. [S.l.], 2012. p. 241–252. Citado na página 131.
- PINTO, R. S. **Data repository of MineBin project**. 2016. [Http://adaptsys.lasdpc.icmc.usp.br/minebin](http://adaptsys.lasdpc.icmc.usp.br/minebin). Citado na página 44.
- PU, X.; LIU, L.; MEI, Y.; SIVATHANU, S.; KOH, Y.; PU, C. Understanding performance interference of i/o workload in virtualized cloud environments. In: IEEE. **Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on**. [S.l.], 2010. p. 51–58. Citado na página 44.
- RAND, W. M. Objective criteria for the evaluation of clustering methods. **Journal of the American Statistical association**, Taylor & Francis Group, v. 66, n. 336, p. 846–850, 1971. Citado na página 74.
- REDDY, A. L. N.; BANERJEE, P. An evaluation of multiple-disk i/o systems. **Computers, IEEE Transactions on**, IEEE, v. 38, n. 12, p. 1680–1690, 1989. Citado na página 44.
- SAHS, J.; KHAN, L. A machine learning approach to android malware detection. In: IEEE. **Intelligence and security informatics conference (eisis), 2012 european**. [S.l.], 2012. p. 141–147. Citado na página 131.

- SANCHES, A.; CARDOSO, J. M.; DELBEM, A. C. Identifying merge-beneficial software kernels for hardware implementation. In: IEEE. **Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on**. [S.l.], 2011. p. 74–79. Citado nas páginas 29, 34, 38 e 53.
- SCHILLING, J. **CDRecord**. 2014. Available at <<http://cdrecord.org/>>. Last access on November, 2014. Citado na página 45.
- SEIBOLD, S.; BOWDEN, T.; BAUER, B.; NERIN, J.; FENG, S. **The /proc Filesystem**. 2009. Disponível em <<https://www.kernel.org/doc/Documentation/filesystems/proc.txt>>. Acessado em 27 de Dezembro de 2013. Citado na página 48.
- SEKAR, V.; MANIATIS, P. Verifiable resource accounting for cloud computing services. In: ACM. **Proceedings of the 3rd ACM workshop on Cloud computing security workshop**. [S.l.], 2011. p. 21–26. Citado na página 28.
- SILVA, B. A.; DELBEM, A. C. B.; DENIZ, P. C.; BONATO, V. Runtime mapping and scheduling for energy efficiency in heterogeneous multi-core systems. In: **International Conference on Reconfigurable Computing and FPGAs**. Mayan Riviera: [s.n.], 2015. p. 1–6. Citado na página 38.
- SILVA, B. d. A. **Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogêneas em FPGA**. Tese (Doutorado) — Universidade de São Paulo, 2016. Citado na página 38.
- SILVA, B. d. A.; CUMINATO, L. A.; DELBEM, A. C. B.; DINIZ, P. C.; BONATO, V. Application-oriented cache memory configuration for energy efficiency in multi-cores. **IET Computers Digital Techniques**, v. 9, n. 1, p. 73–81, 2015. ISSN 1751-8601. Citado na página 38.
- SOBEL, I. **History and definition of the sobel operator**. 2014. Disponível em <https://www.researchgate.net/publication/239398674_An_Isotropic_3_3_Image_Gradient_Operator>. Acessado em Junho de 2017. Citado na página 103.
- SU, S.; LI, J.; HUANG, Q.; HUANG, X.; SHUANG, K.; WANG, J. Cost-efficient task scheduling for executing large programs in the cloud. **Parallel Computing**, Elsevier, v. 39, n. 4, p. 177–188, 2013. Citado na página 28.
- SUAREZ-TANGIL, G.; TAPIADOR, J. E.; PERIS-LOPEZ, P.; BLASCO, J. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. **Expert Systems with Applications**, Elsevier, v. 41, n. 4, p. 1104–1117, 2014. Citado na página 131.
- SUMOYAMA, A. S. **Classificador de kernels para mapeamento em plataforma de computação híbrida composta por FPGA e GPP**. Tese (Doutorado) — Universidade de São Paulo, 2016. Citado na página 38.
- TAKEFUSA, A.; CASANOVA, H.; MATSUOKA, S.; BERMAN, F. A study of deadline scheduling for client-server systems on the computational grid. In: IEEE. **High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on**. [S.l.], 2001. p. 406–415. Citado na página 44.

- TELLES, G. P.; MINGHIM, R.; PAULOVICH, F. V. Visual mapping of text collections using an approximation of kolmogorov complexity. **Tech. Report**, 2005. Citado na página 29.
- TRIDGELL, A.; MACKERRAS, P. **Rsync**. 2014. Available at <<http://rsync.samba.org/>>. Last access on November, 2014. Citado na página 45.
- TRMAC, M. **Mlocate**. 2014. Available at <<https://fedorahosted.org/mlocate/>>. Last access on November, 2014. Citado na página 45.
- VANMECHELEN, K.; DEPOORTER, W.; BROECKHOVE, J. Economic grid resource management for cpu bound applications with hard deadlines. In: IEEE. **Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on**. [S.l.], 2008. p. 258–266. Citado na página 44.
- WARNEKE, D.; KAO, O. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. **Parallel and Distributed Systems, IEEE Transactions on**, IEEE, v. 22, n. 6, p. 985–997, 2011. Citado na página 28.
- WILSHIRE, P. Real-time linux: Testing and evaluation. In: **Real Time Linux Workshop, Orlando, FL**. [S.l.: s.n.], 2000. p. 13. Citado na página 44.
- WU, J.; KUO, T.-W. Real-time scheduling of cpu-bound and i/o-bound processes. In: IEEE. **Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on**. [S.l.], 1999. p. 303–310. Citado na página 44.
- WU, W.; CHI, Y.; HACÍGÜMÜŞ, H.; NAUGHTON, J. F. Towards predicting query execution time for concurrent and dynamic database workloads. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 6, n. 10, p. 925–936, 2013. Citado na página 31.
- YANG, Y.; TAN, Z.; XIE, C.; LIANG, W.; YU, J.; HE, J. A study of hotspot data prediction model in i/o workloads. **International Journal of Computer Mathematics**, Taylor & Francis, n. just-accepted, p. 1–50, 2013. Citado na página 32.
- YERIMA, S. Y.; SEZER, S.; MCWILLIAMS, G.; MUTTIK, I. A new android malware detection approach using bayesian classification. In: IEEE. **Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on**. [S.l.], 2013. p. 121–128. Citado na página 131.
- YING, A. T.; MURPHY, G. C.; NG, R.; CHU-CARROLL, M. C. Predicting source code changes by mining change history. **Software Engineering, IEEE Transactions on**, IEEE, v. 30, n. 9, p. 574–586, 2004. Citado na página 81.
- ZHAO, X.; YIN, J.; CHEN, Z.; HE, S. Workload classification model for specializing virtual machine operating system. In: IEEE. **Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on**. [S.l.], 2013. p. 343–350. Citado na página 31.

CÓDIGOS UTILIZADOS NO REPOSITÓRIO DE TESTES DO CAPÍTULO 9

A.1 Algoritmos Modulação/Demodulação ADPCM

Código-fonte 8 – Código do algoritmo de Modulação ADPCM

```
1: #define DATASIZE 8096
2:
3: short indata[DATASIZE];
4: char outdata[(DATASIZE/2)+1];
5:
6: static int indexTable[16] = {
7:     -1, -1, -1, -1, 2, 4, 6, 8,
8:     -1, -1, -1, -1, 2, 4, 6, 8,
9: };
10:
11: static int stepsizeTable[89] = {
12:     7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
13:     19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
14:     50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
15:     130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
16:     337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
17:     876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
18:     2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
19:     5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635,
20:     13899,
21:     15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794,
22:     32767
```

```
21: };
22:
23: int main() {
24:     int i;
25:     int len;
26:     int sign;
27:     int delta;
28:     int step;
29:     int valpred = 0;
30:     int vpdiff;
31:     int index = 0;
32:     int bufferstep;
33:     int outputbuffer = 0;
34:     int diff;
35:     int val;
36:     step = stepsizeTable[index];
37:     bufferstep = 1;
38:     i=0;
39:
40:     for (len = 0 ; len < DATASIZE ; len++ ) {
41:         val = indata[len];
42:         diff = val - valpred;
43:         sign = (diff < 0) ? 8 : 0;
44:         if ( sign )
45:             diff = (-diff);
46:         delta = 0;
47:         vpdiff = (step >> 3);
48:         if ( diff >= step ) {
49:             delta = 4;
50:             diff -= step;
51:             vpdiff += step;
52:         }
53:         step >>= 1;
54:         if ( diff >= step ) {
55:             delta |= 2;
56:             diff -= step;
57:             vpdiff += step;
58:         }
59:         step >>= 1;
60:         if ( diff >= step ) {
61:             delta |= 1;
62:             vpdiff += step;
```



```

63:     }
64:     if ( sign )
65:         valpred -= vpdiff;
66:     else
67:         valpred += vpdiff;
68:     if ( valpred > 32767 )
69:         valpred = 32767;
70:     else if ( valpred < -32768 )
71:         valpred = -32768;
72:     delta |= sign;
73:     index += indexTable[delta];
74:     if ( index < 0 )
75:         index = 0;
76:     if ( index > 88 )
77:         index = 88;
78:     step = stepsizeTable[index];
79:     if ( bufferstep ) {
80:         outputbuffer = (delta << 4) & 0xf0;
81:     } else {
82:         outdata[i] = (delta & 0x0f) | outputbuffer;
83:     }
84:     bufferstep = !bufferstep;
85: }
86:
87: if ( !bufferstep )
88:     outdata[i] = outputbuffer;
89:
90: return outputbuffer;
91: }

```

Código-fonte 9 – Código do algoritmo de Demodulação ADPCM

```

1: #define DATASIZE 1024
2:
3: short indata[(DATASIZE/2)+1];
4: char outdata[DATASIZE];
5:
6: static int indexTable[16] = {
7:     -1, -1, -1, -1, 2, 4, 6, 8,
8:     -1, -1, -1, -1, 2, 4, 6, 8,
9: };
10:

```

```
11: static int stepsizeTable[89] = {
12:     7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
13:     19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
14:     50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
15:     130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
16:     337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
17:     876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
18:     2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
19:     5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635,
    13899,
20:     15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794,
    32767
21: };
22:
23: int main() {
24:     int i;
25:     int len;
26:     int sign;
27:     int delta;
28:     int step;
29:     int valpred = 0;
30:     int vpdiff;
31:     int index = 0;
32:     int bufferstep;
33:     int inputbuffer = 0;
34:     step = stepsizeTable[index];
35:     bufferstep = 0;
36:     i=0;
37:
38:     valpred = 0;
39:     vpdiff;
40:     index = 0;
41:     bufferstep;
42:     inputbuffer = 0;
43:     step = stepsizeTable[index];
44:     bufferstep = 0;
45:     i=0;
46:
47:     for (len = 0 ; len < DATASIZE ; len++ ) {
48:         if ( bufferstep ) {
49:             delta = inputbuffer & 0xf;
50:         }
```

```
51:     else {
52:         inputbuffer = indata[i];
53:         i++;
54:         delta = (inputbuffer >> 4) & 0xf;
55:     }
56:     bufferstep = !bufferstep;
57:     index += indexTable[delta];
58:     if ( index < 0 )
59:         index = 0;
60:     if ( index > 88 )
61:         index = 88;
62:     sign = delta & 8;
63:     delta = delta & 7;
64:     vpdiff = step >> 3;
65:     if ( delta & 4 )
66:         vpdiff += step;
67:     else
68:         if ( delta & 2 )
69:             vpdiff += step>>1;
70:         else
71:             if ( delta & 1 )
72:                 vpdiff += step>>2;
73:     if ( sign )
74:         valpred -= vpdiff;
75:     else
76:         valpred += vpdiff;
77:     if ( valpred > 32767 )
78:         valpred = 32767;
79:     else if ( valpred < -32768 )
80:         valpred = -32768;
81:     step = stepsizeTable[index];
82:     outdata[len] = valpred;
83: }
84: return valpred;
85: }
```

A.2 Algoritmo Sobel

```

1: void originalSobel()
2: {
3:     int i, j;
4:
5:     for(i = 1; i < M-1; i++) {
6:         for(j = 1; j < N-1; j++) {
7:             r00 = imgIn[i-1][j-1];
8:             r10 = imgIn[i][j-1];
9:             r20 = imgIn[i+1][j-1];
10:
11:             r01 = imgIn[i-1][j];
12:             r11 = imgIn[i][j];
13:             r21 = imgIn[i+1][j];
14:
15:             r02 = imgIn[i-1][j+1];
16:             r12 = imgIn[i][j+1];
17:             r22 = imgIn[i+1][j+1];
18:
19:             num_loads_external += 9;
20:             imgOut[i][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
                r22);
21:             num_stores_external += 1;
22:         }
23:     }
24: }

```

Código-fonte 11 – Sobel modificado com *loop peeling* para a primeira coluna

```

1: void reuseColsSobel()
2: {
3:     int i, j;
4:
5:     for(i = 1; i < M-1; i++) {
6:         /* --- peeled j = 1 iteration of the j loop */
7:         r00 = imgIn[i-1][0];
8:         r10 = imgIn[i][0];
9:         r20 = imgIn[i+1][0];
10:
11:         r01 = imgIn[i-1][1];
12:         r11 = imgIn[i][1];
13:         r21 = imgIn[i+1][1];
14:

```

```
15:     r02 = imgIn[i-1][2];
16:     r12 = imgIn[i][2];
17:     r22 = imgIn[i+1][2];
18:
19:     num_loads_external += 9;
20:     imgOut[i][1] = compute(r00,r01,r02,r10,r11,r12,r20,r21,r22)
    ;
21:     num_stores_external += 1;
22:
23:     line0[0] = r00; line0[1] = r01; line0[2] = r02;
24:     line1[0] = r10; line1[1] = r11; line1[2] = r12;
25:     line2[0] = r20; line2[1] = r21; line2[2] = r22;
26:
27:     idx0 = 0;
28:     idx1 = 0;
29:     idx2 = 0;
30:
31:     for (j = 2; j < N-1; j++) {
32:         r00 = line0[idx0+1];
33:         r10 = line1[idx1+1];
34:         r20 = line2[idx2+1];
35:
36:         r01 = line0[idx0+2];
37:         r11 = line1[idx1+2];
38:         r21 = line2[idx2+2];
39:
40:         r02 = imgIn[i-1][j+1];
41:         r12 = imgIn[i][j+1];
42:         r22 = imgIn[i+1][j+1];
43:
44:         num_loads_external += 3;
45:         imgOut[i][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
r22);
46:         num_stores_external += 1;
47:
48:         line0[idx0+3] = r02;
49:         line1[idx1+3] = r12;
50:         line2[idx2+3] = r22;
51:
52:         idx0++;
53:         idx1++;
54:         idx2++;
```

```
55:     }  
56: }  
57: }
```

Código-fonte 12 – Sobel modificado com *loop peeling* para a primeira linha e coluna

```
1: void reuseRowsColsSobel()  
2: {  
3:     int i, j;  
4:     int *temp;  
5:  
6:     idx0 = 0;  
7:     idx1 = 0;  
8:     idx2 = 0;  
9:  
10:    /* -- peeled first iteration of i = 1 */  
11:    /* -- peeled first iteration of j = 1 */  
12:    {  
13:        r00 = imgIn[0][0];  
14:        r10 = imgIn[1][0];  
15:        r20 = imgIn[2][0];  
16:  
17:        r01 = imgIn[0][1];  
18:        r11 = imgIn[1][1];  
19:        r21 = imgIn[2][1];  
20:  
21:        r02 = imgIn[0][2];  
22:        r12 = imgIn[1][2];  
23:        r22 = imgIn[2][2];  
24:  
25:        num_loads_external += 9;  
26:  
27:        imgOut[1][1] = compute(r00,r01,r02,r10,r11,r12,r20,r21,r22)  
28:        ;  
29:  
30:        line0[0] = r00; line0[1] = r01; line0[2] = r02;  
31:        line1[0] = r10; line1[1] = r11; line1[2] = r12;  
32:        line2[0] = r20; line2[1] = r21; line2[2] = r22;  
33:  
34:        num_stores_external += 1;  
35:        idx0++;
```

```
36:     idx1++;
37:     idx2++;
38: }
39:
40: for (j = 2; j < N-1; j++) {
41:     r00 = line0[idx0];
42:     r10 = line1[idx1];
43:     r20 = line2[idx2];
44:
45:     r01 = line0[idx0+1];
46:     r11 = line1[idx1+1];
47:     r21 = line2[idx2+1];
48:
49:     r02 = imgIn[0][j+1];
50:     r12 = imgIn[1][j+1];
51:     r22 = imgIn[2][j+1];
52:
53:     num_loads_external += 3;
54:
55:     imgOut[1][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,r22)
    ;
56:
57:     line0[idx0+2] = r02; line1[idx1+2] = r12; line2[idx2+2] =
    r22;
58:
59:     num_stores_external += 1;
60:
61:     idx0++;
62:     idx1++;
63:     idx2++;
64: }
65:
66: for (i = 2; i < M-1; i++) {
67:     /* --- reset pointers and rotate lines */
68:     idx0 = 0;
69:     idx1 = 0;
70:     idx2 = 0;
71:
72:     temp = line0;
73:     line0 = line1;
74:     line1 = line2;
75:     line2 = temp;
```

```

76:
77:     /* --- peeled the first iteration j = 1 */
78:     {
79:         r00 = line0[0];    r01 = line0[1];    r02 = line0[2];
80:         r10 = line1[0];    r11 = line1[1];    r12 = line1[2];
81:
82:         r20 = imgIn[i+1][0];    r21 = imgIn[i+1][1];    r22 = imgIn
[i+1][2];
83:
84:
85:         num_loads_external += 3;
86:
87:         imgOut[i][1] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
r22);
88:
89:         line2[0] = r20;    line2[1] = r21;    line2[2] = r22;
90:
91:         num_stores_external += 1;
92:
93:         idx0++;
94:         idx1++;
95:         idx2++;
96:     }
97:     for (j = 2; j < N-1; j++) {
98:         r00 = line0[idx0];
99:         r10 = line1[idx1];
100:        r20 = line2[idx2];
101:
102:        r01 = line0[idx0+1];
103:        r11 = line1[idx1+1];
104:        r21 = line2[idx2+1];
105:
106:        r02 = line0[idx0+2];
107:        r12 = line1[idx1+2];
108:        r22 = imgIn[i+1][j+1];
109:
110:        num_loads_external += 1;
111:
112:        imgOut[i][j] = compute(r00,r01,r02,r10,r11,r12,r20,r21,
r22);
113:
114:        line2[idx2+2] = r22;

```



```
115:
116:     num_stores_external += 1;
117:
118:     idx0++;
119:     idx1++;
120:     idx2++;
121: }
122: }
123: }
```

A.3 Algoritmo de método numérico

Código-fonte 13 – Código do algoritmo Simpson $\frac{1}{3}$

```
1: #include <stdio.h>
2: #include <math.h>
3:
4: #define tnum double
5:
6: tnum f_clapeyron(tnum t);
7: tnum simpson13(tnum a, tnum b, tnum h, unsigned int n);
8:
9: int main(int argc, char **argv) {
10:     tnum p, h;
11:     tnum resultado;
12:     tnum p0 = 0.028;
13:     int npontos;
14:     int i;
15:
16:     npontos = 9;
17:     h = 5;
18:
19:     resultado = simpson13(185,225,h,npontos);
20:     p = p0 * exp(resultado);
21:
22:     return(0);
23: }
24:
25: tnum f_clapeyron(tnum t) {
26:     tnum temps[9] = { 185, 190, 195, 200, 205, 210, 215, 220, 225
        };
```

```
27:  tnum deltas[9] = { 81.307, 80.472, 79.714, 78.714, 77.859,
    77.002, 76.141, 75.272, 74.395 };
28:  tnum R      = 16.14;
29:  tnum val = 0;
30:  int i;
31:
32:  for(i=0;i<9;i++)
33:      if(t == temps[i]) {
34:          val = deltas[i] / (R * temps[i]);
35:          break;
36:      }
37:
38:  return(val);
39: }
40:
41: tnum simpson13(tnum a, tnum b, tnum h, unsigned int n) {
42:  tnum total = 0;
43:  tnum x, s1, s2;
44:  int i;
45:
46:  s1 = s2 = 0;
47:  for(i=1;i<=(n/2);i++) {
48:      x = a + ((2*i)-1) * h;
49:      s1 += f_clapeyron(x);
50:
51:      if(i < (n/2)) {
52:          x = a + (2*i) * h;
53:          s2 += f_clapeyron(x);
54:      }
55:  }
56:
57:  total = f_clapeyron(a) + f_clapeyron(b) + (4*s1) + (2*s2);
58:  total *= (h/3);
59:
60:  return(total);
61: }
```

A.4 Algoritmo de sombreamento

```
1: #define TP
2: #define N 200
3:
4: int p[N];
5:
6: unsigned int n;
7: unsigned int rd;
8: unsigned int r;
9: unsigned int gd;
10: unsigned int g;
11: unsigned int bd;
12: int b;
13:
14: main() {
15:     unsigned int mask = 0xF800F800, i;
16:
17:     for (i = 0; i < N; i++) {
18:         r += rd;
19:         g += gd;
20:         b += bd;
21:         p[i] = (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
22:     }
23: }
```

A.5 Algoritmos de ordenação

Código-fonte 15 – Código do algoritmo *Bubble sort*

```
1: #define N 32
2:
3: int ar[N];
4:
5: main() {
6:     int i, j;
7:     int temp;
8:     for (i=0; i<N; i++) {
9:         for (j=0; j<N-1; j++) {
10:             if (ar[j] > ar[j+1]) {
11:                 temp = ar[j+1];
12:                 ar[j+1] = ar[j];
```

```
13:         ar[j] = temp;
14:     }
15: }
16: }
17: }
```

Código-fonte 16 – Código do algoritmo *Heap sort*

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <time.h>
4:
5: #define VETSIZE 20000
6:
7: void downHeap(int a[], int root, int bottom)
8: {
9:     int maxchild, temp, child;
10:    while (root*2 < bottom)
11:    {
12:        child = root * 2 + 1;
13:        if (child == bottom)
14:        {
15:            maxchild = child;
16:        }
17:        else
18:        {
19:            if (a[child] > a[child + 1])
20:                maxchild = child;
21:            else
22:                maxchild = child + 1;
23:        }
24:
25:        if (a[root] < a[maxchild])
26:        {
27:            temp = a[root];
28:            a[root] = a[maxchild];
29:            a[maxchild] = temp;
30:        }
31:        else return;
32:
33:        root = maxchild;
34:    }
```

```
35: }
36:
37: void heapsort(int a[], int array_size)
38: {
39:     int i;
40:     for (i = (array_size/2 - 1); i >= 0; --i)
41:     {
42:         downHeap(a, i, array_size-1);
43:     }
44:
45:     for (i = array_size-1; i >= 0; --i)
46:     {
47:         int temp;
48:         temp = a[i];
49:         a[i] = a[0];
50:         a[0] = temp;
51:         downHeap(a, 0, i-1);
52:     }
53: }
54:
55: int main(int argc, const char *argv[])
56: {
57:     int x[VETSIZE];
58:     int i;
59:
60:     srand(clock());
61:
62:     for (i = 0; i < VETSIZE; i++) {
63:         x[i] = (rand() % 2000);
64:     }
65:
66:     heapsort(x, VETSIZE);
67:
68:     return 0;
69: }
```

Código-fonte 17 – Código do algoritmo *Radix sort*

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <time.h>
4:
```

```
5: #define false 0
6: #define true 1
7: #define bool char
8:
9: #define VETSIZE 20000
10: #define BASE 10
11:
12: void radixsort(int *a, int n)
13: {
14:     int i, b[VETSIZE], m = a[0], exp = 1;
15:
16:     //Get the greatest value in the array a and assign it to m
17:     for (i = 1; i < n; i++)
18:     {
19:         if (a[i] > m)
20:             m = a[i];
21:     }
22:
23:     //Loop until exp is bigger than the largest number
24:     while (m / exp > 0)
25:     {
26:         int bucket[BASE] = { 0 };
27:
28:         //Count the number of keys that will go into each bucket
29:         for (i = 0; i < n; i++)
30:             bucket[(a[i] / exp) % BASE]++;
31:
32:         //Add the count of the previous buckets to acquire the
indexes after the end of each bucket location in the array
33:         for (i = 1; i < BASE; i++)
34:             bucket[i] += bucket[i - 1];
35:
36:         //Starting at the end of the list, get the index
corresponding to the a[i]'s key, decrement it, and use it to
place a[i] into array b.
37:         for (i = n - 1; i >= 0; i--)
38:             b[--bucket[(a[i] / exp) % BASE]] = a[i];
39:
40:         //Copy array b to array a
41:         for (i = 0; i < n; i++)
42:             a[i] = b[i];
43:
```

```
44:      //Multiply exp by the BASE to get the next group of keys
45:      exp *= BASE;
46:  }
47: }
48:
49: int main(int argc, const char *argv[])
50: {
51:     int x[VETSIZE];
52:     int i;
53:
54:     srand(clock());
55:
56:     for (i = 0; i < VETSIZE; i++) {
57:         x[i] = (rand() % 2000);
58:     }
59:
60:     radixsort(x, VETSIZE);
61:
62:     return 0;
63: }
```

Código-fonte 18 – Código do algoritmo *Selection sort*

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <time.h>
4:
5: #define VETSIZE 10000
6:
7: void SelectionSort(int a[], int array_size)
8: {
9:     int i;
10:    for (i = 0; i < array_size - 1; ++i)
11:    {
12:        int j, min, temp;
13:        min = i;
14:        for (j = i+1; j < array_size; ++j)
15:        {
16:            if (a[j] < a[min])
17:                min = j;
18:        }
19:
```

```
20:         temp = a[i];
21:         a[i] = a[min];
22:         a[min] = temp;
23:     }
24: }
25:
26: int main(int argc, const char *argv[])
27: {
28:     int x[VETSIZE];
29:     int i;
30:
31:     srand(clock());
32:
33:     for (i = 0; i < VETSIZE; i++) {
34:         x[i] = (rand() % 1000);
35:     }
36:
37:     SelectionSort(x, VETSIZE);
38:
39:     return 0;
40: }
```

Código-fonte 19 – Código do algoritmo *Insertion sort*

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <time.h>
4:
5: #define VETSIZE 20000
6:
7: void insertionSort(int a[], int array_size)
8: {
9:     int i, j, index;
10:    for (i = 1; i < array_size; ++i)
11:    {
12:        index = a[i];
13:        for (j = i; j > 0 && a[j-1] > index; j--)
14:            a[j] = a[j-1];
15:
16:        a[j] = index;
17:    }
18: }
```



```

19:
20: int main(int argc, const char *argv[])
21: {
22:     int x[VETSIZE];
23:     int i;
24:
25:     srand(clock());
26:
27:     for (i = 0; i < VETSIZE; i++) {
28:         x[i] = (rand() % 2000);
29:     }
30:
31:     insertionSort(x, VETSIZE);
32:
33:     return 0;
34: }

```

A.6 Códigos assembly “desmontados” dos programas

Código-fonte 20 – Código assembly do programa *Bubble sort*

Disassembly of section .text.startup:

```

0000000000000000 <main>:
    0:  bf 20 00 00 00      mov     $0x20,%edi
    5:  be 00 00 00 00      mov     $0x0,%esi
   a:  66 0f 1f 44 00 00    nopw    0x0(%rax,%rax,1)
  10:  b8 00 00 00 00      mov     $0x0,%eax
  15:  0f 1f 00             nopl     (%rax)
  18:  8b 08               mov     (%rax),%ecx
 1a:  8b 50 04             mov     0x4(%rax),%edx
 1d:  39 d1               cmp     %edx,%ecx
 1f:  7e 05               jle     26 <main+0x26>
 21:  89 48 04             mov     %ecx,0x4(%rax)
 24:  89 10               mov     %edx,(%rax)
 26:  48 83 c0 04          add     $0x4,%rax
 2a:  48 39 c6             cmp     %rax,%rsi
 2d:  75 e9               jne     18 <main+0x18>
 2f:  83 ef 01             sub     $0x1,%edi
 32:  75 dc               jne     10 <main+0x10>

```

```

34: 31 c0          xor    %eax,%eax
36: c3            retq

```

Código-fonte 21 – Código assembly do programa *Heap sort*

Disassembly of section .text:

0000000000000000 <downHeap>:

```

0: 44 8d 0c 36      lea    (%rsi,%rsi,1),%r9d
4: 44 39 ca         cmp    %r9d,%edx
7: 7e 79           jle    82 <downHeap+0x82>
9: 48 63 f6         movslq %esi,%rsi
c: 48 63 c2         movslq %edx,%rax
f: 55              push   %rbp
10: 44 8b 1c b7      mov    (%rdi,%rsi,4),%r11d
14: 53              push   %rbx
15: 48 8d 2c 87      lea    (%rdi,%rax,4),%rbp
19: eb 42           jmp    5d <downHeap+0x5d>
1b: 0f 1f 44 00 00   nopl   0x0(%rax,%rax,1)
20: 4c 63 c0         movslq %eax,%r8
23: 49 c1 e0 02      shl    $0x2,%r8
27: 4e 8d 14 07      lea    (%rdi,%r8,1),%r10
2b: 4e 8d 44 07 04   lea    0x4(%rdi,%r8,1),%r8
30: 41 8b 0a         mov    (%r10),%ecx
33: 41 8b 18         mov    (%r8),%ebx
36: 39 d9           cmp    %ebx,%ecx
38: 7f 09           jg     43 <downHeap+0x43>
3a: 41 8d 41 02      lea    0x2(%r9),%eax
3e: 89 d9           mov    %ebx,%ecx
40: 4d 89 c2         mov    %r8,%r10
43: 41 39 cb         cmp    %ecx,%r11d
46: 48 8d 34 b7      lea    (%rdi,%rsi,4),%rsi
4a: 7d 34           jge    80 <downHeap+0x80>
4c: 44 8d 0c 00      lea    (%rax,%rax,1),%r9d
50: 89 0e           mov    %ecx,(%rsi)
52: 45 89 1a         mov    %r11d,(%r10)
55: 41 39 d1         cmp    %edx,%r9d
58: 7d 26           jge    80 <downHeap+0x80>
5a: 48 63 f0         movslq %eax,%rsi
5d: 41 8d 41 01      lea    0x1(%r9),%eax
61: 39 c2           cmp    %eax,%edx

```

```

63: 75 bb                jne     20 <downHeap+0x20>
65: 8b 4d 00            mov     0x0(%rbp),%ecx
68: 49 89 ea            mov     %rbp,%r10
6b: 89 d0              mov     %edx,%eax
6d: 48 8d 34 b7         lea     (%rdi,%rsi,4),%rsi
71: 41 39 cb            cmp     %ecx,%r11d
74: 7c d6              jl      4c <downHeap+0x4c>
76: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
7d: 00 00 00
80: 5b                 pop     %rbx
81: 5d                 pop     %rbp
82: f3 c3              repz   retq
84: 66 90              xchg    %ax,%ax
86: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
8d: 00 00 00

0000000000000090 <heapsort>:
90: 41 56              push    %r14
92: 41 55              push    %r13
94: 44 8d 56 ff         lea     -0x1(%rsi),%r10d
98: 41 54              push    %r12
9a: 55                 push    %rbp
9b: 89 f5              mov     %esi,%ebp
9d: c1 ed 1f           shr     $0x1f,%ebp
a0: 53                 push    %rbx
a1: 01 f5              add     %esi,%ebp
a3: d1 fd              sar     %ebp
a5: 83 ed 01           sub     $0x1,%ebp
a8: 0f 88 a6 00 00 00   js      154 <heapsort+0xc4>
ae: 48 63 c5           movslq  %ebp,%rax
b1: 44 8d 64 2d 00      lea     0x0(%rbp,%rbp,1),%r12d
b6: 4c 8d 2c 87         lea     (%rdi,%rax,4),%r13
ba: 49 63 c2           movslq  %r10d,%rax
bd: 48 8d 1c 87         lea     (%rdi,%rax,4),%rbx
c1: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
c8: 45 39 d4           cmp     %r10d,%r12d
cb: 45 89 e0           mov     %r12d,%r8d
ce: 7d 70              jge     140 <heapsort+0xb0>
d0: 45 8b 5d 00         mov     0x0(%r13),%r11d
d4: 48 63 cd           movslq  %ebp,%rcx
d7: eb 46              jmp     11f <heapsort+0x8f>
d9: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)

```

```

e0:  48 63 f0          movslq %eax,%rsi
e3:  48 c1 e6 02       shl     $0x2,%rsi
e7:  4c 8d 0c 37       lea     (%rdi,%rsi,1),%r9
eb:  48 8d 74 37 04    lea     0x4(%rdi,%rsi,1),%rsi
f0:  41 8b 11          mov     (%r9),%edx
f3:  44 8b 36          mov     (%rsi),%r14d
f6:  44 39 f2          cmp     %r14d,%edx
f9:  7f 0a            jg      105 <heapsort+0x75>
fb:  41 8d 40 02       lea     0x2(%r8),%eax
ff:  44 89 f2          mov     %r14d,%edx
102: 49 89 f1          mov     %rsi,%r9
105: 41 39 d3          cmp     %edx,%r11d
108: 48 8d 0c 8f       lea     (%rdi,%rcx,4),%rcx
10c: 7d 32            jge     140 <heapsort+0xb0>
10e: 44 8d 04 00       lea     (%rax,%rax,1),%r8d
112: 89 11            mov     %edx,(%rcx)
114: 45 89 19          mov     %r11d,(%r9)
117: 45 39 d0          cmp     %r10d,%r8d
11a: 7d 24            jge     140 <heapsort+0xb0>
11c: 48 63 c8          movslq %eax,%rcx
11f: 41 8d 40 01       lea     0x1(%r8),%eax
123: 44 39 d0          cmp     %r10d,%eax
126: 75 b8            jne     e0 <heapsort+0x50>
128: 8b 13            mov     (%rbx),%edx
12a: 49 89 d9          mov     %rbx,%r9
12d: 44 89 d0          mov     %r10d,%eax
130: 48 8d 0c 8f       lea     (%rdi,%rcx,4),%rcx
134: 41 39 d3          cmp     %edx,%r11d
137: 7c d5            jl      10e <heapsort+0x7e>
139: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
140: 83 ed 01          sub     $0x1,%ebp
143: 41 83 ec 02       sub     $0x2,%r12d
147: 49 83 ed 04       sub     $0x4,%r13
14b: 83 fd ff          cmp     $0xffffffff,%ebp
14e: 0f 85 74 ff ff ff jne     c8 <heapsort+0x38>
154: 45 85 d2          test    %r10d,%r10d
157: 0f 88 a8 00 00 00 js      205 <heapsort+0x175>
15d: 49 63 d2          movslq %r10d,%rdx
160: 8b 07            mov     (%rdi),%eax
162: 48 8d 2c 97       lea     (%rdi,%rdx,4),%rbp
166: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
16d: 00 00 00

```

```

170: 44 8b 5d 00      mov     0x0(%rbp),%r11d
174: 41 83 ea 01      sub     $0x1,%r10d
178: 89 45 00         mov     %eax,0x0(%rbp)
17b: 41 83 fa 00      cmp     $0x0,%r10d
17f: 44 89 1f         mov     %r11d,(%rdi)
182: 7e 7f           jle     203 <heapsort+0x173>
184: 49 63 c2         movslq  %r10d,%rax
187: 31 c9           xor     %ecx,%ecx
189: 45 31 c0         xor     %r8d,%r8d
18c: 48 8d 1c 87      lea     (%rdi,%rax,4),%rbx
190: eb 45           jmp     1d7 <heapsort+0x147>
192: 66 0f 1f 44 00 00 nopw    0x0(%rax,%rax,1)
198: 48 63 f0         movslq  %eax,%rsi
19b: 48 c1 e6 02      shl     $0x2,%rsi
19f: 4c 8d 0c 37      lea     (%rdi,%rsi,1),%r9
1a3: 48 8d 74 37 04   lea     0x4(%rdi,%rsi,1),%rsi
1a8: 41 8b 11         mov     (%r9),%edx
1ab: 44 8b 26         mov     (%rsi),%r12d
1ae: 44 39 e2         cmp     %r12d,%edx
1b1: 7f 0a           jg      1bd <heapsort+0x12d>
1b3: 41 8d 40 02      lea     0x2(%r8),%eax
1b7: 44 89 e2         mov     %r12d,%edx
1ba: 49 89 f1         mov     %rsi,%r9
1bd: 41 39 d3         cmp     %edx,%r11d
1c0: 48 8d 0c 8f      lea     (%rdi,%rcx,4),%rcx
1c4: 7d 32           jge     1f8 <heapsort+0x168>
1c6: 44 8d 04 00      lea     (%rax,%rax,1),%r8d
1ca: 89 11           mov     %edx,(%rcx)
1cc: 45 89 19         mov     %r11d,(%r9)
1cf: 45 39 d0         cmp     %r10d,%r8d
1d2: 7d 24           jge     1f8 <heapsort+0x168>
1d4: 48 63 c8         movslq  %eax,%rcx
1d7: 41 8d 40 01      lea     0x1(%r8),%eax
1db: 41 39 c2         cmp     %eax,%r10d
1de: 75 b8           jne     198 <heapsort+0x108>
1e0: 8b 13           mov     (%rbx),%edx
1e2: 49 89 d9         mov     %rbx,%r9
1e5: 44 89 d0         mov     %r10d,%eax
1e8: 48 8d 0c 8f      lea     (%rdi,%rcx,4),%rcx
1ec: 41 39 d3         cmp     %edx,%r11d
1ef: 7c d5           jl      1c6 <heapsort+0x136>
1f1: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)

```

```

1f8: 8b 07          mov    (%rdi),%eax
1fa: 48 83 ed 04    sub    $0x4,%rbp
1fe: e9 6d ff ff ff jmpq   170 <heapsort+0xe0>
203: 74 09          je     20e <heapsort+0x17e>
205: 5b            pop    %rbx
206: 5d            pop    %rbp
207: 41 5c          pop    %r12
209: 41 5d          pop    %r13
20b: 41 5e          pop    %r14
20d: c3            retq
20e: 44 89 d8       mov    %r11d,%eax
211: eb e7          jmp    1fa <heapsort+0x16a>

```

Disassembly of section .text.startup:

0000000000000000 <main>:

```

0: 41 54          push   %r12
2: 55            push   %rbp
3: 53            push   %rbx
4: bb d3 4d 62 10 mov    $0x10624dd3,%ebx
9: 48 81 ec 90 38 01 00 sub    $0x13890,%rsp
10: 48 8d ac 24 80 38 01 lea    0x13880(%rsp),%rbp
17: 00
18: 49 89 e4       mov    %rsp,%r12
1b: 64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
22: 00 00
24: 48 89 84 24 88 38 01 mov    %rax,0x13888(%rsp)
2b: 00
2c: 31 c0          xor     %eax,%eax
2e: e8 00 00 00 00 callq   33 <main+0x33>
33: 89 c7          mov     %eax,%edi
35: e8 00 00 00 00 callq   3a <main+0x3a>
3a: 66 0f 1f 44 00 00 nopw    0x0(%rax,%rax,1)
40: e8 00 00 00 00 callq   45 <main+0x45>
45: 89 c1          mov     %eax,%ecx
47: 49 83 c4 04    add     $0x4,%r12
4b: f7 eb          imul    %ebx
4d: 89 c8          mov     %ecx,%eax
4f: c1 f8 1f       sar     $0x1f,%eax
52: c1 fa 07       sar     $0x7,%edx
55: 29 c2          sub     %eax,%edx
57: 69 d2 d0 07 00 00 imul    $0x7d0,%edx,%edx

```

```

5d: 29 d1          sub    %edx,%ecx
5f: 41 89 4c 24 fc  mov    %ecx,-0x4(%r12)
64: 49 39 ec        cmp    %rbp,%r12
67: 75 d7          jne    40 <main+0x40>
69: be 20 4e 00 00  mov    $0x4e20,%esi
6e: 48 89 e7        mov    %rsp,%rdi
71: e8 00 00 00 00  callq  76 <main+0x76>
76: 31 c0          xor    %eax,%eax
78: 48 8b b4 24 88 38 01  mov    0x13888(%rsp),%rsi
7f: 00
80: 64 48 33 34 25 28 00  xor    %fs:0x28,%rsi
87: 00 00
89: 75 0c          jne    97 <main+0x97>
8b: 48 81 c4 90 38 01 00  add    $0x13890,%rsp
92: 5b            pop    %rbx
93: 5d            pop    %rbp
94: 41 5c          pop    %r12
96: c3            retq
97: e8 00 00 00 00  callq  9c <heapsort+0xc>

```

Código-fonte 22 – Código assembly do programa *Insertion sort*

Disassembly of section .text:

```

0000000000000000 <insertionSort>:
 0: 83 fe 01        cmp    $0x1,%esi
 3: 4c 8d 4f 04      lea    0x4(%rdi),%r9
 7: 41 bb 04 00 00 00  mov    $0x4,%r11d
 d: 41 ba 01 00 00 00  mov    $0x1,%r10d
13: 7e 4f          jle    64 <insertionSort+0x64>
15: 0f 1f 00        nopl   (%rax)
18: 45 85 d2        test   %r10d,%r10d
1b: 4c 89 c8        mov    %r9,%rax
1e: 45 8b 01        mov    (%r9),%r8d
21: 7e 2d          jle    50 <insertionSort+0x50>
23: 41 8b 51 fc      mov    -0x4(%r9),%edx
27: 41 39 d0        cmp    %edx,%r8d
2a: 7d 24          jge    50 <insertionSort+0x50>
2c: 4c 89 d8        mov    %r11,%rax
2f: 44 89 d1        mov    %r10d,%ecx
32: eb 11          jmp    45 <insertionSort+0x45>

```

```

34: 0f 1f 40 00      nopl    0x0(%rax)
38: 48 83 e8 04      sub     $0x4,%rax
3c: 8b 54 07 fc      mov     -0x4(%rdi,%rax,1),%edx
40: 41 39 d0          cmp     %edx,%r8d
43: 7d 2b            jge     70 <insertionSort+0x70>
45: 83 e9 01          sub     $0x1,%ecx
48: 89 14 07          mov     %edx,(%rdi,%rax,1)
4b: 75 eb            jne     38 <insertionSort+0x38>
4d: 48 89 f8          mov     %rdi,%rax
50: 41 83 c2 01      add     $0x1,%r10d
54: 49 83 c1 04      add     $0x4,%r9
58: 49 83 c3 04      add     $0x4,%r11
5c: 44 39 d6          cmp     %r10d,%esi
5f: 44 89 00          mov     %r8d,(%rax)
62: 75 b4            jne     18 <insertionSort+0x18>
64: f3 c3            repz    retq
66: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
6d: 00 00 00
70: 48 01 f8          add     %rdi,%rax
73: eb db            jmp     50 <insertionSort+0x50>

```

Disassembly of section .text.startup:

0000000000000000 <main>:

```

0: 41 54            push    %r12
2: 55              push    %rbp
3: 53              push    %rbx
4: bb d3 4d 62 10   mov     $0x10624dd3,%ebx
9: 48 81 ec 90 38 01 00 sub     $0x13890,%rsp
10: 48 8d ac 24 80 38 01 lea     0x13880(%rsp),%rbp
17: 00
18: 49 89 e4          mov     %rsp,%r12
1b: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
22: 00 00
24: 48 89 84 24 88 38 01 mov     %rax,0x13888(%rsp)
2b: 00
2c: 31 c0            xor     %eax,%eax
2e: e8 00 00 00 00   callq   33 <main+0x33>
33: 89 c7            mov     %eax,%edi
35: e8 00 00 00 00   callq   3a <main+0x3a>
3a: 66 0f 1f 44 00 00 nopw    0x0(%rax,%rax,1)
40: e8 00 00 00 00   callq   45 <main+0x45>

```



```

45: 89 c1          mov    %eax,%ecx
47: 49 83 c4 04    add    $0x4,%r12
4b: f7 eb          imul   %ebx
4d: 89 c8          mov    %ecx,%eax
4f: c1 f8 1f      sar    $0x1f,%eax
52: c1 fa 07      sar    $0x7,%edx
55: 29 c2          sub    %eax,%edx
57: 69 d2 d0 07 00 00 imul   $0x7d0,%edx,%edx
5d: 29 d1          sub    %edx,%ecx
5f: 41 89 4c 24 fc  mov    %ecx,-0x4(%r12)
64: 49 39 ec          cmp    %rbp,%r12
67: 75 d7          jne    40 <main+0x40>
69: be 20 4e 00 00  mov    $0x4e20,%esi
6e: 48 89 e7          mov    %rsp,%rdi
71: e8 00 00 00 00  callq  76 <main+0x76>
76: 31 c0          xor    %eax,%eax
78: 48 8b b4 24 88 38 01 mov    0x13888(%rsp),%rsi
7f: 00
80: 64 48 33 34 25 28 00 xor    %fs:0x28,%rsi
87: 00 00
89: 75 0c          jne    97 <main+0x97>
8b: 48 81 c4 90 38 01 00 add    $0x13890,%rsp
92: 5b            pop    %rbx
93: 5d            pop    %rbp
94: 41 5c          pop    %r12
96: c3            retq
97: e8 00 00 00 00  callq  9c <main+0x9c>

```

Código-fonte 23 – Código assembly do programa *Radix sort*

Disassembly of section .text:

0000000000000000 <radixsort>:

```

0: 41 57          push   %r15
2: 41 56          push   %r14
4: 48 89 f9      mov    %rdi,%rcx
7: 41 55          push   %r13
9: 41 54          push   %r12
b: 41 89 f6      mov    %esi,%r14d
e: 55            push   %rbp
f: 53            push   %rbx

```

```

10: 48 81 ec d8 38 01 00    sub    $0x138d8,%rsp
17: 44 8b 2f                mov    (%rdi),%r13d
1a: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
21: 00 00
23: 48 89 84 24 c8 38 01    mov    %rax,0x138c8(%rsp)
2a: 00
2b: 31 c0                xor    %eax,%eax
2d: 83 fe 01                cmp    $0x1,%esi
30: 0f 8e a4 01 00 00      jle    1da <radixsort+0x1da>
36: 48 8d 47 04            lea    0x4(%rdi),%rax
3a: 8d 76 ff                lea    -0x1(%rsi),%esi
3d: 83 e0 0f                and    $0xf,%eax
40: 48 c1 e8 02            shr    $0x2,%rax
44: 48 f7 d8                neg    %rax
47: 83 e0 03                and    $0x3,%eax
4a: 39 f0                cmp    %esi,%eax
4c: 0f 47 c6                cmova  %esi,%eax
4f: 83 fe 08                cmp    $0x8,%esi
52: 0f 87 69 03 00 00      ja     3c1 <radixsort+0x3c1>
58: 89 f0                mov    %esi,%eax
5a: 8b 51 04                mov    0x4(%rcx),%edx
5d: 41 39 d5                cmp    %edx,%r13d
60: 44 0f 4c ea            cmovl  %edx,%r13d
64: 83 f8 01                cmp    $0x1,%eax
67: 0f 84 70 03 00 00      je     3dd <radixsort+0x3dd>
6d: 8b 51 08                mov    0x8(%rcx),%edx
70: 41 39 d5                cmp    %edx,%r13d
73: 44 0f 4c ea            cmovl  %edx,%r13d
77: 83 f8 02                cmp    $0x2,%eax
7a: 0f 84 67 03 00 00      je     3e7 <radixsort+0x3e7>
80: 8b 51 0c                mov    0xc(%rcx),%edx
83: 41 39 d5                cmp    %edx,%r13d
86: 44 0f 4c ea            cmovl  %edx,%r13d
8a: 83 f8 03                cmp    $0x3,%eax
8d: 0f 84 5e 03 00 00      je     3f1 <radixsort+0x3f1>
93: 8b 51 10                mov    0x10(%rcx),%edx
96: 41 39 d5                cmp    %edx,%r13d
99: 44 0f 4c ea            cmovl  %edx,%r13d
9d: 83 f8 04                cmp    $0x4,%eax
a0: 0f 84 55 03 00 00      je     3fb <radixsort+0x3fb>
a6: 8b 51 14                mov    0x14(%rcx),%edx
a9: 41 39 d5                cmp    %edx,%r13d

```

```

ac:  44 0f 4c ea          cmovl  %edx,%r13d
b0:  83 f8 05            cmp    $0x5,%eax
b3:  0f 84 4c 03 00 00    je     405 <radixsort+0x405>
b9:  8b 51 18              mov     0x18(%rcx),%edx
bc:  41 39 d5              cmp     %edx,%r13d
bf:  44 0f 4c ea          cmovl  %edx,%r13d
c3:  83 f8 06              cmp     $0x6,%eax
c6:  0f 84 43 03 00 00    je     40f <radixsort+0x40f>
cc:  8b 51 1c              mov     0x1c(%rcx),%edx
cf:  41 39 d5              cmp     %edx,%r13d
d2:  44 0f 4c ea          cmovl  %edx,%r13d
d6:  83 f8 08              cmp     $0x8,%eax
d9:  0f 85 f4 02 00 00    jne    3d3 <radixsort+0x3d3>
df:  8b 51 20              mov     0x20(%rcx),%edx
e2:  41 39 d5              cmp     %edx,%r13d
e5:  44 0f 4c ea          cmovl  %edx,%r13d
e9:  ba 09 00 00 00        mov     $0x9,%edx
ee:  39 c6                 cmp     %eax,%esi
f0:  0f 84 e4 00 00 00    je     1da <radixsort+0x1da>
f6:  29 c6                 sub     %eax,%esi
f8:  45 8d 46 fe          lea     -0x2(%r14),%r8d
fc:  41 89 c2              mov     %eax,%r10d
ff:  8d 7e fc             lea     -0x4(%rsi),%edi
102:  41 29 c0              sub     %eax,%r8d
105:  c1 ef 02              shr     $0x2,%edi
108:  83 c7 01              add     $0x1,%edi
10b:  41 83 f8 02           cmp     $0x2,%r8d
10f:  44 8d 0c bd 00 00 00  lea     0x0(,%rdi,4),%r9d
116:  00
117:  0f 86 88 00 00 00    jbe    1a5 <radixsort+0x1a5>
11d:  44 89 2c 24           mov     %r13d,(%rsp)
121:  4e 8d 44 91 04        lea     0x4(%rcx,%r10,4),%r8
126:  31 c0                 xor     %eax,%eax
128:  66 0f 6e 1c 24        movd    (%rsp),%xmm3
12d:  66 0f 70 c3 00        pshufd  $0x0,%xmm3,%xmm0
132:  66 41 0f 6f 10        movdqa  (%r8),%xmm2
137:  66 0f 6f c8           movdqa  %xmm0,%xmm1
13b:  83 c0 01              add     $0x1,%eax
13e:  49 83 c0 10           add     $0x10,%r8
142:  66 0f 66 ca           pcmptgd %xmm2,%xmm1
146:  39 f8                 cmp     %edi,%eax
148:  66 0f db c1           pand    %xmm1,%xmm0

```

14c:	66 0f df ca	pandn	%xmm2,%xmm1
150:	66 0f eb c1	por	%xmm1,%xmm0
154:	72 dc	jb	132 <radixsort+0x132>
156:	66 0f 6f c8	movdqa	%xmm0,%xmm1
15a:	44 01 ca	add	%r9d,%edx
15d:	41 39 f1	cmp	%esi,%r9d
160:	66 0f 73 d9 08	psrldq	\$0x8,%xmm1
165:	66 0f 6f d1	movdqa	%xmm1,%xmm2
169:	66 0f 66 d0	pcmpgtd	%xmm0,%xmm2
16d:	66 0f db ca	pand	%xmm2,%xmm1
171:	66 0f df d0	pandn	%xmm0,%xmm2
175:	66 0f 6f c2	movdqa	%xmm2,%xmm0
179:	66 0f eb c1	por	%xmm1,%xmm0
17d:	66 0f 6f c8	movdqa	%xmm0,%xmm1
181:	66 0f 73 d9 04	psrldq	\$0x4,%xmm1
186:	66 0f 6f d1	movdqa	%xmm1,%xmm2
18a:	66 0f 66 d0	pcmpgtd	%xmm0,%xmm2
18e:	66 0f db ca	pand	%xmm2,%xmm1
192:	66 0f df d0	pandn	%xmm0,%xmm2
196:	66 0f 6f c2	movdqa	%xmm2,%xmm0
19a:	66 0f eb c1	por	%xmm1,%xmm0
19e:	66 41 0f 7e c5	movd	%xmm0,%r13d
1a3:	74 35	je	1da <radixsort+0x1da>
1a5:	48 63 c2	movslq	%edx,%rax
1a8:	8b 04 81	mov	(%rcx,%rax,4),%eax
1ab:	41 39 c5	cmp	%eax,%r13d
1ae:	44 0f 4c e8	cmovl	%eax,%r13d
1b2:	8d 42 01	lea	0x1(%rdx),%eax
1b5:	41 39 c6	cmp	%eax,%r14d
1b8:	7e 20	jle	1da <radixsort+0x1da>
1ba:	48 98	cltq	
1bc:	8b 04 81	mov	(%rcx,%rax,4),%eax
1bf:	41 39 c5	cmp	%eax,%r13d
1c2:	44 0f 4c e8	cmovl	%eax,%r13d
1c6:	8d 42 02	lea	0x2(%rdx),%eax
1c9:	41 39 c6	cmp	%eax,%r14d
1cc:	7e 0c	jle	1da <radixsort+0x1da>
1ce:	48 98	cltq	
1d0:	8b 04 81	mov	(%rcx,%rax,4),%eax
1d3:	41 39 c5	cmp	%eax,%r13d
1d6:	44 0f 4c e8	cmovl	%eax,%r13d
1da:	45 85 ed	test	%r13d,%r13d

```

1dd: 0f 8e 99 01 00 00      jle     37c <radixsort+0x37c>
1e3: 41 8d 56 ff            lea     -0x1(%r14),%edx
1e7: 48 8d 1c 95 04 00 00    lea     0x4(,%rdx,4),%rbx
1ee: 00
1ef: 48 89 d0              mov     %rdx,%rax
1f2: 48 98                cltq
1f4: 48 89 5c 24 08        mov     %rbx,0x8(%rsp)
1f9: 4c 8d 24 19          lea     (%rcx,%rbx,1),%r12
1fd: 48 8d 1c 81          lea     (%rcx,%rax,4),%rbx
201: 48 29 d0            sub     %rdx,%rax
204: 48 8d 6c 81 fc        lea     -0x4(%rcx,%rax,4),%rbp
209: 48 89 1c 24          mov     %rbx,(%rsp)
20d: bb 01 00 00 00        mov     $0x1,%ebx
212: 66 0f 1f 44 00 00      nopw    0x0(%rax,%rax,1)
218: 45 85 f6            test    %r14d,%r14d
21b: 48 c7 44 24 10 00 00    movq    $0x0,0x10(%rsp)
222: 00 00
224: 48 c7 44 24 18 00 00    movq    $0x0,0x18(%rsp)
22b: 00 00
22d: 48 c7 44 24 20 00 00    movq    $0x0,0x20(%rsp)
234: 00 00
236: 48 c7 44 24 28 00 00    movq    $0x0,0x28(%rsp)
23d: 00 00
23f: 48 c7 44 24 30 00 00    movq    $0x0,0x30(%rsp)
246: 00 00
248: 0f 8e 57 01 00 00      jle     3a5 <radixsort+0x3a5>
24e: 48 89 ce            mov     %rcx,%rsi
251: bf 67 66 66 66        mov     $0x66666667,%edi
256: 66 2e 0f 1f 84 00 00    nopw    %cs:0x0(%rax,%rax,1)
25d: 00 00 00
260: 8b 06            mov     (%rsi),%eax
262: 48 83 c6 04        add     $0x4,%rsi
266: 99                cltd
267: f7 fb            idiv    %ebx
269: 41 89 c1            mov     %eax,%r9d
26c: f7 ef            imul    %edi
26e: 44 89 c8            mov     %r9d,%eax
271: c1 f8 1f          sar     $0x1f,%eax
274: c1 fa 02          sar     $0x2,%edx
277: 29 c2            sub     %eax,%edx
279: 44 8d 04 92        lea     (%rdx,%rdx,4),%r8d
27d: 44 89 ca            mov     %r9d,%edx

```

280:	45 01 c0	add	%r8d,%r8d
283:	44 29 c2	sub	%r8d,%edx
286:	48 63 d2	movslq	%edx,%rdx
289:	83 44 94 10 01	addl	\$0x1,0x10(%rsp,%rdx,4)
28e:	4c 39 e6	cmp	%r12,%rsi
291:	75 cd	jne	260 <radixsort+0x260>
293:	8b 44 24 10	mov	0x10(%rsp),%eax
297:	03 44 24 14	add	0x14(%rsp),%eax
29b:	44 8b 5c 24 18	mov	0x18(%rsp),%r11d
2a0:	44 8b 54 24 1c	mov	0x1c(%rsp),%r10d
2a5:	44 8b 4c 24 20	mov	0x20(%rsp),%r9d
2aa:	44 8b 44 24 24	mov	0x24(%rsp),%r8d
2af:	8b 7c 24 28	mov	0x28(%rsp),%edi
2b3:	8b 74 24 2c	mov	0x2c(%rsp),%esi
2b7:	41 01 c3	add	%eax,%r11d
2ba:	8b 54 24 30	mov	0x30(%rsp),%edx
2be:	44 8b 7c 24 34	mov	0x34(%rsp),%r15d
2c3:	45 01 da	add	%r11d,%r10d
2c6:	45 01 d1	add	%r10d,%r9d
2c9:	45 01 c8	add	%r9d,%r8d
2cc:	44 01 c7	add	%r8d,%edi
2cf:	01 fe	add	%edi,%esi
2d1:	01 f2	add	%esi,%edx
2d3:	41 01 d7	add	%edx,%r15d
2d6:	45 85 f6	test	%r14d,%r14d
2d9:	44 89 44 24 24	mov	%r8d,0x24(%rsp)
2de:	89 74 24 2c	mov	%esi,0x2c(%rsp)
2e2:	89 44 24 14	mov	%eax,0x14(%rsp)
2e6:	44 89 5c 24 18	mov	%r11d,0x18(%rsp)
2eb:	41 b8 67 66 66 66	mov	\$0x66666667,%r8d
2f1:	44 89 54 24 1c	mov	%r10d,0x1c(%rsp)
2f6:	44 89 4c 24 20	mov	%r9d,0x20(%rsp)
2fb:	89 7c 24 28	mov	%edi,0x28(%rsp)
2ff:	89 54 24 30	mov	%edx,0x30(%rsp)
303:	44 89 7c 24 34	mov	%r15d,0x34(%rsp)
308:	48 8b 34 24	mov	(%rsp),%rsi
30c:	7e 5b	jle	369 <radixsort+0x369>
30e:	66 90	xchg	%ax,%ax
310:	8b 3e	mov	(%rsi),%edi
312:	48 83 ee 04	sub	\$0x4,%rsi
316:	89 f8	mov	%edi,%eax
318:	99	cld	

```

319:  f7 fb          idiv    %ebx
31b:  41 89 c1        mov     %eax,%r9d
31e:  41 f7 e8        imul    %r8d
321:  44 89 c8        mov     %r9d,%eax
324:  c1 f8 1f        sar     $0x1f,%eax
327:  c1 fa 02        sar     $0x2,%edx
32a:  29 c2          sub     %eax,%edx
32c:  8d 04 92        lea     (%rdx,%rdx,4),%eax
32f:  44 89 ca        mov     %r9d,%edx
332:  01 c0          add     %eax,%eax
334:  29 c2          sub     %eax,%edx
336:  48 63 d2        movslq  %edx,%rdx
339:  8b 44 94 10     mov     0x10(%rsp,%rdx,4),%eax
33d:  83 e8 01        sub     $0x1,%eax
340:  48 39 ee        cmp     %rbp,%rsi
343:  89 44 94 10     mov     %eax,0x10(%rsp,%rdx,4)
347:  48 98          cltq
349:  89 7c 84 40     mov     %edi,0x40(%rsp,%rax,4)
34d:  75 c1          jne     310 <radixsort+0x310>
34f:  45 85 f6        test    %r14d,%r14d
352:  7e 15          jle     369 <radixsort+0x369>
354:  48 8b 54 24 08  mov     0x8(%rsp),%rdx
359:  48 8d 74 24 40  lea     0x40(%rsp),%rsi
35e:  48 89 cf        mov     %rcx,%rdi
361:  e8 00 00 00 00  callq   366 <radixsort+0x366>
366:  48 89 c1        mov     %rax,%rcx
369:  8d 1c 9b        lea     (%rbx,%rbx,4),%ebx
36c:  44 89 e8        mov     %r13d,%eax
36f:  99            cltd
370:  01 db          add     %ebx,%ebx
372:  f7 fb          idiv    %ebx
374:  85 c0          test    %eax,%eax
376:  0f 8f 9c fe ff ff  jg      218 <radixsort+0x218>
37c:  48 8b 84 24 c8 38 01  mov     0x138c8(%rsp),%rax
383:  00
384:  64 48 33 04 25 28 00  xor     %fs:0x28,%rax
38b:  00 00
38d:  0f 85 86 00 00 00  jne     419 <radixsort+0x419>
393:  48 81 c4 d8 38 01 00  add     $0x138d8,%rsp
39a:  5b            pop     %rbx
39b:  5d            pop     %rbp
39c:  41 5c          pop     %r12

```

```

39e: 41 5d                pop    %r13
3a0: 41 5e                pop    %r14
3a2: 41 5f                pop    %r15
3a4: c3                  retq
3a5: 45 31 ff            xor     %r15d,%r15d
3a8: 31 d2                xor     %edx,%edx
3aa: 31 f6                xor     %esi,%esi
3ac: 31 ff                xor     %edi,%edi
3ae: 45 31 c0            xor     %r8d,%r8d
3b1: 45 31 c9            xor     %r9d,%r9d
3b4: 45 31 d2            xor     %r10d,%r10d
3b7: 45 31 db            xor     %r11d,%r11d
3ba: 31 c0                xor     %eax,%eax
3bc: e9 15 ff ff ff      jmpq    2d6 <radixsort+0x2d6>
3c1: 85 c0                test    %eax,%eax
3c3: 0f 85 91 fc ff ff   jne     5a <radixsort+0x5a>
3c9: ba 01 00 00 00      mov     $0x1,%edx
3ce: e9 23 fd ff ff      jmpq    f6 <radixsort+0xf6>
3d3: ba 08 00 00 00      mov     $0x8,%edx
3d8: e9 11 fd ff ff      jmpq    ee <radixsort+0xee>
3dd: ba 02 00 00 00      mov     $0x2,%edx
3e2: e9 07 fd ff ff      jmpq    ee <radixsort+0xee>
3e7: ba 03 00 00 00      mov     $0x3,%edx
3ec: e9 fd fc ff ff      jmpq    ee <radixsort+0xee>
3f1: ba 04 00 00 00      mov     $0x4,%edx
3f6: e9 f3 fc ff ff      jmpq    ee <radixsort+0xee>
3fb: ba 05 00 00 00      mov     $0x5,%edx
400: e9 e9 fc ff ff      jmpq    ee <radixsort+0xee>
405: ba 06 00 00 00      mov     $0x6,%edx
40a: e9 df fc ff ff      jmpq    ee <radixsort+0xee>
40f: ba 07 00 00 00      mov     $0x7,%edx
414: e9 d5 fc ff ff      jmpq    ee <radixsort+0xee>
419: e8 00 00 00 00      callq   41e <radixsort+0x41e>

```

Disassembly of section .text.startup:

0000000000000000 <main>:

```

0: 41 54                push    %r12
2: 55                  push    %rbp
3: 53                  push    %rbx
4: bb d3 4d 62 10      mov     $0x10624dd3,%ebx
9: 48 81 ec 90 38 01 00 sub     $0x13890,%rsp

```

```

10: 48 8d ac 24 80 38 01    lea     0x13880(%rsp),%rbp
17: 00
18: 49 89 e4                mov     %rsp,%r12
1b: 64 48 8b 04 25 28 00    mov     %fs:0x28,%rax
22: 00 00
24: 48 89 84 24 88 38 01    mov     %rax,0x13888(%rsp)
2b: 00
2c: 31 c0                  xor     %eax,%eax
2e: e8 00 00 00 00          callq   33 <main+0x33>
33: 89 c7                  mov     %eax,%edi
35: e8 00 00 00 00          callq   3a <main+0x3a>
3a: 66 0f 1f 44 00 00       nopw    0x0(%rax,%rax,1)
40: e8 00 00 00 00          callq   45 <main+0x45>
45: 89 c1                  mov     %eax,%ecx
47: 49 83 c4 04             add     $0x4,%r12
4b: f7 eb                  imul    %ebx
4d: 89 c8                  mov     %ecx,%eax
4f: c1 f8 1f               sar     $0x1f,%eax
52: c1 fa 07               sar     $0x7,%edx
55: 29 c2                  sub     %eax,%edx
57: 69 d2 d0 07 00 00       imul    $0x7d0,%edx,%edx
5d: 29 d1                  sub     %edx,%ecx
5f: 41 89 4c 24 fc          mov     %ecx,-0x4(%r12)
64: 49 39 ec                cmp     %rbp,%r12
67: 75 d7                  jne     40 <main+0x40>
69: be 20 4e 00 00          mov     $0x4e20,%esi
6e: 48 89 e7                mov     %rsp,%rdi
71: e8 00 00 00 00          callq   76 <main+0x76>
76: 31 c0                  xor     %eax,%eax
78: 48 8b b4 24 88 38 01    mov     0x13888(%rsp),%rsi
7f: 00
80: 64 48 33 34 25 28 00    xor     %fs:0x28,%rsi
87: 00 00
89: 75 0c                  jne     97 <main+0x97>
8b: 48 81 c4 90 38 01 00    add     $0x13890,%rsp
92: 5b                      pop     %rbx
93: 5d                      pop     %rbp
94: 41 5c                  pop     %r12
96: c3                      retq
97: e8 00 00 00 00          callq   9c <main+0x9c>

```

Código-fonte 24 – Código assembly do programa *Selection sort*

Disassembly of section .text:

```

0000000000000000 <SelectionSort>:
  0:  83 fe 01                cmp     $0x1,%esi
  3:  7e 66                  jle     6b <SelectionSort+0x6b>
  5:  41 55                  push    %r13
  7:  44 8d 6e ff            lea     -0x1(%rsi),%r13d
  b:  41 54                  push    %r12
  d:  55                    push    %rbp
  e:  53                    push    %rbx
  f:  49 89 fb              mov     %rdi,%r11
 12:  45 31 c9              xor     %r9d,%r9d
 15:  0f 1f 00             nopl    (%rax)
 18:  41 8d 59 01          lea     0x1(%r9),%ebx
 1c:  39 de                cmp     %ebx,%esi
 1e:  7e 4d                jle     6d <SelectionSort+0x6d>
 20:  45 8b 23              mov     (%r11),%r12d
 23:  49 8d 6b 04          lea     0x4(%r11),%rbp
 27:  89 d9                mov     %ebx,%ecx
 29:  49 89 e8              mov     %rbp,%r8
 2c:  44 89 e0              mov     %r12d,%eax
 2f:  90                    nop
 30:  41 8b 10              mov     (%r8),%edx
 33:  4d 63 d1             movslq  %r9d,%r10
 36:  4e 8d 14 97          lea     (%rdi,%r10,4),%r10
 3a:  39 c2                cmp     %eax,%edx
 3c:  44 0f 4c c9          cmovl   %ecx,%r9d
 40:  4d 0f 4c d0          cmovl   %r8,%r10
 44:  39 d0                cmp     %edx,%eax
 46:  0f 4f c2             cmovg   %edx,%eax
 49:  83 c1 01             add     $0x1,%ecx
 4c:  49 83 c0 04          add     $0x4,%r8
 50:  39 ce                cmp     %ecx,%esi
 52:  75 dc                jne     30 <SelectionSort+0x30>
 54:  41 39 dd              cmp     %ebx,%r13d
 57:  41 89 03              mov     %eax,(%r11)
 5a:  41 89 d9              mov     %ebx,%r9d
 5d:  45 89 22              mov     %r12d,(%r10)
 60:  49 89 eb              mov     %rbp,%r11

```

```

63: 75 b3                jne     18 <SelectionSort+0x18>
65: 5b                  pop     %rbx
66: 5d                  pop     %rbp
67: 41 5c                pop     %r12
69: 41 5d                pop     %r13
6b: f3 c3                repz    retq
6d: 41 8b 03             mov     (%r11),%eax
70: 4d 89 da             mov     %r11,%r10
73: 49 8d 6b 04          lea     0x4(%r11),%rbp
77: 41 89 c4             mov     %eax,%r12d
7a: eb d8                jmp     54 <SelectionSort+0x54>

```

Disassembly of section .text.startup:

0000000000000000 <main>:

```

0: 41 54                push    %r12
2: 55                  push    %rbp
3: 53                  push    %rbx
4: 48 81 ec 50 9c 00 00 sub     $0x9c50,%rsp
b: 48 8d ac 24 40 9c 00 lea     0x9c40(%rsp),%rbp
12: 00
13: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
1a: 00 00
1c: 48 89 84 24 48 9c 00 mov     %rax,0x9c48(%rsp)
23: 00
24: 31 c0                xor     %eax,%eax
26: e8 00 00 00 00       callq   2b <main+0x2b>
2b: 89 c7                mov     %eax,%edi
2d: e8 00 00 00 00       callq   32 <main+0x32>
32: 49 89 e4             mov     %rsp,%r12
35: bb d3 4d 62 10       mov     $0x10624dd3,%ebx
3a: 66 0f 1f 44 00 00    nopw    0x0(%rax,%rax,1)
40: e8 00 00 00 00       callq   45 <main+0x45>
45: 89 c1                mov     %eax,%ecx
47: 49 83 c4 04          add     $0x4,%r12
4b: f7 eb                imul    %ebx
4d: 89 c8                mov     %ecx,%eax
4f: c1 f8 1f             sar     $0x1f,%eax
52: c1 fa 06             sar     $0x6,%edx
55: 29 c2                sub     %eax,%edx
57: 69 d2 e8 03 00 00    imul    $0x3e8,%edx,%edx
5d: 29 d1                sub     %edx,%ecx

```

```

5f: 41 89 4c 24 fc      mov     %ecx,-0x4(%r12)
64: 4c 39 e5            cmp     %r12,%rbp
67: 75 d7              jne     40 <main+0x40>
69: 4c 8d 4c 24 04      lea     0x4(%rsp),%r9
6e: 31 ff              xor     %edi,%edi
70: 45 8b 59 fc        mov     -0x4(%r9),%r11d
74: 44 8d 57 01        lea     0x1(%rdi),%r10d
78: 4c 89 ce            mov     %r9,%rsi
7b: 44 89 d1            mov     %r10d,%ecx
7e: 44 89 d8            mov     %r11d,%eax
81: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
88: 8b 16              mov     (%rsi),%edx
8a: 4c 63 c7            movslq  %edi,%r8
8d: 4e 8d 04 84        lea     (%rsp,%r8,4),%r8
91: 39 c2              cmp     %eax,%edx
93: 0f 4c f9            cmovl   %ecx,%edi
96: 4c 0f 4c c6        cmovl   %rsi,%r8
9a: 39 d0              cmp     %edx,%eax
9c: 0f 4f c2            cmovg   %edx,%eax
9f: 83 c1 01            add     $0x1,%ecx
a2: 48 83 c6 04        add     $0x4,%rsi
a6: 81 f9 10 27 00 00   cmp     $0x2710,%ecx
ac: 75 da              jne     88 <main+0x88>
ae: 41 89 41 fc        mov     %eax,-0x4(%r9)
b2: 49 83 c1 04        add     $0x4,%r9
b6: 41 81 fa 0f 27 00 00 cmp     $0x270f,%r10d
bd: 45 89 18            mov     %r11d,(%r8)
c0: 44 89 d7            mov     %r10d,%edi
c3: 75 ab              jne     70 <main+0x70>
c5: e9 68 ff ff ff     jmpq    32 <main+0x32>

```

Código-fonte 25 – Código assembly do programa *Simpson* $\frac{1}{3}$

Disassembly of section .text:

```

0000000000000000 <f_clapeyron>:
 0: 48 83 ec 58        sub     $0x58,%rsp
 4: f2 0f 10 0d 00 00 00 movsd   0x0(%rip),%xmm1      # c
    <f_clapeyron+0xc>
 b: 00
 c: f2 0f 10 15 00 00 00 movsd   0x0(%rip),%xmm2      #

```

```

14 <f_clapeyron+0x14>
13: 00
14: f2 0f 10 1d 00 00 00 movsd 0x0(%rip),%xmm3      # 1
   c <f_clapeyron+0x1c>
1b: 00
1c: f2 0f 10 25 00 00 00 movsd 0x0(%rip),%xmm4      #
   24 <f_clapeyron+0x24>
23: 00
24: 64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
2b: 00 00
2d: 48 89 44 24 48      mov    %rax,0x48(%rsp)
32: 31 c0              xor    %eax,%eax
34: 66 0f 2e c1        ucomisd %xmm1,%xmm0
38: f2 0f 11 14 24      movsd  %xmm2,(%rsp)
3d: f2 0f 11 5c 24 08   movsd  %xmm3,0x8(%rsp)
43: f2 0f 11 64 24 10   movsd  %xmm4,0x10(%rsp)
49: f2 0f 10 2d 00 00 00 movsd 0x0(%rip),%xmm5      #
   51 <f_clapeyron+0x51>
50: 00
51: f2 0f 10 35 00 00 00 movsd 0x0(%rip),%xmm6      #
   59 <f_clapeyron+0x59>
58: 00
59: f2 0f 10 3d 00 00 00 movsd 0x0(%rip),%xmm7      #
   61 <f_clapeyron+0x61>
60: 00
61: f2 0f 10 15 00 00 00 movsd 0x0(%rip),%xmm2      #
   69 <f_clapeyron+0x69>
68: 00
69: f2 0f 10 1d 00 00 00 movsd 0x0(%rip),%xmm3      #
   71 <f_clapeyron+0x71>
70: 00
71: f2 0f 10 25 00 00 00 movsd 0x0(%rip),%xmm4      #
   79 <f_clapeyron+0x79>
78: 00
79: f2 0f 11 6c 24 18   movsd  %xmm5,0x18(%rsp)
7f: f2 0f 11 74 24 20   movsd  %xmm6,0x20(%rsp)
85: f2 0f 11 7c 24 28   movsd  %xmm7,0x28(%rsp)
8b: f2 0f 11 54 24 30   movsd  %xmm2,0x30(%rsp)
91: f2 0f 11 5c 24 38   movsd  %xmm3,0x38(%rsp)
97: f2 0f 11 64 24 40   movsd  %xmm4,0x40(%rsp)
9d: 7a 06              jp     a5 <f_clapeyron+0xa5>
9f: 0f 84 db 00 00 00   je     180 <f_clapeyron+0x180>

```

```

a5:  f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    ad <f_clapeyron+0xad>
ac:  00
ad:  66 0f 2e c1            ucomisd %xmm1,%xmm0
b1:  7a 06                  jp      b9 <f_clapeyron+0xb9>
b3:  0f 84 cf 00 00 00      je      188 <f_clapeyron+0x188>
b9:  f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    c1 <f_clapeyron+0xc1>
c0:  00
c1:  66 0f 2e c1            ucomisd %xmm1,%xmm0
c5:  7a 06                  jp      cd <f_clapeyron+0xcd>
c7:  0f 84 c3 00 00 00      je      190 <f_clapeyron+0x190>
cd:  f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    d5 <f_clapeyron+0xd5>
d4:  00
d5:  66 0f 2e c1            ucomisd %xmm1,%xmm0
d9:  7a 06                  jp      e1 <f_clapeyron+0xe1>
db:  0f 84 bf 00 00 00      je      1a0 <f_clapeyron+0x1a0>
e1:  f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    e9 <f_clapeyron+0xe9>
e8:  00
e9:  66 0f 2e c1            ucomisd %xmm1,%xmm0
ed:  7a 06                  jp      f5 <f_clapeyron+0xf5>
ef:  0f 84 bb 00 00 00      je      1b0 <f_clapeyron+0x1b0>
f5:  f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    fd <f_clapeyron+0xfd>
fc:  00
fd:  66 0f 2e c1            ucomisd %xmm1,%xmm0
101: 7a 06                  jp      109 <f_clapeyron+0x109>
103: 0f 84 b7 00 00 00      je      1c0 <f_clapeyron+0x1c0>
109: f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    111 <f_clapeyron+0x111>
110: 00
111: 66 0f 2e c1            ucomisd %xmm1,%xmm0
115: 7a 06                  jp      11d <f_clapeyron+0x11d>
117: 0f 84 b3 00 00 00      je      1d0 <f_clapeyron+0x1d0>
11d: f2 0f 10 0d 00 00 00    movsd  0x0(%rip),%xmm1      #
    125 <f_clapeyron+0x125>
124: 00
125: 66 0f 2e c1            ucomisd %xmm1,%xmm0
129: 7a 02                  jp      12d <f_clapeyron+0x12d>
12b: 74 33                  je      160 <f_clapeyron+0x160>

```

```

12d:  f2 0f 10 0d 00 00 00    movsd    0x0(%rip),%xmm1          #
      135 <f_clapeyron+0x135>
134:  00
135:  66 0f 2e c1              ucomisd  %xmm1,%xmm0
139:  7a 07                    jp       142 <f_clapeyron+0x142>
13b:  b8 08 00 00 00          mov      $0x8,%eax
140:  74 26                    je       168 <f_clapeyron+0x168>
142:  66 0f ef c0              pxor     %xmm0,%xmm0
146:  48 8b 44 24 48          mov      0x48(%rsp),%rax
14b:  64 48 33 04 25 28 00    xor      %fs:0x28,%rax
152:  00 00
154:  0f 85 7d 00 00 00      jne      1d7 <f_clapeyron+0x1d7>
15a:  48 83 c4 58              add      $0x58,%rsp
15e:  c3                      retq
15f:  90                      nop
160:  b8 07 00 00 00          mov      $0x7,%eax
165:  0f 1f 00                nopl     (%rax)
168:  f2 0f 59 0d 00 00 00    mulsd    0x0(%rip),%xmm1          #
      170 <f_clapeyron+0x170>
16f:  00
170:  f2 0f 10 04 c4          movsd    (%rsp,%rax,8),%xmm0
175:  f2 0f 5e c1              divsd    %xmm1,%xmm0
179:  eb cb                    jmp      146 <f_clapeyron+0x146>
17b:  0f 1f 44 00 00          nopl     0x0(%rax,%rax,1)
180:  31 c0                    xor      %eax,%eax
182:  eb e4                    jmp      168 <f_clapeyron+0x168>
184:  0f 1f 40 00              nopl     0x0(%rax)
188:  b8 01 00 00 00          mov      $0x1,%eax
18d:  eb d9                    jmp      168 <f_clapeyron+0x168>
18f:  90                      nop
190:  b8 02 00 00 00          mov      $0x2,%eax
195:  eb d1                    jmp      168 <f_clapeyron+0x168>
197:  66 0f 1f 84 00 00 00    nopw     0x0(%rax,%rax,1)
19e:  00 00
1a0:  b8 03 00 00 00          mov      $0x3,%eax
1a5:  eb c1                    jmp      168 <f_clapeyron+0x168>
1a7:  66 0f 1f 84 00 00 00    nopw     0x0(%rax,%rax,1)
1ae:  00 00
1b0:  b8 04 00 00 00          mov      $0x4,%eax
1b5:  eb b1                    jmp      168 <f_clapeyron+0x168>
1b7:  66 0f 1f 84 00 00 00    nopw     0x0(%rax,%rax,1)
1be:  00 00

```

```

1c0:  b8 05 00 00 00      mov     $0x5,%eax
1c5:  eb a1               jmp     168 <f_clapeyron+0x168>
1c7:  66 0f 1f 84 00 00 00 nopw    0x0(%rax,%rax,1)
1ce:  00 00
1d0:  b8 06 00 00 00      mov     $0x6,%eax
1d5:  eb 91               jmp     168 <f_clapeyron+0x168>
1d7:  e8 00 00 00 00      callq  1dc <f_clapeyron+0x1dc>
1dc:  0f 1f 40 00         nopl    0x0(%rax)

0000000000000001e0 <simpson13>:
1e0:  48 83 ec 58         sub     $0x58,%rsp
1e4:  64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
1eb:  00 00
1ed:  48 89 44 24 48      mov     %rax,0x48(%rsp)
1f2:  31 c0               xor     %eax,%eax
1f4:  d1 ef               shr     %edi
1f6:  0f 84 a1 05 00 00   je      79d <simpson13+0x5bd>
1fc:  66 0f ef ff         pxor    %xmm7,%xmm7
200:  b9 01 00 00 00      mov     $0x1,%ecx
205:  ba 01 00 00 00      mov     $0x1,%edx
20a:  f2 0f 10 25 00 00 00 movsd   0x0(%rip),%xmm4      #
    212 <simpson13+0x32>
211:  00
212:  f2 44 0f 10 05 00 00 movsd   0x0(%rip),%xmm8      #
    21b <simpson13+0x3b>
219:  00 00
21b:  66 0f 28 f7         movapd  %xmm7,%xmm6
21f:  66 0f 28 ef         movapd  %xmm7,%xmm5
223:  f2 44 0f 10 0d 00 00 movsd   0x0(%rip),%xmm9      #
    22c <simpson13+0x4c>
22a:  00 00
22c:  f2 44 0f 10 15 00 00 movsd   0x0(%rip),%xmm10     #
    235 <simpson13+0x55>
233:  00 00
235:  f2 44 0f 10 1d 00 00 movsd   0x0(%rip),%xmm11     #
    23e <simpson13+0x5e>
23c:  00 00
23e:  f2 44 0f 10 25 00 00 movsd   0x0(%rip),%xmm12     #
    247 <simpson13+0x67>
245:  00 00
247:  f2 44 0f 10 2d 00 00 movsd   0x0(%rip),%xmm13     #
    250 <simpson13+0x70>

```



```

24e: 00 00
250: f2 44 0f 10 35 00 00    movsd    0x0(%rip),%xmm14          #
    259 <simpson13+0x79>
257: 00 00
259: f2 44 0f 10 3d 00 00    movsd    0x0(%rip),%xmm15          #
    262 <simpson13+0x82>
260: 00 00
262: 66 0f 1f 44 00 00      nopw     0x0(%rax,%rax,1)
268: 66 0f ef db            pxor     %xmm3,%xmm3
26c: 8d 04 12              lea      (%rdx,%rdx,1),%eax
26f: f2 0f 2a d9            cvtsi2sd %ecx,%xmm3
273: f2 0f 59 da            mulsd    %xmm2,%xmm3
277: f2 0f 58 d8            addsd    %xmm0,%xmm3
27b: 66 0f 2e dc            ucomisd   %xmm4,%xmm3
27f: 7a 06                jp       287 <simpson13+0xa7>
281: 0f 84 d9 03 00 00      je       660 <simpson13+0x480>
287: 66 41 0f 2e d8        ucomisd   %xmm8,%xmm3
28c: 7a 06                jp       294 <simpson13+0xb4>
28e: 0f 84 dc 03 00 00      je       670 <simpson13+0x490>
294: 66 41 0f 2e d9        ucomisd   %xmm9,%xmm3
299: 7a 06                jp       2a1 <simpson13+0xc1>
29b: 0f 84 df 03 00 00      je       680 <simpson13+0x4a0>
2a1: 66 41 0f 2e da        ucomisd   %xmm10,%xmm3
2a6: 7a 06                jp       2ae <simpson13+0xce>
2a8: 0f 84 e2 03 00 00      je       690 <simpson13+0x4b0>
2ae: 66 41 0f 2e db        ucomisd   %xmm11,%xmm3
2b3: 7a 06                jp       2bb <simpson13+0xdb>
2b5: 0f 84 e5 03 00 00      je       6a0 <simpson13+0x4c0>
2bb: 66 41 0f 2e dc        ucomisd   %xmm12,%xmm3
2c0: 7a 06                jp       2c8 <simpson13+0xe8>
2c2: 0f 84 e8 03 00 00      je       6b0 <simpson13+0x4d0>
2c8: 66 41 0f 2e dd        ucomisd   %xmm13,%xmm3
2cd: 7a 06                jp       2d5 <simpson13+0xf5>
2cf: 0f 84 eb 03 00 00      je       6c0 <simpson13+0x4e0>
2d5: 66 41 0f 2e de        ucomisd   %xmm14,%xmm3
2da: 7a 06                jp       2e2 <simpson13+0x102>
2dc: 0f 84 ee 03 00 00      je       6d0 <simpson13+0x4f0>
2e2: 66 41 0f 2e df        ucomisd   %xmm15,%xmm3
2e7: 7a 0a                jp       2f3 <simpson13+0x113>
2e9: f2 0f 10 1d 00 00 00    movsd    0x0(%rip),%xmm3          # 2
    f1 <simpson13+0x111>
2f0: 00

```

```

2f1: 74 04                je      2f7 <simpson13+0x117>
2f3: 66 0f 28 df         movapd  %xmm7,%xmm3
2f7: 39 d7                cmp     %edx,%edi
2f9: f2 0f 58 eb         addsd   %xmm3,%xmm5
2fd: 0f 86 a8 00 00 00    jbe     3ab <simpson13+0x1cb>
303: 66 0f ef db         pxor    %xmm3,%xmm3
307: f2 0f 2a d8         cvtsi2sd %eax,%xmm3
30b: f2 0f 59 da         mulsd   %xmm2,%xmm3
30f: f2 0f 58 d8         addsd   %xmm0,%xmm3
313: 66 0f 2e dc         ucomisd %xmm4,%xmm3
317: 7a 06                jp      31f <simpson13+0x13f>
319: 0f 84 01 04 00 00    je      720 <simpson13+0x540>
31f: 66 0f 2e 1d 00 00 00 ucomisd 0x0(%rip),%xmm3      #
    327 <simpson13+0x147>
326: 00
327: 7a 06                jp      32f <simpson13+0x14f>
329: 0f 84 01 04 00 00    je      730 <simpson13+0x550>
32f: 66 0f 2e 1d 00 00 00 ucomisd 0x0(%rip),%xmm3      #
    337 <simpson13+0x157>
336: 00
337: 7a 06                jp      33f <simpson13+0x15f>
339: 0f 84 01 04 00 00    je      740 <simpson13+0x560>
33f: 66 0f 2e 1d 00 00 00 ucomisd 0x0(%rip),%xmm3      #
    347 <simpson13+0x167>
346: 00
347: 7a 06                jp      34f <simpson13+0x16f>
349: 0f 84 01 04 00 00    je      750 <simpson13+0x570>
34f: 66 0f 2e 1d 00 00 00 ucomisd 0x0(%rip),%xmm3      #
    357 <simpson13+0x177>
356: 00
357: 7a 06                jp      35f <simpson13+0x17f>
359: 0f 84 01 04 00 00    je      760 <simpson13+0x580>
35f: 66 0f 2e 1d 00 00 00 ucomisd 0x0(%rip),%xmm3      #
    367 <simpson13+0x187>
366: 00
367: 7a 06                jp      36f <simpson13+0x18f>
369: 0f 84 01 04 00 00    je      770 <simpson13+0x590>
36f: 66 0f 2e 1d 00 00 00 ucomisd 0x0(%rip),%xmm3      #
    377 <simpson13+0x197>
376: 00
377: 7a 06                jp      37f <simpson13+0x19f>
379: 0f 84 01 04 00 00    je      780 <simpson13+0x5a0>

```

```

37f: 66 0f 2e 1d 00 00 00    ucomisd 0x0(%rip),%xmm3      #
    387 <simpson13+0x1a7>
386: 00
387: 7a 06                    jp      38f <simpson13+0x1af>
389: 0f 84 01 04 00 00        je      790 <simpson13+0x5b0>
38f: 66 0f 2e 1d 00 00 00    ucomisd 0x0(%rip),%xmm3      #
    397 <simpson13+0x1b7>
396: 00
397: 7a 0a                    jp      3a3 <simpson13+0x1c3>
399: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      # 3
    a1 <simpson13+0x1c1>
3a0: 00
3a1: 74 04                    je      3a7 <simpson13+0x1c7>
3a3: 66 0f 28 df              movapd  %xmm7,%xmm3
3a7: f2 0f 58 f3              addsd   %xmm3,%xmm6
3ab: 83 c2 01                  add     $0x1,%edx
3ae: 83 c1 02                  add     $0x2,%ecx
3b1: 39 d7                    cmp     %edx,%edi
3b3: 0f 83 af fe ff ff        jae     268 <simpson13+0x88>
3b9: f2 0f 59 2d 00 00 00    mulsd   0x0(%rip),%xmm5      # 3
    c1 <simpson13+0x1e1>
3c0: 00
3c1: f2 0f 58 f6              addsd   %xmm6,%xmm6
3c5: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      # 3
    cd <simpson13+0x1ed>
3cc: 00
3cd: 66 0f 2e c4              ucomisd %xmm4,%xmm0
3d1: f2 0f 11 1c 24          movsd   %xmm3,(%rsp)
3d6: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      # 3
    de <simpson13+0x1fe>
3dd: 00
3de: f2 0f 11 5c 24 08        movsd   %xmm3,0x8(%rsp)
3e4: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      # 3
    ec <simpson13+0x20c>
3eb: 00
3ec: f2 0f 11 5c 24 10        movsd   %xmm3,0x10(%rsp)
3f2: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      # 3
    fa <simpson13+0x21a>
3f9: 00
3fa: f2 0f 11 5c 24 18        movsd   %xmm3,0x18(%rsp)
400: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
    408 <simpson13+0x228>

```

```

407: 00
408: f2 0f 11 5c 24 20      movsd  %xmm3,0x20(%rsp)
40e: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    416 <simpson13+0x236>
415: 00
416: f2 0f 11 5c 24 28      movsd  %xmm3,0x28(%rsp)
41c: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    424 <simpson13+0x244>
423: 00
424: f2 0f 11 5c 24 30      movsd  %xmm3,0x30(%rsp)
42a: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    432 <simpson13+0x252>
431: 00
432: f2 0f 11 5c 24 38      movsd  %xmm3,0x38(%rsp)
438: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    440 <simpson13+0x260>
43f: 00
440: f2 0f 11 5c 24 40      movsd  %xmm3,0x40(%rsp)
446: 7a 06                  jp      44e <simpson13+0x26e>
448: 0f 84 68 03 00 00      je      7b6 <simpson13+0x5d6>
44e: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    456 <simpson13+0x276>
455: 00
456: 66 0f 2e c3            ucomisd %xmm3,%xmm0
45a: 7a 06                  jp      462 <simpson13+0x282>
45c: 0f 84 73 03 00 00      je      7d5 <simpson13+0x5f5>
462: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    46a <simpson13+0x28a>
469: 00
46a: 66 0f 2e c3            ucomisd %xmm3,%xmm0
46e: 7a 06                  jp      476 <simpson13+0x296>
470: 0f 84 69 03 00 00      je      7df <simpson13+0x5ff>
476: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    47e <simpson13+0x29e>
47d: 00
47e: 66 0f 2e c3            ucomisd %xmm3,%xmm0
482: 7a 06                  jp      48a <simpson13+0x2aa>
484: 0f 84 b0 03 00 00      je      83a <simpson13+0x65a>
48a: f2 0f 10 1d 00 00 00    movsd  0x0(%rip),%xmm3      #
    492 <simpson13+0x2b2>
491: 00
492: 66 0f 2e c3            ucomisd %xmm3,%xmm0

```

```

496: 7a 06                jp      49e <simpson13+0x2be>
498: 0f 84 92 03 00 00    je      830 <simpson13+0x650>
49e: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          # 4
    a6 <simpson13+0x2c6>
4a5: 00
4a6: 66 0f 2e c3          ucomisd %xmm3,%xmm0
4aa: 7a 06                jp      4b2 <simpson13+0x2d2>
4ac: 0f 84 56 03 00 00    je      808 <simpson13+0x628>
4b2: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          # 4
    ba <simpson13+0x2da>
4b9: 00
4ba: 66 0f 2e c3          ucomisd %xmm3,%xmm0
4be: 7a 06                jp      4c6 <simpson13+0x2e6>
4c0: 0f 84 38 03 00 00    je      7fe <simpson13+0x61e>
4c6: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          # 4
    ce <simpson13+0x2ee>
4cd: 00
4ce: 66 0f 2e c3          ucomisd %xmm3,%xmm0
4d2: 7a 06                jp      4da <simpson13+0x2fa>
4d4: 0f 84 24 02 00 00    je      6fe <simpson13+0x51e>
4da: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          # 4
    e2 <simpson13+0x302>
4e1: 00
4e2: 66 0f 2e c3          ucomisd %xmm3,%xmm0
4e6: 7a 0b                jp      4f3 <simpson13+0x313>
4e8: b8 08 00 00 00      mov     $0x8,%eax
4ed: 0f 84 15 02 00 00    je      708 <simpson13+0x528>
4f3: 66 0f 28 c7          movapd  %xmm7,%xmm0
4f7: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          # 4
    ff <simpson13+0x31f>
4fe: 00
4ff: 66 0f 2e cc          ucomisd %xmm4,%xmm1
503: f2 0f 11 1c 24      movsd   %xmm3,(%rsp)
508: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          #
    510 <simpson13+0x330>
50f: 00
510: f2 0f 11 5c 24 08    movsd   %xmm3,0x8(%rsp)
516: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          #
    51e <simpson13+0x33e>
51d: 00
51e: f2 0f 11 5c 24 10    movsd   %xmm3,0x10(%rsp)
524: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3          #

```

```

52c: <simpson13+0x34c>
52b: 00
52c: f2 0f 11 5c 24 18      movsd   %xmm3,0x18(%rsp)
532: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
53a: <simpson13+0x35a>
539: 00
53a: f2 0f 11 5c 24 20      movsd   %xmm3,0x20(%rsp)
540: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
548: <simpson13+0x368>
547: 00
548: f2 0f 11 5c 24 28      movsd   %xmm3,0x28(%rsp)
54e: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
556: <simpson13+0x376>
555: 00
556: f2 0f 11 5c 24 30      movsd   %xmm3,0x30(%rsp)
55c: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
564: <simpson13+0x384>
563: 00
564: f2 0f 11 5c 24 38      movsd   %xmm3,0x38(%rsp)
56a: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
572: <simpson13+0x392>
571: 00
572: f2 0f 11 5c 24 40      movsd   %xmm3,0x40(%rsp)
578: 7a 06                  jp      580 <simpson13+0x3a0>
57a: 0f 84 69 02 00 00      je      7e9 <simpson13+0x609>
580: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
588: <simpson13+0x3a8>
587: 00
588: 66 0f 2e cb            ucomisd %xmm3,%xmm1
58c: 7a 06                  jp      594 <simpson13+0x3b4>
58e: 0f 84 2d 02 00 00      je      7c1 <simpson13+0x5e1>
594: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      #
59c: <simpson13+0x3bc>
59b: 00
59c: 66 0f 2e cb            ucomisd %xmm3,%xmm1
5a0: 7a 06                  jp      5a8 <simpson13+0x3c8>
5a2: 0f 84 23 02 00 00      je      7cb <simpson13+0x5eb>
5a8: f2 0f 10 1d 00 00 00    movsd   0x0(%rip),%xmm3      # 5
b0: <simpson13+0x3d0>
5af: 00
5b0: 66 0f 2e cb            ucomisd %xmm3,%xmm1
5b4: 7a 06                  jp      5bc <simpson13+0x3dc>

```

```

5b6: 0f 84 38 02 00 00      je      7f4 <simpson13+0x614>
5bc: f2 0f 10 1d 00 00 00  movsd   0x0(%rip),%xmm3      # 5
      c4 <simpson13+0x3e4>
5c3: 00
5c4: 66 0f 2e cb           ucomisd %xmm3,%xmm1
5c8: 7a 06                jp      5d0 <simpson13+0x3f0>
5ca: 0f 84 56 02 00 00      je      826 <simpson13+0x646>
5d0: f2 0f 10 1d 00 00 00  movsd   0x0(%rip),%xmm3      # 5
      d8 <simpson13+0x3f8>
5d7: 00
5d8: 66 0f 2e cb           ucomisd %xmm3,%xmm1
5dc: 7a 06                jp      5e4 <simpson13+0x404>
5de: 0f 84 38 02 00 00      je      81c <simpson13+0x63c>
5e4: f2 0f 10 1d 00 00 00  movsd   0x0(%rip),%xmm3      # 5
      ec <simpson13+0x40c>
5eb: 00
5ec: 66 0f 2e cb           ucomisd %xmm3,%xmm1
5f0: 7a 06                jp      5f8 <simpson13+0x418>
5f2: 0f 84 1a 02 00 00      je      812 <simpson13+0x632>
5f8: f2 0f 10 1d 00 00 00  movsd   0x0(%rip),%xmm3      #
      600 <simpson13+0x420>
5ff: 00
600: 66 0f 2e cb           ucomisd %xmm3,%xmm1
604: 7a 06                jp      60c <simpson13+0x42c>
606: 0f 84 d1 00 00 00      je      6dd <simpson13+0x4fd>
60c: f2 0f 10 1d 00 00 00  movsd   0x0(%rip),%xmm3      #
      614 <simpson13+0x434>
613: 00
614: 66 0f 2e cb           ucomisd %xmm3,%xmm1
618: 7a 0b                jp      625 <simpson13+0x445>
61a: b8 08 00 00 00        mov     $0x8,%eax
61f: 0f 84 c3 00 00 00      je      6e8 <simpson13+0x508>
625: f2 0f 5e 15 00 00 00  divsd   0x0(%rip),%xmm2      #
      62d <simpson13+0x44d>
62c: 00
62d: f2 0f 58 f8           addsd   %xmm0,%xmm7
631: 48 8b 44 24 48        mov     0x48(%rsp),%rax
636: 64 48 33 04 25 28 00  xor     %fs:0x28,%rax
63d: 00 00
63f: f2 0f 58 ef           addsd   %xmm7,%xmm5
643: f2 0f 58 f5           addsd   %xmm5,%xmm6
647: f2 0f 59 d6           mulsd   %xmm6,%xmm2

```

```

64b: 66 0f 28 c2          movapd %xmm2,%xmm0
64f: 0f 85 ef 01 00 00    jne     844 <simpson13+0x664>
655: 48 83 c4 58          add     $0x58,%rsp
659: c3                  retq
65a: 66 0f 1f 44 00 00    nopw   0x0(%rax,%rax,1)
660: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    668 <simpson13+0x488>
667: 00
668: e9 8a fc ff ff      jmpq    2f7 <simpson13+0x117>
66d: 0f 1f 00            nopl    (%rax)
670: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    678 <simpson13+0x498>
677: 00
678: e9 7a fc ff ff      jmpq    2f7 <simpson13+0x117>
67d: 0f 1f 00            nopl    (%rax)
680: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    688 <simpson13+0x4a8>
687: 00
688: e9 6a fc ff ff      jmpq    2f7 <simpson13+0x117>
68d: 0f 1f 00            nopl    (%rax)
690: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    698 <simpson13+0x4b8>
697: 00
698: e9 5a fc ff ff      jmpq    2f7 <simpson13+0x117>
69d: 0f 1f 00            nopl    (%rax)
6a0: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      # 6
    a8 <simpson13+0x4c8>
6a7: 00
6a8: e9 4a fc ff ff      jmpq    2f7 <simpson13+0x117>
6ad: 0f 1f 00            nopl    (%rax)
6b0: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      # 6
    b8 <simpson13+0x4d8>
6b7: 00
6b8: e9 3a fc ff ff      jmpq    2f7 <simpson13+0x117>
6bd: 0f 1f 00            nopl    (%rax)
6c0: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      # 6
    c8 <simpson13+0x4e8>
6c7: 00
6c8: e9 2a fc ff ff      jmpq    2f7 <simpson13+0x117>
6cd: 0f 1f 00            nopl    (%rax)
6d0: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      # 6
    d8 <simpson13+0x4f8>

```



```

6d7: 00
6d8: e9 1a fc ff ff      jmpq    2f7 <simpson13+0x117>
6dd: b8 07 00 00 00      mov     $0x7,%eax
6e2: 66 0f 1f 44 00 00    nopw    0x0(%rax,%rax,1)
6e8: f2 0f 59 1d 00 00 00 mulsd    0x0(%rip),%xmm3      # 6
    f0 <simpson13+0x510>
6ef: 00
6f0: f2 0f 10 3c c4      movsd   (%rsp,%rax,8),%xmm7
6f5: f2 0f 5e fb        divsd   %xmm3,%xmm7
6f9: e9 27 ff ff ff      jmpq    625 <simpson13+0x445>
6fe: b8 07 00 00 00      mov     $0x7,%eax
703: 0f 1f 44 00 00      nopl    0x0(%rax,%rax,1)
708: f2 0f 59 1d 00 00 00 mulsd    0x0(%rip),%xmm3      #
    710 <simpson13+0x530>
70f: 00
710: f2 0f 10 04 c4      movsd   (%rsp,%rax,8),%xmm0
715: f2 0f 5e c3        divsd   %xmm3,%xmm0
719: e9 d9 fd ff ff      jmpq    4f7 <simpson13+0x317>
71e: 66 90              xchg    %ax,%ax
720: f2 0f 10 1d 00 00 00 movsd    0x0(%rip),%xmm3      #
    728 <simpson13+0x548>
727: 00
728: e9 7a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
72d: 0f 1f 00            nopl    (%rax)
730: f2 0f 10 1d 00 00 00 movsd    0x0(%rip),%xmm3      #
    738 <simpson13+0x558>
737: 00
738: e9 6a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
73d: 0f 1f 00            nopl    (%rax)
740: f2 0f 10 1d 00 00 00 movsd    0x0(%rip),%xmm3      #
    748 <simpson13+0x568>
747: 00
748: e9 5a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
74d: 0f 1f 00            nopl    (%rax)
750: f2 0f 10 1d 00 00 00 movsd    0x0(%rip),%xmm3      #
    758 <simpson13+0x578>
757: 00
758: e9 4a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
75d: 0f 1f 00            nopl    (%rax)
760: f2 0f 10 1d 00 00 00 movsd    0x0(%rip),%xmm3      #
    768 <simpson13+0x588>
767: 00

```

```

768: e9 3a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
76d: 0f 1f 00            nopl    (%rax)
770: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    778 <simpson13+0x598>
777: 00
778: e9 2a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
77d: 0f 1f 00            nopl    (%rax)
780: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    788 <simpson13+0x5a8>
787: 00
788: e9 1a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
78d: 0f 1f 00            nopl    (%rax)
790: f2 0f 10 1d 00 00 00 movsd   0x0(%rip),%xmm3      #
    798 <simpson13+0x5b8>
797: 00
798: e9 0a fc ff ff      jmpq    3a7 <simpson13+0x1c7>
79d: 66 0f ef ff        pxor    %xmm7,%xmm7
7a1: f2 0f 10 25 00 00 00 movsd   0x0(%rip),%xmm4      # 7
    a9 <simpson13+0x5c9>
7a8: 00
7a9: 66 0f 28 f7          movapd  %xmm7,%xmm6
7ad: 66 0f 28 ef          movapd  %xmm7,%xmm5
7b1: e9 0f fc ff ff      jmpq    3c5 <simpson13+0x1e5>
7b6: 31 c0                xor     %eax,%eax
7b8: 66 0f 28 dc          movapd  %xmm4,%xmm3
7bc: e9 47 ff ff ff      jmpq    708 <simpson13+0x528>
7c1: b8 01 00 00 00      mov     $0x1,%eax
7c6: e9 1d ff ff ff      jmpq    6e8 <simpson13+0x508>
7cb: b8 02 00 00 00      mov     $0x2,%eax
7d0: e9 13 ff ff ff      jmpq    6e8 <simpson13+0x508>
7d5: b8 01 00 00 00      mov     $0x1,%eax
7da: e9 29 ff ff ff      jmpq    708 <simpson13+0x528>
7df: b8 02 00 00 00      mov     $0x2,%eax
7e4: e9 1f ff ff ff      jmpq    708 <simpson13+0x528>
7e9: 31 c0                xor     %eax,%eax
7eb: 66 0f 28 dc          movapd  %xmm4,%xmm3
7ef: e9 f4 fe ff ff      jmpq    6e8 <simpson13+0x508>
7f4: b8 03 00 00 00      mov     $0x3,%eax
7f9: e9 ea fe ff ff      jmpq    6e8 <simpson13+0x508>
7fe: b8 06 00 00 00      mov     $0x6,%eax
803: e9 00 ff ff ff      jmpq    708 <simpson13+0x528>
808: b8 05 00 00 00      mov     $0x5,%eax

```

```

80d:  e9 f6 fe ff ff      jmpq    708 <simpson13+0x528>
812:  b8 06 00 00 00      mov     $0x6,%eax
817:  e9 cc fe ff ff      jmpq    6e8 <simpson13+0x508>
81c:  b8 05 00 00 00      mov     $0x5,%eax
821:  e9 c2 fe ff ff      jmpq    6e8 <simpson13+0x508>
826:  b8 04 00 00 00      mov     $0x4,%eax
82b:  e9 b8 fe ff ff      jmpq    6e8 <simpson13+0x508>
830:  b8 04 00 00 00      mov     $0x4,%eax
835:  e9 ce fe ff ff      jmpq    708 <simpson13+0x528>
83a:  b8 03 00 00 00      mov     $0x3,%eax
83f:  e9 c4 fe ff ff      jmpq    708 <simpson13+0x528>
844:  e8 00 00 00 00      callq   849 <simpson13+0x669>

```

Disassembly of section .text.startup:

0000000000000000 <main>:

```

0:  48 83 ec 08          sub     $0x8,%rsp
4:  bf 09 00 00 00      mov     $0x9,%edi
9:  f2 0f 10 15 00 00 00 movsd   0x0(%rip),%xmm2      #
11 <main+0x11>
10:  00
11:  f2 0f 10 0d 00 00 00 movsd   0x0(%rip),%xmm1      #
19 <main+0x19>
18:  00
19:  f2 0f 10 05 00 00 00 movsd   0x0(%rip),%xmm0      #
21 <main+0x21>
20:  00
21:  e8 00 00 00 00      callq   26 <main+0x26>
26:  66 0f 2e 05 00 00 00 ucomisd 0x0(%rip),%xmm0      #
2e <main+0x2e>
2d:  00
2e:  72 05              jb      35 <main+0x35>
30:  e8 00 00 00 00      callq   35 <main+0x35>
35:  31 c0              xor     %eax,%eax
37:  48 83 c4 08          add     $0x8,%rsp
3b:  c3                retq

```