Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Otimizando Simulador RISC-V para Ambiente de Linha de Comando

Autor: Rodrigo Dadamos Lopes da Silva
Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF
2023

Rodrigo Dadamos Lopes da Silva

Otimizando Simulador RISC-V para Ambiente de Linha de Comando

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF

2023

Rodrigo Dadamos Lopes da Silva
Otimizando Simulador RISC-V para Ambiente de Linha de Comando/ Rodrigo Dadamos Lopes da Silva. – Brasília, DF, 2023-
60 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Tiago Alves da Fonseca

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2023.

1. CLI. 2. RISC-V. I. Prof. Dr. Tiago Alves da Fonseca. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Otimizando Simulador RISC-V para Ambiente de Linha de Comando

CDU 02:141:005.6

Rodrigo Dadamos Lopes da Silva

Otimizando Simulador RISC-V para Ambiente de Linha de Comando

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, :

Prof. Dr. Tiago Alves da Fonseca
Orientador

Prof. Dr. Bruno Cesar Ribas
Convidado 1

Prof. Dr. Daniel Sundfeld Lima
Convidado 2

Brasília, DF
2023

Este trabalho é dedicado à minha mãe.

Agradecimentos

Agradeço primeiramente a Priscila pelo o apoio e incentivo durante todo este curso. Agradeço a minha família e agradeço também ao Professor Tiago pela orientação e pela oportunidade de realizar este trabalho.

Resumo

Simuladores de conjunto de instruções são utilizados em disciplinas de arquitetura de computadores para o ensino e avaliação de provas e exercícios. Atualmente, no ensino de arquitetura de computadores, cresce o uso de um conjunto de instruções chamado RISC-V. A proposta deste trabalho é oferecer um simulador RISC-V otimizado para a interface de linha de comando (CLI) apropriado para o uso junto a juízes eletrônicos como o CD-MOJ. Foi feita uma pesquisa por simuladores e, após uma análise, o simulador RARS foi escolhido e teve sua CLI otimizada. Para verificar a eficiência das otimizações foram utilizados métodos de coleta do tempo de execução. Uma ferramenta de *profiler* também foi utilizada. O tempo de execução do simulador RARS no modo CLI foi reduzido com as otimizações realizadas. Também foram identificadas melhorias a serem feitas.

Palavras-chave: Arquitetura de computadores; Simulador; RISC-V; CLI.

Abstract

Instruction set simulators are used in computer architecture disciplines for the teaching and evaluation of tests and exercises. Nowadays, in the teaching of computer architecture, the use of a set of instructions called RISC-V grows. The proposal of this work is to offer an optimized RISC-V simulator for the command line interface (CLI) suitable for use with electronic judges such as CD-MOJ. To verify the efficiency of the optimizations were used methods of collection of the execution time. A profiler tool has also been used. The RARS simulator execution time in Cli mode has been reduced with the optimizations performed. Improvements to be made were also identified.

Key-words: Computer architecture; Simulator; RISC-V; CLI.

Lista de ilustrações

Figura 1 – Formatos de instruções do RV32I 24
Figura 2 – Execução de um programa no SPIM com entrada de dados 28
Figura 3 – Execução de um programa no RARS com entrada de dados 30
Figura 4 – Árvore de chamadas por métodos com crivo gerada no JProfiler 41
Figura 5 – Árvore de chamadas por métodos com crivo no RARS 41
Figura 6 – Árvore de chamadas por classes com crivo no RARS 42
Figura 7 – Árvore de chamadas por pacotes com crivo no RARS 42
Figura 8 – Sequência de comandos para clonar, criar e testar o RARS 43
Figura 9 – Árvore de chamadas por métodos com crivo no RARS-CLI 48
Figura 10 – Gráfico de comparação das médias e desvios-padrão com crivo adaptado 49
Figura 11 – Gráfico de comparação das médias e desvios-padrão com *selection sort* 49
Figura 12 – Criando um executável nativo com GraalVM 50

Lista de tabelas

Tabela 1 – Critérios de inclusão de simuladores 30
Tabela 2 – Critérios de exclusão de simuladores 30
Tabela 3 – Questões para a pesquisa de simuladores 31
Tabela 4 – Resultados da pesquisa de simuladores 34
Tabela 5 – SPIM e RARS, média do crivo com *date* 38
Tabela 6 – SPIM e RARS, desvio-padrão do crivo com *date* 38
Tabela 7 – SPIM e RARS, média com crivo adaptado com *date* 39
Tabela 8 – SPIM e RARS, desvio-padrão com crivo adaptado com *date* 39
Tabela 9 – SPIM e RARS, média com *selection sort* 39
Tabela 10 – SPIM e RARS, desvio-padrão com *selection sort* 39
Tabela 11 – Removendo bibliotecas gráficas - média com *selection sort* 44
Tabela 12 – Removendo configurações não utilizadas - média com *selection sort* . . 44
Tabela 13 – Removendo bibliotecas não utilizadas - média com *selection sort* 45
Tabela 14 – Aplicando sugestões do *Effective Java* - média com *selection sort* . . . 47
Tabela 15 – Aplicando sugestões do *Effective Java* - desvio-padrão 47
Tabela 16 – Aplicando sugestões do *Effective Java* - média com crivo adaptado . . 47
Tabela 17 – Aplicando sugestões do *Effective Java* - desvio-padrão crivo adaptado . 48

Lista de códigos

Listagem 1 – Algumas instruções do conjunto base RV32I 24
Listagem 2 – Trecho do shell script para coletar medidas 37
Listagem 3 – *While loop* na classe *Globals* 45
Listagem 4 – *For loop* substituindo o *while loop* na classe *Globals* 46
Listagem 5 – *Loop* na classe *Launch* 46
Listagem 6 – *For-each loop* substituindo o *loop* na classe *Launch* 46
Listagem 7 – Programa em C com base no Crivo de Eratóstenes 56
Listagem 8 – Programa em RISC-V com base no Crivo de Eratóstenes 57
Listagem 9 – Shell Script para coletar medidas dos tempos de execução 59

Lista de abreviaturas e siglas

BSD	Berkeley Software Distribution
CISC	Complex Instruction Set Computer
CLI	Command-Line Interface
CPU	Central Processing Unit
GNU	GNU's Not Unix!
GUI	Graphical User Interface
IBM	International Business Machines
ISA	Instruction Set Architecture
JNI	Java Native Interface
JVM	Java Virtual Machine
MIPS	Microprocessor without Interlocked Pipeline Stages
RISC	Reduced Instruction Set Computer
SoC	System on a Chip
VLSI	Very Large-Scale Integrated

Sumário

1 INTRODUÇÃO **14**
1.1 Justificativa **15**
1.2 Objetivos **16**
1.2.1 Objetivo Geral 16
1.2.2 Objetivos Específicos 16
1.3 Estrutura do Documento **16**

2 FUNDAMENTAÇÃO TEÓRICA **17**
2.1 ISA **17**
2.1.1 CISC 18
2.1.2 RISC 19
2.2 MIPS **21**
2.2.1 SPIM 22
2.3 RISC-V **22**
2.3.1 Conjuntos Base de Instruções 23
2.3.2 Instruções do Conjunto Base RV32I 23
2.3.3 Extensões 25
2.4 CD-MOJ **25**

3 PROPOSTA DE TRABALHO **27**
3.1 CLI **28**

4 METODOLOGIA **29**
4.1 Metodologia do Desenvolvimento **29**
4.1.1 Comparação entre SPIM e RARS 29
4.1.2 Pesquisa por Simuladores 29
4.1.2.1 Critérios de Inclusão e Exclusão 29
4.1.2.2 Questões para a Pesquisa por Simuladores 30
4.1.2.3 Estratégia de Análise 30
4.1.3 Coleta das Métricas do Tempo de Execução 31
4.1.3.1 Crivo de Eratóstenes 31
4.1.3.2 *Selection Sort* 32
4.1.3.3 Medindo o Tempo de Execução 32
4.2 Políticas de Desenvolvimento do Projeto **33**

5 RESULTADOS **34**

5.1 Resultados da Pesquisa por Simuladores 34
5.2 Discussão dos Resultados da Pesquisa por Simuladores 35
5.2.1 Simulador Ripes 35
5.2.2 Simulador RARS 35
5.3 Programas para Calcular Primos com Crivo de Eratóstenes 36
5.4 Shell Script para Medir o Tempo de Execução 36
5.4.1 Configurações do Ambiente de Execução do Shell Script 37
5.5 Comparação entre SPIM e RARS 37
5.6 Otimização do Simulador RARS 39
5.6.1 Medidas Iniciais do RARS 40
5.6.2 Otimizações Realizadas 43
5.6.2.1 Removendo bibliotecas gráficas 43
5.6.2.2 Removendo configurações não utilizadas 44
5.6.2.3 Removendo bibliotecas e funcionalidades não utilizadas 44
5.6.2.4 Aplicando sugestões do livro *Effective Java* 45
5.6.3 Comparação entre RARS e RARS-CLI 47
5.7 Executável Nativo 48

6 CONCLUSÃO 51
6.1 Trabalhos futuros 51

REFERÊNCIAS 53

APÊNDICES 55

APÊNDICE A – PROGRAMA EM C 56

APÊNDICE B – PROGRAMA EM RISC-V 57

APÊNDICE C – SHELL SCRIPT 59

1 Introdução

Para aprender a arquitetura de um computador é necessário aprender sua linguagem e a grande maioria das arquiteturas definem as mesmas instruções básicas, como adição e subtração, que operam na memória ou nos registradores. Isso acontece devido à forma como os computadores são construídos com tecnologias de *hardware* que seguem os mesmos princípios e pelo fato dos computadores necessitarem oferecer as mesmas operações básicas. No entanto, escrever e ler linguagem (binária) de máquina é muito tedioso para os desenvolvedores e os projetistas optaram por representar o conjunto de instruções em uma linguagem de montagem (*assembly*) (HARRIS; HARRIS, 2013).

Tanto Harris e Harris (2013) como Patterson e Hennessy (2005) consideram que os conjuntos de instruções das diferentes arquiteturas existentes possuem mais similaridades entre si (como dialetos de uma mesma linguagem) em vez de diferenças (como em linguagens independentes). Dessa forma, aprender um conjunto de instruções simplifica o aprendizado de outros conjuntos.

Ainda de acordo com Patterson e Hennessy (2005), abstração é um modelo que facilita o desenvolvimento de sistemas mais complexos detalhando níveis inferiores. A interface entre o *hardware* e *software* de baixo nível é uma das abstrações mais importantes usadas no estudo de sistemas computacionais e é chamada de arquitetura do conjunto de instruções (ISA).

Na disciplina de Fundamentos de Arquitetura de Computadores, as atividades e exercícios de linguagem de montagem (*assembly*) são corrigidas utilizando o meta juiz *online* CD-MOJ[1]. No ensino de *assembly* utilizando MIPS[2], uma ISA desenvolvida seguindo os princípios de computador com um conjunto reduzido de instruções (RISC), são utilizados *softwares* simuladores como o SPIM[3] que possui a implementação de uma interface de linha de comando (CLI[4]) utilizada pelo CD-MOJ para a correção automática das atividades e exercícios.

Nos últimos anos, uma ISA livre e aberta, também criada seguindo os princípios RISC e chamada de RISC-V[5], vem tornando-se cada vez mais relevante e com estimativas de crescimento cada vez maiores chegando a mais de 65 bilhões de *cores*[6] de CPUs com RISC-V implementados até 2025 (RISC-V, 2019). Na era da computação aberta,

---

[1] *Contest Driven Meta Online Judge*, disponível em <https://github.com/cd-moj/cdmoj>.
[2] Sigla em inglês para microprocessador sem estágios intertravados de *pipeline*.
[3] <http://spimsimulator.sourceforge.net/>
[4] *Command-line interface*
[5] <https://riscv.org/about/>
[6] Um componente é chamado de *core*, ou núcleo do processador, se possuir uma unidade independente de busca de instruções (WATERMAN; ASANOVIC, 2019).

tanto nos mercados de consumidores quanto nas empresas, o RISC-V lidera com investimentos crescentes e previsão de existirem mais de 25 bilhões de sistema em um chip (SoC) com RISC-V até 2027 demandados por implementações de inteligência artificial (RISC-V, 2022). Recentemente a NASA selecionou a SiFive[7], uma empresa de semicondutores e chips comerciais que utilizam a ISA RISC-V, para fornecer o ecossitema das futuras missões espaciais (SIFIVE, 2022).

Para o ensino da ISA RISC-V, os principais simuladores utilizados em sala de aula são o Ripes[8] e o RARS[9]. Entretanto, enquanto o Ripes não possui a linha de comando sem dependência da interface gráfica[10] com entrada de dados implementada, o RARS possui o uso a partir da linha de comando, porém o tempo consumido para sustentar a execução do simulador[11] dificulta as avaliações feitas pelo CD-MOJ.

Esse trabalho pretende analisar e avaliar a implementação de softwares de simulação da ISA RISC-V, encontrados através da realização de uma pesquisa. Questões e critérios de inclusão e de exclusão utilizados na pesquisa por simuladores foram definidos.

Com os resultados da pesquisa, um simulador será escolhido entre os simuladores encontrados para a implementação de uma CLI. Por fim, espera-se entregar um simulador RISC-V com CLI para suporte às atividades de correção do CD-MOJ.

1.1 Justificativa

Para o funcionamento correto de um programa em linguagem de máquina, os desenvolvedores necessitam conhecer e utilizar a arquitetura do conjunto de instruções incluindo operandos e operadores, dispositivos de entrada e saída, entre outros.

Patterson e Hennessy (2005) consideram que aprender a representar instruções programando em *assembly* demonstra como o desempenho é impactado pelas otimizações dos compiladores e pelas linguagens de programação. E, no ensino do *assembly*, a utilização de simuladores pode fornecer um ambiente mais propício para a aprendizagem. Simuladores possuem interfaces que facilitam a modificação ou adição de instruções e podem detectar e examinar erros.

A automatização da correção de atividades e exercícios na disciplina Fundamentos de Arquitetura de Computadores depende de um simulador RISC-V que possua uma CLI

[7] <https://www.sifive.com/>
[8] <https://github.com/mortbopet/Ripes>
[9] <https://github.com/TheThirdOne/rars>
[10] GUI em inglês, sigla para interface gráfica do usuário.
[11] O RARS está escrito em Java e, como explicado pelo Patterson e Hennessy (2005), as instruções de programas Java (chamadas de *bytecode*) são interpretadas na máquina virtual Java (JVM) para aumentar a portabilidade. A compilação do conjunto de instruções do interpretador do Java para os conjuntos de instruções nativos causa mais impacto ao desempenho se comparado, por exemplo, com programas escritos em C.

que permita a análise acurada do tempo de execução ao ser utilizado pelo CD-MOJ. Os simuladores ISA RISC-V utilizados atualmente em sala de aula possuem interfaces gráficas úteis no ensino de *assembly* mas não possuem uma CLI pura como o simulador SPIM possui.

O desenvolvimento de um modo CLI puro para um simulador ISA RISC-V com interface gráfica utilizado em sala de aula permitiria o uso do mesmo simulador, tanto para o ensino, quanto para correção automática das atividades e exercícios.

1.2 Objetivos

1.2.1 Objetivo Geral

Disponibilizar uma ferramenta de simulação de RISC-V com CLI que possa ser utilizada junto ao CD-MOJ automatizando a correção das atividades e exercícios no ensino de *assembly* em disciplinas de arquitetura de computadores.

1.2.2 Objetivos Específicos

- Avaliar simuladores de RISC-V existentes para a implementação da CLI;
- Desenvolver ou otimizar uma CLI para um simulador ISA RISC-V.

1.3 Estrutura do Documento

Esse trabalho está dividido da seguinte forma: no Capítulo 2, é apresentado a Fundamentação Teórica; em seguida, no Capítulo 3, a Proposta de Trabalho; no Capítulo 4, Metodologia, são definidas as questões para a pesquisa de simuladores RISC-V e os critérios de inclusão e exclusão; e, no Capítulo 5, são apresentados os resultados. No Capítulo 6 é apresentada a conclusão.

2 Fundamentação Teórica

Nesse capítulo são apresentados os conceitos de arquitetura do conjunto de instruções (ISA) e os dois princípios básicos de arquitetura de computadores: o conjunto complexo de instruções (CISC) e o conjunto reduzido de instruções (RISC). Apresentaremos as arquiteturas MIPS e RISC-V que são baseadas nos princípios do conjunto reduzido de instruções, além do simulador MIPS e do juiz *online* CD-MOJ.

2.1 ISA

ISA é a sigla em inglês para arquitetura do conjunto de instruções. É uma abstração que funciona como uma interface entre o *hardware* e o *software* definindo como um processador deverá ser operado abrangendo todas as informações necessárias para executar instruções em linguagem de máquina. As instruções definem a operação a ser executada e quais operandos utilizar. Os operandos podem estar em registradores, na memória, ou na própria instrução (PATTERSON; HENNESSY, 2005).

A implementação de uma arquitetura é o *hardware* que segue a abstração de uma ISA. Sistemas computacionais complexos são projetados com *hardware* e *software* compostos por camadas. As camadas inferiores abstraem detalhes para as camadas superiores. Essa abstração permite que o mesmo *software* seja executado por diferentes implementações (PATTERSON; HENNESSY, 2005).

De acordo com Tanenbaum e Austin (2013), no final da década de 1950, a IBM, então principal empresa de computadores do mundo, introduziu o termo arquitetura para descrever a camada que permitia a compatibilidade das mesmas instruções em diferentes computadores. Executar o mesmo conjunto de instruções em diferentes máquinas representava uma vantagem para a empresa e para os clientes pois, permitia a implementação de uma única arquitetura em computadores diferentes de forma que os mesmos programas podiam ser executados em *hardwares* de velocidades e preços variados.

Como processadores só compreendem linguagem de máquina, composta por instruções formadas por sequências de números zero e números um, o *assembler*, ou montador, traduz ou interpreta o código da linguagem *assembly*, ou linguagem de montagem, para a linguagem de máquina. Instruções em *assembly* são simbólicas e possuem correspondentes diretos em linguagem de máquina (DANDAMUDI, 2005).

A arquitetura e o projeto de um processador relaciona-se diretamente com a linguagem *assembly*. Estas linguagens são chamadas de linguagem de programação de baixo nível pois suas instruções são nativas ao processador e não podem ser executadas em

processadores com arquiteturas diferentes. Ao contrário das linguagens de alto nível como C ou Java, *assembly* é dependente do sistema computacional (DANDAMUDI, 2005).

A arquitetura do conjunto de instruções de um computador pode possuir um conjunto complexo de instruções (CISC), mas também há arquiteturas que seguem os princípios do computador com conjunto reduzido de instruções (RISC). Algumas das arquiteturas existentes são a x86, JVM, MIPS, RISC-V, SPARC, entre outras.

2.1.1 CISC

Computador com conjunto complexo de instruções (CISC, Complex Instruction Set Computer) é um dos dois tipos básicos de processadores. Enquanto somar dois inteiros é um exemplo de uma instrução simples, uma instrução que copia um elemento de um *array* para outro automaticamente é um exemplo de uma instrução complexa (DANDAMUDI, 2005).

Um dos efeitos da busca por computadores mais poderosos no início dos sistemas computacionais foi o surgimento de instruções individuais complexas em substituição às instruções simples dos primeiros computadores. Havia uma lacuna semântica entre o que o *hardware* podia fazer e o que linguagens de programação de alto nível necessitavam, e a tentativa dos projetistas de preencher essa lacuna resultou na complexificação das instruções (TANENBAUM; AUSTIN, 2013).

Patterson e Ditzel (1980), no artigo *The case for the reduced instruction set computer* citam como razões que tornaram os computadores mais complexos:

- a velocidade superior do processador em relação à memória principal, o que tornava a execução de primitivas implementadas como instruções mais rápidas do que primitivas implementadas como sub-rotinas[1], assim, instruções complexas resultavam em computadores mais econômicos;
- a densidade do código, programas compactados com conjuntos de instruções complexos eram mais efetivos devido ao alto custo da memória dos primeiros computadores;
- estratégia de *marketing*, um conjunto de instruções complexas promovem uma arquitetura convencendo os clientes de que determinado projeto é superior aos projetos dos concorrentes;
- compatibilidade, um novo projeto era incrementado adicionando cada vez mais instruções sem remover as instruções anteriores, resultando em um aumento gradual da complexidade;

[1] Sub-rotinas, ou métodos, permitem que o programador crie uma nova abstração para uma operação comum (PATTERSON; HENNESSY, 2005).

- suporte para linguagens de alto nível, que conforme se popularizavam, recebiam instruções mais poderosas fornecidas pelos projetistas;
- e multiprogramação, pois para compartilhar o tempo do processador e permitir que processos em execução sejam interrompidos e depois reiniciados do ponto onde estavam é necessário manter os estados dos processos acarrentando adição de instruções e modos de endereçamento de memória.

Computadores executam apenas instruções escritas em linguagem de máquina enquanto para as pessoas é mais conveniente escrever em linguagens simbólicas. Existem duas abordagens, ou técnicas, para executar instruções escritas em linguagem simbólica *assembly* em um computador: tradução e interpretação (TANENBAUM; AUSTIN, 2013).

Na tradução, cada instrução da linguagem *assembly* é primeiramente substituída, uma por uma, pela instrução equivalente em linguagem de máquina. Então, o novo programa traduzido é executado.

Na interpretação, um programa chamado interpretador carrega e examina uma instrução de cada vez do programa escrito em *assembly* e executa diretamente a instrução equivalente em linguagem de máquina (TANENBAUM; AUSTIN, 2013). Memórias somente de leitura de rápido acesso, chamadas de memórias de controle, favoreceram a interpretação servindo para armazenar o interpretador. A interpretação permitiu o projeto de computadores mais simples com baixo custo que mesmo assim podiam executar instruções complexas. Um projeto complexo de *hardware* pode ser escrito em *software* mantendo a complexidade somente no interpretador armazenado na memória (TANENBAUM; AUSTIN, 2013).

Em projetos CISC, não há imposição de restrições de forma que há muitos modos de endereçamento de memória e as instruções possuem tamanhos variados. A arquitetura x86 da Intel, uma empresa de tecnologia mais conhecida pela fabricação de processadores, é um exemplo de arquitetura CISC (HARRIS; HARRIS, 2013).

2.1.2 RISC

A sigla em inglês RISC significa computador com conjunto reduzido de instruções. O termo RISC, em oposição ao CISC, foi criado no ano de 1980, na Universidade da Califórnia, por um grupo liderado por David Patterson e Carlo Séquin, para designar o conceito utilizado no projeto de processador com circuito VLSI[2] que não utilizava inter-

[2] VLSI é a abreviação em inglês para um circuito integrado em escala muito grande. Circuitos integrados combinam até centenas de transistores em um único *chip*. Um circuito VLSI pode combinar milhões de transistores. Transistor é basicamente um chave controlada por eletricidade que liga e desliga (PATTERSON; HENNESSY, 2005).

pretação. Os dois circuitos de processadores implementados pelo grupo foram chamados de RISC I e RISC II (TANENBAUM; AUSTIN, 2013).

RISC é um dos dois tipos básicos de processadores. Uma das principais diferenças entre RISC e CISC, é que processadores RISC utilizam instruções com tamanhos fixos. Projetos baseados em RISC usam somente instruções simples assumindo que os operandos estão sempre nos registradores e não na memória principal. Essas restrições, além de simplificarem o projeto do processador, resultam em uma melhora no desempenho das aplicações executadas (DANDAMUDI, 2005).

Para os criadores do RISC, um projeto de computador deveria ter um número pequeno de instruções simples. Essas instruções simples deveriam ser executadas em um único ciclo do caminho de dados, operando em dois registradores e armazenando o resultado da operação também em um registrador (TANENBAUM; AUSTIN, 2013).

Algumas razões que Patterson e Ditzel (1980) citam para utilizar o RISC como alternativa aos projetos CISC são:

- o menor tempo de elaboração de projetos baseados em RISC, pois, arquiteturas complexas possuem projetos de maior duração e, devido ao constante avanço da tecnologia de circuitos VLSI, projetos que demoram menos tempo para serem concluídos podem utilizar tecnologias superiores;
- e a maior viabilidade de implementação de projetos baseados em RISC pois, arquiteturas complexas são mais difíceis de implementar.

Na década de 1980, a memória principal alcançou a velocidade das memórias de controle, reduzindo, nas arquiteturas CISC, o ganho de desempenho adquirido anteriormente com a técnica da interpretação. Esse fato favoreceu os princípios RISC que começaram a ganhar espaço, substituindo arquiteturas CISC por arquiteturas mais simples, porém mais rápidas (TANENBAUM; AUSTIN, 2013).

Dandamudi (2005) explica que os princípios RISC, seguidos por muitos processadores recentes e propostos como alternativas aos projetos CISC, foram baseados em estudos empíricos com processadores CISC. Um desses estudos mostrou que compiladores utilizavam instruções simples para sintetizar instruções complexas levando os projetistas a repensarem os princípios de projeto de processadores.

Um conjunto de princípios de projeto RISC elaborados e aceitos pelos projetistas como um bom modo de fazer computadores são apresentados por Tanenbaum e Austin (2013):

- instruções devem ser executadas diretamente pelo *hardware*, sem uso de interpretadores;

- instruções devem ser fáceis de decodificar, como instruções de tamanho fixo e com número menor de formatos diferentes;
- instruções não devem acessar a memória, somente instruções de *load* e *store*[3] podem movimentar operandos entre a memória e os registradores, todas as demais instruções devem operar nos registradores;
- oferecer muitos registradores, pois o acesso a eles é muito mais rápido se comparado com o acesso à memória.

Como em arquiteturas RISC somente instruções de *load* e *store* podem acessar a memória, todas as demais instruções utilizam operandos a partir dos registradores. Por esta razão que os processadores RISC possuem uma quantidade maior de registradores se comparados com os processadores CISC (DANDAMUDI, 2005).

Tanenbaum e Austin (2013) explicam que, apesar das vantagens de desempenho da tecnologia RISC, computadores RISC não resultaram no fim das vendas de computadores CISC, primeiro, por uma questão de compatibilidade, já que as empresas já haviam investido muito em *softwares*, e segundo, pelo fato da Intel ter conseguido utilizar princípios da arquitetura RISC em arquiteturas CISC. Nessa abordagem híbrida, iniciada no processador 486, instruções mais simples são executadas em um núcleo RISC em um único ciclo do caminho de dados, resultando em um desempenho competitivo e permitindo a execução de código legado.

A proposta dos princípios RISC foi motivada pela mudança tecnológica e pelo conhecimento acumulado de projetos CISC. Os projetistas procuraram alternativas aos projetos CISC quando observaram alguns fatos: compiladores sintetizam instruções complexas em instruções simples; estruturas de dados complexas são utilizadas com pouca frequência e podem ser sintetizadas com poucos tipos de dados simples; instruções com tamanhos variados levam a decodificação e agendamentos ineficientes; e que conjuntos numerosos de registradores proveem eficiência, evitando referências à memória nas chamadas e retornos de procedimentos (DANDAMUDI, 2005).

MIPS e RISC-V são exemplos de ISA que utilizam princípios de arquitetura baseados em RISC.

2.2 MIPS

MIPS é a sigla em inglês para microprocessador sem estágios intertravados de pipeline. MIPS é um conjunto de liguagens de ISA que evoluiram a partir de um *chip*

[3] *Load* e *store* são instruções para recuperar ou armazenar um dado na memória.

fabricado em São Francisco, em 1981, por John Hennessy (TANENBAUM; AUSTIN, 2013).

Como todo processador baseado em princípios RISC, MIPS utiliza instruções exclusivas para ler e gravar dados na memória de forma que as demais instruções operam somente nos registradores. Além disso, diferente de processadores CISC, o MIPS aceita somente um modo de endereçamento de memória. Alguns outros diferentes modos de endereçamento são suportados pelo *assembler* (DANDAMUDI, 2005).

MIPS fornece 32 registradores, além de dois registradores especiais onde são armazenados os resultados das operações de multiplicação e divisão, e do registrador que armazena o contador do programa. Todos esses registradores são de 32 *bits*.

2.2.1 SPIM

SPIM é um software simulador da arquitetura MIPS com a capacidade de executar programas escritos em linguagem *assembly*. SPIM é mais lento do que um processador que implementa a arquitetura MIPS diretamente, mas não possui os custos que *hardwares* possuem. O SPIM fornece um depurador e outras ferramentas disponíveis em sistemas operacionais que auxiliam na escrita de programas em *assembly*. Há diferentes versões para diferentes sistemas operacionais e a versão mais simples é implementada em interface de linhas de comando. Versões gráficas exibem os comandos e os valores armazenados nos registradores (PATTERSON; HENNESSY, 2005).

O SPIM foi criado por James Larus e é distribuído sobre a licença BSD. A versão CLI do SPIM é escrita na linguagem de programação C e seu código fonte pode ser encontrado em <https://spimsimulator.sourceforge.net/>.

2.3 RISC-V

RISC-V é uma ISA livre e aberta que segue os princípios RISC originalmente desenvolvido em um projeto iniciado em 2010 na Universidade da Califórnia oferecendo um modelo de como uma ISA simples e moderna deve ser (PATTERSON; HENNESSY, 2018).

Em 2015, a Fundação RISC-V, que alterou seu nome em 2020 para RISC-V International[4], foi criada para manter as especificações do RISC-V.

Projetada inicialmente com fins educacionais e para ser utilizada por pesquisadores, mas tornando-se cada vez mais uma arquitetura utilizada para implementações comerciais, a recente arquitetura do conjunto de instruções RISC-V possui objetivos defi-

[4] <https://riscv.org/>

nidos em seu manual *The RISC-V Instruction Set Manual* (WATERMAN; ASANOVIC, 2019). São alguns desses objetivos:

- disponibilizar gratuitamente uma ISA de código aberto tanto para a academia quanto para a indústria;
- adequar a ISA apropriadamente para ser implementada diretamente no *hardware*;
- separar a ISA em uma ISA menor para servir como base com suporte para extensões opcionais e variantes especializadas;
- suportar o padrão IEEE-754 para ponto flutuante;
- oferecer espaços de endereços de 32 e de 64 *bits*;
- suportar implementações de *multicore* e paralelismo;
- expandir o espaço de codificação para suportar instruções de comprimento variável;
- facilitar a virtualização da ISA;

2.3.1 Conjuntos Base de Instruções

RISC-V é uma família de ISAs relacionadas. Toda implementação da ISA RISC-V possui uma ISA base onde extensões adicionais podem ser implementadas. Uma ISA base é restrita a um conjunto mínimo de instruções com números inteiros, suficientes para serem utilizadas por *assemblers*, sistemas operacionais e compiladores. Atualmente existem quatro conjuntos bases de instruções com inteiros que se diferenciam pela quantidade e tamanho dos registradores e do espaços de endereçamentos correspondentes. Esses conjuntos contém instruções computacionais com inteiros, carregamento e armazenamento de inteiros e instruções de controle de fluxo e são identificadas pela letra "I". As duas variantes principais de conjuntos base de instruções com inteiros são a RV32I e a RV64I que fornecem, respectivamente, espaços de endereçamento de 32 e 64 *bits*. A ISA base RV32E, que possui metade dos registradores da ISA base RV32I, foi adicionada para suportar pequenos microcontroladores para sistemas embarcados, e a ISA base RV128I, com suporte para espaços de endereçamento de 128 bits está sendo adicionada. Números inteiros com sinal são representados com complemento de dois nos conjuntos base de instruções com inteiros (WATERMAN; ASANOVIC, 2019).

A vantagem principal em separar os conjuntos base de instruções em diferentes ISAs é que, por não precisarem suportar operações necessárias para outros conjuntos base, cada ISA base é otimizada para suas necessidades. E a desvantagem principal de não projetar como uma ISA única é o aumento de *hardware* necessário para emular um conjunto base em outro. Entretanto, um conjunto grande com todas as instruções dos

conjuntos bases reunidas, necessitariam de *hardwares* controladores adicionais da mesma maneira. Os conjuntos base também são semelhantes entre si o bastante para reduzirem os custos de implementações que suportam múltiplas versões (WATERMAN; ASANOVIC, 2019).

2.3.2 Instruções do Conjunto Base RV32I

O conjunto base RV32I possui quatro tipos principais de formato de instruções:

- tipo R (*register-register*), um formato utilizado por instruções com 3 registradores que realizam operações lógicas e aritméticas, o conjunto base RV32I define muitas dessas instruções;
- tipo I (*short immediates*), um formato utilizado por instruções com imediatos e carregamentos;
- tipo S (*stores*), um formato utilizado por instruções de armazenamento; e
- tipo U (*long immediates*), um formato utilizado por instruções com imediatos de 20 *bits*.

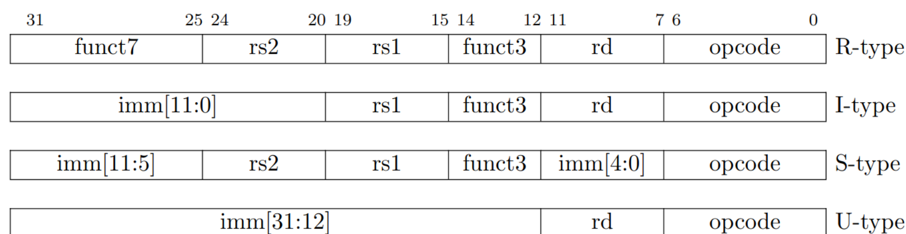

Figura 1 – Formatos de instruções do RV32I (WATERMAN; ASANOVIC, 2019)

Os quatro tipos principais de formatos de instruções do conjunto base RV32I do RISC-V estão representados na Figura 1. Os campos rotulados com *funct7* e *funct3* selecionam o tipo de operação.

Existem ainda os formatos do tipo B (*conditional branches*), uma variação do tipo S sendo chamado também de tipo SB, usado para instruções condicionais; e o tipo J (*jumps*), uma variação do tipo U sendo chamado também de tipo UJ, usado para instruções de saltos incondicionais.

Todos os formatos de instrução do conjunto base RV32I do RISC-V possuem um comprimento fixo de 32 *bits* e mantêm os registradores de origem (campos *rs1* e *rs2* na Fig. 1) e o registrador de destino (*rd*) na mesma posição simplificando a decodificação (WATERMAN; ASANOVIC, 2019).

O conjunto base RV32I utiliza 32 registradores sendo 31 deles disponíveis para uso geral e um registrador fixado com a constante 0 (zero). Esse conjunto possui 47 instruções únicas (WATERMAN; ASANOVIC, 2019). Alguns exemplos de instruções podem ser encontrados em Listagem 1.

Listagem 1 – Algumas instruções do conjunto base RV32I

```
1 add  t0, t1, t2  # tipo  R: t0 = t1 + t2
2 sub  t0, t1, t2  # tipo  R: t0 = t1 - t2
3 addi t0, t1, 42  # tipo  I: t0 = t1 + 42
4 jal  s0, foo     # tipo UJ: pula para foo e armazena a instrucao seguinte em s0
5 beq  t0, t1, foo # tipo SB: se t0 == t1; pula para foo
```

2.3.3 Extensões

Cada conjunto base de instruções com inteiros pode ser ampliada com conjuntos de instruções opcionais chamadas de extensões. Existem diferentes extensões padrão para o suporte de desenvolvimento de *software*. As principais extensões são:

- a extensão padrão de multiplicação e divisão de inteiros, identificada pela letra “M”, adiciona instruções para multiplicar e dividir números inteiros;
- a extensão padrão de instruções atômicas, identificada pela letra “A”, adiciona instruções de acesso a memória para sincronização entre processadores;
- a extensão padrão de ponto flutuante de precisão simples, identificada pela letra “F”, adiciona o suporte para registradores e instruções de ponto flutuante com precisão simples;
- a extensão padrão de ponto flutuante de precisão dupla, identificada pela letra “D”, expande os registradores e adiciona instruções de ponto flutuante com precisão dupla;
- a extensão padrão de ponto flutuante de precisão quádrupla, identificada pela letra “Q”, expande os registradores e adiciona instruções de ponto flutuante com precisão quádrupla (128 *bits*);
- a extensão padrão de instruções compactadas, identificada pela letra “C”, fornece versões menores, de 16 *bits*, de instruções comuns.

Diferente de arquiteturas que mudam para um nova versão conforme novas instruções são adicionadas, a intenção dos projetistas do RISC-V é manter seus conjuntos base e suas extensões sem modificações ao longo do tempo, e disponibilizar novos conjuntos de instruções em futuras extensões (WATERMAN; ASANOVIC, 2019).

2.4 CD-MOJ

Com o objetivo de oferecer um ambiente de submissão de algoritmos onde os professores possam configurar competições ou listas de exercícios para seus alunos, o CD-MOJ é um meta juiz *online* escrito em *Shell Script* que envia os códigos submetidos para outros juízes *online* (RIBAS, 2016).

O CD-MOJ, implementado pelo Prof. Dr. Bruno Cesar Ribas, foi criado inicialmente para ser utilizado no treinamento de equipes da Maratona de Programação da Universidade Tecnológica Federal do Paraná, campus Pato Branco (RIBAS, 2013).

Na Universidade de Brasília, o CD-MOJ funciona como uma plataforma de apoio em disciplinas de programação e, atualmente, possui um repositório de problemas para as disciplinas de Algoritmos e Programação de Computadores, Compiladores, Estrutura de Dados I e II, Fundamentos de Arquitetura de Computadores e Fundamentos de Sistemas Operacionais (SILVA, 2022).

3 Proposta de Trabalho

O presente trabalho pretende desenvolver uma interface de linha de comando (CLI) de um simulador com uma arquitetura do conjunto de instruções (ISA) RISC-V que possa ser utilizado pelo CD-MOJ na automatização das atividades e exercícios de disciplinas de arquitetura de computadores. Essa interface deverá ser similar à CLI do SPIM, um simulador ISA MIPS já utilizado no ensino de *assembly*.

Para a execução desta proposta de trabalho, após o levantamento e estudo da bibliografia, foi realizada uma pesquisa por simuladores ISA RISC-V.

Os simuladores encontrados foram avaliados considerando perguntas elaboradas de acordo com critérios de inclusão e exclusão. Esses critérios foram definidos com base nas seguintes premissas:

- O código do simulador deve estar disponível publicamente, em um repositório para que possa ser acessado, avaliado e utilizado no desenvolvimento da CLI;
- A licença do simulador deve permitir a livre utilização e modificação do código, para que o simulador com a CLI implementada possa ser utilizado em sala de aula;
- O simulador deve estar escrito em uma linguagem que permita a avaliação do tempo de execução pelo CD-MOJ;
- O simulador deve possuir comunidade e desenvolvedores ativos permitindo contato e contribuição no desenvolvimento;
- O simulador deve possuir uma interface gráfica útil no ensino da ISA RISC-V.

As perguntas elaboradas para a pesquisa de simuladores foram respondidas para cada simulador encontrado e alguns simuladores foram escolhidos para serem analisados. O resultado da pesquisa será apresentado com uma discussão sobre os simuladores escolhidos.

Uma comparação do tempo de execução entre os simuladores SPIM e RARS foi realizada evidenciando a inviabilidade do uso do RARS, um simulador RISC-V, pelo CD-MOJ na automatização da correção de atividades e exercícios.

Será feita uma análise sobre a implementação da CLI no SPIM e os simuladores escolhidos na pesquisa terão seus códigos analisados. Dessa forma, espera-se reunir informações suficientes para determinar a escolha de um simulador para a implementação da CLI.

3.1 CLI

A CLI deverá possuir suporte para entrada padrão nos programas executados semelhante ao SPIM, como visto na Fig. 2.

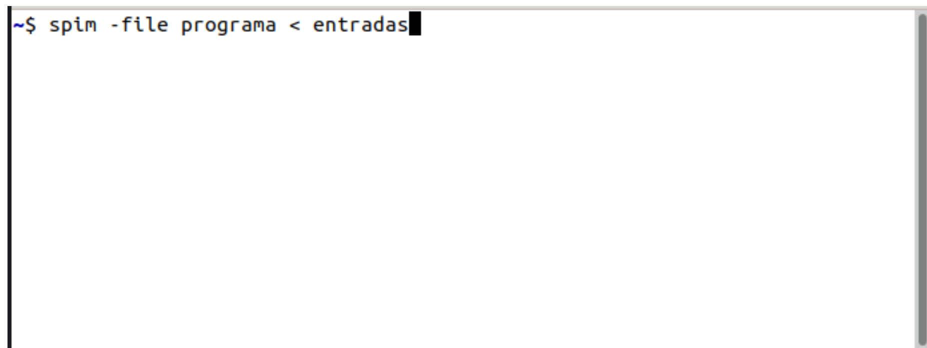

Figura 2 – Execução de um programa no SPIM com entrada de dados

A CLI será desenvolvida independente da interface gráfica do simulador selecionado possuindo suas saídas e métricas próprias adaptadas à utilização pelo CD-MOJ.

A CLI também deverá permitir a análise acurada do tempo de execução com latência mínima no estabelecimento do ambiente para sustentar a execução do simulador.

A comparação do tempo de execução realizada entre os simuladores SPIM e RARS utilizou um programa equivalente escrito em MIPS e em RISC-V e executados com as mesmas entradas.

O SPIM, descrito na Seção 2.2.1, é um simulador escrito em C com uma CLI implementada que aceita dados de entrada nos programas executados sendo suportado pelo CD-MOJ.

O RARS é um simulador RISC-V escrito em Java que possui uma CLI que aceita dados de entrada, mas seu uso junto ao CD-MOJ é prejudicado pela latência e por eventuais *overheads* observadas durante a execução.

Uma comparação do tempo de execução entre os simuladores SPIM e o RARS foi realizada.

Espera-se que a CLI desenvolvida para um simulador RISC-V ao final deste trabalho possua um tempo de execução próximo ao tempo de execução do SPIM para um programa equivalente.

4 Metodologia

4.1 Metodologia do Desenvolvimento

O desenvolvimento deste trabalho está dividido em duas etapas. Na primeira etapa, com a proposta definida, foram realizadas uma pesquisa bibliográfica sobre o tema e uma pesquisa por simuladores RISC-V. Foi feita uma pesquisa por simuladores e, após a análise inicial dos resultados da pesquisa, foi realizada uma comparação do tempo de execução de um programa equivalente entre os simuladores SPIM e RARS.

Na segunda etapa deste trabalho, são feitas análises do simulador SPIM, utilizado em disciplinas de arquiteturas de computadores no ensino de MIPS, e dos simuladores selecionados utilizando os critérios e questões de pesquisa por simuladores. Entre os simuladores selecionados, após as análises, o simulador escolhido para o desenvolvimento ou otimização da interface de linha de comando tem o tempo de execução comparado e avaliado com outros simuladores. Um *shell script* é utilizado na coleta das métricas executando diferentes métodos de medição do tempo de execução. Um programa escrito com base no Crivo de Eratóstenes é utilizado pelo *script* para mensurar os simuladores.

4.1.1 Comparação entre SPIM e RARS

Para a comparação do tempo de execução entre os simuladores SPIM e RARS foi utilizado um programa equivalente escrito em MIPS e em RISC-V. Eles foram executados em seus respectivos simuladores com as mesmas entradas.

O comando para a execução no SPIM encontra-se na Fig. 2 e o comando para a execução no RARS encontra-se na Fig. 3.

4.1.2 Pesquisa por Simuladores

Com o intuito de encontrar um simulador RISC-V onde possa ser implementada uma CLI para ser utilizada pelo CD-MOJ são definidos critérios de inclusão e exclusão e identificadas questões para a pesquisa de simuladores. A estratégia utilizada para análise inicial e seleção desses simuladores também é apresentada.

4.1.2.1 Critérios de Inclusão e Exclusão

O conjunto de critérios de inclusão e exclusão para analisar e identificar os simuladores mais relevantes entre aqueles encontrados foi formulado considerando as premissas descritas no Capítulo 3.

Figura 3 – Execução de um programa no RARS com entrada de dados

Os critérios de inclusão definidos são apresentados na Tabela 1. A primeira coluna representa um identificador para aquele críterio.

Tabela 1 – Critérios de inclusão de simuladores

Identificador	Critério de Inclusão
CI1	O código do simulador está disponível
CI2	O código pode ser modificado e distribuído
CI3	O simulador possui interface gráfica (GUI)

A Tabela 2 apresenta os critérios de exclusão.

Tabela 2 – Critérios de exclusão de simuladores

Identificador	Critério de Exclusão
CE1	O simulador não foi desenvolvido com fins educacionais
CE2	A linguagem possui latência incerta no estabelecimento do ambiente
CE3	Repositório inativo há mais de dois anos

4.1.2.2 Questões para a Pesquisa por Simuladores

Considerando os critérios de inclusão e exclusão, foram definidas cinco questões para a pesquisa de simuladores RISC-V. As questões e os critérios avaliados por cada uma delas é apresentada na Tabela 3.

4.1.2.3 Estratégia de Análise

As questões listadas na Tabela 3 foram respondidas para cada um dos simuladores encontrados e são apresentadas no Capítulo 5.

Tabela 3 – Questões para a pesquisa de simuladores

Identificador	Questão	Critério avaliado
QP1	Possui repositório?	CI1
QP2	Possui fins educacionais?	CE1
QP3	Qual a licença atribuída?	CI2
QP4	Possui GUI?	CI3
QP5	Em qual linguagem o simulador está escrito?	CE2
QP6	O último *commit* foi há menos de dois anos?	CE3

Para responder a primeira questão foram realizadas buscas por simuladores de RISC-V e por seus repositórios e códigos-fontes, que foram então analisados para responder as demais perguntas.

4.1.3 Coleta das Métricas do Tempo de Execução

As medições do tempo de execução podem ser usadas para caracterizar o comportamento dinâmico e auxiliar na identificação de trechos de código que precisam ser otimizados (STEWART, 2001).

Aém disso, neste trabalho, as medições do tempo de execução são utilizadas para comparar diferentes simuladores. Também são coletadas os tempos de execução dos algoritmos utilizados escrito em C e compilados com GCC[1].

Na coleta das métricas do tempo de execução dos simuladores escolhidos, um programa escrito com base no Crivo de Eratóstenes e um algoritmo de ordenação com complexidade quadrática serão utilizados com os métodos de coletar métricas do tempo de execução.

4.1.3.1 Crivo de Eratóstenes

Eratóstenes foi um matemático grego que viveu há mais de dois mil anos e o Crivo de Eratóstenes é um algoritmo que determina se cada um dos inteiros positivos começando pelo número 2 até um inteiro positivo dado como limite é número primo ou não (SORENSON, 1990).

Esse algoritmo funciona da seguinte maneira (SORENSON, 1990):

- um *array* do tamanho do limite deve ser iniciado com todas as posições em 1;
- para cada primo p menor ou igual que a raiz quadrada do limite (começando com p igual a 2), a posição m do *array* deve ser setada como 0 para todos os múltiplos m do p, comecando por p^2;

[1] A compilação dos algoritmos em C foi realizada utilizando a *flag* -O0.

- ao final, na posição n do *array* (começando com $n = 2$) teremos 1 se o número for primo.

4.1.3.2 *Selection Sort*

O algoritmo *Selection Sort* também será utilizado para medir o tempo de execução dos simuladores. Esse algoritmo possui uma complexidade quadrática e foi implementado em C e depois adaptado para RISC-V e MIPS e podem ser encontrados no repositório do GitHub[2].

Foram geradas três listas com números desordenados com quantidades de cem, mil e dois mil inteiros, respectivamente. Estas listas são utilizadas como entradas das execuções do algoritmo *selection sort* nas suas diferentes versões.

4.1.3.3 Medindo o Tempo de Execução

Para medir o tempo de execução, existem diferentes métodos que podem variar na precisão, granularidade[3] e dificuldade em obter-se as medidas. Os métodos com maior granularidade são utilizados para medir o tempo de execução de todo um programa ou processo não interativos (STEWART, 2001).

Não podemos dizer que um dos métodos é melhor que todos os demais. Escolher um método de medição do tempo de execução depende do motivo da medição e, também, dos recursos de *hardware* e das ferramentas de instrumentação disponíveis. Para analisar o desempenho em tempo real, pode-se utilizar métodos com maior granularidade. Para medições com o intuito de otimizar o código, tanto métodos com maior ou menor granularidade podem ser utilizados (STEWART, 2001).

Os métodos com maior granularidade apresentados por Stewart (2001) são:

- *stop-watch*, uma técnica simples com um precisão típica de 5×10^{-1} segundos. Consiste em utilizar um relógio iniciando o cronômetro quando o programa inicia. Quando o programa terminar a execução, para-se o cronômetro;

- comando *date*, usado como um cronômetro mas utiliza o relógio embutido no computador. Sua precisão típica é de 2×10^{-1} segundos. O uso mais comum é com um *shell script* executando o comando *date* antes e depois da execução do programa medido[4]; e

[2] Disponível em <https://github.com/dadamos-tcc/scripts>

[3] Nível da medição, pode ser desde o programa inteiro, um processo, uma função, um pequeno trecho de código ou até mesmo única instrução (STEWART, 2001).

[4] O comando *date*, assim como o método *stop-watch*, fornece uma estimativa do tempo de execução do programa completo sem calcular preempção, interrupções ou operações de entrada e saída (STEWART, 2001).

- comando *time*, com um precisão típica de 2×10^{-1} segundos. Sua utilização é feita acrescentando o comando *time* antes do comando que deseja-se medir o tempo de execução. Além do tempo entre o início e o fim do programa executado, o comando *time* considera preempção, interrupções ou operações de entrada e saída.

O comando *time* está embutido na maioria dos *shells*, mas também temos disponível o *software* (comando GNU) */usr/bin/time*. Os dois apresentam resultados que podem diferir pois são implementações diferentes que utilizam maneiras diferentes de medir o tempo de execução. O comando *time* embutido no *shell* oferece menos recursos se comparado com o comando */usr/bin/time* (GNU, 2018).

As principais medidas que o *time* e o */usr/bin/time* fornecem, de acordo com Stewart (2001) são:

- o tempo gasto pela CPU excluído o tempo gasto com preempção, interrupções ou operações de entrada e saída;
- o tempo gasto pelo sistema incluindo o tempo de execução de drivers de dispositivo, manipuladores de interrupções ou demais chamadas de sistema associadas;
- o tempo total do programa, do início ao fim, seja em execução, bloqueado ou aguardando na fila. No comando *time* é apresentado no campo chamado *real* e no comando */usr/bin/time* é apresentado no campo *Elapsed (wall clock) time*; e
- a porcentagem média do tempo de CPU, apresentada somente pelo comando */usr/-bin/time*, uma medida que depende da quantidade dos demais processos executados pela CPU sendo de pouca relevância para a medição do tempo de execução.

Um método que pode ser utilizado para otimizar códigos apresentado por Stewart (2001) consiste em utilizar *softwares* analisadores. Esses *softwares* são ferramentas projetadas especificamente para medir o tempo de execução e a granularidade das medidas depende da implementação de cada analizador.

Para coletar métricas do tempo de execução, os métodos utilizados são o comando *date*, o comando *time* e o */usr/bin/time*, além da utilização de um *software* analisador para medir o tempo de execução dos métodos considerando a pilha de chamadas.

4.2 Políticas de Desenvolvimento do Projeto

Serão seguidas as políticas de contribuição do simulador escolhido após análise dos simuladores selecionados na pesquisa. Caso o simulador escolhido não possua as políticas de contribuição com as políticas de desenvolvimento definidas, será utilizado um repositório privado na plataforma GitHub.

5 Resultados

Os resultados da pesquisa por simuladores RISC-V são apresentados nesse capítulo seguido dos resultados das análises e trabalhos realizados com os simuladores selecionados a partir da pesquisa. São apresentados os programas escritos com base no Crivo de Eratóstenes utilizados para coletar as medidas dos tempos de execução de um programa. Para realizar comparações entre diferentes simuladores foi escrito um *shell script* que utiliza dois métodos, e uma variação de um dos métodos, de medição do tempo de execução. Uma comparação entre os tempos de execução dos simuladores SPIM e RARS foi realizada. As otimizações realizadas no simulador escolhido e os resultados são apresentados.

5.1 Resultados da Pesquisa por Simuladores

A Tabela 4 foi elaborada ao analisar cada um dos simuladores encontrados considerando os critérios e as questões definidas anteriormente. A discussão sobre os resultados é apresentada em seguida.

Tabela 4 – Resultados da pesquisa de simuladores

Simulador	QP1	QP2	QP3	QP4	QP5	QP6
DBT-RISE-RISCV	sim	não	-	-	-	-
emulsiV	sim	sim	MPL-2.0	sim	JavaScript	sim
FireSim	sim	não	-	-	-	-
jor1k	sim	não	-	-	-	-
Jupiter	sim	sim	GPL-3.0	sim	Java	não
MARSS-RISCV	sim	não	-	-	-	-
RARS	sim	sim	MIT	sim	Java	sim
Ripes	sim	sim	MIT	sim	C++	sim
RISC-V Virtual Prototype	sim	sim	MIT	não	C++	sim
riscv-simulator	sim	sim	(sem licença)	não	C++	não
riscv-vm	sim	sim	MIT	não	C	não
riscvOVPsim	sim	não	-	-	-	-
riscvOVPsimPlus	não	-	-	-	-	-
rv32emulator	sim	não	-	-	-	-
rv8	sim	sim	MIT	não	C++	não
Spike	sim	não	-	-	-	-

Simuladores que não possuam repositório (QP1) ou não tenham fins educacionais (QP2) não tiveram as demais questões respondidas devidos aos critérios de inclusão e exclusão.

5.2 Discussão dos Resultados da Pesquisa por Simuladores

Dos 16 simuladores encontrados, um não possui repositório (riscvOVPsimPlus). 7 simuladores encontrados não possuem fins educacionais sendo 2 deles usados para testar *softwares* para *chips* (riscvOVPsim e MARSS-RISCV), um para executar em instâncias FPGAs na AWS (FireSim), um para ser usado em implementação própria (DBT-RISE-RISCV), um online para executar em navegadores (jor1k), um emula um sistema completamente (Spike) e um executa instruções simples somente no CLI (rv32emulator).

8 simuladores encontrados possuem fins educacionais sendo que em metade deles o último *commit* foi há mais de dois anos (Jupiter, riscv-simulator, riscv-vm e rv8). Dos demais, um deles é feito para ser utilizado em navegadores web (emulsiV) e um não possui interface gráfica (RISC-V Virtual Prototype).

De acordo com o resultado das pesquisas por simuladores RISC-V atendendo aos critérios de inclusão e exclusão, os simuladores mais adequados para implementação de uma CLI propícia para ser utilizada junto ao CD-MOJ são o Ripes e o RARS. Entre os simuladores selecionados, optou-se por otimizar o simulador RARS.

5.2.1 Simulador Ripes

O Ripes[1], um simulador de arquitetura de computadores RISC-V, é um *software* de código fonte aberto escrito em C++ focado na visualização das tarefas executadas por um processador ao executar um programa. A simulação feita pelo Ripes inclui o *datapath* do processador, acesso ao cache, aos registradores e às instruções carregadas na memória fornecendo uma visualização interativa. Ripes é considerado um simulador visual de microarquitetura adequado para o ensino dos fundamentos da arquitetura de computadores (PETERSEN, 2021).

Durante a execução de uma simulação, a troca e coleta de informações do estado do modelo do processador estão muito entrelaçadas com a interface gráfica. Por esse motivo, o simulador Ripes não foi escolhido.

5.2.2 Simulador RARS

O RARS, (*RISC-V Assembler, Simulator, and Runtime*), é um simulador RISC-V de código fonte aberto escrito em Java e construído a partir do MARS, um simulador da arquitetura MIPS. O RARS está disponível em <https://github.com/TheThirdOne/rars> e seu principal objetivo é ser um ambiente propício para quem está iniciando nos estudos do RISC-V. Além da interface gráfica, o RARS já possui uma interface de linha de comando mas com um tempo de execução maior (na casa de duas ordens de grandeza)

[1] <https://github.com/mortbopet/Ripes>

do que um programa equivalente em MIPS com as mesmas entradas executado pelo SPIM como pode ser visto na Seção 5.5.

5.3 Programas para Calcular Primos com Crivo de Eratóstenes

Um programa em RISC-V e um equivalente em MIPS foram escritos a partir do Crivo de Eratóstenes para serem utilizados na coleta de medidas dos tempos de execução.

Apoiado na definição de algoritmo do Crivo de Eratóstenes exposto por Sorenson (1990) e discutido na Seção 4.1.3.1, foi elaborado um programa em C que implementa uma versão desse algoritmo que pode ser encontrado no Apêndice A.

O programa em C serviu como base para a elaboração do algoritmo do Crivo de Eratóstenes em RISC-V que, depois, foi adaptado para MIPS. O programa em RISC-V pode ser encontrado no Apêndice B.

5.4 Shell Script para Medir o Tempo de Execução

Utilizou-se um *shell script* para automatizar a coleta e organização das medidas do tempo de execução considerando diferentes métodos. Esse *script* gera tabelas para comparar o tempo de execução do mesmo programa em diferentes simuladores e foi escrito considerando os métodos de medição apresentados por Stewart (2001) e expostos na Seção 4.1.3.3.

O *shell script* escrito para automatizar a coleta das medidas do tempo de execução de programas em *assembly* pode executar mais de um programa por vez, com suas respectivas entradas e versões MIPS e RISC-V. Cada um dos programas listados no *shell script* são usados na medição do tempo de execução dos simuladores. São feitas coletas das medidas do tempo de execução por programa em cada simulador utilizando */usr/bin/time* como método de medição do tempo de execução. O comando *date* é utilizado na coleta dos algoritmos executados escritos em C devido ao fato de tempos de execução muito próximos de zero não serem mensurados adequadamente pelo */usr/bin/time*. Também é possível definir no *script* o números de repetições que cada execução será feita. Os resultados obtidos com as medições são salvos em um arquivo organizados em tabelas. A Listagem 2 apresenta algumas linhas desse *shell script* que foi utilizado para coletar as medidas do tempo de execução a cada tentativa de otimização.

O *shell script* completo pode ser encontratado no Apêndice C e suas adaptações podem ser encontradas no repositório do GitHub[2].

[2] Disponível em <https://github.com/dadamos-tcc/scripts>

Listagem 2 – Trecho do shell script para coletar medidas dos tempos de execução

```
# date
start=`date +%s.%3N`
for (( run = 1; run <= $repeticao; run++ ))
do
    java -jar rars.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
done
end=`date +%s.%3N`; date=$( echo "$end - $start" | bc -l )
# time
(time (for (( run = 1; run <= $repeticao; run++ )); do
    java -jar rars.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
done)) 2> output_time
time=`grep real output_time | cut -f2 -d $'\x09'`
# /usr/bin/time
/usr/bin/time -vo output_bin_time bash -c "for (( run = 1; run <= $repeticao; run++ )); do
    java -jar rars.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
done"
bin_time=`grep 'wall clock' output_bin_time | cut -f8 -d' '`
```

5.4.1 Configurações do Ambiente de Execução do Shell Script

Para executar a coleta das medidas do tempo de execução de programas escritos em *assembly* em diferentes simuladores, o computador utilizado possui um processador modelo *Intel Xeon CPU E3-1505M* de 2.80 GHz com 4 núcleos (cada um com duas *threads* de execução) e 32 *gigabytes* de memória RAM DDR4. O sistema operacional utilizado é o Ubuntu 20.04 (64-*bit*).

5.5 Comparação entre SPIM e RARS

Os programas utilizados para a comparação dos tempos de execução entre os simuladores SPIM e RARS foram o algoritmo escrito a partir do Crivo de Eratóstenes e o algoritmo *selection sort*, ambos com suas versões equivalentes tanto em MIPS quanto em RISC-V.

O comando */usr/bin/time* foi utilizado inicialmente para comparar o tempo de execução do algoritmo do Crivo de Eratóstenes entre os simuladores SPIM e RARS, mas como muitas das medidas coletadas para o simulador SPIM apresentaram zero como valor de *wall clock* para entradas com números primos de 2 e 8 *bits*, optou-se por utilizar o comando *date*. O manual do */usr/bin/time* explica que alguns valores podem ser relatados como zero quando o tempo de execução de um comando é muito próximo de zero (CANONICAL, 2019).

Para a comparação entre os simuladores SPIM e RARS, optou-se por calcular a média do tempo de 10 execuções para os algoritmos do Crivo de Eratóstenes e do *selection sort*. A partir das médias, foram calculados os desvios-padrão para cada execução. No Crivo de Eratóstenes, as entradas utilizadas foram os maiores números primos de 2, 8 e 14 *bits* e no *selection sort* foram utilizadas três listas com numéros desordenados com cem, mil e dois mil inteiros respectivamente.

Para calcular o desvio-padrão, calculam-se os desvios médios subtraindo-se a média de cada um dos valores coletados. Depois, é calculado a variância, que é a soma da média do quadrado dos desvios. A raiz quadrada da variância é o desvio-padrão. A variância quantifica a dispersão dos valores em torno da média e tem como unidade de medida o quadrado da unidade de medida da variável, já o desvio-padrão possui a mesma unidade de medida da variável. Quanto maior o valor do desvio-padrão, mais disperso da média estarão as medidas coletas (MORETTIN; BUSSAB, 2010).

Tabela 5 – SPIM e RARS, média do crivo com *date*

crivo	entrada 2 bits	entrada 8 bits	entrada 14 bits
Algoritmo em C	0,0018 s	0,0019 s	0,0071 s
SPIM	0,0090 s	0,0094 s	0,4958 s
RARS	0,4165 s	0,4229 s	0,7871 s

Tabela 6 – SPIM e RARS, desvio-padrão do crivo com *date*

crivo	entrada 2 bits	entrada 8 bits	entrada 14 bits
Algoritmo em C	0,0004 s	0,0005 s	0,0007 s
SPIM	0,0154 s	0,0004 s	0,0224 s
RARS	0,0676 s	0,0422 s	0,0811 s

A média do tempo de execução do programa com base no Crivo de Eratóstenes na Tabela 5 no SPIM, um simulador MIPS escrito em C, é menor em duas ordens de grandeza em comparação com a média do tempo de execução no RARS, um simulador de RISC-V escrito em Java, para as entradas com 2 e 8 *bits*. Para a entrada de 14 *bits*, a ordem de grandeza da média do tempo de execução nos dois simuladores se iguala. Na Tabela 6, o desvio-padrão aumenta para a entrada de 14 *bits* no SPIM indicando uma dispersão maior. Entendemos que, devido ao alto uso que o programa com base no Crivo de Eratóstenes faz de chamadas de sistemas para imprimir os números primos, quanto maior for a entrada fornecida como limite, mais o desempenho do SPIM aproxima-se do desempenho do RARS.

Assim, foi utilizado uma adaptação do algoritmo do Crivo de Eratóstenes nas coletas realizadas de medidas do tempo de execução. Nesta adaptação, somente a primalidade do número fornecido como entrada é verificada. No crivo adaptado, as entradas utilizadas foram os maiores números primos de 2, 8, 14, 20 e 26 *bits*.

Na Tabela 7, temos as médias das medidas coletadas para um conjunto de 10 execuções por entrada e, na Tabela 8, os desvios-padrão para essas medidas. Na Tabela 7, a média do tempo de execução do SPIM é menor em duas ordens de grandeza, em comparação com a média do tempo de execução no RARS, até a entrada com 14 *bits*, mas, na Tabela 8, o desvio-padrão é duas ordens de grandeza menor até a entrada com 26

Tabela 7 – SPIM e RARS, média com crivo adaptado com *date*

crivo adaptado	entradas				
	2 bits	8 bits	14 bits	20 bits	26 bits
Algoritmo em C	0,0020 s	0,0019 s	0,0020 s	0,0017 s	0,0020 s
SPIM	0,0026 s	0,0026 s	0,0038 s	0,0106 s	0,0583 s
RARS	0,4030 s	0,4170 s	0,4065 s	0,4415 s	0,4368 s

Tabela 8 – SPIM e RARS, desvio-padrão com crivo adaptado com *date*

crivo adaptado	entradas				
	2 bits	8 bits	14 bits	20 bits	26 bits
Algoritmo em C	0,0003 s	0,0005 s	0,0005 s	0,0004 s	0,0004 s
SPIM	0,0004 s	0,0004 s	0,0004 s	0,0014 s	0,0007 s
RARS	0,0407 s	0,0464 s	0,0474 s	0,0522 s	0,0443 s

bits. No programa do crivo adaptado para verificar a primalidade da entrada fornecida, os tempos de execução são mais uniformes se comparados com o programa que exibe todos os primos até o limite fornecido.

Tabela 9 – SPIM e RARS, média com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,0036 s	0,0033 s	0,0074 s
SPIM	0,0910 s	7,8234 s	31,2778 s
RARS	0,4821 s	5,1792 s	19,2142 s

Tabela 10 – SPIM e RARS, desvio-padrão com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,0005 s	0,0004 s	0,0004 s
SPIM	0,0044 s	0,1110 s	0,2158 s
RARS	0,0375 s	0,0304 s	0,1030 s

Nas Tabelas 9 e 10, o simulador SPIM apresenta uma média e desvio-padrão dos tempos de execução maiores que os do RARS quando se executa a implementação do algoritmo *selection sort* de complexidade quadrática para listas desordenadas com dois mil inteiros.

5.6 Otimização do Simulador RARS

As otimizações realizadas no simulador RARS, os princípios utilizados e os resultados obtidos são expostos nesta seção.

Ao realizar otimizações, é importante medir o desempenho do *software* antes e depois de cada tentativa de otimização efetuada, pois muitas das tentativas de otimizações não surtem efeito (BLOCH, 2018).

Nem sempre é fácil descobrir qual trecho do código do *software* está consumindo mais tempo para ser executado. Os trechos onde os esforços de otimização devem ser realizados podem ser encontrados com a ajuda de ferramentas de *profiling* (BLOCH, 2018).

Ferramentas de *profiling* são *softwares* analisadores que podem possuir diferentes níveis de granularidade. Utilizar *profilings* para mensurar o tempo de execução de um programa, ou de partes dele, é um dos métodos citados por Stewart (2001).

Criado pela empresa EJ-Technologies, o JProfiler é um *software* para analisar programas escritos em Java e, neste trabalho, foi utilizado para encontrar trechos de código do RARS onde o esforço de otimização deve ser empenhado.

O JProfiler é uma ferramenta profissional que analisa a JVM em tempo de execução. Uma de suas principais funcionalidades é analisar as chamadas de métodos, conhecida como *CPU profiling*. Essa funcionalidade ajuda a entender o funcionamento da aplicação e ajuda a encontrar formas de melhorar o desempenho da mesma (EJ-TECHNOLOGIES, 2018).

O *CPU profiling* mede o tempo de execução dos métodos considerando a pilha de chamadas. Todas as chamadas de métodos observadas pelo JProfiler são acumuladas em uma árvore. Além dos métodos, também é possível visualizar a árvore de chamadas agregadas em nível de classes ou de pacotes (EJ-TECHNOLOGIES, 2018).

5.6.1 Medidas Iniciais do RARS

A árvore de chamadas do RARS sem alterações no código foi mensurado com o JProfiler executando inicialmente o algoritmo escrito em RISC-V do Crivo de Eratóstenes. O número de entrada utilizado como limite foi 500.000. A árvore de chamadas e as métricas coletadas resultante da análise do JProfiler pode ser vista na Figura 4.

Podemos confirmar, na Figura 4, que as chamadas de sistema utilizadas para imprimir na tela os números primos até o limite fornecido consomem a maior parte do tempo de execução do programa. Dessa maneira, para evidenciar os trechos de código passíveis de serem otimizados, foi utilizado a adaptação do algoritmo do Crivo de Eratóstenes que verifica a primalidade da entrada fornecida.

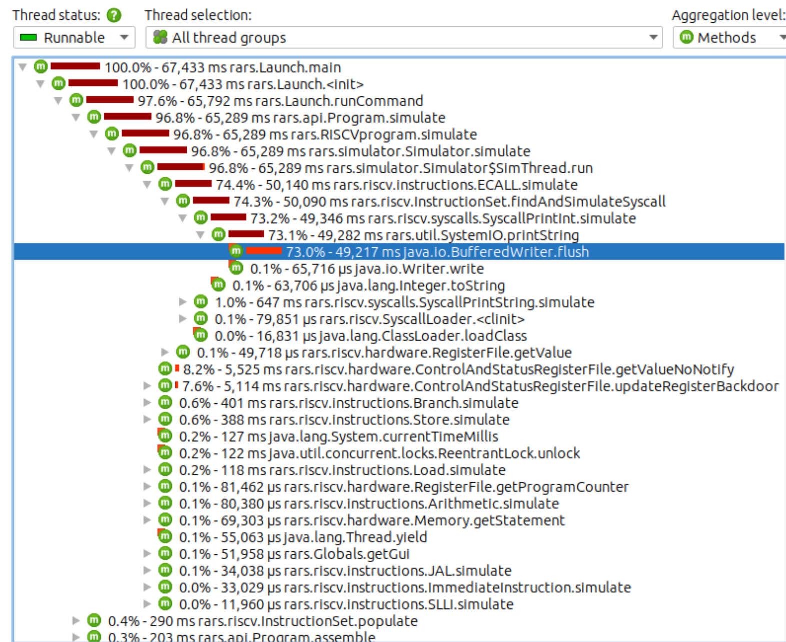

Figura 4 – Árvore de chamadas por métodos com crivo gerada no JProfiler

Utilizando o algoritmo do crivo adaptado no RARS sem alterações com um número primo de 31 *bits* como entrada, foram geradas árvores de chamadas no JProfiler. As árvores de chamadas encontram-se agregadas por métodos na Figura 5, por classes na Figura 6 e por pacotes na Figura 7.

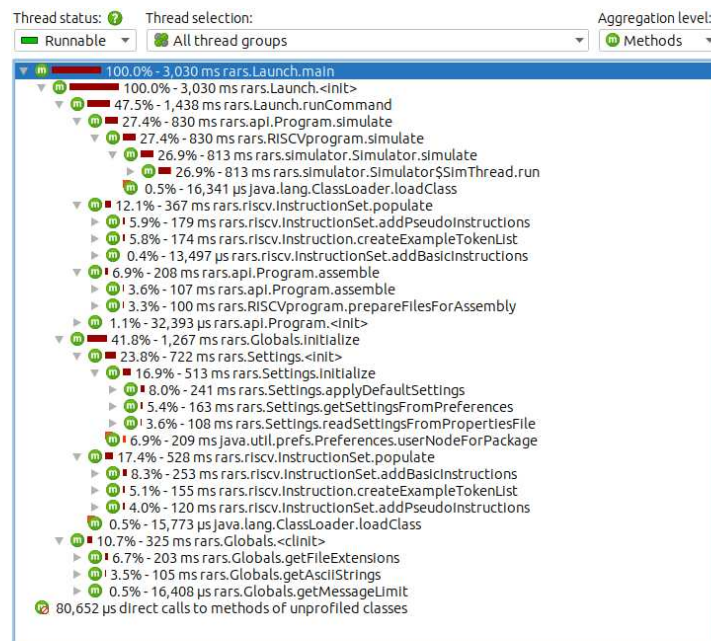

Figura 5 – Árvore de chamadas por métodos com o programa crivo adaptado no RARS sem alterações gerada no JProfiler

Observando as Figuras 5 e 6, notamos que a classe *Globals* e seus métodos consumiram mais da metade do tempo de execução. Na Figura 7, notamos que mesmo sendo

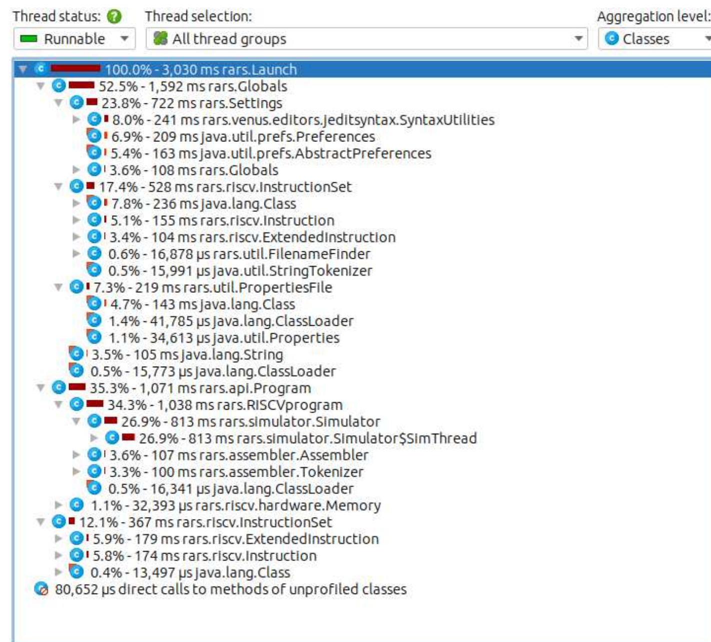

Figura 6 – Árvore de chamadas por classes com o programa crivo adaptado no RARS sem alterações gerada no JProfiler

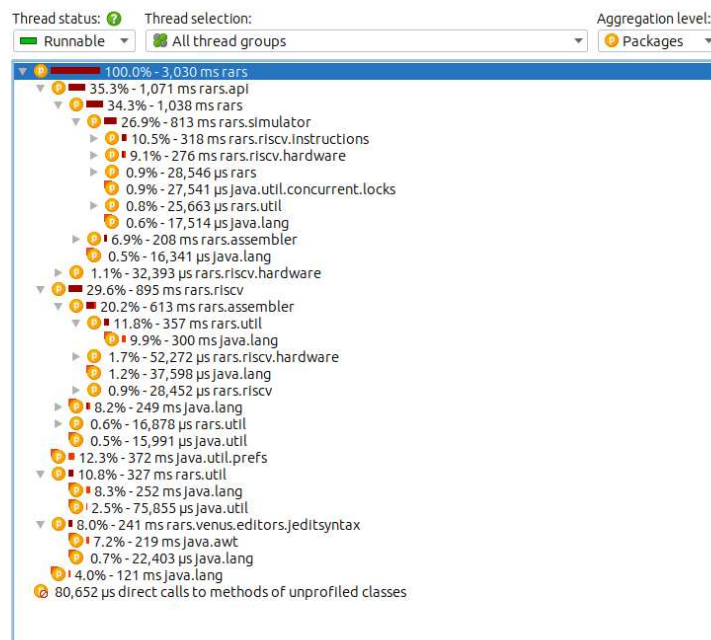

Figura 7 – Árvore de chamadas por pacotes com o programa crivo adaptado no RARS sem alterações gerada no JProfiler

executado pela linha de comando, o RARS sem alterações consome tempo de execução com bibliotecas de pacotes gráficos.

O repositório do RARS possui *scripts* para criar o arquivo *.jar* e para executar testes no programa. Após clonar o repositório, para gerar o *.jar* o *script* build-jar.sh deve ser executado e para os testes o *script* test.sh. Os testes foram executados para cada novo

.jar criado com as otimizações realizadas.

```
$ git clone https://github.com/TheThirdOne/rars --recursive
Cloning into 'rars'...
remote: Enumerating objects: 5156, done.
remote: Counting objects: 100% (365/365), done.
remote: Compressing objects: 100% (172/172), done.
remote: Total 5156 (delta 245), reused 267 (delta 192), pack-reused 4791
Receiving objects: 100% (5156/5156), 4.66 MiB | 7.12 MiB/s, done.
Resolving deltas: 100% (4257/4257), done.
Submodule 'src/jsoftfloat' (https://github.com/thethirdone/jsoftfloat/) registered for path 'src/jsoftfloat'
Cloning into '/home/dadamos/tcc2/rars/src/jsoftfloat'...
remote: Enumerating objects: 351, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 351 (delta 0), reused 5 (delta 0), pack-reused 344
Receiving objects: 100% (351/351), 59.96 KiB | 653.00 KiB/s, done.
Resolving deltas: 100% (141/141), done.
Submodule path 'src/jsoftfloat': checked out '75c3a5d1ab1322ce4dde0b5994d6f9f6ff820529'
$ cd rars/
$ ./build-jar.sh
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ ./test.sh
..X...................................................................................................
Crashed while executing ./test/selfmod.s

Unexpected number of psuedo-instructions skipped.
$ 
```

Figura 8 – Sequência de comandos executados para clonar, criar e testar o RARS

Na Figura 8, é possível observar que um dos testes falha com o *.jar* criado após o repositório ser clonado e sem nenhuma alteração nas linhas de código ser realizada. Foi aberta uma *issue* no repositório do projeto RARS[3] relatando este problema. Nos demais testes e em diversos programas executados, as saídas ocorreram conforme o esperado.

5.6.2 Otimizações Realizadas

Para cada tentativa de otimização realizada no RARS, o *shell script* foi executado para coletar as medidas de tempo e gerar as tabelas. As medidas de tempo de execução foram coletadas para o algoritmo em C, para o RARS sem alterações no código e para o RARS com modificações para otimizar a interface de linha de comando. Nas tabelas onde o RARS modificado com alterações aparece, ele é chamado de RARS-CLI. O programa utilizado pelo *shell script* foi o *selection sort*.

Quais otimizações e onde realizá-las foram decididas a partir da análise da árvore de chamadas e dos tempos de execução do RARS sem alterações coletados pelo JProfiler, além das sugestões propostas por Bloch (2018). O foco inicial da otimização foram as partes relevantes do sistema. Ao final das alterações realizadas para otimizar a CLI, os resultados dos tempos de execução do RARS-CLI[4] foram verificados com o JProfiler para serem comparados com os tempos de execução do RARS sem alterações.

[3] <https://github.com/TheThirdOne/rars/issues/183>
[4] Disponível no repositório do GitHub <https://github.com/dadamos-tcc/rars-cli>

5.6.2.1 Removendo bibliotecas gráficas

Na Figura 5, é possível observar que em torno de 25 por cento do tempo de execução é consumido pelos métodos chamados pela classe *Settings*. De acordo com a Figura 6, uma parte do tempo consumido pela classe *Settings* é utilizado por classes da biblioteca gráfica Venus ao carregar configurações do ambiente gráfico. E, na Figura 7, podemos ver que a biblioteca Venus está consumindo o pacote AWT utilizado para desenvolver bibliotecas gráficas.

Dessa forma, a primeira tentativa de otimização consistiu em remover as chamadas para pacotes gráficos na classe *Settings*. Foram removidas as demais referências encontradas para os pacotes Venus, AWT e Swing.

O resultado da coleta das medidas do tempo de execução encontram-se na Tabela 11, na qual foi possível notar que houve uma redução no tempo de execução do RARS-CLI quando comparado com o RARS sem alterações.

Tabela 11 – Removendo bibliotecas gráficas - média com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,005 s	0,010 s	0,019 s
RARS	0,456 s	5,024 s	8,642 s
RARS-CLI	0,400 s	4,747 s	7,766 s

5.6.2.2 Removendo configurações não utilizadas

Analisando a classe *Settings*, notamos que mesmo no modo CLI, o RARS sem modificações carrega configurações utilizadas pela interface gráfica e pela IDE embutida, o que também pode ser visto nas Figuras 5 e 6. Dessa forma, foram removidas as configurações não utilizadas.

Tabela 12 – Removendo configurações não utilizadas - média com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,004 s	0,010 s	0,014 s
RARS	0,447 s	5,016 s	8,572 s
RARS-CLI	0,341 s	4,705 s	7,760 s

Na Tabela 12, é possível observar que houve redução no tempo de execução do RARS-CLI em comparação com a Tabela 11.

5.6.2.3 Removendo bibliotecas e funcionalidades não utilizadas

Foram removidas as bibliotecas *Observables*, *Observers*, *Notices* e suas dependências, assim como chamadas de sistemas para *dialogs* utilizadas somente pela interface gráfica do RARS. Também foi removida a funcionalidade de retornar para instruções já executadas (*BackStepper*), que não é utilizada pela linha de comando. Notamos na Tabela 13 que houve um ganho no tempo de execução do RARS-CLI.

Tabela 13 – Removendo bibliotecas e funcionalidades não utilizadas - média com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,004 s	0,010 s	0,010 s
RARS	0,450 s	5,013 s	8,574 s
RARS-CLI	0,292 s	3,775 s	4,105 s

5.6.2.4 Aplicando sugestões do livro *Effective Java*

De acordo com Bloch (2018), escrever bons programas em Java resulta em programas com menor tempo de execução. Alguns itens, que apresentam boas práticas derivadas de princípios fundamentais como clareza e simplicidade (BLOCH, 2018), foram aplicados na tentativa de otimizar o CLI do RARS.

Devem ser evitados os usos de finalizadores como o método *System.gc* pois, no Java, quando um objeto se torna inacessível, o *garbage collector* é responsável por liberar o armazenamento ocupado por esse objeto sem a necessidade de intervenção do programador (BLOCH, 2018).

No método *initialize* da classe *Memory*, foi removido uma chamada para o método *System.gc*.

Raw types não devem ser utilizados pois podem levar a exceções em tempo de execução sendo fornecidos por questões de compatibilidade e interoperabilidade antes da introdução de genéricos no Java. Uma alternativa segura é utilizar o *wildcard type* (BLOCH, 2018).

Os *raw types* encontrados nas classes *InstructionSet*, *SyscallLoader* e *DumpFormatLoader*, foram substituídos por *wildcard types*.

Bloch (2018) explica que o escopo das variáveis locais deve ser minimizado e que os *loops* apresentam uma oportunidade para isso. É preferível o *for loop* em vez do *while loop*, pois o *for loop* permite declarar variáveis utilizadas no *loop* limitando seu escopo para onde são necessárias.

Os *While loops* nas classes *Globals*, *ExtendedInstruction* e *FilenameFinder* foram substituídos por *for loops* de forma que as variáveis de interação tiveram seu escopo limitado. Na Listagem 3, temos um exemplo de um *while loop* encontrado na classe *Globals*. O resultado da substituição por um *for loop* pode ser visto na Listagem. 4.

Listagem 3 – *While loop* na classe *Globals*

```
1 Enumeration keys = properties.keys();
2 while (keys.hasMoreElements()) {
3     String key = (String) keys.nextElement();
4     overrides.add(new SyscallNumberOverride(key, properties.getProperty(key)));
5 }
```

Listagem 4 – *For loop* substituindo o *while loop* na classe *Globals*

```
1 for (Enumeration<Object> keys = properties.keys(); keys.hasMoreElements(); ) {
2     String key = (String) keys.nextElement();
3     overrides.add(new SyscallNumberOverride(key, properties.getProperty(key)));
4 }
```

Por questões de clareza, flexibilidade e prevenção de erros, é mais vantajoso utilizar um *for-each loop* do que um *loop* tradicional (BLOCH, 2018). O *for-each loop* permite a iteração em um tipo definido sem necessitar da implementação de um *Iterator* próprio.

Na Listagem 5, temos um *loop* na classe *Launch* que depende de uma implementação de um *Iterator* próprio. Na Listagem 6, o resultado da substituição por um *for-each loop*. A implementação do *Iterator* foi removida por não ser mais necessária. Um *loop* da classe *MemoryConfigurations* também foi substituído por um *for-each loop* e teve a implementação do seu *Iterator* removida.

Listagem 5 – *Loop* na classe *Launch*

```
1 Iterator<String> memIter = memoryDisplayList.iterator();
2 int addressStart = 0, addressEnd = 0;
3 while (memIter.hasNext()) {
```

Listagem 6 – *For-each loop* substituindo o *loop* na classe *Launch*

```
1 int addressStart = 0, addressEnd = 0;
2 for (String memIter : memoryDisplayList ) {
```

A classe *Settings*, utilizada no RARS sem alterações para carregar as configurações padrão ou carregar as configurações feitas na interface gráfica a partir de um arquivo, também possui algumas configurações que podem ser ajustadas pela linha de comando. Para reduzir instâncias de objetos e dependências, as configurações que podem ser feitas pela linha de comando foram implementadas na classe *Globals* mantendo a capacidade de serem configuradas no RARS-CLI.

As médias das medidas dos tempos de execução coletadas após a aplicação de alguns dos princípios sugeridos por Bloch (2018) e da remoção de bibliotecas não mais utilizadas são exibidas na Tabela 14. Podemos observar um ganho no tempo de execução do RARS-CLI em comparação com o RARS sem modificações. Na Tabela 15 temos os desvios-padrão.

Tabela 14 – Aplicando sugestões do livro *Effective Java* - média com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,005 s	0,010 s	0,013 s
RARS	0,442 s	5,045 s	8,725 s
RARS-CLI	0,272 s	3,709 s	3,958 s

Tabela 15 – Aplicando sugestões do livro *Effective Java* - desvio-padrão com *selection sort*

selection sort	quantidade de números para ordenar		
	cem	mil	dois mil
Algoritmo em C	0,0009 s	0,0004 s	0,0056 s
RARS	0,0227 s	0,0628 s	0,2034 s
RARS-CLI	0,0153 s	0,0544 s	0,0895 s

Nas Tabelas 16 e 17, temos a comparação entre o RARS e RARS-CLI das médias e dos desvios-padrão utilizando o algortimo do crivo adaptado após a aplicação de alguns dos princípios sugeridos por Bloch (2018) e da remoção de bibliotecas não mais utilizadas. Notamos que o ganho no tempo de execução foi maior para um algoritmo com complexidade quadrática.

Tabela 16 – Aplicando sugestões do livro *Effective Java* - média com crivo adaptado

crivo adaptado	entradas				
	2 bits	8 bits	14 bits	20 bits	26 bits
Algoritmo em C	0,0059 s	0,0054 s	0,0056 s	0,0055 s	0,0057 s
RARS	0,355 s	0,351 s	0,357 s	0,355 s	0,404 s
RARS-CLI	0,189 s	0,184 s	0,189 s	0,200 s	0,214 s

Na Figura 9, com a árvore de chamadas agregadas por métodos do RARS-CLI utilizando o crivo adaptado, é possível observar que houve uma redução do tempo de execução se comparado com o RARS sem modificações na Figura 5. O método *runCommmand* da classe *Launch*, que no RARS consome quase metade do tempo de execução, no RARS-CLI passou a consumir por volta de um terço. O método *simulate* da classe *Program*, chamado pelo método *runCommmand*, continua consumindo por volta de um quarto do tempo de

Tabela 17 – Aplicando sugestões do livro *Effective Java* - desvio-padrão com crivo adaptado

crivo adaptado	entradas				
	2 bits	8 bits	14 bits	20 bits	26 bits
Algoritmo em C	0,0008 s	0,0004 s	0,0004 s	0,0005 s	0,0004 s
RARS	0,0102 s	0,0157 s	0,0184 s	0,0102 s	0,0261 s
RARS-CLI	0,0083 s	0,0066 s	0,0104 s	0,0100 s	0,0048 s

execução no RAR-CLI. É possivel notar também que nenhum método da classe *Settings* é chamado pelo *initialize* da *Globals*.

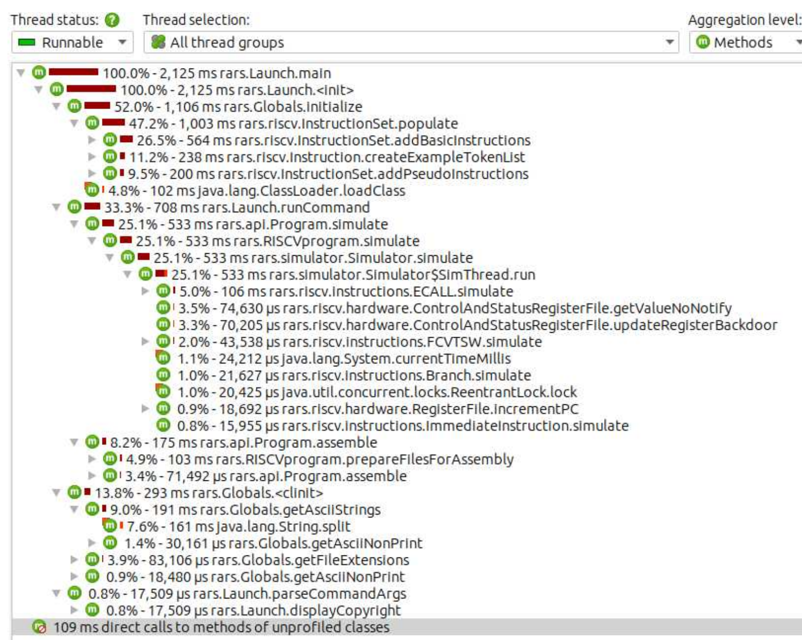

Figura 9 – Árvore de chamadas por métodos com o programa crivo adaptado no RARS-CLI gerada no JProfiler

5.6.3 Comparação entre RARS e RARS-CLI

Os programas utilizados para a comparação da média e do desvio-padrão do tempo de execução entre os simuladores RARS sem alterações e o RARS-CLI, com as otimizações implementadas, foram o algoritmo do crivo adaptado e o *selection sort*.

Utilizando as médias e desvios-padrão coletados anteriormente para a versão em C dos algoritmos, o simulador SPIM, o RARS sem alterações e o RARS-CLI com as últimas otimizações implementadas, foram gerados um gráfico para cada algoritmo utilizado com suas respectivas entradas. Na Figura 10, é apresentado o gráfico do crivo adaptado, e na Figura 11, é apresentado o gráfico do *selection sort*.

A média dos tempos de execução para cada entrada é menor no RARS-CLI tanto

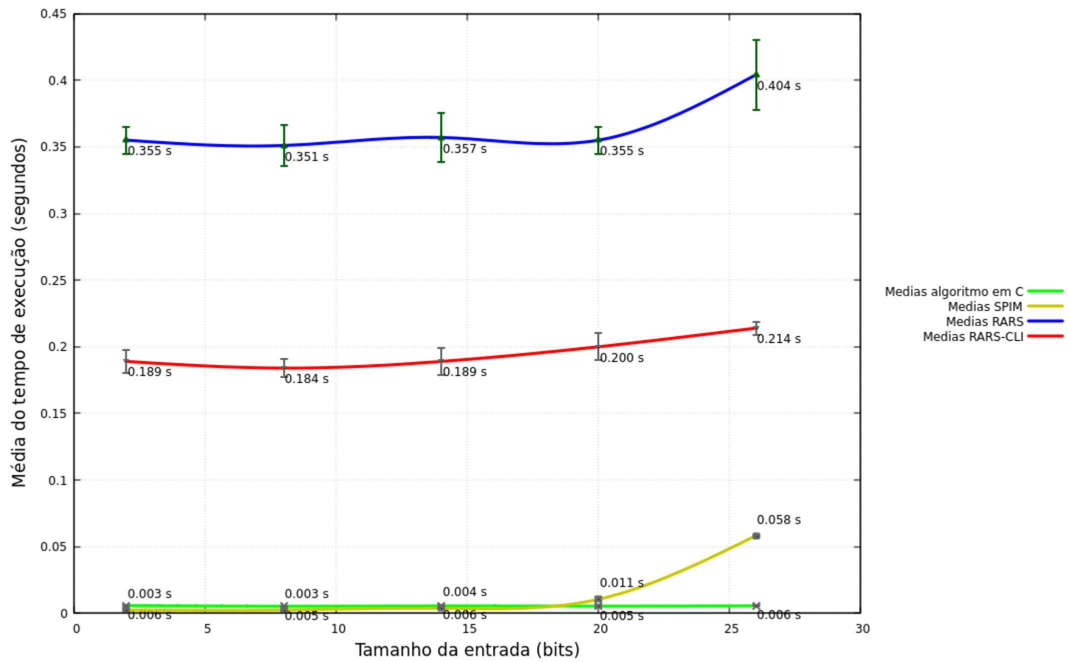

Figura 10 – Gráfico de comparação das médias e desvios-padrão com crivo adaptado

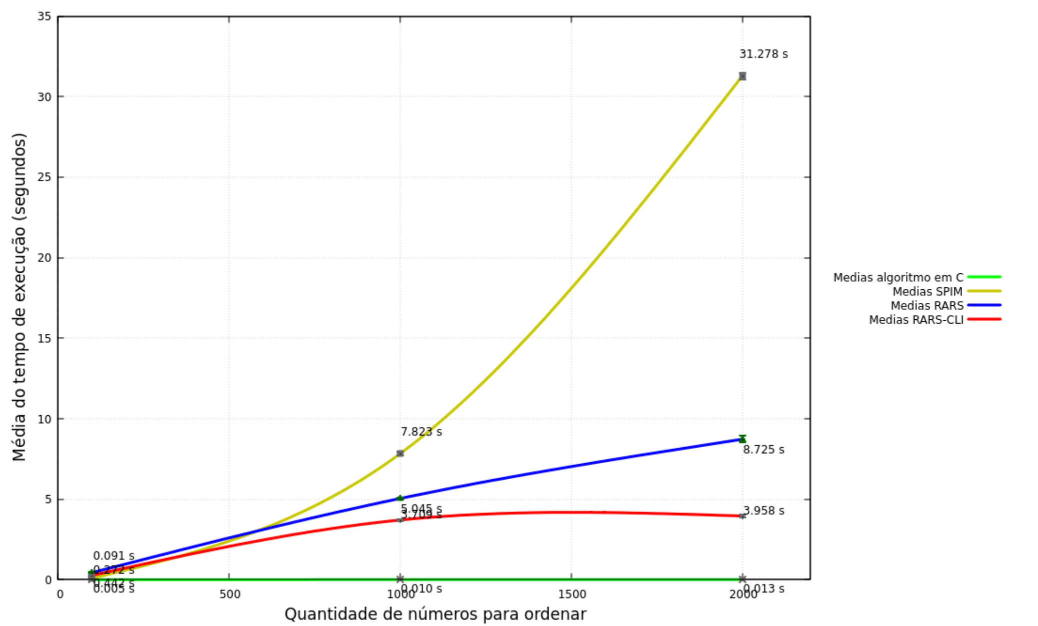

Figura 11 – Gráfico de comparação das médias e desvios-padrão com *selection sort*

com o crivo adaptado quanto no *selection sort* como visto nos gráficos das Figuras 10 e 11.

Considerando as medidas dos tempos de execução coletadas do RARS sem alterações e do RARS-CLI com as otimizações propostas implementadas, com o crivo adaptado houve uma redução em torno de 38 por cento no tempo consumido pelo RARS-CLI e com o *selection sort* houve uma redução em torno de 48 por cento no tempo consumido pelo RARS-CLI. Para calcular essa redução, foi feita a média das médias coletadas para cada

simulador para cada um dos algoritmos.

5.7 Executável Nativo

É possivel utilizar a ferramenta GraalVM[5] para criar executáveis nativos para aplicações escritas em Java. Desenvolvido pela *Oracle Labs*, o GraalVM é uma plataforma que otimiza aplicações Java tornando-as independentes da JVM e reduzindo o uso de memória e de CPU (GRAALVM, 2023).

A partir do *jar* do RARS-CLI, a ferramenta GraalVM foi utilizada para criar um executável nativo. Na Figura 12, é apresentado o comando executado e a sua saída. Uma desvantagem notada foi o tamanho do executável comparado com o *jar*. Enquanto o *RARS-CLI.jar* possui o tamanho de aproximadamente 1,3 MB, o executável nativo gerado pelo GraalVM possui 15,1 MB.

```
$ native-image -H:IncludeResources=".*/*.(txt|properties)$" --no-fallback -jar rars-cli.jar
========================================================================================================
GraalVM Native Image: Generating 'rars-cli' (executable)...
========================================================================================================
[1/7] Initializing...                                                                  (3,4s @ 0,20GB)
 Version info: 'GraalVM 22.3.1 Java 17 CE'
 Java version info: '17.0.6+10-jvmci-22.3-b13'
 C compiler: gcc (linux, x86_64, 9.4.0)
 Garbage collector: Serial GC
[2/7] Performing analysis...  [*****]                                                 (12,5s @ 1,22GB)
   2.974 (74,37%) of  3.999 classes reachable
   3.788 (52,60%) of  7.202 fields reachable
  13.325 (43,72%) of 30.479 methods reachable
      28 classes,     0 fields, and   316 methods registered for reflection
      59 classes,    59 fields, and    52 methods registered for JNI access
       4 native libraries: dl, pthread, rt, z
[3/7] Building universe...                                                             (1,5s @ 1,74GB)
[4/7] Parsing methods...      [*]                                                      (1,1s @ 0,97GB)
[5/7] Inlining methods...     [***]                                                    (0,7s @ 1,38GB)
[6/7] Compiling methods...    [***]                                                    (9,6s @ 0,44GB)
[7/7] Creating image...                                                                (1,5s @ 0,87GB)
   4,49MB (31,23%) for code area:     7.571 compilation units
   9,34MB (64,94%) for image heap:  104.339 objects and 200 resources
 564,37KB ( 3,83%) for other data
  14,39MB in total
--------------------------------------------------------------------------------------------------------
Top 10 packages in code area:                          Top 10 object types in image heap:
 685,87KB java.util                                       1,83MB byte[] for embedded resources
 336,69KB java.lang                                     994,93KB byte[] for code metadata
 264,68KB java.text                                     971,13KB java.lang.String
 216,40KB java.util.regex                               885,45KB byte[] for general heap data
 195,24KB java.util.concurrent                          652,87KB java.lang.Class
 149,57KB rars.assembler                                588,85KB byte[] for java.lang.String
 146,96KB java.math                                     484,73KB java.util.HashMap$Node
 117,08KB java.lang.invoke                              232,34KB c.o.svm.core.hub.DynamicHubCompanion
 114,90KB com.oracle.svm.core.genscavenge               231,83KB java.util.HashMap$Node[]
 103,73KB java.util.logging                             173,73KB java.lang.String[]
   2,16MB for 127 more packages                           1,63MB for 790 more object types
--------------------------------------------------------------------------------------------------------
                0,5s (1,6% of total time) in 18 GCs | Peak RSS: 3,22GB | CPU load: 6,07
--------------------------------------------------------------------------------------------------------
Produced artifacts:
 /home/dadamos/tcc/my_rars/rars/rars-cli (executable)
 /home/dadamos/tcc/my_rars/rars/rars-cli.build_artifacts.txt (txt)
========================================================================================================
Finished generating 'rars-cli' in 31,8s.
$
```

Figura 12 – Criando um executável nativo com GraalVM

O executável nativo do RARS-CLI não reconheceu as instruções dos programas RISC-V. De acordo com a *issue Native executable with JDK 9*[6], o carregamento das instruções não funciona adequadamente devido a forma como está implementado no RARS.

[5] <https://www.graalvm.org/>.
[6] <https://github.com/TheThirdOne/rars/issues/4>.

Nesta mesma *issue* do repositório do RARS, houve uma decisão por não corrigir o carregamento de instruções para funcionar com métodos nativos.

6 Conclusão

Medir os efeitos das tentativas de otimizações em aplicações escritas em Java é mais necessário do que em aplicações escritas em linguagens como C e C++ pois, de acordo com Bloch (2018), o custo das operações básicas é menos definido no Java. É difícil prever os efeitos das otimizações pois, no Java, o *gap* de abstração entre o que o programador escreve e o que a CPU executa é maior do que em outras linguagens. O modelo de desempenho do Java não é claramente definido e pode variar de acordo com a implementação, *release* e processador utilizado.

Métodos da aplicação e trechos de códigos críticos para o desempenho podem ser escritos em linguagens como C e C++ através da JNI (*Java Native Interface*), a interface nativa do Java, mas não são recomendados por não terem garantia de ganho e por possuirem desvantagens. De acordo com Bloch (2018), nas primeiras versões do Java, o uso dos chamados métodos nativos muitas vezes era necessário, mas a JVM ficou mais eficiente nas versões mais recentes. Os métodos nativos possuem a desvantagem de não serem seguros quanto ao vazamento de memória, além de reduzirem a portabilidade da aplicação e serem mais difícies de depurar. As chamadas para os métodos nativos também possuem um custo associado.

Com a GraalVM, é possivel criar executáveis nativos otimizados a partir de aplicações Java. O executável nativo do RARS-CLI foi gerado mas não foi possível verificar o resultado da otimização devido a um problema na implementação do carregamento de instruções.

O trabalho cumpriu o seu objetivo ao otimizar o tempo de execução de uma interface de linha de comando para um simulador RISC-V. Os resultados obtidos também indicam que a continuação desta proposta de trabalho pode conseguir reduzir ainda mais os tempos de execução do RARS-CLI.

6.1 Trabalhos futuros

Depois das otimizações realizadas, com os métodos críticos identificados com a ajuda de uma ferramenta de *profiler*, os algoritmos utilizados podem ser analisados e substituídos por algoritmos mais eficientes. Outras boas práticas e princípios propostos por Bloch (2018) podem ser aplicados.

Arquivos de configurações, instruções e chamadas de sistemas estão implementados no RARS sendo carregadas por uma *ClassLoader*. Modificar a implementação do carregamento desses recursos pode resultar em uma melhora no tempo de execução e

também permitiria a criação de um executável nativo.

Referências

BLOCH, J. *Effective Java*. 3th. ed. Boston, MA: Addison-Wesley, 2018. ISBN 978-0-13-468599-1. Citado 5 vezes nas páginas 40, 43, 45, 46 e 51.

CANONICAL. *MAN TIME(1)*. 2019. Disponível em: <https://manpages.ubuntu.com/manpages/xenial/man1/time.1.html>. Acesso: 8 jul. 2023. Citado na página 37.

DANDAMUDI, S. P. *Introduction to Assembly Language Programming*. 2th. ed. USA: Springer, 2005. ISBN 0-387-20636-1. Citado 5 vezes nas páginas 17, 18, 20, 21 e 22.

EJ-TECHNOLOGIES. *Introduction To JProfiler*. 2018. Disponível em: <https://www.ej-technologies.com/resources/jprofiler/v/13.0/help/doc/main/introduction.html>. Acesso: 20 mai. 2023. Citado na página 40.

GNU. *GNU Time*. 2018. Disponível em: <https://www.gnu.org/software/time/>. Acesso: 30 mai. 2023. Citado na página 33.

GRAALVM. *Why GraalVM?* 2023. Disponível em: <https://www.graalvm.org/why-graalvm/>. Acesso: 04 jul. 2023. Citado na página 49.

HARRIS, D. M.; HARRIS, S. L. *Digital Design and Computer Architecture*. 2th. ed. USA: Morgan Kaufmann, 2013. ISBN 978-0-12-394424-5. Citado 2 vezes nas páginas 14 e 19.

MORETTIN, P. A.; BUSSAB, W. de O. *Estatística básica*. 6. ed. São Paulo: Saraiva, 2010. ISBN 978-85-02-08177-2. Citado na página 38.

PATTERSON, D. A.; DITZEL, D. R. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, Association for Computing Machinery, New York, NY, USA, v. 8, n. 6, p. 25–33, oct 1980. ISSN 0163-5964. Disponível em: <https://doi.org/10.1145/641914.641917>. Citado 2 vezes nas páginas 18 e 20.

PATTERSON, D. A.; HENNESSY, J. L. *Organização e Projeto de Computadores: a Interface Hardware/Software*. 3a. ed. Rio de Janeiro: Elsevier Editora, 2005. ISBN 978-85-352-1521-2. Citado 6 vezes nas páginas 14, 15, 17, 18, 19 e 22.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: the Hardware/Software Interface: RISC-V Edition*. USA: Morgan Kaufmann, 2018. ISBN 78-0-12-812275-4. Citado na página 22.

PETERSEN, M. B. Ripes: A visual computer architecture simulator. In: IEEE. *2021 ACM/IEEE Workshop on Computer Architecture Education (WCAE)*. [S.l.], 2021. p. 1–8. Citado na página 35.

RIBAS, B. C. *Sobre o CD-MOJ*. 2013. Disponível em: <https://moj.naquadah.com.br/about.shtml>. Acesso: 30 jun. 2023. Citado na página 25.

RIBAS, B. C. *A Contest Driven Meta Online Judge*. 2016. Disponível em: <https://github.com/cd-moj/cdmoj>. Acesso: 30 jun. 2023. Citado na página 25.

RISC-V. *Semico Forecasts Strong Growth for RISC-V*. 2019. Disponível em: <https://riscv.org/announcements/2019/11/9679/>. Acesso: 10 set. 2022. Citado na página 14.

RISC-V. *Semico Research's New Report Predicts There Will Be 25 Billion RISC-V-Based AI SoCs By 2027*. 2022. Disponível em: <https://riscv.org/blog/2022/02/semico-researchs-new-report-predicts-there-will-be-25-billion-risc-v-based-ai-socs-by-2027/>. Acesso: 10 set. 2022. Citado na página 15.

SIFIVE. *NASA Selects SiFive and Makes RISC-V the Go-to Ecosystem for Future Space Missions*. 2022. Disponível em: <https://www.sifive.com/press/nasa-selects-sifive-and-makes-risc-v-the-go-to-ecosystem>. Acesso: 10 set. 2022. Citado na página 15.

SILVA, L. dos S. Cd-moj: Contribuições para melhorias no sistema. 2022. Citado na página 26.

SORENSON, J. *An introduction to prime number sieves*. [S.l.: s.n.], 1990. Citado 2 vezes nas páginas 31 e 36.

STEWART, D. B. Measuring execution time and real-time performance. In: CITESEER. *Embedded Systems Conference (ESC)*. [S.l.], 2001. v. 141. Citado 5 vezes nas páginas 31, 32, 33, 36 e 40.

TANENBAUM, A. S.; AUSTIN, T. *Organização estruturada de computadores*. São Paulo: Pearson Prentice Hall, 2013. ISBN 978-85-8143-539-8. Citado 5 vezes nas páginas 17, 18, 19, 20 e 21.

WATERMAN, A.; ASANOVIC, K. (Ed.). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*. RISC-V Foundation, Dezembro, 2019. Citado 5 vezes nas páginas 14, 22, 23, 24 e 25.

Apêndices

APÊNDICE A – Programa em C

Listagem 7 – Programa em C com base no Crivo de Eratóstenes

```
#include <stdio.h>

int main() {

    int limite, p, m;
    do
        scanf("%d", &limite);
    while (limite < 2);

    int lista[limite + 1];
    for (p = 2; p <= limite; p++)
        lista[p] = 1;

    for (p = 2; p * p <= limite; p++)
        if (lista[p])
            for (m = p * p; m <= limite; m += p)
                lista[m] = 0;

    for (p = 2; p <= limite; p++)
        if (lista[p])
            printf("%d\n", p);

    return 0;
}
```

APÊNDICE B – Programa em RISC-V

Listagem 8 – Programa em RISC-V com base no Crivo de Eratóstenes

```
.data
espaco: .asciz  " "
lista:  .word   0

# s0: limite
# s1: lista
# s2: p
# s3: p * p
# s4: m
# s5: 1
.text
main:
    li   a7, 5                  # a7 eh o servico
    ecall                       # le um inteiro em a0
    mv   s0, a0                 # armazena o inteiro em s0

# inicia lista
    li   s5, 1
    la   s1, lista              # posicao inicial da lista
    li   s2, 2                  # posicao inicial p = 2
inicia_lista:
    blt  s0, s2, inicia_crivo   # limite < posicao corrente (p <= limite)
    sw   s5, 8(s1)              # lista[p] = 1; 8() eh para comecar em lista[2]
    addi s2, s2, 1              # p++
    addi s1, s1, 4              # proxima posicao da lista, cada word tem tamanho 4
    j    inicia_lista

# crivo de Eratostenes
inicia_crivo:
    la   s1, lista              # posicao inicial da lista
    li   s2, 2                  # posicao inicial p = 2
crivo:
    mul  s3, s2, s2             # p * p
    blt  s0, s3, inicia_imprime # limite < quadrado da posicao corrente (p * p <= limite)
    mv   s4, s3                 # m = p * p
    slli t0, s2, 2              # posicao corrente vezes o tamanho da word (p * 4)
    add  s1, s1, t0             # proxima posicao da lista (primeira posicao sera 8, lista
         [2])
    lw   t0, 0(s1)              # lista[p]
    beq  t0, s5, marca_nao_primo # if (lista[p])
loop_crivo:
    la   s1, lista              # vai para o inicio da lista
    addi s2, s2, 1              # p++
    j    crivo
marca_nao_primo:
    blt  s0, s4, loop_crivo     # limite < posicao corrente (m <= limite)
    la   s1, lista              # volta para o comeco da lista
    slli t1, s4, 2              # posicao corrente vezes o tamanho da word (m * 4)
    add  s1, s1, t1             # proxima posicao da lista
    sw   zero, 0(s1)            # lista[m] = 0
```

```
50      add  s4, s4, s2              # m += p
51      j    marca_nao_primo
52
53  # imprime primos
54  inicia_imprime:
55      la   s1, lista               # posicao inicial da lista
56      li   s2, 2                   # posicao inicial p = 2
57  imprime_lista:
58      blt  s0, s2, encerra         # limite < posicao corrente (p <= limite)
59      lw   t0, 8(s1)               # 8() eh para comecar em lista[2]
60      beq  t0, s5, imprimir        # if (lista[p])
61      j    loop_imprime
62  imprimir:
63      mv   a0, s2                  # copia a posicao corrente para a0
64      li   a7, 1                   # a7 eh o servico, 1 imprime inteiros
65      ecall                        # imprime a0
66      la   a0, espaco              # a0 eh a entrada do ecall
67      li   a7, 4                   # a7 eh o servico, 4 imprime strings
68      ecall                        # imprime a0
69  loop_imprime:
70      addi s2, s2, 1               # p++
71      addi s1, s1, 4               # proxima posicao da lista, cada word tem tamanho 4
72      j    imprime_lista
73
74  # return 0;
75  encerra:
76      mv   a0, zero                # copia 0 para a0, sera o retorno do encerramento do programa
77      li   a7, 93                  # a7 eh o servico, 93 encerra o programa
78      ecall                        # encerra o programa
```

APÊNDICE C – Shell Script

Listagem 9 – Shell Script para coletar medidas dos tempos de execução

```
#!/bin/bash

# a proxima linha define os programas que serao executadas com suas entradas e repeticoes
programas="crivo,entrada,1-1-10 exercicio,entradas,1-1-10"

for programa in $programas; do
    arquivo=$(echo $programa | cut -f1 -d,)
    entrada=$(echo $programa | cut -f2 -d,)
    repeticoes=$(echo $programa | cut -f3 -d,)

    echo "
\begin{table}[h]
    \centering
    \caption{Teste de Execucao com ${arquivo}}
    \label{tab${arquivo}_teste}
    \begin{tabular}{ccccc}
    \toprule
    & repeticoes & date +\%s.\%3N & time (real) & /usr/bin/time (wall clock) \\\\
    \midrule" >> $1

    # SPIM
    for repeticao in ${repeticoes//-/ }; do
        # date
        start=`date +%s.%3N`
        for (( run = 1; run <= $repeticao; run++ )); do
            spim -file ${arquivo}.mips < $entrada >> /dev/null 2>&1
        done
        end=`date +%s.%3N`; date=$( echo "$end - $start" | bc -l )
        # time
        (time (for (( run = 1; run <= $repeticao; run++ )); do
            spim -file ${arquivo}.mips < $entrada >> /dev/null 2>&1
        done)) 2> output_time
        time=`grep real output_time | cut -f2 -d $'\x09'`
        # /usr/bin/time
        /usr/bin/time -vo output_bin_time bash -c "for (( run = 1; run <= $repeticao; run++ ))
            ; do
            spim -file ${arquivo}.mips < $entrada >> /dev/null 2>&1
        done"
        bin_time=`grep 'wall clock' output_bin_time | cut -f8 -d' '`
        echo "
        SPIM & $repeticao & $date & $time & $bin_time (m:ss) \\\\" >> $1
    done

    # RARS
    for repeticao in ${repeticoes//-/ }; do
        # date
        start=`date +%s.%3N`
        for (( run = 1; run <= $repeticao; run++ ))
        do
            java -jar rars.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
```

```
50            done
51            end=`date +%s.%3N`; date=$( echo "$end - $start" | bc -l )
52            # time
53            (time (for (( run = 1; run <= $repeticao; run++ )); do
54                java -jar rars.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
55            done)) 2> output_time
56            time=`grep real output_time | cut -f2 -d $'\x09'`
57            # /usr/bin/time
58            /usr/bin/time -vo output_bin_time bash -c "for (( run = 1; run <= $repeticao; run++ ))
                  ; do
59                java -jar rars.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
60            done"
61            bin_time=`grep 'wall clock' output_bin_time | cut -f8 -d' '`
62            echo "
63            RARS & $repeticao & $date & $time & $bin_time (m:ss) \\\\" >> $1
64        done
65
66        # RARS-CLI
67        for repeticao in ${repeticoes//-/ }; do
68            # date
69            start=`date +%s.%3N`
70            for (( run = 1; run <= $repeticao; run++ ))
71            do
72                java -jar rars-cli.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
73            done
74            end=`date +%s.%3N`; date=$( echo "$end - $start" | bc -l )
75            # time
76            (time (for (( run = 1; run <= $repeticao; run++ )); do
77                java -jar rars-cli.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
78            done)) 2> output_time
79            time=`grep real output_time | cut -f2 -d $'\x09'`
80            # /usr/bin/time
81            /usr/bin/time -vo output_bin_time bash -c "for (( run = 1; run <= $repeticao; run++ ))
                  ; do
82                java -jar rars-cli.jar ${arquivo}.riscv < $entrada >> /dev/null 2>&1
83            done"
84            bin_time=`grep 'wall clock' output_bin_time | cut -f8 -d' '`
85            echo "
86            RARS-CLI & $repeticao & $date & $time & $bin_time (m:ss) \\\\" >> $1
87        done
88
89        echo "
90            \bottomrule
91        \end{tabular}
92    \end{table}" >> $1
93    done
```