Universidade Federal de Santa Catarina
Centro Tecnológico
Curso de Sistemas de Informação

Wilson de Almeida

Avaliação de Performance de Interpretadores Ruby

Florianópolis
2010

Wilson de Almeida

Avaliação de Performance de Interpretadores Ruby

Monografia apresentada ao Curso de Sistemas de Informação da UFSC, como requisito para a obtenção parcial do grau de BACHAREL em Sistemas de Informação.

Orientador: Lúcia Helena Martins Pacheco

Doutora em Engenharia

Florianópolis

2010

Almeida, Wilson

Avaliação de Performance de Interpretadores Ruby / Wilson Almeida - 2010

xx.p

1.Performance 2. Interpretadores.. I.Título.

CDU 536.21

Wilson de Almeida

Avaliação de Performance de Interpretadores Ruby

Monografia apresentada ao Curso de Sistemas de Informação da UFSC, como requisito para a obtenção parcial do grau de BACHAREL em Sistemas de Informação.

Aprovado em 21 de junho de 2010

BANCA EXAMINADORA

Lúcia Helena Martins Pacheco

Doutora em Engenharia

José Eduardo De Lucca

Mestre em Ciências da Computação

Eduardo Bellani

Bacharel em Sistemas de Informação

Aos meus pais e meu irmão.

Aos familiares e amigos, em especial pra minha eterna amiga Liliana, que está torcendo por mim de onde ela estiver.

Agradecimentos

Agradeço ao meu amigo, colega de curso, parceiro de trabalhos e orientador Eduardo Bellani, pelo encorajamento, apoio e seus ricos conselhos sobre o melhor direcionamento deste trabalho.

A professora Lúcia Helena Martins Pacheco pela orientação, amizade, e pela paciência, sem a qual este trabalho não se realizaria.

Ao professor José Eduardo Delucca, por seus conselhos objetivos e pontuais.

Todos os meus amigos que incentivaram e compreenderam a minha ausência nesse período de corrida atrás do objetivo de concluir o curso.

Obrigado Mariana, Priscilla, Rafael e Rafaela.

Sumário

Lista de Figuras

Lista de Tabelas

1 Objetivo Geral

Estudar o desempenho de implementações da linguagem de programação Ruby por meio de testes de avaliação de desempenho (benchmarks) de seus principais interpretadores disponíveis atualmente no mercado.

2 Objetivos Específicos

1. Analisar as qualidades e limitações dos principais interpretadores da linguagem Ruby por meio de algoritmos de teste de performance de forma a obter dados de desempenho.

2. Divulgar conhecimento e informações sobre a linguagem de programação Ruby e distribuí-lo para a comunidade acadêmica.

3 Justificativa

O presente trabalho tem como justificativa divulgar uma linguagem de programação interpretada orientada a objetos e de fácil aprendizado. Desta forma visa-se encontrar as melhores características de cada um dos interpretadores e com isso disseminar o conhecimento sobre a linguagem de programação Ruby, com seus benefícios e suas limitações.

Com a criação do Framework para desenvolvimento voltado para Web chamado Ruby on Rails[1], que foi desenvolvido em Ruby, a linguagem ganhou evidência. Assim, torna-se interessante para os estudiosos desta área testes em seus diferentes interpretadores. Testes utilizando algoritmos simples que verificam funções matemáticas, recursividade e manipulação de arquivos permitem conclusões sobre a performance dos interpretadores.

Diversas notórias aplicações para Web foram desenvolvidas no referido framework, o microblog Twitter[2], o website de gerenciamento BaseCamp da 37 signals[3], Shopify[4], Yellow Pages[5], Justin TV[6] e o Github[7]. Todas famosas aplicações Web desenvolvidas neste framework.

Ruby caracteriza-se por ter como principal objetivo a facilidade para o programador Fitzgerald [2007]página 1. Devido a isto torna-se interessante para o curso de Sistemas de Informação o uso desta linguagem pelas suas características de orientação a objetos, podendo ser utilizada nas implementações de programação necessárias nas várias disciplinas do curso.

Nos próximos capítulos falaremos sobre os assuntos que formam a base fundamental do trabalho. Falaremos sobre as características das linguagens de programação e dos conceitos básicos de interpretadores e compiladores, e por último mais especificamente sobre Ruby.

[1] http://rubyonrails.org
[2] http://twitter.com
[3] http://basecamphq.com
[4] http://www.shopify.com
[5] http://www.yellowpages.com
[6] http://www.justin.tv
[7] http://github.com

4 Linguagens de programação

Neste capítulo descreveremos sucintamente as linguagens de programação segundo seus paradigmas, seu modelo de execução, seu grau de abstração e outras maneiras de classificação.

4.1 Paradigma

Diferentes linguagens de programação podem ser agrupadas em diversos grupos dependendo dos mais diversos pontos de vista, como sintaxe, modelo de execução, paradigma ou a sua data de criação. Analisando segundo o paradigma que a linguagem segue em sua sintaxe podemos definir:

4.1.1 Programação funcional

Podemos definir de modo bem restrito, que programação funcional define as saídas de um programa como uma função matemática de suas entradas, sem opção de mudanças de estado e sem efeitos colaterais, ou seja evita dados e estados mutáveis (Scott [2006]).

As linguagens de programação funcional são linguagens declarativas, que utilizam funções. Esta classe de linguagens de programação baseia-se em avaliação de expressões. Entre as reivindicações dos defensores das linguagens de programação funcional são que programas podem ser escritos mais rapidamente, mais enxutos, mais concisos, mais propícios para uma análise formal e podem ser executados mais facilmente em um ambiente de arquitetura paralela. Ressaltando que esta é uma opinião dos defensores deste paradigma de programação (Hudak [1989]).

Diversas características de linguagens funcionais, tais como construtores, recursividade e coletor de lixo (*garbage collection*) podem ser encontradas em algumas, porém não todas, linguagens imperativas. Uma importante característica da programação funcional é o polimorfismo, pois ele permite que funções de mesmo nome possam atuar com os mais diversos tipos de parâmetros, formando assim, assinaturas diferentes

para a mesma função. Listas também são importantes em programação funcional, pois definem recursividade naturalmente. As funções são tratadas como valores que podem ser passados como parâmetro, podem ser o retorno de subrotinas, ou mesmo definidas como valor de uma variável.

Exemplos de linguagens de programação funcional segundo Scott [2006] temos Miranda, Haskell, Sisal como puramente funcionais e Scheme, um dialeto de Lisp que inclui algumas características de programação imperativa.

4.1.2 Programação estruturada

Programação estruturada foi o carro-chefe dos paradigmas de programação nos anos 70, assim como o paradigma orientado a objetos foi nos anos 90. Programação estruturada enfatiza o modelo *top-down* de descrição de um problema. Definição de constantes e estruturas de tipo, como os arrays, conjuntos e ponteiros, iteração, seleção e laços de repetição são algumas das características mais marcantes deste paradigma. Desenvolver em programação estruturada é criar uma sequência de comandos que descrevem como o problema será resolvido.

Muitas das estruturas de controles mais utilizadas como a seleção (comando *if*), os laços de repetição (comandos *for* e *while*) atualmente provém da programação estruturada, sendo primeiramente utilizados em *algol 60*. A estrutura de controle *case* foi introduzida pela linguagem Algol W.

4.1.3 Programação orientada a objetos

Com o aumento da complexidade das aplicações computacionais, a abstração de dados tornou-se essencial para a engenharia de software. A abstração consegue atingir três benefícios por meio do conceito de modularização do problema. Modularizar o problema é dividir um problema grande em pedaços ou problemas menores, facilitando a compreensão. Primeiro benefício é a redução da carga conceitual pela diminuição de detalhes a serem pensados em uma só vez. Segundo, a limitação de escopo de um determinado componente, pois pelo isolamento das suas funções conseguimos identificar mais facilmente os erros ou causas de erros no componente, e com isto o terceiro benefício é a independência dos componentes de um programa entre si, facilitando suas modificações internas não

precisando modificar sua camada externa (*interface*) (Scott [2006]).

Algumas das características mais marcantes do programação orientada a objetos, paradigma ao qual Ruby está inserido são:

1. Classes

   Uma Classe é uma abstração das características comuns de um grupo de objetos. Um modelo ao qual diversos objetos podem ser criados contendo as mesmas funcionalidades, por meio de seus métodos, e manter os mesmos estados, através de seus atributos.

2. Objetos

   Um *objeto* é uma instância de uma *classe*. Um objeto tem seu estado determinado pelos seus atributos e contém comportamentos definidos pelos seus métodos. Um exemplo básico da relação entre classe e objeto seria definir uma classe chamada *Pessoa*, com um atributo chamado *nome* e um método chamado *respirar*. Podemos criar, por exemplo, 3 objetos da classe *Pessoa*, com 3 valores diferentes para o nome (*João*, *José* e *Wilson*) e podemos realizar em cada uma das 3 instâncias chamadas ao método *respirar* .

3. Herança

   Programação orientada a objetos permite que classes herdem estados e comportamentos de outras classes. Herança provê um poderoso e natural mecanismo de organização e estruturação do software (Tutorials [2009]). Um exemplo de herança pode ser um conjunto de classes começando pela classe *Automóvel*. A classe *Automóvel* será a super-classe das classes *Carro* e *Caminhão*, onde ambas herdam os atributos e métodos da super-classe. Cada classe pode conter um número ilimitado de sub-classes.

4. Métodos públicos e privados

   As linguagens de programação orientadas a objetos definem o acesso às suas funções, ou métodos, segundo o nível de visibilidade geralmente pública ou privada. Os métodos públicos de um objeto podem ser acessados por outros objetos dentro do sistema por meio da interface do mesmo, e métodos privados são de uso exclusivo dos objetos da mesma classe ou do próprio objeto dependendo da especificação da

linguagem. Java, por exemplo, ainda contém mais uma cláusula que define um acesso intermediário, chamado *protected*, ao qual objetos das sub-classes podem acessar o método que contém *protected* em sua assinatura.

5. Polimorfismo

   Polimorfismo nas linguagens de programação orientadas a objeto é a capacidade de definir mais de um método com o mesmo nome, porém com diferentes argumentos. Uma coleção de diferentes assinaturas para o mesmo método, assim podendo responder de diferentes maneiras uma chamada para um método dependendo dos argumentos passados para este método.

6. Subrotinas menores

   Segundo Scott [2006] programas orientados a objetos tendem a criar mais chamadas para subrotinas do que programas em outros paradigmas, e as subrotinas tendem a ser menores.

7. Classes derivadas

   Derivando novas classes de classes anteriores, o desenvolvedor tem a possibilidade de criar uma hierarquia de classes com funcionalidades adicionais nas classes dos níveis mais baixos da sua árvore hierárquica.

8. Sobrecarga de construtores

   Construtores assim como métodos em geral podem conter diferentes tipos de argumentos. Logo, uma classe pode conter mais de um construtor.

9. Encapsulamento

   O mecanismo de encapsulamento permite que o desenvolvedor agrupar os dados e rotinas que operam juntos em um mesmo local, escondendo detalhes que não são relevantes ao usuário desta abstração.

As linguagens de programação suportam os conceitos fundamentais de programação em diferentes níveis. Algumas aprofundam-se mais que as outras em determinados conceitos, tal como da programação orientada a objetos. As linguagens se diferem na medida que elas forçam o desenvolvedor em escrever um código fonte da maneira voltada à orientação a objetos. Alguns autores pensam que uma linguagem de programação orientada a objetos deve dificultar ou mesmo tornar impossível que o programador consiga

escrever um código que não seja nesse estilo. Este autores defendem a maneira mais pura da implementação da orientação a objetos, onde cada tipo de dados seja de uma classe, que as variáveis referenciem os objetos e que cada subrotina seja um método. Duas linguagens que são exemplos desse pensamento são Smalltalk e Ruby (Scott [2006]).

4.1.4 Programação lógica

De acordo com Scott [2006] programação lógica é uma idéia atraente que sugere um modelo de computação em que listamos as propriedades lógicas de um valor desconhecido e, em seguida, o computador demonstra como este valor podem ser encontrado (ou que não existe).

O sistema de Programação Lógica permite ao desenvolvedor recolher uma coleção de *axiomas* a partir dos quais podem ser provados teoremas, ou seja, permite ter em mãos uma coleção de proposições ainda não provadas de tal forma que possam ser aceitas como teorias comprovadas. O desenvolvedor que se utiliza da Programação Lógica afirma um teorema, ou objetivo, e a implementação da linguagem executa tentativas de encontrar uma coleção de *axiomas* e inferências de passos que juntos implicam o objetivo final. A linguagem mais utilizada neste ramo é *Prolog*.

4.1.5 Programação imperativa

Linguagens de programação imperativa descrevem suas ações por meio de uma sequência de comandos que alteram o estado do programa por suas variáveis. O foco das linguagens imperativas está em como o algoritmo irá executar o programa, ou seja, quais passos.

Segundo Scott [2006] as linguagens de programação imperativas podem se subdividir em 3 categorias.
As que seguem o modelo de Von Neumann como C, ADA e Fortran.
As linguagens de Script como Perl, Python e PHP.
As linguagens orientadas a objeto como Smalltalk, Eiffel, C++ e Java.

O paradigma de programação imperativo é hoje o dominante no mercado de trabalho. Vide o sucesso de linguagens de programação como Java, C++ e PHP.

Assim como citado pelo mesmo autor, toda a classificação sempre está aberta

ao debate.

4.2 Estrutura de Tipos

Segundo a estrutura de tipos as linguagens classificam-se em diferentes níveis de avaliação de compatibilidade com as regras da especificação da linguagem.

4.2.1 Fortemente Tipada

Linguagens fortemente tipadas definem com rigor a declaração de tipo de uma variável. As variáveis devem ser de um tipo específico e não podem ser alteradas para um tipo diferente. Linguagens que utilizam esse modelo são Java, C++, Ada, Pascal, etc.

4.2.2 Dinamicamente Tipada

Linguagens dinamicamente tipadas são aquelas ao qual a declaração do tipo das variáveis é mutável. Uma mesma variável pode receber em determinados momentos valores de diferentes tipos. Exemplos desse modelo são Php, Ruby, etc.

4.3 Grau de abstração

4.3.1 Linguagem de programação de baixo nível

Linguagens de programação de baixo nível são linguagens que se aproximam da arquitetura da máquina ao qual foram desenvolvidas para executar. O exemplo mais utilizado são as linguagens assembly. Linguagens assembly foram desenvolvidas para conectar suas operações para serem traduzidas para a linguagem da máquina. Esse trabalho de tradução é feito pelo sistema chamado de *Assembler*.

4.3.2 Linguagem de programação de alto nível

Linguagens de programação de alto nível foram originalmente desenvolvidas para substituir as linguagens assembly. O modelo de abstração da linguagem de alto nível busca ser

mais compreensível aos humanos visando a facilitação da compreensão do código fonte.

O termo abstração neste caso remete ao grau de separação que a linguagem está da arquitetura do computador. As linguagens de programação de alto nível tem por objetivo ser independente de máquina e de fácil assimilação para o desenvolvimento por humanos, ou seja, fácil de programar (Scott [2006]).

Ser independente de máquina é uma característica, ou conceito, simples, e que determina uma das características do que é ser uma linguagem de alto nível, pois as linguagens de baixo nível estão intimamente ligadas com o hardware de determinada máquina. As linguagens de alto nível procuram independência desse fator para poder atingir uma gama maior de tipos de máquinas as quais podem executar.

4.4 Modelo de execução

4.4.1 Linguagem de programação compiladas

Uma linguagem de programação compilada tem seu código fonte primeiramente traduzido para um código intermediário, por meio de um programa chamado compilador, e este código intermediário será posteriormente montado para um código de baixo-nível ou código de máquina para ser executado pelo sistema operacional.

4.4.2 Linguagem de programação interpretadas

Uma linguagem de programação interpretada define-se por ser executada diretamente do seu código fonte pelo sistema operacional por meio de um programa de chamado interpretador. O código fonte é executado diretamente sem a geração de código intermediário.

4.4.3 Linguagens de Script

As linguagens de Script tem dois conjuntos de ancestrais. Um conjunto são o interpretadores de comando ou *shells* e os terminais de linha de comando, tais como o *MS-DOS command interpreter* e o *sh* utilizado no sistema operacional *Unix*. Outro são o conjunto de ferramentas de processamento de texto e geração de relatórios como *RPG* da *IBM*, e *sed* e *awk* utilizados em *Unix*. Algumas linguagens de Script são ActionScript, BASIC,

JavaScript, Lua, PHP, Python, Ruby e Tcl Scott [2006].

Como citado por John Ousterhout, o criador do Tcl:

> Linguagens de Script assumem que uma coleção utilizável de componentes existem em outras linguagens. Eles não são destinados para escrever aplicações a partir do início mas para a combinação de componentes [1].

Ou seja, as linguagens de programação em geral são construídas para aceitar uma determinada gama de entradas, manipular estes dados de uma determinada forma e produzir as saídas apropriadas. Porém, com o crescimento da complexidade dos sistemas em geral, e a existência de sistemas que se utilizam de mais de uma linguagem de programação, necessita-se de flexibilidade para a troca de dados entre os mesmos.

Alguns autores denominam o termo *scripting languages* para as linguagens que agrupam múltiplos programas, porém, atualmente o termo é utilizado também para o *web scripting*. Um bom exemplo é o Tk, utilizado para programação de interfaces gráficas, originalmente desenvolvido para ser usado com Tcl, e com o tempo foi incorporado em outras linguagens de script como Perl, Python e Ruby.

Algumas características das Linguagens de Script são:

1. Interatividade

   Algumas Linguagens de Script como Perl utilizam um compilador de tempo real (*just-in-time compiler*) que lê o código fonte completo antes de produzir resultados. A maioria das outras linguagens são propensas em interpretar suas entradas linha por linha. Python, Tcl, e Ruby aceitam comandos vindos do teclado.

2. Expressões curtas

   Para suportar o rápido desenvolvimento e a interatividade, as Linguagens de Script utilizam o mínimo de comandos. Algumas fazem uso de muita pontuação e poucos e pequenos identificadores tal como em Perl, enquanto outras procuram aproximar sua sintaxe à lingua inglesa (um estilo chamado de "English-like") com muitas palavras e pouca pontuação tal como em Ruby. Todas buscam evitar as declarações extensivas.

[1] Scripting languages assume that a collection of useful components already exist in other languages. They are intended not for writing applications from scratch but rather for combining components Ousterhout [1998].

Um exemplo clássico chamado de "Hello World!", onde a única tarefa é imprimir na tela esta expressão, para exemplificar a busca pela simplicidade. Em Java (uma linguagem que não é de script):

hello_world.java

```
1 class Hello {
2   public static void main(String[] args) {
3     System.out.println("Hello World!");
4   }
5 }
```

Em Ruby (uma linguagem de script):

hello_world.rb

```
1 print "Hello World!"
```

3. Regras simples de escopo

   Linguagens de Script utilizam regras simples para definir escopo. Em algumas linguagens, como Perl, tudo, por padrão, é global. Em outras linguagens, como PHP, tudo, por padrão, é local. Em Python qualquer variável que é atribuída um valor é local no bloco ao qual a atribuição aparece.

4. Tipos dinâmicos e flexíveis

   A maioria das Linguagens de Script são dinamicamente tipadas. Em PHP, Python, Ruby e Scheme o tipo de uma variável é verificado durante o uso. Em Perl e Tcl a variável será interpretada diferentemente em diferentes contextos.

5. Fácil acesso para outros programas

   Muitas linguagens contém maneiras de acessar o sistema operacional para executar outros programas, ou para executar alguma operação diretamente. Em Linguagens de Script estas requisições são fundamentais, e tem um forte suporte. Perl contém mais de 100 comandos que acessam funções do sistema operacional para manipulação de arquivos, acesso em bases de dados ou mesmo comunicação de redes.

6. Padrões de busca e manipulação de *Strings* sofisticados

De acordo com sua origem em processadores de texto e para facilitar a manipulação textual de entradas e saídas de programas externos, as Linguagens de Script geralmente contém facilidades para buscas e manipulação de *strings* e são tipicamente baseadas em expressões regulares.

7. Tipos de dados de alto nível

   Tipos de dados de alto nível como listas, dicionários, conjuntos e tuplas são comuns em bibliotecas de linguagens de programação. Em Linguagens de Script essa característica vai um passo além por construir tipos de alto nível dentro de sua sintaxe e semântica.

Muito do crescimento e da popularização das Linguagens de Script deve-se ao crescimento contínuo da Web, do dinamismo da comunidade do software livre e do baixo investimento para desenvolver um projeto em comparação com outras linguagens de maior apelo industrial como Java ou C#.

5 Compiladores

Neste capítulo serão apresentados alguns conceitos básicos sobre compiladores.

Compiladores são programas de computador que traduzem código fonte em código objeto.

Conforme Scott [2006], traduzir a partir de uma linguagem de alto nível para a linguagem de montagem (assembly) ou linguagem de máquina é uma tarefa de sistemas conhecidos como compiladores.

> *Um compilador traduz de uma linguagem de alto nível para uma linguagem de baixo nível.* Watt and Brown [2000]página 27[1]

Código fonte, também chamado de código em linguagem de alto nível, e código objeto, também chamado de código em linguagem de baixo nível.

Avaliando de uma visão de alto nível o objetivo do compilador é traduzir o código fonte num código objeto semanticamente equivalente (Alfred V. Aho and Ullman [1986]).

> *Programas de computador são formulados em uma linguagem de programação e especificam classes de processos computacionais.* Wirth [1981]página 6[2]

Computadores interpretam sequências de instruções em linguagens de baixo nível, como Assembly e as linguagens simbólicas específicas da máquina, tendo como principal objetivo a criação de programas executáveis. Nisto a tradução necessita ser automatizada entre a linguagem de alto nível e a de baixo nível. Antes de realizar qualquer tradução, um compilador verifica se o texto do código objeto está bem formatado sob as definições da linguagem fonte. Caso não esteja o compilador gera relatórios de erro.

Dentro deste processo de *compilação* do código de alto nível, temos toda uma

[1] Tradução livre do autor

[2] Tradução livre do autor

sequência de passos de Análise Léxica, pré-processamento, Análise Semântica, Geração de Código e Otimização do Código segundo os passos descritos por Watt and Brown [2000].

Ao fim destes passos, um código intermediário que será *montado* para uma linguagem de máquina.

A geração de código intermediário é uma das etapa da compilação como citado por Alfred V. Aho and Ullman [1986], já os interpretadores, ao invés de produzirem um programa objeto fruto da tradução, executam diretamente as operações especificadas no código fonte. Ou seja, não existe geração de código intermediário, uma característica que marca a diferença entre compiladores e interpretadores.

A geração de código intermediário é uma das etapa da compilação, já os interpretadores, ao invés de produzirem um programa objeto fruto da tradução, executam diretamente as operações especificadas no código fonte. Ou seja, não existe geração de código intermediário, uma característica que marca a diferença entre compiladores e interpretadores.

> *Todos os compiladores executam a mesma coleção de tarefas e as variações aparentes acontecem porque as tarefas são colocadas em conjunto de forma ligeiramente diferente ou porque algumas das tarefas são dadas maior importância do que outras em algumas variações.* Bornat [1979]página 3[3]

> *Desenvolvimento de compiladores deve ser guiado pela especificação da linguagem.* Watt and Brown [2000]página 338[4]

Segundo Alfred V. Aho and Ullman [1986] um compilador é uma simples caixa que mapeia programas em código fonte para um programa semanticamente equivalente. Esta estrutura de mapeamento se divide em duas partes: Análise e Síntese.

Nas figuras da página 2 do livro temos o exemplo desta "simples"caixa que é o compilador e interpretador.

[3]Tradução livre do autor

[4]Tradução livre do autor

Figura 5.1: Um compilador - baseado no exemplo do livro Alfred V. Aho and Ullman [1986]

Figura 5.2: Um interpretador - baseado no exemplo do livro Alfred V. Aho and Ullman [1986]

5.1 Análise

Esta parte do processo de compilação cria uma estrutura onde o programa-fonte é analisado, em cada pedaço do código, por uma visão gramatical, no fim criando uma representação intermediária do programa-fonte. Se esta parte de análise detectar erros tanto sintáticos quanto semânticos, o compilador fornecerá mensagens de erro para que o desenvolvedor possa tomar providências. A tarefa de análise também coleta informação sobre o código e o armazena numa estrutura de dados chamada tabela de símbolos, que é passada para a fase de síntese com a representação intermediária.

5.2 Síntese

Na fase de síntese é construído o programa objeto por meio da tabela de símbolos e do código intermediário.

Se examinarmos mais ao fundo podemos decompor o processo de compilação em mais fases. Ainda seguindo o modelo descrito por Alfred V. Aho and Ullman [1986]

5.3 Análise Léxica

Esta primeira fase de um compilador é também chamada de *scanning*. O analisador léxico lê as sequências de caracteres agrupando-os em grupos de que formem sequências inteligíveis chamadas *lexemes*. Para cada *lexeme*, o analisador léxico produz uma saída como *token* no formato [nome do token, valor do atributo] aos quais são passados para a fase posterior chamada de Análise Sintática.

1. Um *token* é um par no modelo chave-valor que contém o nome do *token* como chave e um valor opcional. O nome é um símbolo abstrato que representa uma palavra reservada, uma unidade léxica ou uma sequência de caracteres que denotam um identificador. Os nomes de *token* são os símbolos de entrada que são processados pelo *parser*.

2. Um *padrão* é a descrição da forma dos *lexemes* de um *token*. No caso de uma palavra reservada, o padrão é apenas uma sequência de caracteres que formam a palavra-chave (palavra reservada). Para identificadores e alguns outros *tokens*, o padrão é uma estrutura um pouco mais complexa.

3. Um *lexeme* é a sequência de caracteres no código fonte que corresponde ao padrão pra um *token* e é identificado pelo Analisador Léxico como uma instância deste *token*.

No *token*, a primeira parte chamada de *nome* é um símbolo abstrato que é usado na fase de Análise Sintática, e a segunda parte, o *valor do atributo* aponta para uma entrada na tabela de símbolos. Esta informação será posteriormente utilizada na Análise Semântica e na geração de código.

Utilizando um exemplo do mesmo livro na página 6:
*posição = inicial + taxa * 60.*

1. *posição* é o lexeme que vai ser mapeado para [id, 1], onde *id* é um símbolo abstrato e *1* referencia uma entrada na tabela de símbolos para *posição*. A entrada na tabela de símbolos armazena informação sobre este id, tal qual nome e tipo.

2. = é mapeado para [=]. Este símbolo não precisa de valor do atributo pois é reconhecido como o elemento de igualdade.

3. *inicial* é mapeado para [id, 2].

4. *+* é mapeado para [+] que também não contém valor do atributo pois é reconhecido como valor de soma.

5. *taxa* é mapeado para [id, 3].

6. * é mapeado para [*] contendo o valor reconhecido como multiplicação.

7. *60* é mapeado diretamente para [60] pois é reconhecido como um valor numérico. Porém em estudos mais avançados de técnicas de compilação poderiam utilizar a notação [numero, 60].

Então o analisador léxico vai ler da seguinte forma:

[id, 1] [=] [id, 2] [+] [id, 3] [*] [60]

Neste exemplo os *tokens* =, *+* e * são conhecidos pelo compilador como símbolos abstratos para igualdade, adição e multiplicação. Já são figuras conhecidas.

E a tabela de símbolos teria.

1 = posição
2 = inicial
3 = taxa

O Analisador lerá que o identificador da posição 1 chama-se *posição*, da posição 2 *inicial* e da posição *taxa*.

Como o Analisador Léxico é a parte do compilador que lê o texto fonte, ele realiza certas tarefas enquanto rastreia os *lexemes*. Uma delas é remover os espaços em branco e os comentários no código fonte.

5.4 Análise Sintática

A segunda fase também pode ser chamada de *parsing*. O *parser* utiliza os primeiros componentes dos *tokens*, o nome dos tokens ou seja, as chaves do par chave-valor, produzidos

na Análise Léxica para criar uma representação intermediária em forma de árvore que reproduz a estrutura gramatical da sequência de *tokens*. Uma representação típica é uma árvore sintática no qual o interior de cada nó representa uma operação e cada filho deste nó representa os argumentos dessa operação.

Utilizando o exemplo da Análise Léxica:
*posição = inicial + taxa * 60*

O nó chamado de * contém [id, 3] que contém o identificador *taxa* como seu filho esquerdo e [60] como seu filho direito. O nó * explicita que deve-se multiplicar o valor de *taxa* por 60. O nó + indica que deve-se adicionar o resultado da multiplicação no valor do [id, 2] que contém o identificador *inicial*. A raiz desta árvore que contém = indica que devemos guardar o valor da soma na posição do identificador *posição*. Esta ordem das operações segue um padrão consistente com a aritmética que diz que a multiplicação tem precedência sobre a soma.

5.5 Análise Semântica

O analisador semântico utiliza a árvore sintática, a tabela de símbolos e verifica se existe consistência semântica com a definição da linguagem.

Uma importante parte da Análise Semântica é a verificação de tipo (*type checking*), onde o compilador compara se cada operador contém os seus operandos corretos. Por exemplo, muitas linguagens de programação definem que o índice de um *Array* deve ser um número inteiro, então o compilador deve produzir um erro caso encontre um número de ponto-flutuante sendo usado como índice do *Array*. Algumas definições de linguagem também permitem que o operador trabalhe com diferentes tipos de operandos, permitindo algumas conversões de tipo, tal como se um operador de soma possa ser aplicado tanto em números inteiros como em números de ponto-flutuante. Caso o operador utilize um número inteiro e em um número de ponto-flutuante como operandos o compilador deve converter o número inteiro para seu valor em ponto-flutuante, por exemplo.

Utilizando o exemplo das análises léxica e semântica, supondo que os valores de *posição*, *inicial* e *taxa* fossem declarados como números de ponto-flutuante e o lexeme 60 como um valor inteiro. O verificador de tipo do compilador iria processar que o operador * utiliza um número de ponto-flutuante (*taxa*) e um número inteiro como seus operandos.

Neste caso o valor inteiro seria convertido para o seu valor em ponto-flutuante para depois realizar a multiplicação.

5.6 Geração de Código Intermediário

No processo de traduzir código fonte em código objeto o compilador pode criar uma ou mais representações intermediárias de diversas formas. Árvores sintáticas são uma forma de representação intermediária que são utilizadas durante as fases de Análise Sintática e Análise Semântica. Após a Análise Sintática e Análise Semântica do programa fonte, muitos compiladores geram explicitamente representações intermediárias de baixo nível ou em formato de parecido com o da linguagem de máquina, que podemos visualizar como um programa de uma máquina abstrata. Esta representação intermediária deve conter duas propriedades: ser fácil de produzir e fácil de traduzir para a linguagem da máquina.

5.7 Otimização de Código

A fase de Otimização de Código independente de máquina e tem seu foco no aprimoramento do código intermediário para obter como resultado um código objeto melhor, mais eficiente. Geralmente melhor quer dizer mais rápido, mas outros objetivos podem ser desejados, tal como um código mais curto, ou uma código objeto que consuma menos energia. Um algoritmo simples de geração de código intermediário seguido de Otimização de Código é um razoável meio para gerar um bom código objeto.

5.8 Geração de Código

O Gerador de Código utiliza como entrada a representação intermediária do programa fonte e mapeia para código objeto. Se a linguagem objeto é código de máquina, registradores ou locais de memória são selecionados para cada variável do programa. Então, as instruções intermediárias são traduzidas em sequências de instruções de máquina que desempenham a mesma tarefa. Um aspecto crucial da geração de código é a sensata atribuição dos registradores para alocar as variáveis.

6 Interpretadores

Neste capítulo falaremos sobre interpretadores e as diferenças entre compiladores e interpretadores.

Interpretador é uma espécie de tradutor de linguagens de alto nível que executa diretamente o código fonte para a máquina sem a criação de código intermediário, eliminando uma etapa que o diferencia dos compiladores, ou seja, ao invés de produzir um programa alvo como uma tradução, um interpretador executa diretamente na máquina as operações especificadas no programa fonte nas entradas fornecidas pelo usuário.

Inevitavelmente a velocidade da execução dos programas gerados por compiladores é maior que a velocidade dos programas gerados por interpretadores. O programa alvo em linguagem de máquina produzido por um compilador geralmente é muito mais rápido do que um gerado por um interpretador no mapeamento de entrada e saídas. Um dos fatores é este passo da geração do código intermediário, pois este código é otimizado durante uma das etapas do processo de compilação, para melhor execução na linguagem assembly ou de máquina. A perda na velocidade de execução é um ponto negativo na execução final do programa, porém durante a fase de desenvolvimento de um software, a utilização de interpretadores numa forma geral tem uma maior flexibilidade no processo, pois mapeia e diagnostica melhor os erros no programa, porque a execução é direta e as declarações são lidas na medida que aparecem, ou seja, ele executa o programa fonte declaração por declaração, e normalmente pode dar um melhor diagnóstico de erro do que um compilador (Alfred V. Aho and Ullman [1986]).

Um interpretador também pode lidar com linguagens nas quais características fundamentais do programa, tais como tamanho e tipos de variáveis, ou mesmo quais nomes remetem quais variáveis, podem depender dos dados de entrada. Algumas características da linguagem são quase impossíveis de implementar sem interpretação: em Lisp e Prolog, por exemplo, um programa pode escrever partes de si próprio e executar durante a sua própria execução. Um interpretador permite que o programa seja reiniciado imediatamente após uma modificação, sem esperar por uma recompilação, fato que é uma característica valorosa durante do desenvolvimento do programa. Alguns dos me-

lhores ambientes de programação para linguagens imperativas incluem um compilador e um interpretador juntos.

Implementações de linguagens baseada em compilador tendem a ser mais eficiente que as implementações baseadas em interpretador porque eles fazem decisões mais cedo. Por exemplo, um compilador analisa a sintaxe e a semântica de variáveis globais uma vez, antes de cada execução do programa. Então decide sobre a maneira como essas variáveis serão armazenadas na memória, e gera código eficiente para acessá-las onde quer que eles apareçam no programa. Um interpretador puro, em contraste, precisa analisar as declarações cada vez que o programa começa execução (Scott [2006]).

Depuração em tempo de execução usando um interpretador é muito simples. O programa interpretado em execução roda até o ponto em que um erro aparece. O erro pode ser solucionado e o programa pode continuar sua execução: com certeza não será necessário recarregar o programa e normalmente não será necessário reiniciá-lo segundo Bornat [1979].

O custo real de encontrar e corrigir diversos erros em um programa pode efetivamente ser menor quando se utiliza um interpretador do que quando se utiliza um compilador, pois o interpretador evita o caro processo de compilação, carregamento e iniciação. Ao desenvolver um programa tão grande e complexo como um compilador, por exemplo, é difícil de encaixar em mais de três ou quatro recarregas e iniciações em uma hora de sessão em um terminal interativo ligado em uma máquina de médio porte. Durante a maior parte do tempo o usuário está apenas à espera da mais recente compilação ou do carregamento terminar. Utilizando a mesma quantidade de tempo da máquina para interpretar o programa muitas vezes pode ser mais eficaz e certamente a hora do usuário pode ser mais bem gasta do que olhando para um terminal à espera de alguma coisa para acontecer. Todo programador sabe que em tempo de execução a interpretação leva mais tempo do que a execução do código compilado para o mesmo programa. A abismo de eficiência pode ser enorme e mesmo os melhores interpretadores Lisp levam pelo menos vinte vezes mais tempo para interpretar um programa do que seria necessário para executar uma versão compilada do mesmo programa.

Outra citação interessante sobre o comportamento dos interpretadores e compiladores está também em Bornat [1979], onde cita que um interpretador é simplesmente um dispositivo que tem uma certa representação do programa e realiza as operações que o

programa precisa. Já um compilador tem uma representação semelhante de um programa e produz instruções que, quando processados por uma máquina, vão efetuar as operações que o programa precisa. A diferença entre um interpretador e uma máquina sob esta definição não é muito grande: o micro-programa de um computador é um interpretador que lê um programa em código de máquina e imita o comportamento que uma máquina "real"iria expressar dado este programa.

Ao contrário de um compilador, um interpretador roda em cima da execução da aplicação. Na verdade, o interpretador é o local de controle durante a execução. Na realidade, o interpretador implementa uma máquina virtual cuja "linguagem de máquina"é a linguagem de programação de alto nível. O interpretador lê declarações nesta linguagem uma ou mais de cada vez, executando na medida que elas aparecem. Scott [2006]página 14[1]

É possível imaginar um interpretador que lê um programa escrito em papel ou mesmo rascunhado num envelope e iria interpretá-lo diretamente. Bornat [1979]página 377[2]

Em um ambiente onde a modelagem do programa está constantemente mudando, onde programadores que estão experimentando diferentes modelagem de programas ou quando as pessoas estão simplesmente aprendendo a programar, um interpretador certamente poupa tempo e esforço humano, pois permite uma interação mais efetiva com o programa objeto, coisa que não é possível quando o programa é compilado. Uma implementação de linguagem que permita a execução de um programa incluindo ambos os módulos compilados e interpretados podem conferir algumas das vantagens de ambas as abordagens. Depuração vai ser mais fácil do que com uma tradução totalmente compilada, especialmente se for possível substituir módulos compilados por interpretados quando erros são descobertos. A eficiência da execução do programa aumentará à medida que mais e mais dos módulos são compilados e depurados, eventualmente para rivalizar com um programa totalmente compilado (Bornat [1979]).

Dentre os conceitos mais básicos dos interpretadores e dos compiladores podemos ver que mesmo em fontes mais antigas os princípios básicos tais como a velocidade de execução e a flexibilidade no desenvolvimento mudaram pouca coisa quando analisamos um contra o outro. Na execução final do programa desenvolvido e compilado a velocidade

[1]Tradução livre do autor

[2]Tradução livre do autor

é maior tanto em técnicas e linguagens mais antigas como nas atuais. Mesmo fato ocorre na precisão dos diagnósticos de erros como a depuração em tempo de execução e na flexibilidade de evitar constantes iniciações na fase de desenvolvimento de um programa. Como citado acima grandes projetos de software buscam aglutinar as melhores capacidades de ambos.

7 Ruby

Ruby é uma linguagem de programação de código aberto e orientada a objetos criado por Yukihiro 'Matz' Matsumoto. Sua primeira versão foi liberada no Japão em 1995. Ruby tem ganhado aceitação ao redor do mundo como uma fácil de aprender, poderosa, e expressiva linguagem, especialmente desde o advento do Ruby on Rails, um framework para aplicações voltadas para Web escrito em Ruby[1]*. Ruby tem seu núcleo escrito na linguagem de programação C e roda na maioria das plataformas. É uma linguagem interpretada e não compilada.* Fitzgerald [2007]página 1[2]

Ruby é escrita em C e é altamente portável, foi desenvolvida em grande parte na plataforma GNU/Linux, atualmente está disponível para as plataformas UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, .NET, Solaris, além de ser possível executá-la na máquina virtual Java (www.ruby lang.org [2009] linuxdevcenter [2001]).

Segundo seu criador no site oficial da linguagem www.ruby lang.org [2009], Yukihiro 'Matz' Matsumoto misturou partes de suas linguagens favoritas (Perl, Smalltalk, Eiffel, Ada, e Lisp) para formar uma linguagem que tivesse características de programação funcional com programação imperativa.

Ruby é uma linguagem de programação orientada a objetos pura, mas também adequada para programação procedural e funcional segundo Flanagan and Matsumoto [2008].

Na figura 7.1 mot podemos ver um fragmento, um pedaço de um gráfico que demonstra uma espécie de relação das origens entre as linguagens de programação num contexto histórico. Ruby como sendo uma linguagens baseada em Python, Eiffel e Smalltalk.

No exemplo que segue podemos visualizar como Ruby define uma classe e cria objetos. Ruby define classe por meio do comando *class* assim como em diversas outras linguagens. Nosso caso a classe chama-se *Pessoa*. O método de criação do objeto chamado

[1] http://www.rubyonrails.org

[2] Tradução livre do autor

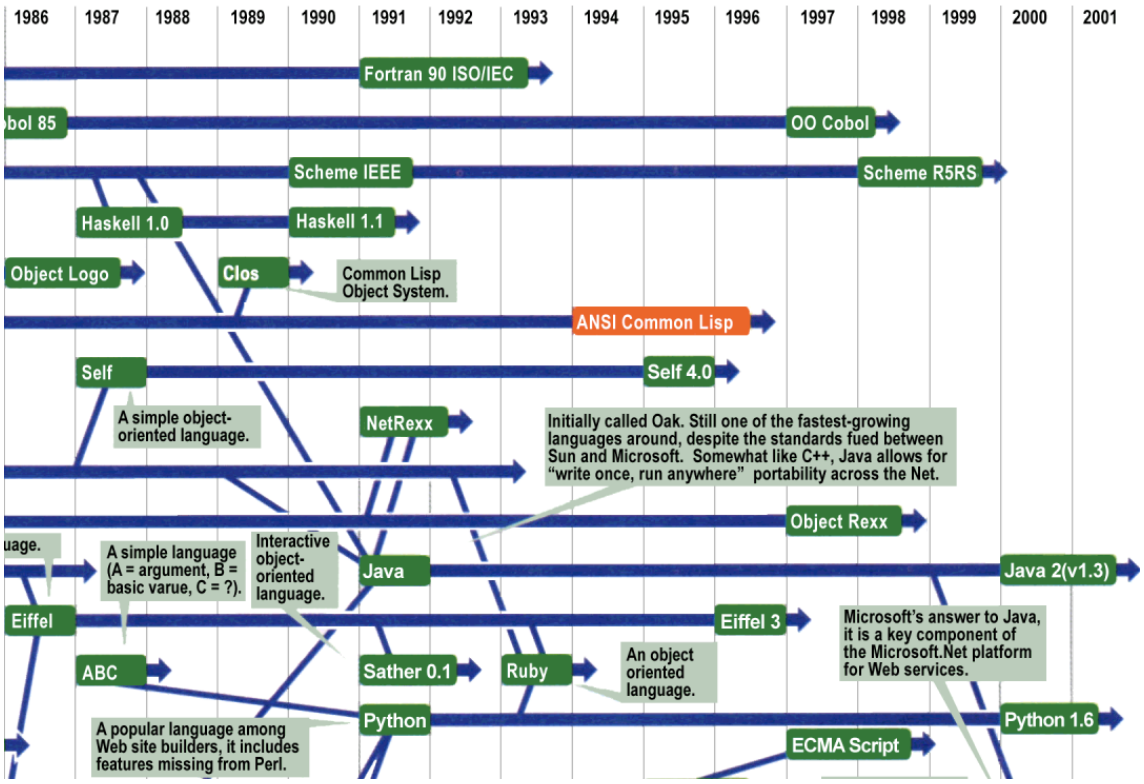

Figura 7.1: Linguagens de programação ancestrais

initialize recebe um argumento *sexo*. Ao criar o objeto com o método *new* devemos passar um string com o sexo do novo objeto da classe Pessoa. Dentro do método *initialize* iremos apontar o argumento sexo para a variável de classe *@sexo*.

O método *agradecer* irá retornar o valor do agradecimento dependendo do gênero que lhe foi passado como atributo no seu construtor como podemos ver na estrutura de controle das linhas 7 até 11.

Nas linhas 15 e 16 criaremos 2 objetos da classe pessoa passando argumentos diferentes para cada um deles. Nas linhas posteriores iremos escrever na tela do console o valores retornados pelo método agradecer que vai retornar o agradecimento correto para *homem* e *mulher*.

exemplo–criacao–objetos.rb

```
1 class Pessoa
2   def initialize(sexo)
3     @sexo = sexo
4   end
5
6   def agradecer
```

```
7       if @sexo == "masculino"
8         return "Obrigado"
9       else
10        return "Obrigada"
11      end
12    end
13  end
14
15  homem = Pessoa.new("masculino")
16  mulher = Pessoa.new("feminino")
17
18  p "Homem: " + homem.agradecer
19  p "Mulher: " + mulher.agradecer
```

Como resultado teremos na tela os valores do método invocado pelos objetos homem e mulher:

$ ruby exemplo-criacao-objetos.rb

"Homem: Obrigado"

"Mulher: Obrigada"

Ruby é uma linguagem de programação orientadas a objetos genuína, pois tudo que é manipulado é um objeto, e os resultados dessas manipulações são também objetos. Cada valor é um objeto, mesmo simples literais numéricos, valores booleanos (true e false), e 'nil' (nil é um valor especial que indica a ausência de valor; este é a versão Ruby de 'null'), strings, arrays, expressões regulares, e muitas outras entidades são na verdade objetos. Em Ruby, todos os objetos são uma instância de alguma classe. Isto explica o fato de diversas fontes citarem Ruby como uma linguagem completamente orientada a objetos. Ruby contém todos os elementos associados à linguagens orientadas a objetos, como objetos com encapsulamento e ofuscamento de dados, métodos com polimorfismo e sobre escrita, e classes com hierarquia e herança. Ela vai mais fundo ao definir e adicionar elementos de meta-classes, como métodos *singletons*, módulos e *mixins* (Dave Thomas and Hunt [2004] Flanagan and Matsumoto [2008] Fulton [2006]).

No exemplo que segue visualizamos que valores numéricos, arrays, valores booleanos podem invocar o método *class* que retorna a sua classe e imprimiremos o resultado

na tela do console.

exemplo–objetos.rb

```
1 p 1.class
2 p 1.1.class
3 p true.class
4 p nil.class
5 p "Wilson de Almeida".class
6 p [1, 2, 3, 4, 5].class
7 p /ruby/.class
```

O resultado gerado na tela pelo exemplo é:

```
$ ruby exemplo-objetos.rb
Fixnum
Float
TrueClass
NilClass
String
Array
Regexp
```

Cada linha é a resposta do nome da classe ao qual cada valor pertence. Por exemplo o número 1 é da classe *Fixnum*, o número 1.1 pertence a classe *Float* e assim por diante.

Polimorfismo em Ruby age diferente da maneira como é utilizado em outras linguagens, como Java por exemplo. Em Ruby o polimorfismo demonstra-se por meio do conceito de herança. Os métodos herdados da super-classe podem ser alterados na classe filho, e com isso as alterações no interior do código de um determinado método serão refletidos nos objetos da classe filho, diferentemente de Java onde tem-se o polimorfismo por meio da herança e também o polimorfismo de interface. Polimorfismo de interface é quando mais de um método tem o mesmo nome, porém com diferentes tipos ou número de argumentos, diferentes assinaturas. Esta segunda maneira não é possível em Ruby, pois caso 2 ou mais métodos tenham o mesmo nome somente um deles será passível de ser invocado. No caso não pode-se ter 2 métodos com o nome *jogar* por exemplo onde

um contém 1 argumento e outro com 2 argumentos em sua assinatura.

Como em muitas outras linguagens de programação orientadas a objetos, Ruby não permite herança múltipla, mas isto não necessariamente a torna menos poderosa. Linguagens de programação orientadas a objetos modernas constantemente seguem o modelo de herança simples (Fulton [2006]).

O encapsulamento de dados em Ruby é considerado restritivo, pois acessar seu estado interno diretamente em seus atributos não é possível. Tal acesso somente é permitido por meio de métodos públicos da classe (Flanagan and Matsumoto [2008]).

Abaixo segue um exemplo de encapsulamento, onde a variável de instância *@nome* só pode ser visualizada por meio de métodos de acesso. Criamos um objeto da classe Pessoa e passamos o *string* "Wilson"como argumento na criação do objeto. Após isto redefine-se o nome no método específico para alterar o nome com o valor "Wilson de Almeida".

exemplo–encapsulamento.rb

```
1  class Pessoa
2    def initialize(nome)
3      self.nome = nome
4    end
5
6    def nome
7      @nome
8    end
9
10   def nome=(nome)
11     @nome = nome
12   end
13 end
14
15 pessoa = Pessoa.new("Wilson")
16 p pessoa.nome
17 pessoa.nome=("Wilson de Almeida")
18 p pessoa.nome
```

O resultado impresso na tela é:

```
$ ruby exemplo-encapsulamento.rb
"Wilson"
"Wilson de Almeida"
```

Ruby é uma linguagem dinâmica, pois seus objetos e classes podem ser alterados em tempo de execução, e tem a capacidade de construir e avaliar pedaços de código no curso da execução de um código estático. Ruby possui uma API de reflexão computacional sofisticada que faz com que ela esteja mais 'consciente de si' e isto permite a criação mais facilitada de *debuggers*, *profilers*, e ferramentas similares que faz com que técnicas avançadas de programação tornem-se possíveis. Linguagens como Smalltalk, Lisp, e Java implementam (em diferentes graus) a noção de uma linguagem de programação reflexiva. Um ambiente ativo pode consultar os objetos que definem, estendem ou modificam a si mesmos em tempo de execução. Ruby permite reflexão extensivamente porém, não tanto quanto em Smalltalk que representa estruturas de controle como objetos. Ruby não controla estruturas e blocos como objetos (Um objeto *Proc* pode ser usado como 'objetificador' de um bloco, mas estruturas de controle nunca são objetos). A linguagem mapeia de perto o domínio do problema em um estilo próximo ao da linguagem humana. Ruby também inclui poderosas capacidades de meta-programação e pode ser usada para criar linguagens de domínio específico porque ela inclui poderosas capacidades de meta-programação e pode ser usada para criar linguagens de domínio específico, que nada mais são que extensões da própria sintaxe da linguagem, como por exemplo a criação de uma sintaxe de manipulação da apresentação de um dado em formato XML (Fulton [2006] Fernandez [2008]).

Ruby é uma linguagem interpretada. Podem existir implementações de compiladores Ruby para propósitos de velocidade, mas mantém-se em mente que um interpretador produz grandes benefícios não apenas em produção rápida de protótipos mas também em diminuir o ciclo de produção em geral. Algumas das características marcantes em Ruby é o de ser uma linguagem de 'muito' alto nível. Um princípio por trás do projeto da linguagem é que o computador deveria fazer o trabalho para o programador ao invés do contrário. A densidade de Ruby significa que operações sofisticadas e complexas podem ser feitas de maneira relativamente fácil se comparada com linguagens de nível mais baixo (Fulton [2006]).

É valido lembrar que uma nova linguagem de programação as vezes é vista como

uma panacéia, especialmente por seus adeptos. Mas nenhuma linguagem irá suplantar as outras, nenhuma ferramenta é a melhor para todas as situações. Existem muitos domínios de problemas no mundo e muitas restrições dentro destes domínios. Fulton [2006]Capítulo 1[3]

[3]Tradução livre do autor

8 Metodologia

Neste capítulo descreveremos o método que será utilizado para construir e analisar os testes de performance.

Segundo o livro Performance and Evaluation of Lisp Systems Gabriel [1985] podemos definir um plano de teste de algoritmos (benchmark) seguindo 5 passos. Deve-se tomar cuidado pois o simples resultado de tempo comparado entre 2 interpretadores pode levar à interpretação de que o resultado está alcançado e que um interpretador é melhor que outro. O objetivo de um benchmark é conseguir testar o mesmo algoritmo nos mais variados ambientes. Explicitar a maneira como foram conduzidos os testes também ajudam o usuário mais bem informado e de maior conhecimento técnico em decidir qual sistema será melhor para ajudar a resolver o seu problema.

Portanto, dentro da metodologia seguida pelo autor temos 5 principais objetivos.

8.1 Saber o que está sendo medido

O primeiro problema com benchmarks é saber o que será testado e medido.

Caso um algoritmo seja utilizado como base para um teste, primeiramente precisamos saber o que o algoritmo faz. Explicar o seu comportamento, e ressaltar qual a sua função, se é uma função aritmética, se é uma operação de entrada e saída etc. Examinando e expondo cuidadosamente cada passo do algoritmo. Como exemplo do algoritmo *Tak* que será testado posteriormente na bateria de testes. No qual utilizaremos os argumentos 18, 12 e 6, e a função será invocada 63.609 vezes por meio de recursividade. Detalhar o algoritmo é o primeiro passo da metodologia.

Após isto precisamos também saber quais dados queremos como retorno de resultado. Ou seja, o que deve ser medido.

No caso da mesma forma no algoritmo *Tak* ao final da execução será gerada a resposta no console.

26840

668132

real 0m1.374s

user 0m1.392s

sys 0m0.096s

8.2 Medir a coisa certa

Em cada passo da função que será usada para teste deve-se detalhar as operações demonstrando o que o processo faz. Saber se realmente o algoritmo testa o que deve ser testado.

Muitas vezes podemos cair na armadilha de estar testando determinado algoritmo dentro de um laço de repetição, porém tal laço de repetição pode ocupar mais linhas de código que o próprio pedaço de código que seria o dito teste. Portanto uma análise de cada passo se faz necessária para deixar bem claro o como o algoritmo funciona.

Ou seja, em casos extremos podemos até mesmo fazer o teste duas vezes, uma com o laço de repetição operando sozinho e outra com o devido algoritmo dentro. Subtraindo o tempo total do laço podemos encontrar o tempo gasto apenas pelo algoritmo que necessita ser testado.

8.3 Saber como as opções se combinam

Muitas vezes um mesmo código-fonte pode ser interpretado internamente de maneiras diferentes por seus respectivos interpretadores ou mesmo compilado de maneira que duas ou mais operações sejam otimizadas. As vezes o coletor de lixo também pode influenciar no desempenho de um algoritmo.

Uma maneira de saber como esses elementos afetam o desempenho de um interpretador seria o teste por meio de grandes benchmarks ou aplicações que trabalhem com o domínio do problema.

O problema é que aplicações nem sempre funcionam exatamente com o mesmo código-fonte em interpretadores diferentes. Ou seja, controlar esse tipo de opções é muito difícil, porém deve ser levado também em consideração, mesmo que apenas para informação.

8.4 Sistemas pessoais contra sistemas de tempo compartilhado

No caso deste trabalho o sistema operacional irá operar em uma única máquina, porém um sistema de "time-sharing", ou multi-tarefa, pode criar elementos de despesa de memória que ficam difíceis de serem controlados.

Máquinas pessoais são mais fáceis de controlar o tempo total e a tempo da CPU para executar determinado código. Sabendo que no fim de todo o processo precisamos saber quanto tempo o usuário leva para ver o resultado de determinado processo.

8.5 Medir com o hardware em todas as condições

A arquitetura de cada máquina interfere também no resultado final de tempo gasto por um algoritmo. Portanto, outro aspecto que deve ser levado em consideração é a arquitetura que será utilizada para o teste. Determinada arquitetura pode utilizar com mais eficiência ou menos a utilização de memória RAM por exemplo, sendo um fator que altera resultados do tempo para a execução de uma tarefa.

Dentro dos passos propostos pelo autor seguiremos os 3 primeiros. São os passos que demonstram a importância da análise do código que será testado e dos resultados das métricas obtidas. Nos demais passos a análise é mais voltada aos ambientes que serão efetuados os testes. Neste trabalho apenas um ambiente será utilizado para os testes, fica então uma futura proposta para a conclusão: os testes em diversos ambientes.

9 Ferramentas

Neste capítulo falaremos sobre as ferramentas que foram utilizadas para os testes de performance.

9.1 RVM - Ruby Version Manager

Ruby Version Manager (RVM) [1] foi iniciado em outubro de 2007, é uma ferramenta de linha de comando que segundo sua descrição é facilmente instalável e simples de trabalhar. A ferramenta tem como objetivo a manipulação de múltiplos ambientes com diferentes versões de interpretadores ruby.

Ruby Version Manager permite que o usuário indique o interpretador ruby que ele deseja utilizar em seu projeto, podendo alterar a versão ou mesmo utilizar outro interpretador ruby diferente.

Ruby Version Manager simplifica o desenvolvimento para multiplos ambientes ruby por meio de simples linhas de comando.

O processo de instalação utilizado pode ser encontrado em 2 locais, ou no próprio website do produto [2], ou numa postagem de um blog chamado DaRoost [3], onde ele faz testes de performance com os interpretadores ruby que irei usar no trabalho.

O processo de instalação foi, primeiramente instalar o *rvm* pela ferramenta *rubygems* que instala pacotes ruby em distribuições linux.

O comando é simples:
$ gem install rvm

Com isto já temos o código-fonte do RVM instalado na máquina. Então o segundo passo é executar o comando:
$ rvm-install

Após instalado agora vamos recarregar o RVM para instalar os interpretadores

[1] `http://rvm.beginrescueend.com/`
[2] `http://rvm.beginrescueend.com/rvm/install/`
[3] `http://greg.nokes.name/2009/12/22/using-rvm-to-benchmark-ruby/`

Ruby com o comando:

$ rvm reload

Agora vamos para a instalação dos 5 interpretadores Ruby desejados:

$ rvm install 1.8.6,1.8.7,1.9.1,ree,jruby

Por último alterasse o arquivo .bashrc encontrado na raiz do usuário linux que está rodando na máquina inserindo esta linha de comando.

if [[-s /home/wilson/.rvm/scripts/rvm]] ; then source /home/wilson/.rvm/scripts/rvm ; fi

Como a máquina em questão está rodando com o usuário *wilson* o caminho físico se inicia com */home/wilson/*.

O RVM suporta uma gama imensa de interpretadores ruby, alguns deles voltados para plataforma windows. Foram escolhidos os mais reconhecidos no momento, as versões 1.8.6, 1.8.7 e 1.9.1 do MRI/YARV Ruby que é considerado o padrão, pois o criador da linguagem ruby ainda é líder do projeto, o JRuby por utilizar a máquina virtual java e portanto ter a capacidade de ser multiplataforma, e o Ruby Enterprise Edition que também tem uma boa documentação na rede.

A lista de interpretadores suportados pelo rvm segundo o website do produto é:

ruby - MRI/YARV Ruby (The Standard) {1.8.6, 1.8.7, 1.9.1, 1.9.2...}
jruby - JRuby {1.3.1,1.4.0}
rbx - rubinius
ree - ruby Enterprise Edition
macruby - MacRuby (Mac OS X Only)
maglev - GemStone Ruby
ironruby - IronRuby
mput - shyouhei(mput)'s github repository
system - use the system ruby (eg. pre-rvm state)
default - use rvm set default ruby and system if it hasn't been set.

Após completados os passos pode-se verificar pelo comando para listar do RVM todos os interpretadores instalados: *$ rvm list*

E na tela do console aparece como saída:

$ rvm list

rvm Rubies

jruby-1.4.0 [[i386-java]]

ree-1.8.7-2010.01 [i386]

ruby-1.8.6-p383 [i386]

ruby-1.8.7-p248 [i386]

ruby-1.9.1-p378 [i386]

System Ruby

system [ ]

Indicando que não selecionamos um Ruby padrão, porém temos os 5 interpretadores instalados, e cada vez que utilizarmos um deles, devemos primeiramente selecionar pelo comando *$ rvm use ...* com o nome de algum dos interpretadores.

9.2 time

A distribuição Ubuntu utilizada no trabalho contém o comando *time*.

O comando será utilizado para medir o tempo utilizado pela máquina para a execução dos algoritmos. Por padrão o comando *time* define um formato após o especificador *-f* para demonstrar os possíveis atributos que o usuário deseja receber o valor de saída.

No padrão da máquina utilizada nos testes o formato é:

comando–time–exemplo–1.sh

```
1 time -f "real\t%E\nuser\t%U\nsys\t%S" ruby tak.rb
```

Neste exemplo ele colocaria na tela do console o seguinte:

real 0m0.070s

user 0m0.060s

sys 0m0.008s

O string \t define um espaço de tabulação e o comando \n uma nova linha.

Cada uma das letras posteriores ao % (E, U, S) tem um diferente significado segundo o manual do comando que é visualizado pela chamada *man time* dentro do console na máquina.

9.3 ps

O comando *ps* será utilizado para demonstrar uma imagem do instante final de execução dos algoritmos. O método retornará os valores de memória RAM utilizados pelo processo. A função do comando *ps* é obter informação dos processos ativos na máquina.

Ao final de todos arquivos de código-fonte ruby que serão utilizados para testes de performance colocaremos nas duas últimas linhas as seguintes linhas de comando.

No caso dos arquivos que testarão o JRuby utilizaremos o comando *ps* para observar o processo com o nome de *java*. Nos arquivos que testarão os outros interpretadores o comando *ps* irá observar o processo com o nome de *ruby*.

ultimas–linhas–interpretador–java.sh

```
1 p `ps -o rss= -C java`.to_i
```

ultimas–linhas–interpretador–ruby.sh

```
1 p `ps -o rss= -C ruby`.to_i
```

Como exemplo podemos ver no arquivo que irá testar o algortimo *Tak* no *JRuby*, pode-se observar nas duas últimas linhas a chamada para o comando *ps*.

tak–java.rb

```
1 def tak(x, y, z)
2   unless y < x
3     z
4   else
5     tak( tak(x-1, y, z),
6          tak(y-1, z, x),
7          tak(z-1, x, y))
8   end
9 end
```

```
10
11 tak(18, 12, 6)
12 p `ps -o rss= -C java`.to_i
```

Em ruby o algoritmo permance o mesmo porém as duas últimas linhas mudam para avaliar o processo chamado *ruby*.

tak–ruby.rb

```
1  def tak(x, y, z)
2    unless y < x
3      z
4    else
5      tak( tak(x-1, y, z),
6           tak(y-1, z, x),
7           tak(z-1, x, y))
8    end
9  end
10
11 tak(18, 12, 6)
12 p `ps -o rss= -C ruby`.to_i
```

As duas linhas iniciam com a letra *p* que é um comando ruby para escrever o resultado do comando posterior na tela do console. Após isto temos uma sequência de comandos entre aspas invertidas. Em ruby estas aspas invertidas significam que o conteúdo é um comando para ser executado pelo sistema operacional. Seria como abrir uma tela de console e digitar o comando do sistema operacional diretamente no console. Ao final este comando retornará uma string contendo o resultado que seria impresso num possível console. Este string será convertido para número inteiro pelo método *to_i*. Visualizando por inteiro o resultado do comando para o sistema operacional retornará o seu resultado como string que será convertido para número inteiro e após isto será escrito na tela pelo método ruby *p*.

Agora dentro das aspas invertidas temos o comando que inicia com *ps*, process status. Logo após temos o argumento *-o* que significa que o formato de saída será definido pelo usuário. O formato é uma lista de argumentos separados por espaços ou vírgulas. Por espaços em nosso caso. o primeiro argumento é *rss=* que define que o valor retornado será o de memória RAM residente em kilobytes utilizado pela tarefa. O argumento seguinte

é o *-C* que seleciona o processo pelo nome, por fim o nome do processo, que em nossos casos será ou *java*, quando utilizarmos o JRuby, ou *ruby* quando utilizarmos os outros interpretadores.

9.4 JRuby

JRuby[4] é uma implementação da linguagem de programação Ruby totalmente criada em Java, com isso é independente de plataforma, e compatível com a versão do Ruby MRI 1.8.7. No presente trabalho iremos utilizar a versão jruby-1.4.0.

9.5 Ruby Enterprise Edition

Ruby Enterprise Edition [5] é uma versão do interpretador Ruby desenvolvida por time de programadores holandeses chamado Phusion [6]. Para os testes a versão utilizada será ree-1.8.7-2010.01.

1. Permite que sua aplicação Ruby on Rails utiliza 33% menos memória quando combinada com o servidor Phusion Passenger.
2. 100% compatível com o Ruby "oficial", o Ruby MRI versão 1.8.7.
3. Bem testado e extremamento estável.
4. Fácil de instalar tanto pelas vias nativas do Linux como pelo instalador do interpretador.
5. Pode ser instalado em paralelo com o outro Ruby com nenhum risco.[7]

9.6 Ruby MRI 1.8.6

Ruby 1.8.6, também conhecido Ruby MRI 1.8.6, ou CRuby 1.8.6, é uma versão estável do interpretador Ruby criado por Yukihiro Matsumoto. O pacote utilizado no trabalho é

[4] `http://jruby.org`
[5] `http://rvm.beginrescueend.com/`
[6] `http://www.phusion.nl`
[7] Tradução livre do autor

o 383.

9.7 Ruby MRI 1.8.7

Ruby 1.8.7, também conhecido Ruby MRI 1.8.7, ou CRuby 1.8.7, é uma versão estável do interpretador Ruby criado por Yukihiro Matsumoto. O pacote utilizado no trabalho é o 248.

9.8 Ruby MRI 1.9.1

A versão Ruby 1.9 também é conhecida como YARV[8] (Yet Another Ruby VM) criada por Koichi Sasada, apelidado de ko1. Seu principal objetivo é o aumento de velocidade na execução. Desde o ano de 2007 o YARV foi incorporado ao projeto principal da linguagem Ruby tornando-se a versão oficial a partir da versão 1.9. Essa versão que conta com um interpretador de bytecode, não será mais chamada de MRI, e sim KRI (Koichi's Ruby Interpreter) por ser de outro autor (`http://ruby.about.com/od/newinruby191/a/YARV.htm`). O pacote utilizado no trabalho é o 378.

[8] `http://www.atdot.net/yarv/`

10 Desenvolvimento

Neste capítulo descreve-se a máquina e a maneira como foi montado o ambiente para a realização dos testes.

10.1 Contexto

O contexto no qual os testes foram realizados baseado no comando *sudo dmidecode*, comando este que informa o "topologia"da máquina, e traduzido livremente foi:

Informação do Sistema
Fabricante: *Hewlett-Packard*
Nome do Produto: *HP Pavilion dv6000*

Processador
Tipo: *Central Processor*
Família: *Opteron*
Fabricante: *AMD*
Assinatura: *Family 15, Model 72, Stepping 2*
Versão: *AMD Turion(tm) 64 X2 Mobile TL*
Voltagem: *1.6 V*
Clock Externo: *200 MHz*
Velocidade Máxima: *1600 MHz*
Velocidade Atual: *1600 MHz*
Estado: *Populated, Enabled*

Dispositivo de Memória
Tamanho: 2.4 MB
Tipo: DDR2
Velocidade: 667 MHz (1.5 ns)

Sistema Operacional Ubuntu - Release 8.04 (Hardy) - Kernel Linux 2.6.24-27-generic - GNOME 2.22.3

10.2 Ambiente

Para selecionar cada interpretador chamaremos o Ruby Version Manager, seguido pelo comando *ruby –version* que escreverá na tela do console o interpretador ruby que está ativo para executar o algoritmo.

Para o caso do JRuby chamaremos o arquivo com o nome seguido por *-java*, pois o nome processo na máquina é *java*. Os demais interpretadores geram um processo com o nome de *-ruby*.

As últimas linhas dos algoritmos que testam no JRuby terão os valores de consumo de memória RAM gerados na tela do console por meio dos comandos:

```
p `ps -o rss= -C java`.to_i
```

Para os demais interpretadores as duas últimas linhas serão:

```
p `ps -o rss= -C ruby`.to_i
```

Portanto, o arquivo que testará o JRuby no algoritmo *Tak* por exemplo vai ser:

tak–java.rb

```
1  def tak(x, y, z)
2    unless y < x
3      z
4    else
5      tak( tak(x-1, y, z),
6           tak(y-1, z, x),
7           tak(z-1, x, y))
8    end
9  end
10
```

```
11 tak(18, 12, 6)
12 p `ps -o rss= -C java`.to_i
```

Um exemplo da saída gerada no teste de um algoritmo com o JRuby seria este:

$ rvm use jruby

Now using jruby 1.4.0

$ ruby –version

jruby 1.4.0 $ruby 1.8.7 patchlevel 174\ 2009-11-0269fbfa3\ JavaHotSpotClientVM 1.6.0_17$ [i386-java]

$ time ruby tak-java.rb

26840

real 0m1.374s

user 0m1.392s

sys 0m0.096s

Onde na primeira linha requisitamos que o Ruby Version Manager selecione o JRuby para ser chamado com o comando ruby. Com isso ele retorna na tela a atual versão que será utilizada. Após isto executamos o comando ruby com o argumento *–version* para demonstrar se a atual versão confere com a seleciona pelo Ruby Version Manager. Em seguida chamaremos o algoritmo precedido pelo comando time que nos retornará os 3 tempos que utilizaremos em nossos comentários sobre o desempenho dos interpretadores, porém antes disso na linha que sucede o comando time temos o valor de memória utilizado, valor este que é retornado pela última linha do arquivo que contém o algoritmo Tak.

Para testar o algoritmo utilizaremos os interpretadores:

Tabela 10.1: Interpretadores

Ruby Enterprise Edition 1.8.7 2010.01
Ruby MRI ruby 1.8.6 p383
Ruby MRI ruby 1.8.7 p248
Ruby MRI ruby 1.9.1 p378
JRuby 1.4.0

11 Testes

Dentro deste capítulo testaremos alguns algoritmos selecionados para testar a performance dos 5 interpretadores escolhidos. Visualizaremos o código-fonte do algoritmo, a maneira como invocamos o algoritmo pelo console do sistema operacional, a tabela com os resultados dos tempos medidos bem como a utilização de memória e gráficos para melhor visualizar os resultados obtidos.

Todos os resultados podem ser conferidos na planilha disponibilizada no Github em `http://github.com/wilsondealmeida/interpretadores-ruby`, no arquivo resultados-dos-testes.xls.

11.1 Tarai

Tarai é um algoritmo que se encontra no repositório de benchmarks da linguagem ruby [1]. O algoritmo testa a recursividade. Chamaremos a função por *tarai(12, 6, 0)* retornando o valor inteiro 12 como resultado. Serão realizadas 12.604.861 chamadas ao método *tarai*.

Na primeira linha do arquivo está a assinatura do método, onde temos o nome da função e os argumentos para a mesma. Na segunda linha inicia uma estrutura de controle que compara se o primeiro argumento (x) é menor ou igual o segundo argumento (y). Em caso de verdade a função retorna o valor do terceiro argumento (z). Porém, caso a condição da segunda linha não seja satisfeita, iremos invocar novamente o método *tarai* com os argumentos que serão também retornados por invocações ao mesmo método com os argumentos alterados em suas posições como podemos ver nas linhas 4 a 6. O processo continuará invocando o método recursivamente até que a condição da segunda linha seja satisfeita e o valor 12 seja retornado.

tarai.rb

```
1  def tarai( x, y, z )
2    if x <= y
3    then y
4    else tarai(tarai(x-1, y, z),
5               tarai(y-1, z, x),
6               tarai(z-1, x, y))
7    end
8  end
9
10 tarai(12, 6, 0)
```

Selecionaremos um dos interpretadores e chamaremos a função *time* para demonstrar os tempos resultantes. Esse passo será executado para cada um dos interpretadores 10 vezes. Posteriormente iremos relatar as médias.

tarai.sh

```
1  #!/bin/bash
2
```

[1] http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/benchmark/bm_app_tarai.rb?view=markup

```
3  TIMES=10;
4
5  rvm use jruby; ruby --version;
6  COUNTER=0;
7  while [ $COUNTER -lt $TIMES ]; do
8  time ruby tarai-java.rb; let COUNTER=COUNTER+1;
9  done;
10
11 rvm use ree; ruby --version;
12 COUNTER=0;
13 while [ $COUNTER -lt $TIMES ]; do
14 time ruby tarai-ruby.rb; let COUNTER=COUNTER+1;
15 done;
16
17 rvm use ruby-1.8.6-p383; ruby --version;
18 COUNTER=0;
19 while [ $COUNTER -lt $TIMES ]; do
20 time ruby tarai-ruby.rb; let COUNTER=COUNTER+1;
21 done;
22
23 rvm use ruby-1.8.7-p248; ruby --version;
24 COUNTER=0;
25 while [ $COUNTER -lt $TIMES ]; do
26 time ruby tarai-ruby.rb; let COUNTER=COUNTER+1;
27 done;
28
29 rvm use ruby-1.9.1-p378; ruby --version;
30 COUNTER=0;
31 while [ $COUNTER -lt $TIMES ]; do
32 time ruby tarai-ruby.rb; let COUNTER=COUNTER+1;
33 done;
```

Como resultado temos a média os tempos medidos em segundos que são: tempo real, tempo do usuário e tempo do sistema. A média do uso da memória RAM obtido é medido em kylobytes.

Tabela 11.1: Resultados do Algoritmo Tarai - Média de tempo medido em segundos e média de consumo memória em kylobytes

Interpretador	Tempo Total	Tempo do usuário	Tempo do Sistema	Memória
JRuby	4.716	4.568	0.190	30557.2
REE	9.077	9.050	0.022	2405.6
MRI 1.8.6	11.591	11.561	0.021	1688
MRI 1.8.7	11.863	11.839	0.016	1750
MRI 1.9.1	2.942	2.924	0.014	2750

Podemos observar os resultados do teste da performance na forma de gráficos.

Comparando o tempo real da tarefa, o tempo que simboliza o total gasto para a execução da tarefa, o Ruby MRI 1.9.1 e o JRuby competiram pela primeira posição, porém o Ruby MRI 1.9.1 conseguiu um concluir no menor tempo e consumindo muito menos memória que o JRruby.

Na comparação do tempo do usuário temos os mesmos resultados com posições praticamente idênticas, o Ruby MRI 1.9.1 novamente surge como opção mais rápida e com um consumo de memória próximo do primeiro lugar.

No tempo utilizado pelo sistema, o tempo que do kernel, temos um agrupamento dos 4 interpretadores sem contar o JRuby. Todos próximos em relação ao tempo e memória utilizada.

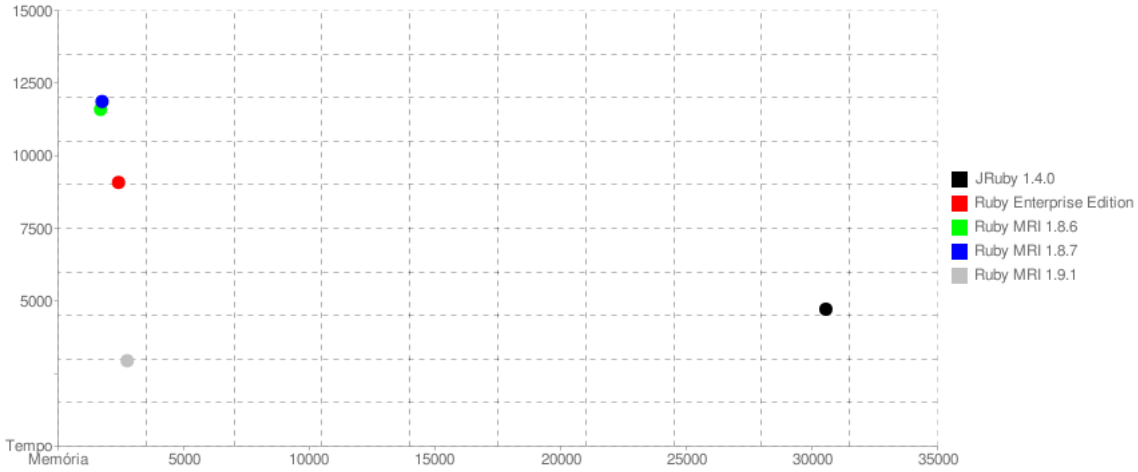

Figura 11.1: Algoritmo Tarai - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes)

No gráfico da figura 11.1 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM.

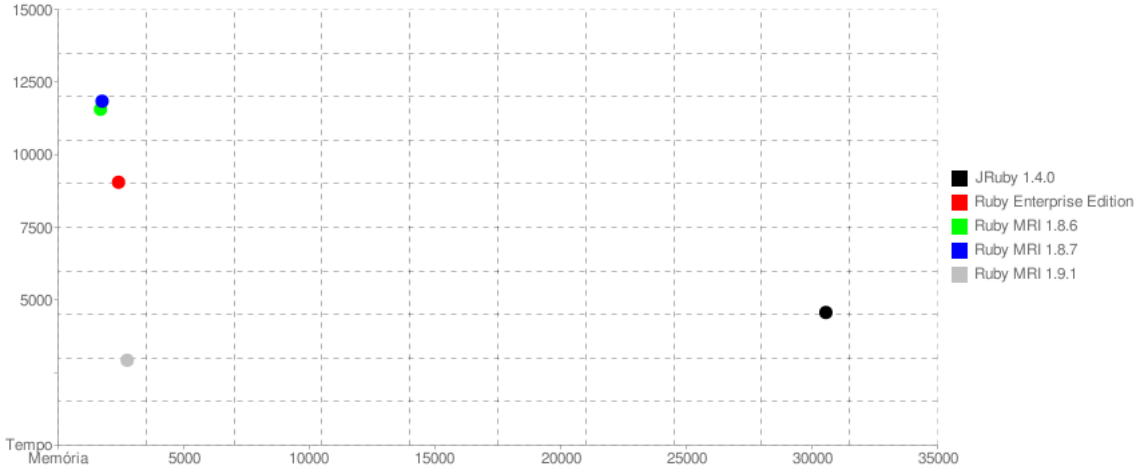

Figura 11.2: Algoritmo Tarai - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes)

No gráfico da figura 11.2 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM.

No gráfico da figura 11.3 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de memória RAM.

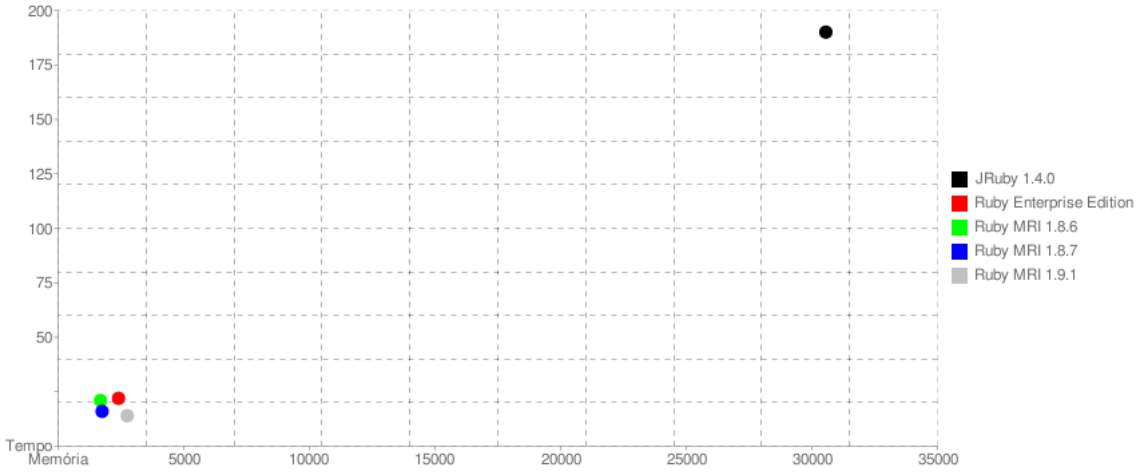

Figura 11.3: Algoritmo Tarai - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes)

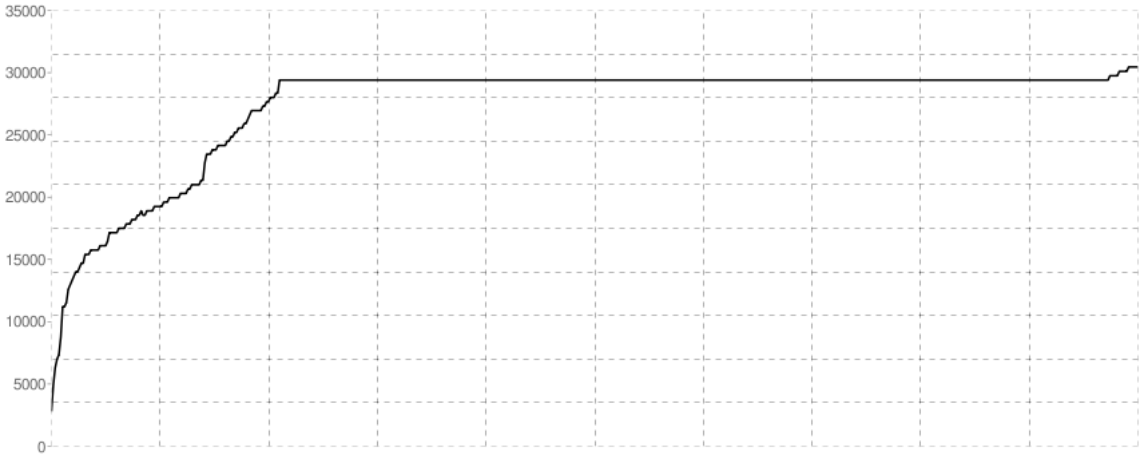

Figura 11.4: Algoritmo Tarai - Linha do tempo do uso de memória RAM pelo JRuby em kilobytes.

No gráfico da figura 11.4 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador JRuby.

No gráfico da figura 11.5 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby Enterprise Edition.

No gráfico da figura 11.6 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.6.

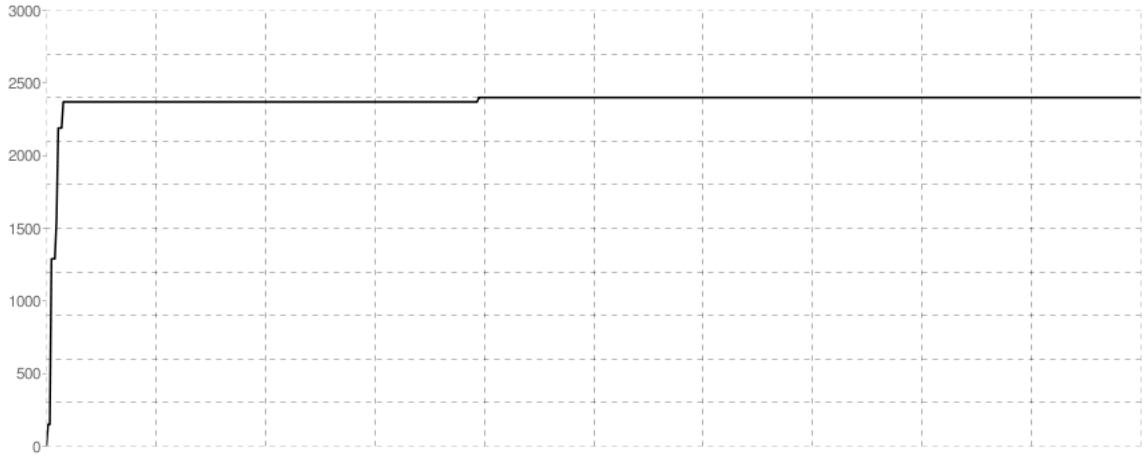

Figura 11.5: Algoritmo Tarai - Linha do tempo do uso de memória RAM pelo Ruby Enterprise Edition em kilobytes.

Figura 11.6: Algoritmo Tarai - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.6 em kilobytes.

No gráfico da figura 11.7 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.7.

No gráfico da figura 11.8 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.9.1.

Figura 11.7: Algoritmo Tarai - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.7 em kilobytes.

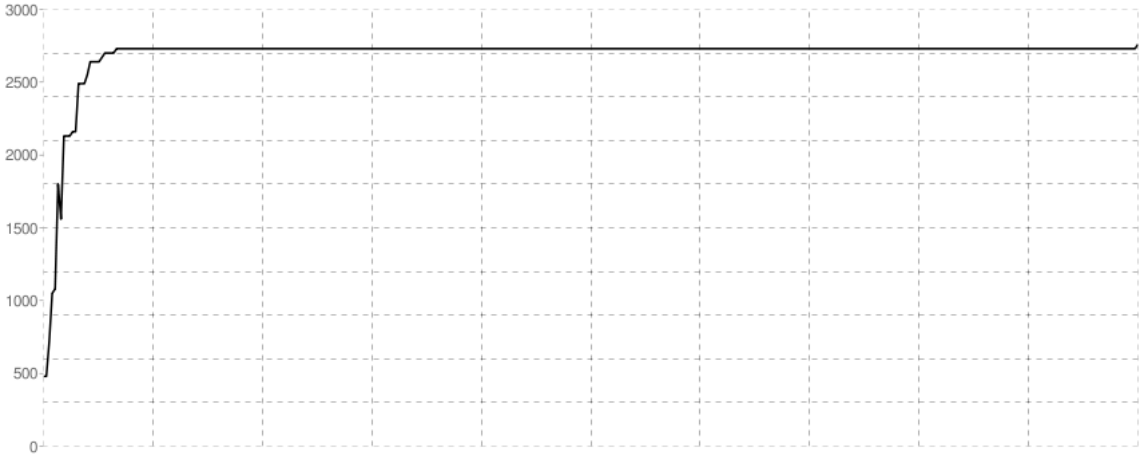

Figura 11.8: Algoritmo Tarai - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.9.1 em kilobytes.

11.2 Tak

Tak é uma variante da função de Takeuchi ao qual o japonês Ikuo Takeuchi usava como um simples teste para *benchmark*. Segundo Gabriel [1985] é um bom algoritmo para testar chamadas de função e recursividade. O algoritmo encontra-se no repositório de benchmarks do projeto Ruby [2] e no próprio livro citado em versão Lisp.

Na primeira linha do arquivo está a assinatura do método, onde temos o nome da função e os argumentos para a mesma. Na segunda linha inicia uma estrutura de controle que compara se o primeiro argumento (x) é maior ou igual o segundo argumento

[2] `http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/benchmark/bm_app_tak.rb?view=markup`

(y). Em caso de verdade a função retorna o valor do terceiro argumento (z). Porém, caso a condição da segunda linha não seja satisfeita, iremos invocar novamente o método *tak* com os argumentos que serão também retornados por invocações ao mesmo método com os argumentos alterados em suas posições como podemos ver nas linhas 5 a 6. O processo continuará invocando o método recursivamente até que a condição da segunda linha seja satisfeita e o valor 7 seja retornado.

Versão Lisp:

tak.lisp

```
1  (defun tak (x y z)
2    (if (not (< y x))
3        z
4        (tak (tak (1- x) y z)
5             (tak (1- y) z x)
6             (tak (1- z) x y))))
```

Versão Ruby:

tak.rb

```
1  def tak(x, y, z)
2    unless y < x
3      z
4    else
5      tak( tak(x-1, y, z),
6           tak(y-1, z, x),
7           tak(z-1, x, y))
8    end
9  end
10
11 tak(18, 12, 6)
```

Chamaremos a função por *tak(18, 12, 6)* que resultará no valor 7, e serão feitas 63.609 chamadas para a função e a recursão não se aprofundará mais que 18 níveis. Como esta função é apropriada para avaliação numérica, apenas números inteiros são possíveis de serem utilizados como argumento na função.

Selecionaremos um dos interpretadores e chamaremos a função *time* para demonstrar os tempos resultantes. Esse passo será executado para cada um dos interpretadores.

tak.sh

```
1  #!/bin/bash
2
3  TIMES=10;
4
5  rvm use jruby; ruby --version;
6  COUNTER=0
7  while [ $COUNTER -lt $TIMES ]; do
8  time ruby tak-java.rb; let COUNTER=COUNTER+1;
9  done;
10
11 rvm use ree; ruby --version;
12 COUNTER=0
13 while [ $COUNTER -lt $TIMES ]; do
14 time ruby tak-ruby.rb; let COUNTER=COUNTER+1;
15 done;
16
17 rvm use ruby-1.8.6-p383; ruby --version;
18 COUNTER=0
19 while [ $COUNTER -lt $TIMES ]; do
20 time ruby tak-ruby.rb; let COUNTER=COUNTER+1;
21 done;
22
23 rvm use ruby-1.8.7-p248; ruby --version;
24 COUNTER=0
25 while [ $COUNTER -lt $TIMES ]; do
26 time ruby tak-ruby.rb; let COUNTER=COUNTER+1;
27 done;
28
29 rvm use ruby-1.9.1-p378; ruby --version;
30 COUNTER=0
31 while [ $COUNTER -lt $TIMES ]; do
```

```
32 time ruby tak-ruby.rb; let COUNTER=COUNTER+1;
33 done;
```

Como resultado temos os tempos medidos em segundos. Tempo real, tempo do usuário e tempo do sistema. O uso da memória RAM obtido é medido em kylobytes.

Tabela 11.2: Resultados do Algoritmo Tak - Tempo medido em segundos e memória em kylobytes

Interpretador	Tempo Total	Tempo do usuário	Tempo do Sistema	Memória
JRuby	1.289	1.183	0.163	30507.6
REE	0.066	0.056	0.010	2364.4
MRI 1.8.6	0.090	0.068	0.008	1600
MRI 1.8.7	0.076	0.067	0.009	1662.4
MRI 1.9.1	0.055	0.033	0.012	2749.2

Podemos observar os resultados do teste da performance na forma de gráficos.

Os primeiros gráficos serão mostrados sem a presença do JRuby, pois a quantidade de memória consumida e o tempo levado para a execução do algoritmo ficaram muito acima dos outros 4 interpretadores. A segunda parte dos gráficos aponta a disparidade do JRuby no uso de memória e nos tempos utilizados.

Visualizando os aspectos medidos que são o tempo utilizado pela máquina para completar o algoritmo e o consumo de memória RAM para efetuar a tarefa, conclui-se que no algoritmo *Tak* a versão 1.9.1 do Ruby MRI tem o tempo mais baixo pra efetuar a tarefa nos tempos, real e do usuário, medidos pelo comando *time*, porém descartando o JRuby que utiliza a Máquina Virtual Java para executar, ele consome o maior montante de memória.

Pode-se concluir que este algoritmo demonstra uma relação entre a ocupação de memória e o tempo gasto para a execução do algoritmo entre as versões utilizadas para teste nos tempos real e do usuário. JRuby diferencia-se pelo uso de memória e pelo tempo gasto nos 3 sentidos ficando longe dos outros concorrentes. Levando em consideração que o tempo real e o tempo do usuário fica mais explícito ainda a relação da memória com o tempo entre os 4 concorrentes, sem o JRuby, pois o gráfico praticamente forma uma linha reta decrescente na relação do memória/tempo.

Considerando o tempo do sistema, os tempos ficam bem próximos porém o Ruby MRI 1.9.1 realizou a tarefa mais rápido também.

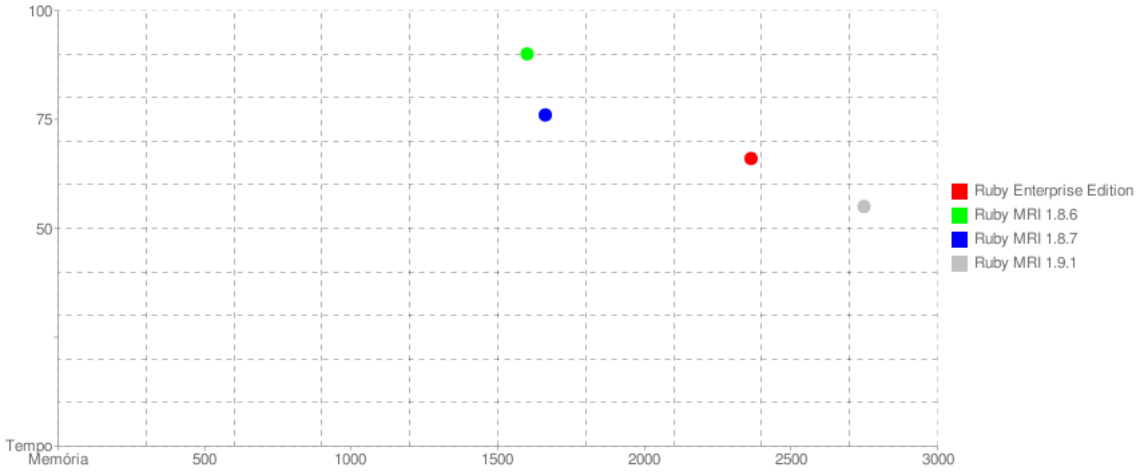

Figura 11.9: Algoritmo Tak - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes) sem a presença do JRuby no gráfico

No gráfico da figura 11.9 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM sem a presença do interpretador JRuby.

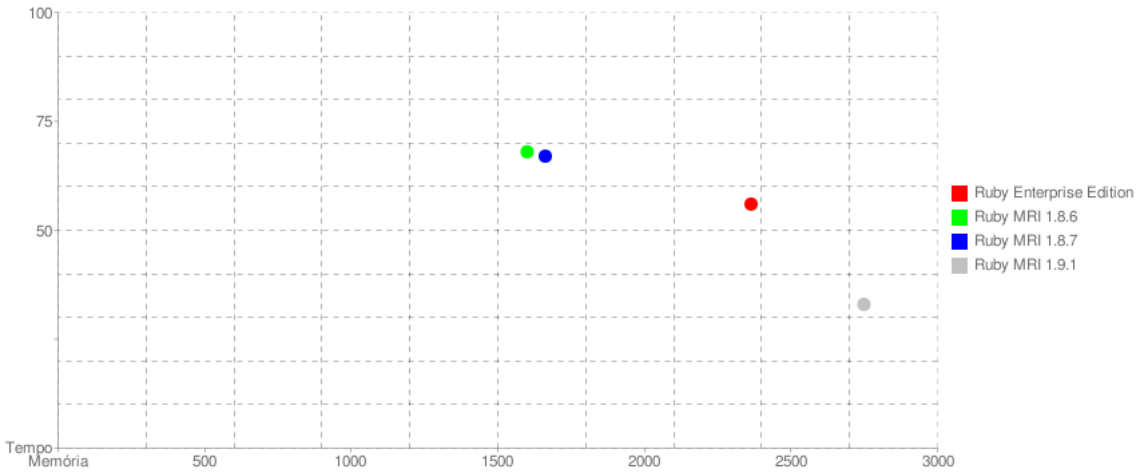

Figura 11.10: Algoritmo Tak - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes) sem a presença do JRuby no gráfico

No gráfico da figura 11.10 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM sem a presença do interpretador JRuby.

No gráfico da figura 11.11 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de

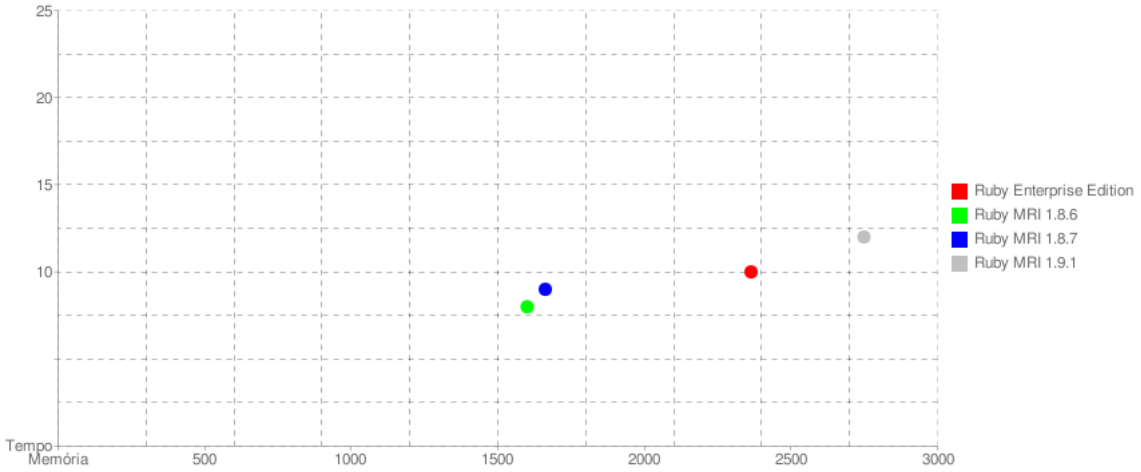

Figura 11.11: Algoritmo Tak - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes) sem a presença do JRuby no gráfico

memória RAM sem a presença do interpretador JRuby.

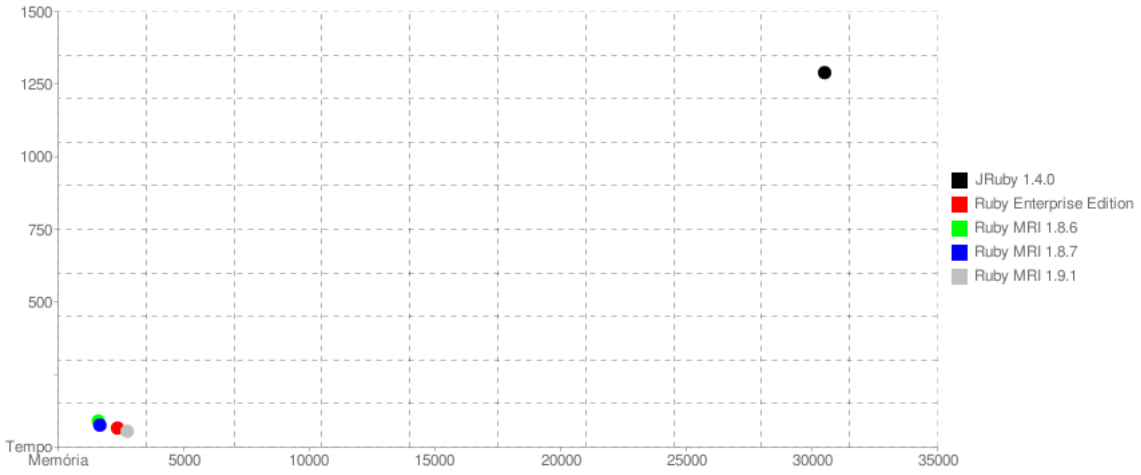

Figura 11.12: Algoritmo Tak - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes)

No gráfico da figura 11.12 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM.

No gráfico da figura 11.13 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM.

No gráfico da figura 11.14 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de memória RAM.

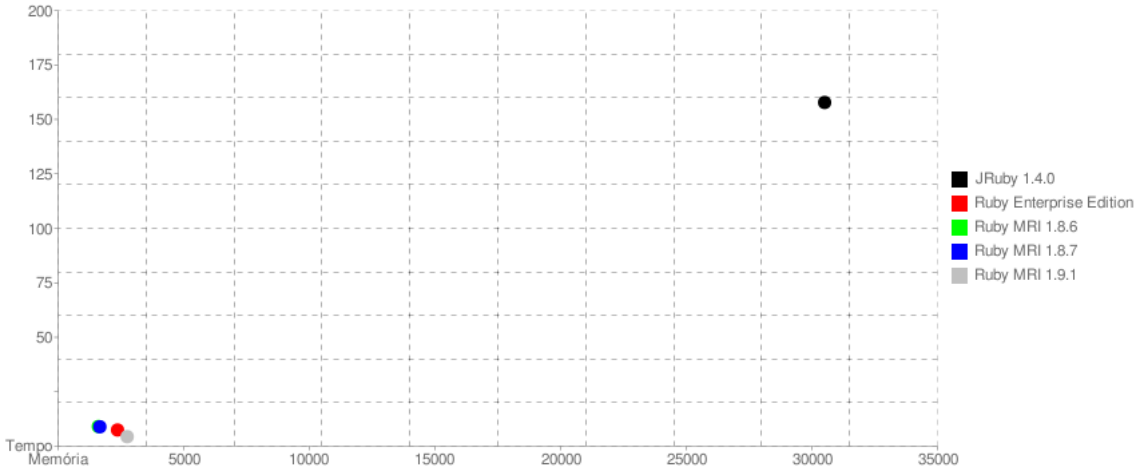

Figura 11.13: Algoritmo Tak - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes)

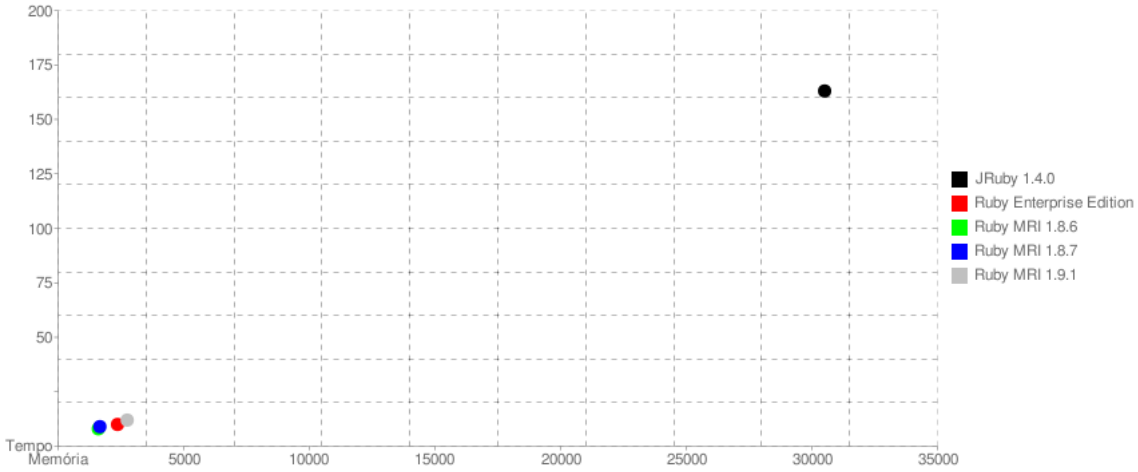

Figura 11.14: Algoritmo Tak - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes)

No gráfico da figura 11.15 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador JRuby.

No gráfico da figura 11.16 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby Enterprise Edition.

No gráfico da figura 11.17 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.6.

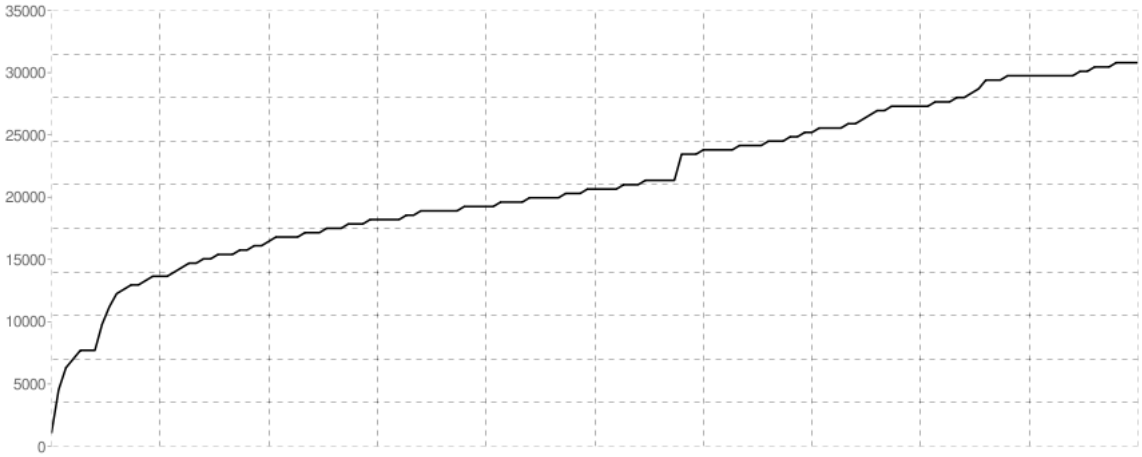

Figura 11.15: Algoritmo Tak - Linha do tempo do uso de memória RAM pelo JRuby em kilobytes

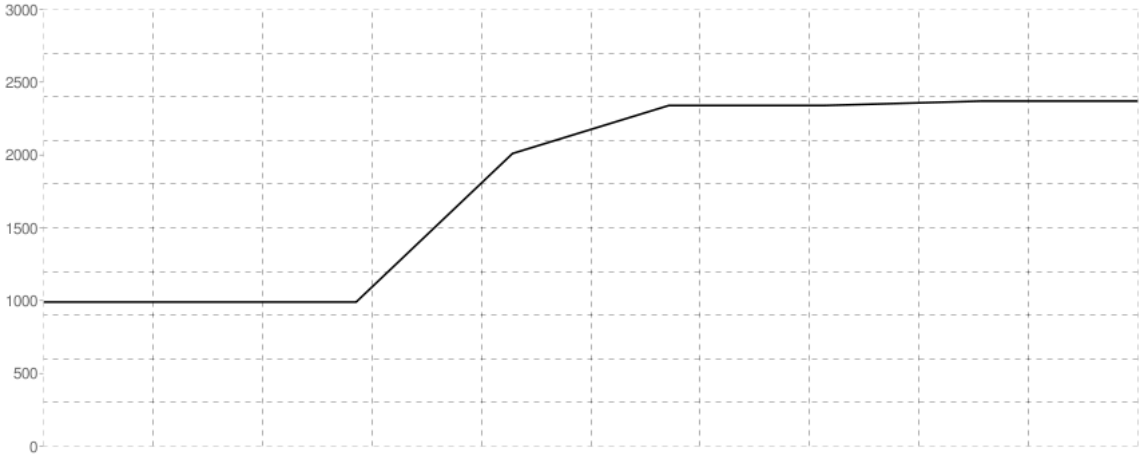

Figura 11.16: Algoritmo Tak - Linha do tempo do uso de memória RAM pelo Ruby Enterprise Edition em kilobytes

No gráfico da figura 11.18 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.7.

No gráfico da figura 11.19 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.9.1.

Figura 11.17: Algoritmo Tak - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.6 em kilobytes

Figura 11.18: Algoritmo Tak - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.7 em kilobytes

11.3 CTak

CTak é uma variante do algoritmo Tak que utiliza o disparo de exceção, por meio das cláusulas *catch* e *throw*, para retornar os valores ao invés de simplesmente chamar a função diretamente para retornar o valor. Gabriel [1985]

No algoritmo que segue temos 2 métodos, um o método *ctak* propriamente dito, e um método auxiliar que irá fazer o papel de disparar as exceções. Nas 3 primeiras linhas do arquivo temos a assinatura do método, onde temos o nome da função e os argumentos para a mesma. Dentro do primeiro método temos apenas uma linha que irá invocar o método auxiliar com os mesmos argumentos recebidos, porém esta linha está

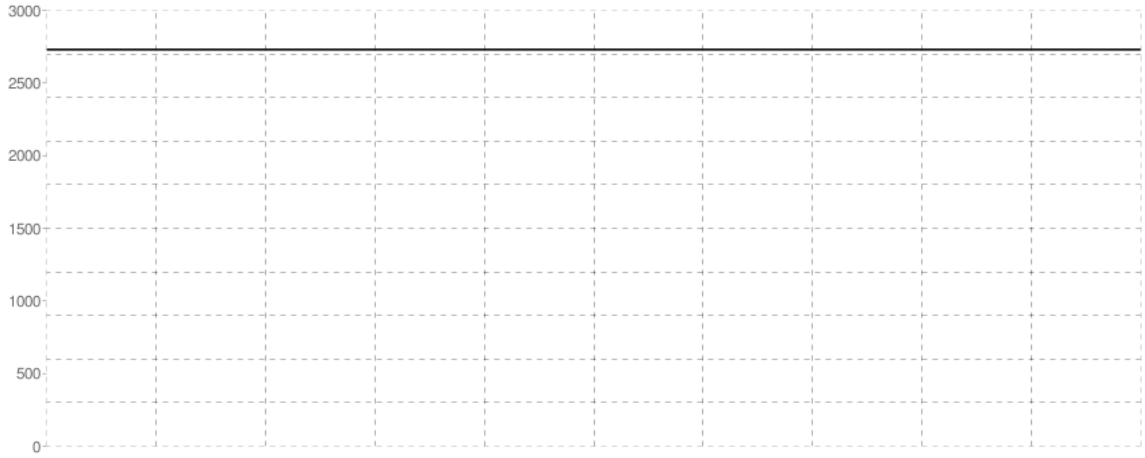

Figura 11.19: Algoritmo Tak - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.9.1 em kilobytes

inclusa dentro da cláusula *catch* da linguagem Ruby, que recebe um símbolo (*:ctak*) como argumento.

O método auxiliar inicia-se na linha 5, onde temos a assinatura do método, e na linha posterior inicia-se uma estrutura de controle que compara se o segundo argumento (y) é menor ou igual o primeiro argumento (x) e retorna o valor inverso do resultado obtido. Em caso de verdade a função retorna o valor do terceiro argumento (z) disparando uma exceção, através da cláusula *throw* com o argumento *:ctak* que retorna o valor de z para o método que a invocou e que tenha o argumento *:ctak* também. Porém, caso a condição da linha 6 não seja satisfeita, iremos invocar novamente o método auxiliar com os argumentos que serão também retornados por invocações ao mesmo método com os argumentos alterados em suas posições como podemos ver nas linhas 10 a 12, todos eles inclusos dentro da clásula *catch* com o argumento *:ctak* para receber o valor retornado pelo disparo da linha 7 que contém a cláusula *throw*. O processo continuará invocando o método recursivamente até que a condição da segunda linha seja satisfeita e o valor 7 seja retornado.

Versão Lisp:

ctak.lisp

```
1 (defun ctak (x y z)
2   (catch 'ctak (ctak-aux x y z)))
3 (defun ctak-aux (x y z)
4   (cond ((not (< y x))
```

```
5           (throw 'ctak z))
6         (t (ctak-aux
7              (catch 'ctak
8                (ctak-aux (1- x)
9                          y
10                         z))
11             (catch 'ctak
12               (ctak-aux (1- y)
13                         z
14                         x))
15             (catch 'ctak
16               (ctak-aux (1- z)
17                         x
18                         y))))))
```

Versão Ruby:

ctak.rb

```
1  def ctak(x, y, z)
2    catch(:ctak) { ctak_aux(x, y, z) }
3  end
4
5  def ctak_aux (x, y, z)
6    if !(y < x)
7      throw :ctak, z
8    else
9      ctak_aux(
10       catch(:ctak) { ctak_aux((x - 1), y, z) },
11       catch(:ctak) { ctak_aux((y - 1), z, x) },
12       catch(:ctak) { ctak_aux((z - 1), x, y) }
13     )
14   end
15 end
16
17 ctak(18, 12, 6)
```

Chamaremos a função por *ctak(18, 12, 6)* que resultará no valor 7, o mesmo

de Tak, e serão feitas as mesmas 63.609 chamadas para a função e a recursão não se aprofundará mais que 18 níveis. Serão disparadas 47.707 exceções durante a execução do algoritmo.

Selecionaremos um dos interpretadores e chamaremos a função *time* para demonstrar os tempos resultantes. Esse passo será executado para cada um dos interpretadores.

ctak.sh

```
1  #!/bin/bash
2
3  TIMES=10;
4
5  rvm use jruby; ruby --version;
6  COUNTER=0
7  while [ $COUNTER -lt $TIMES ]; do
8  time ruby ctak-java.rb; let COUNTER=COUNTER+1;
9  done;
10
11 rvm use ree; ruby --version;
12 COUNTER=0
13 while [ $COUNTER -lt $TIMES ]; do
14 time ruby ctak-ruby.rb; let COUNTER=COUNTER+1;
15 done;
16
17 rvm use ruby-1.8.6-p383; ruby --version;
18 COUNTER=0
19 while [ $COUNTER -lt $TIMES ]; do
20 time ruby ctak-ruby.rb; let COUNTER=COUNTER+1;
21 done;
22
23 rvm use ruby-1.8.7-p248; ruby --version;
24 COUNTER=0
25 while [ $COUNTER -lt $TIMES ]; do
26 time ruby ctak-ruby.rb; let COUNTER=COUNTER+1;
27 done;
```

```
28
29 rvm use ruby-1.9.1-p378; ruby --version;
30 COUNTER=0
31 while [ $COUNTER -lt $TIMES ]; do
32 time ruby ctak-ruby.rb; let COUNTER=COUNTER+1;
33 done;
```

Como resultado temos os tempos medidos em segundos. Tempo real, tempo do usuário e tempo do sistema. O uso da memória RAM obtido é medido em kylobytes.

Tabela 11.3: Resultados do Algoritmo CTak - Tempo medido em segundos e memória em kylobytes

Interpretador	Tempo Total	Tempo do usuário	Tempo do Sistema	Memória
JRuby	1.701	1.654	0.175	31337.6
REE	0.139	0.117	0.019	2402.4
MRI 1.8.6	0.134	0.117	0.017	1649.2
MRI 1.8.7	0.140	0.122	0.018	1704
MRI 1.9.1	0.107	0.084	0.024	2964.4

Podemos observar os resultados do teste da performance na forma de gráficos.

Os primeiros gráficos serão mostrados sem a presença do JRuby, pois a quantidade de memória consumida e o tempo levado para a execução do algoritmo ficaram muito acima dos outros 4 interpretadores. A segunda parte dos gráficos aponta a disparidade do JRuby no uso de memória e nos tempos utilizados.

Visualizando os aspectos medidos que são o tempo utilizado pela máquina para completar o algoritmo e o consumo de memória RAM para efetuar a tarefa, conclui-se que no algoritmo *CTak* a versão 1.9.1 do Ruby MRI tem o tempo mais baixo pra efetuar a tarefa em todos os 3 tempos medidos pelo comando *time*, porém descartando o JRuby que utiliza a Máquina Virtual Java para executar, ele consome o maior montante de memória.

No algoritmo *CTak* a utilização de memória não mostra a uma relação tão forte na questão tempo/memória, pois o tempo dos 4 interpretadores, excluindo JRuby ficaram muito próximos, o Ruby MRI 1.9.1 utilizando um pouco mais de memória. Porém no tempo do sistema ele utilizou um tempo baixíssimo em relação aos outros interpretadores.

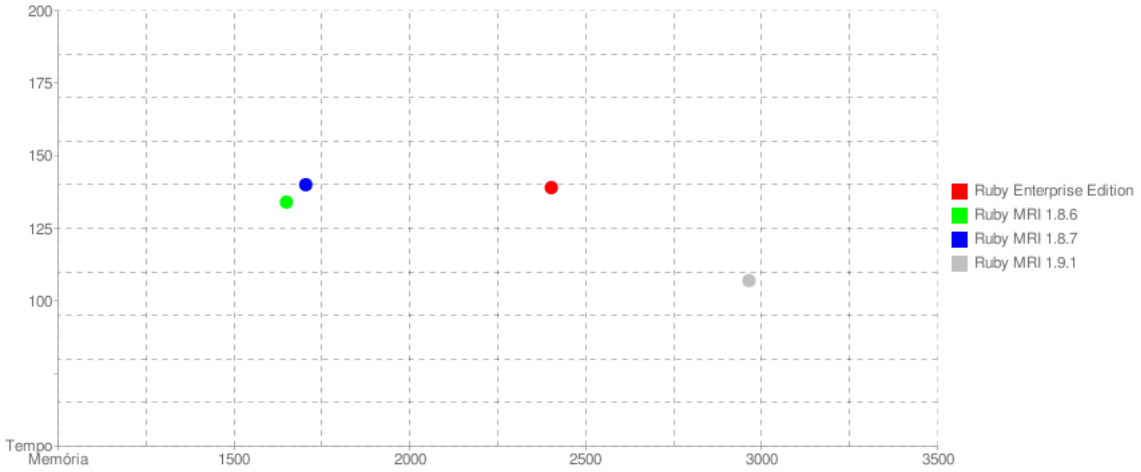

Figura 11.20: Algoritmo CTak - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes) sem a presença do JRuby no gráfico

Na gráfico da figura 11.20 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM sem a presença do interpretador JRuby.

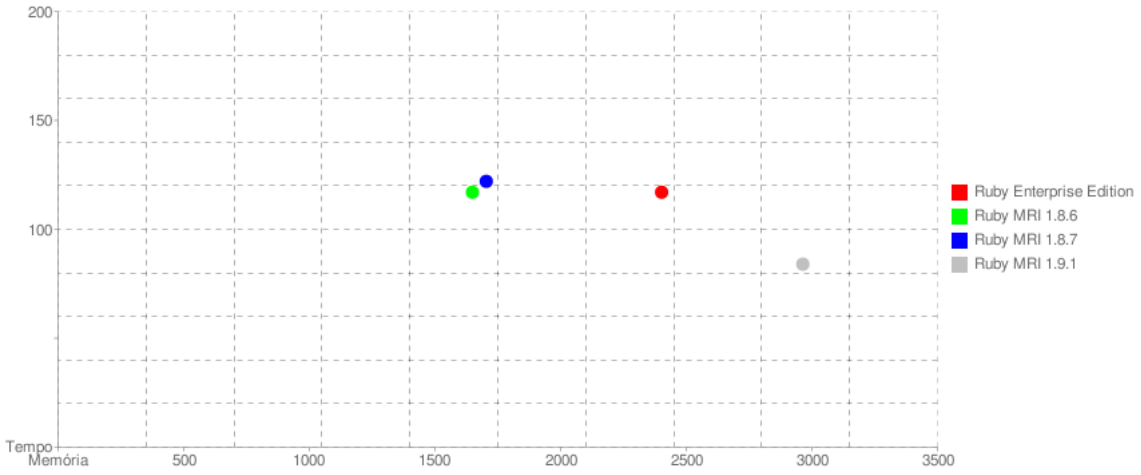

Figura 11.21: Algoritmo CTak - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes) sem a presença do JRuby no gráfico

Na gráfico da figura 11.21 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM sem a presença do interpretador JRuby.

Na gráfico da figura 11.22 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de memória RAM sem a presença do interpretador JRuby.

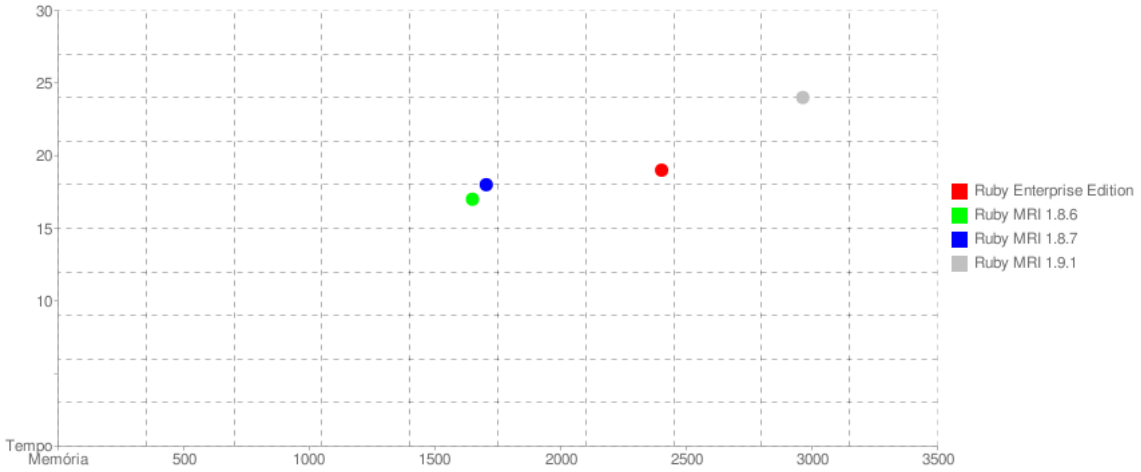

Figura 11.22: Algoritmo CTak - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes) sem a presença do JRuby no gráfico

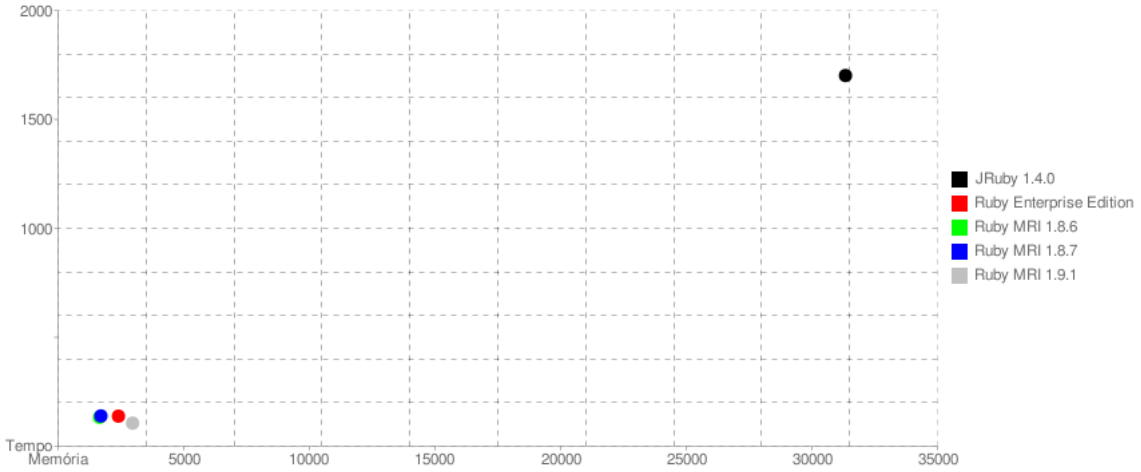

Figura 11.23: Algoritmo CTak - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes)

Na gráfico da figura 11.23 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM.

Na gráfico da figura 11.24 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM.

Na gráfico da figura 11.25 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de memória RAM.

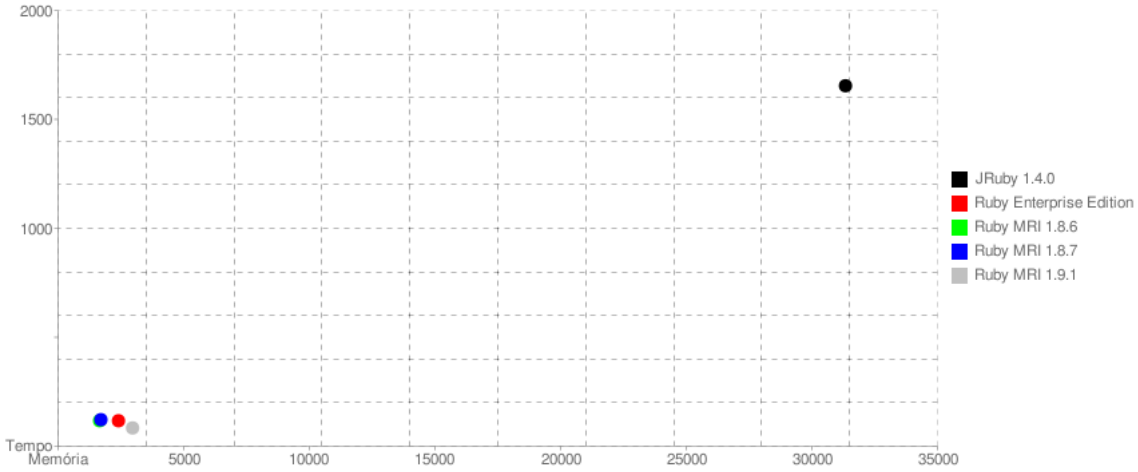

Figura 11.24: Algoritmo CTak - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes)

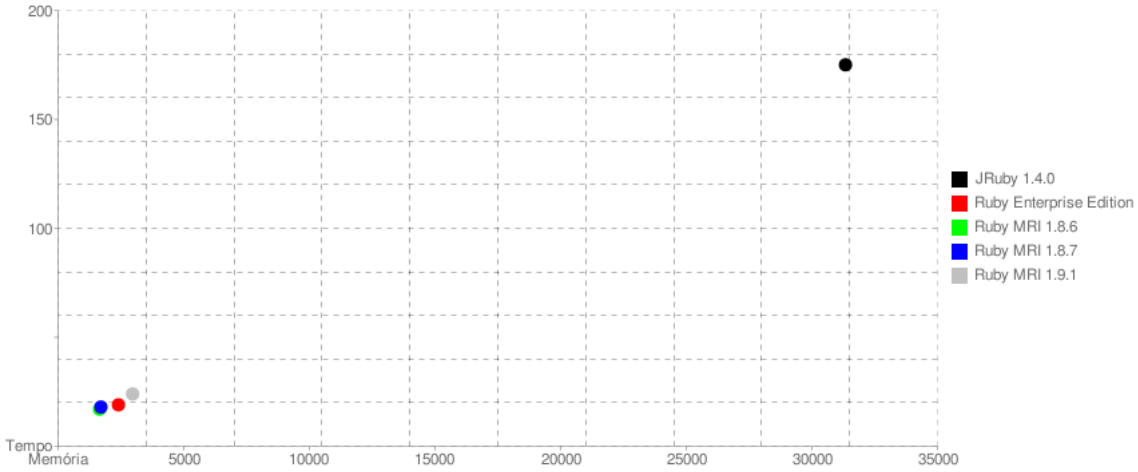

Figura 11.25: Algoritmo CTak - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes)

No gráfico da figura 11.26 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador JRuby.

No gráfico da figura 11.27 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby Enterprise Edition.

No gráfico da figura 11.28 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.6.

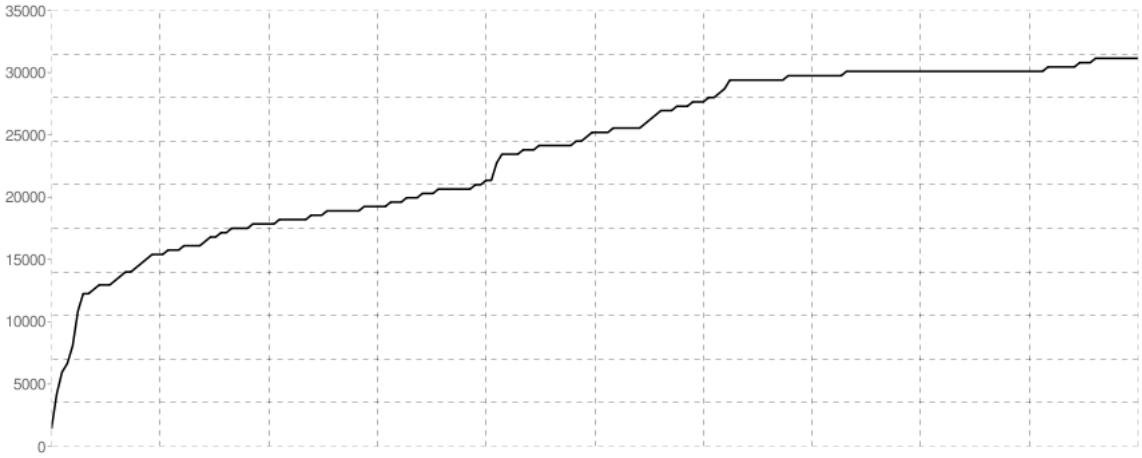

Figura 11.26: Algoritmo CTak - Linha do tempo do uso de memória RAM pelo JRuby em kilobytes

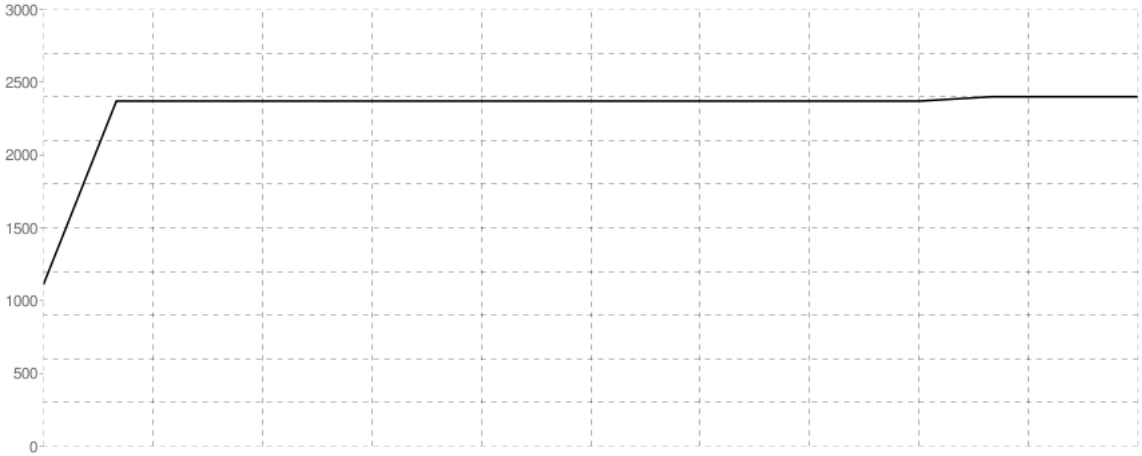

Figura 11.27: Algoritmo CTak - Linha do tempo do uso de memória RAM pelo Ruby Enterprise Edition em kilobytes

No gráfico da figura 11.29 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.7.

No gráfico da figura 11.30 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.9.1.

Figura 11.28: Algoritmo CTak - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.6 em kilobytes

Figura 11.29: Algoritmo CTak - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.7 em kilobytes

11.4 Escrita em arquivos

Outra tarefa muito utilizada pelas linguagens de programação em geral é a escrita em arquivos. No livro Performance and Evaluation of Lisp Systems Gabriel [1985] encontramos dentre os últimos algoritmos testados, o teste de escrita de arquivo.

O algoritmo utilizado é encontrado também no repositório de benchmarks do Ruby MRI [3]. Apenas alterado o nome do arquivo temporário que será criado e das variáveis para melhor visualização. O nome do arquivo será *escrita-de-arquivo.rb*.

[3] `http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/benchmark/bm_io_file_write.rb?view=markup`

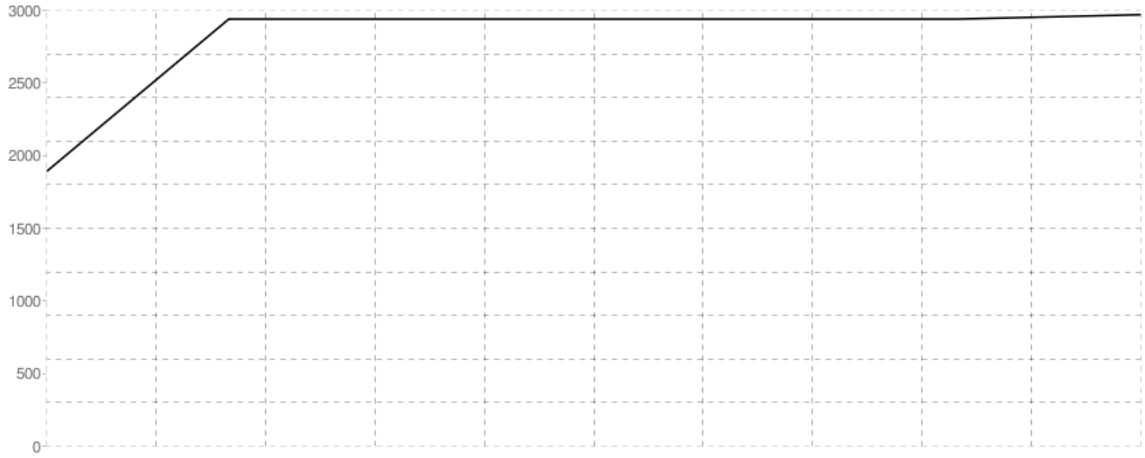

Figura 11.30: Algoritmo CTak - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.9.1 em kilobytes

escrita–de–arquivo.rb

```
1  require 'tempfile'
2
3  max = 20000
4  string = "Hello world! " * 1000
5  file = Tempfile.new('benchmark')
6
7  max.times {
8    file.seek(0)
9    file.write(string)
10 }
```

O algoritmo define as variáveis *max* como um número inteiro (20000) e string "Hello world! "multiplicado mil vezes e armazenado na variável *string*. Após isto cria-se um arquivo temporário que é apontado para a variável *file*. Logo após inicia-se o laço de repetição direciona-se o leitor para a posição inicial do arquivo e posteriormente escreve-se o conteúdo da variável *string* no arquivo temporário.

Para todos os 5 interpretadores utilizaremos a seleção da variável ruby por meio do Ruby Version Manager e logo após mediremos o tempo dos testes pelo comando *time*.

escrita–de–arquivo.sh

```
1  #!/bin/bash
```

```
2
3 TIMES=10;
4
5 rvm use jruby; ruby --version;
6 COUNTER=0
7 while [ $COUNTER -lt $TIMES ]; do
8 time ruby escrita-de-arquivo-java.rb; let COUNTER=COUNTER+1;
9 done;
10
11 rvm use ree; ruby --version;
12 COUNTER=0
13 while [ $COUNTER -lt $TIMES ]; do
14 time ruby escrita-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
15 done;
16
17 rvm use ruby-1.8.6-p383; ruby --version;
18 COUNTER=0
19 while [ $COUNTER -lt $TIMES ]; do
20 time ruby escrita-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
21 done;
22
23 rvm use ruby-1.8.7-p248; ruby --version;
24 COUNTER=0
25 while [ $COUNTER -lt $TIMES ]; do
26 time ruby escrita-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
27 done;
28
29 rvm use ruby-1.9.1-p378; ruby --version;
30 COUNTER=0
31 while [ $COUNTER -lt $TIMES ]; do
32 time ruby escrita-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
33 done;
```

Como resultado temos os tempos medidos em segundos. Tempo real, tempo do usuário e tempo do sistema. O uso da memória RAM obtido é medido em kylobytes.

Tabela 11.4: Resultados da escrita em arquivos - Tempo medido em segundos e memória em kylobytes

Interpretador	Tempo Total	Tempo do usuário	Tempo do Sistema	Memória
JRuby	2.099	1.981	0.450	36984.4
REE	0.532	0.227	0.304	3446
MRI 1.8.6	0.479	0.173	0.299	2386.4
MRI 1.8.7	0.522	0.234	0.286	2561.2
MRI 1.9.1	0.409	0.148	0.261	4029.2

Podemos observar os resultados do teste da performance na forma de gráficos.

Novamente percebemos que o JRuby utiliza mais memória RAM e tempo para executar a tarefa. Nos 3 tempos medidos o JRuby novamente levou mais tempo e utilizou mais recursos da máquina.

Uma disputa interessante alternou dentro dos outros 4 interpretadores a velocidade da execução da tarefa, tempos muito próximos. O Ruby MRI 1.9.1 utilizou um pouco mais de memória e foi o mais rápido o geral e no tempo do usuário. No tempo do sistema o Ruby MRI 1.8.6 foi o mais rápido e também foi o que menos utilizou memória para completar a tarefa.

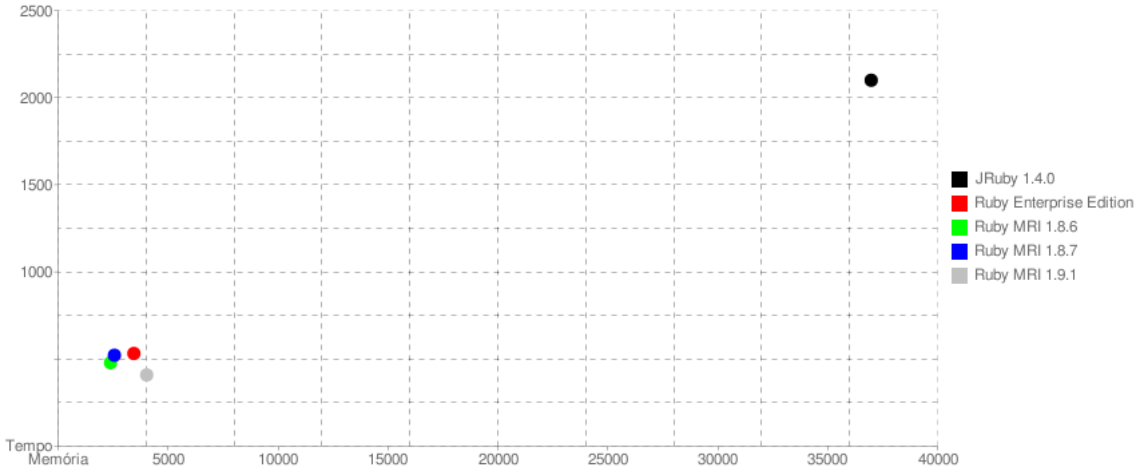

Figura 11.31: Algoritmo de escrita em arquivos - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes)

Na gráfico da figura 11.31 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM.

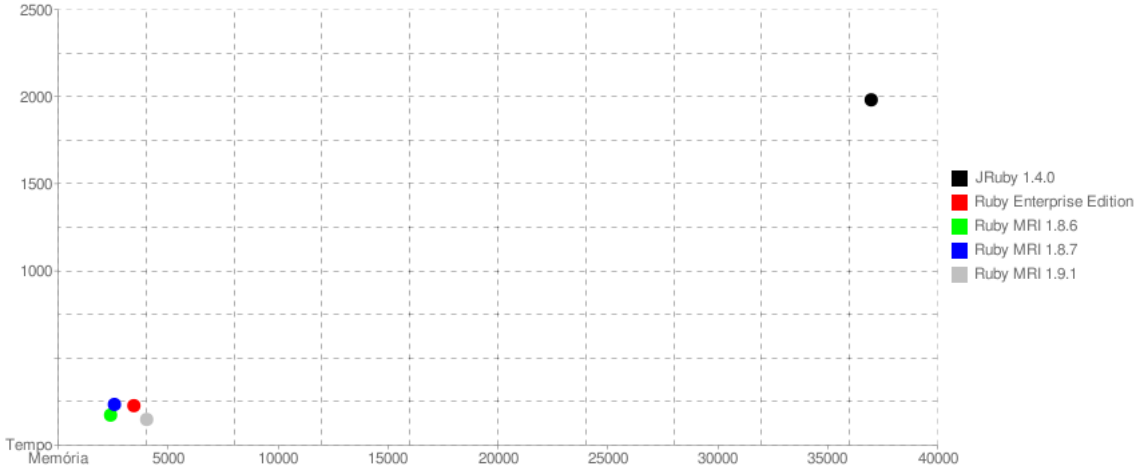

Figura 11.32: Algoritmo de escrita em arquivos - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes)

Na gráfico da figura 11.32 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM.

Na gráfico da figura 11.33 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de memória RAM.

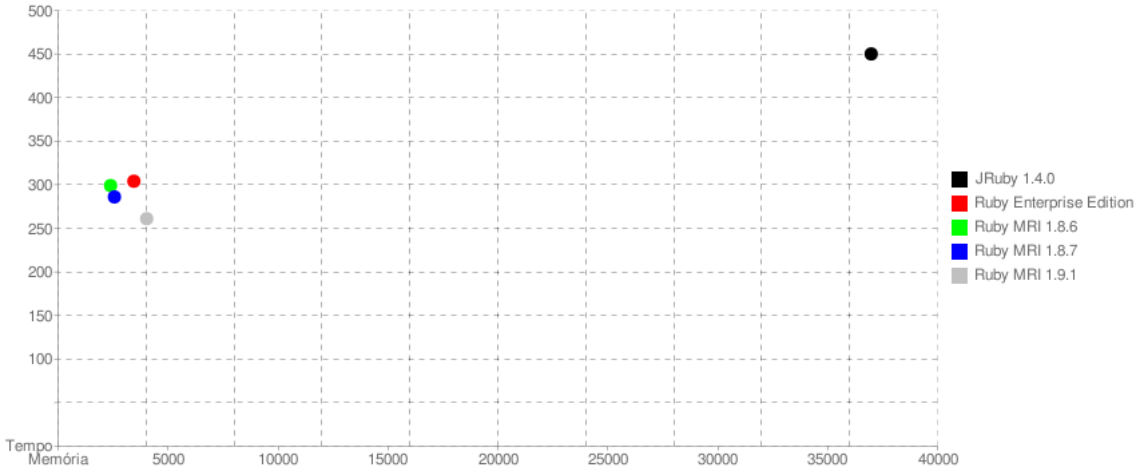

Figura 11.33: Algoritmo de escrita em arquivos - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes)

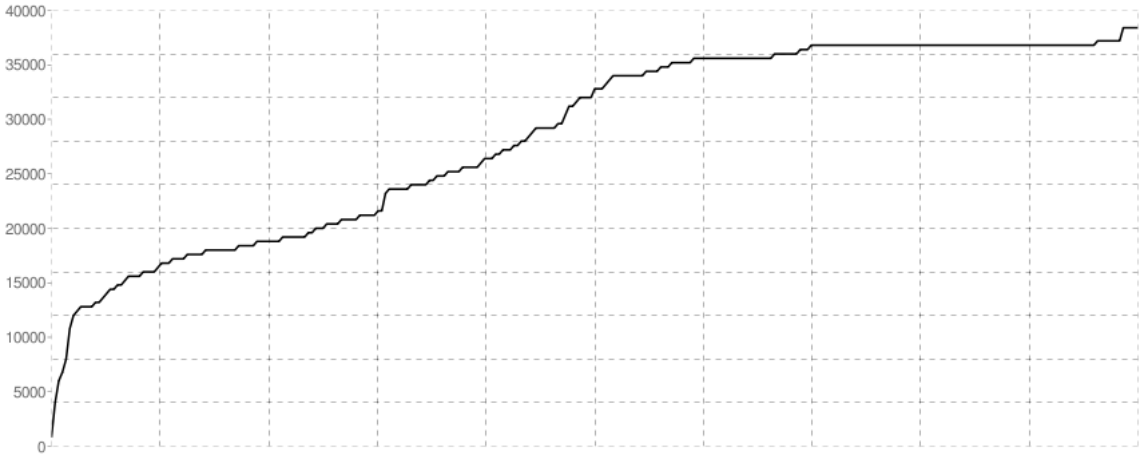

Figura 11.34: Algoritmo de escrita em arquivos - Linha do tempo do uso de memória RAM pelo JRuby em kilobytes

No gráfico da figura 11.34 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador JRuby.

No gráfico da figura 11.35 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby Enterprise Edition.

No gráfico da figura 11.36 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.6.

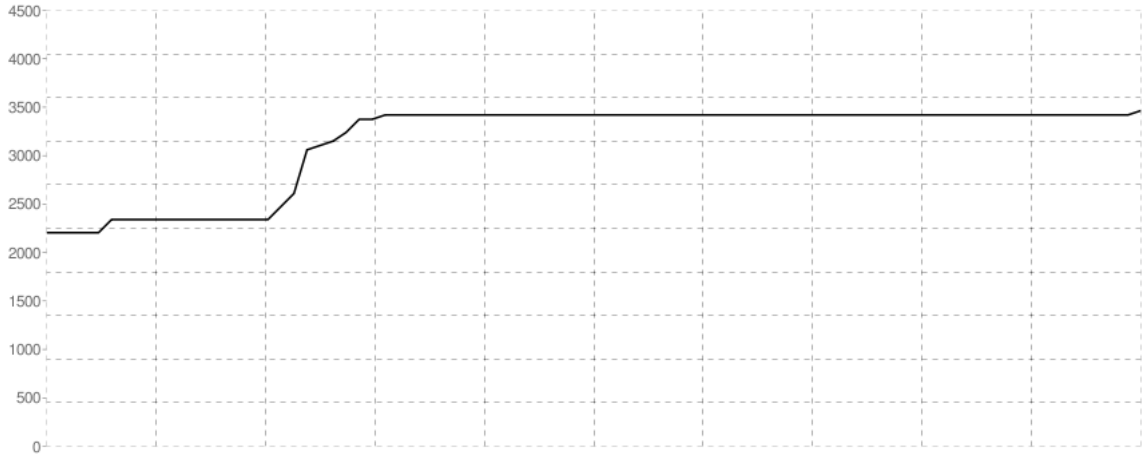

Figura 11.35: Algoritmo de escrita em arquivos - Linha do tempo do uso de memória RAM pelo Ruby Enterprise Edition em kilobytes

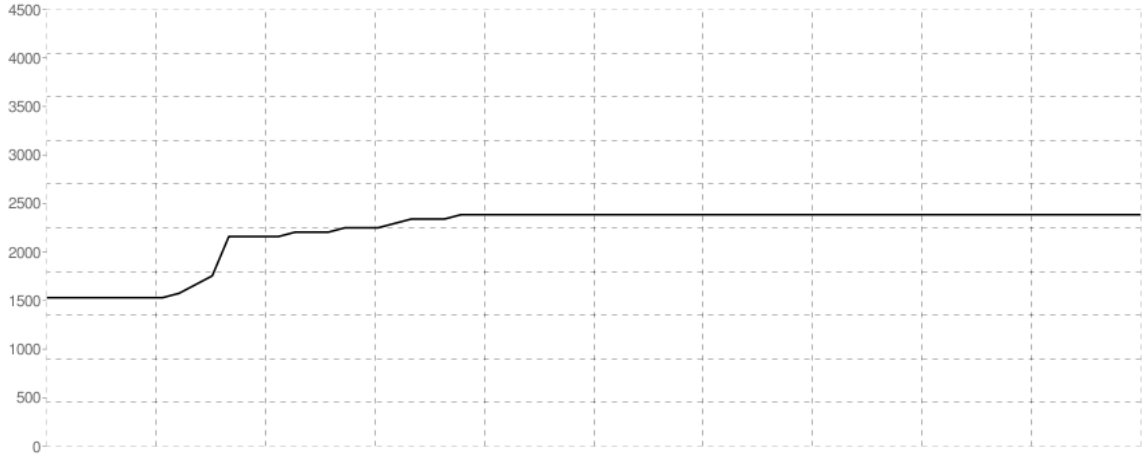

Figura 11.36: Algoritmo de escrita em arquivos - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.6 em kilobytes

No gráfico da figura 11.37 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.7.

No gráfico da figura 11.38 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.9.1.

Figura 11.37: Algoritmo de escrita em arquivos - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.7 em kilobytes

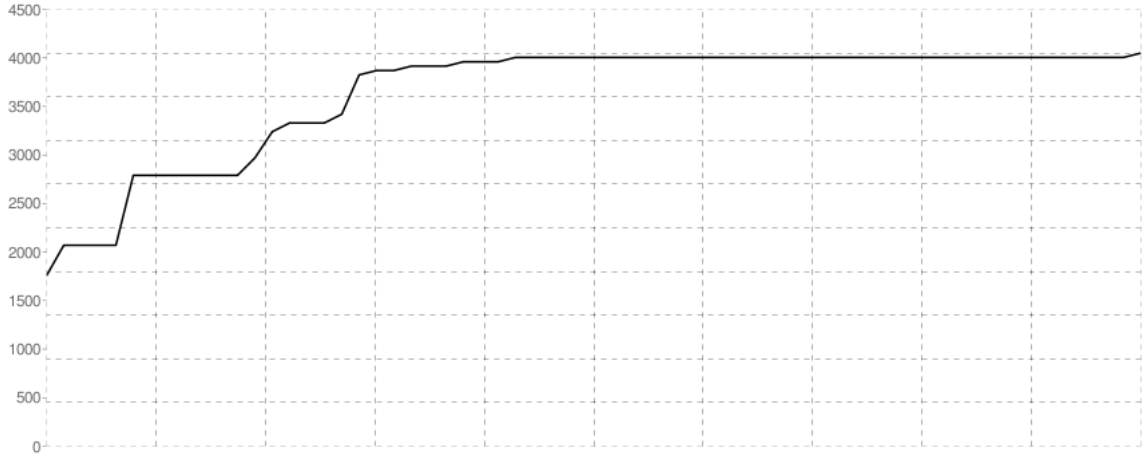

Figura 11.38: Algoritmo de escrita em arquivos - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.9.1 em kilobytes

11.5 Leitura de arquivos

Uma das tarefas que provavelmente é das mais importantes e utilizadas pelas linguagens de programação em geral é a leitura de arquivos. No livro Performance and Evaluation of Lisp Systems Gabriel [1985] encontramos dentre os últimos algoritmos testados, o teste de leitura de arquivo.

O algoritmo utilizado é encontrado também no repositório de benchmarks do Ruby MRI [4]. Apenas alterado o nome do arquivo temporário que será criado e das variáveis para melhor visualização. O nome do arquivo será *leitura-de-arquivo.rb*.

[4] `http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/benchmark/bm_io_file_read.rb?view=markup`

leitura–de–arquivo.rb

```
1  require 'tempfile'
2
3  max = 20000
4  string = "Hello world!  " * 1000
5  file = Tempfile.new('benchmark')
6  file.write(string)
7
8  max.times {
9     file.seek(0)
10    file.read
11 }
```

O algoritmo define as variáveis *max* como um número inteiro (20000) e string "Hello world! "multiplicado mil vezes e armazenado na variável *string*. Após isto cria-se um arquivo temporário que é apontado para a variável *file*. Logo após escrevemos o conteúdo da variável *string* no arquivo temporário. Num laço de repetição que vai rodar 20 mil vezes iremos direcionar o leitor na posição inicial e mandaremos o interpretador ler o conteúdo do arquivo.

Para todos os 5 interpretadores utilizaremos a seleção da variável ruby por meio do Ruby Version Manager e logo após mediremos o tempo dos testes pelo comando *time*.

leitura–de–arquivo.sh

```
1  #!/bin/bash
2
3  TIMES=10;
4
5  rvm use jruby; ruby --version;
6  COUNTER=0
7  while [ $COUNTER -lt $TIMES ]; do
8  time ruby leitura-de-arquivo-java.rb; let COUNTER=COUNTER+1;
9  done;
10
11 rvm use ree; ruby --version;
12 COUNTER=0
```

```
13 while [ $COUNTER -lt $TIMES ]; do
14 time ruby leitura-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
15 done;
16
17 rvm use ruby-1.8.6-p383; ruby --version;
18 COUNTER=0
19 while [ $COUNTER -lt $TIMES ]; do
20 time ruby leitura-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
21 done;
22
23 rvm use ruby-1.8.7-p248; ruby --version;
24 COUNTER=0
25 while [ $COUNTER -lt $TIMES ]; do
26 time ruby leitura-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
27 done;
28
29 rvm use ruby-1.9.1-p378; ruby --version;
30 COUNTER=0
31 while [ $COUNTER -lt $TIMES ]; do
32 time ruby leitura-de-arquivo-ruby.rb; let COUNTER=COUNTER+1;
33 done;
```

Como resultado temos os tempos medidos em segundos. Tempo real, tempo do usuário e tempo do sistema. O uso da memória RAM obtido é medido em kylobytes.

Tabela 11.5: Resultados da leitura em arquivos - Tempo medido em segundos e memória em kylobytes

Interpretador	Tempo Total	Tempo do usuário	Tempo do Sistema	Memória
JRuby	2.214	2.224	0.362	38321.6
REE	0.757	0.514	0.234	12375.2
MRI 1.8.6	0.975	0.447	0.526	2879.2
MRI 1.8.7	1.044	0.488	0.550	3422.8
MRI 1.9.1	1.024	0.825	0.198	6034.4

Podemos observar os resultados do teste da performance na forma de gráficos.

Comparando os aspectos medidos que são os tempos utilizados pela máquina para completar os algoritmos e o consumo de memória RAM para efetuar a tarefa, podemos ver um grande equilíbrio no fator tempo real da máquina entre os interpretadores Ruby MRI 1.8.6, Ruby MRI 1.8.7, Ruby MRI 1.9.1 e Ruby Enterprise Edition. No tempo real eles ficam muito próximos, com o Ruby Enterprise Edition consumindo um pouco a mais de memória. A tarefa é concluída por todos os interpretadores em tempos parecidos. Visualizando o tempo utilizado pelo sistema, temos um equilibrio maior, pois o JRuby consegue ficar muito próximo da melhor coloção. Um destaque para o Ruby Enterprise Edition que conseguiu executar e finalizar o laço de repetição em menos de 1 segundo no tempo real.

No fator memória o JRuby mais uma vez foi o que ocupou maior quantidade para executar e o Ruby MRI 1.8.6 foi o único que menos utilizou memória para a tarefa.

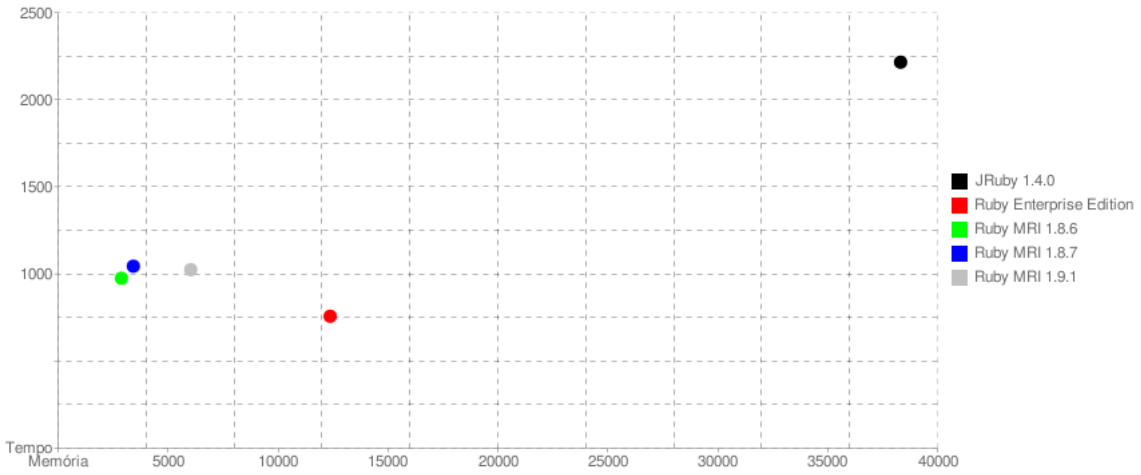

Figura 11.39: Algoritmo de leitura de arquivos - Média do Tempo real (milisegundos) por Média da Memória RAM utilizada (kilobytes)

Na gráfico da figura 11.39 pode-se visualizar as medidas da média de tempo real gasto pelo algoritmo pelo uso médio de memória RAM.

Na gráfico da figura 11.40 pode-se visualizar as medidas da média de tempo utilizado pelo usuário no sistema operacional em relação ao uso médio de memória RAM.

Na gráfico da figura 11.41 pode-se visualizar as medidas da média de tempo utilizado pela execução do algoritmo no sistema operacional em relação ao uso médio de memória RAM.

No gráfico da figura 11.42 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador JRuby.

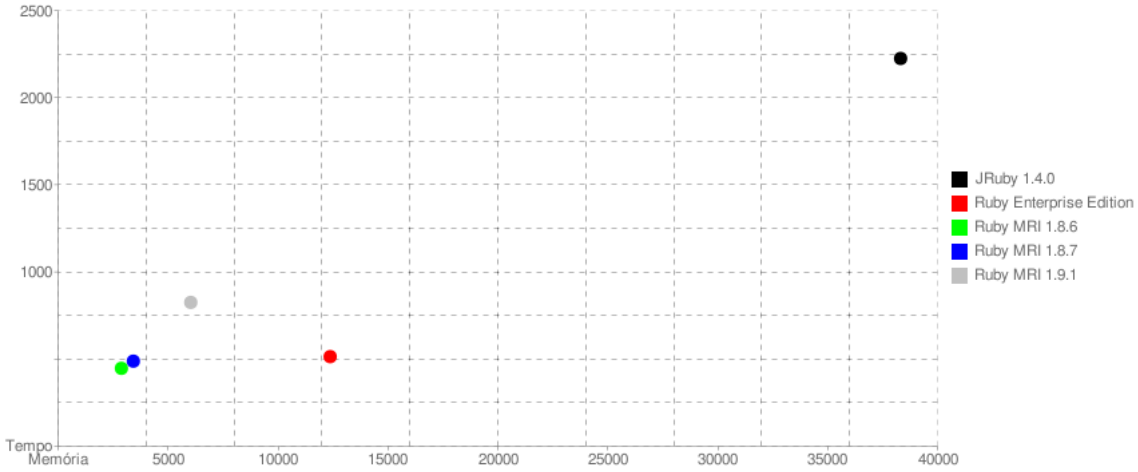

Figura 11.40: Algoritmo de leitura de arquivos - Média do Tempo do usuário (milisegundos) por Média da Memória RAM utilizada (kilobytes)

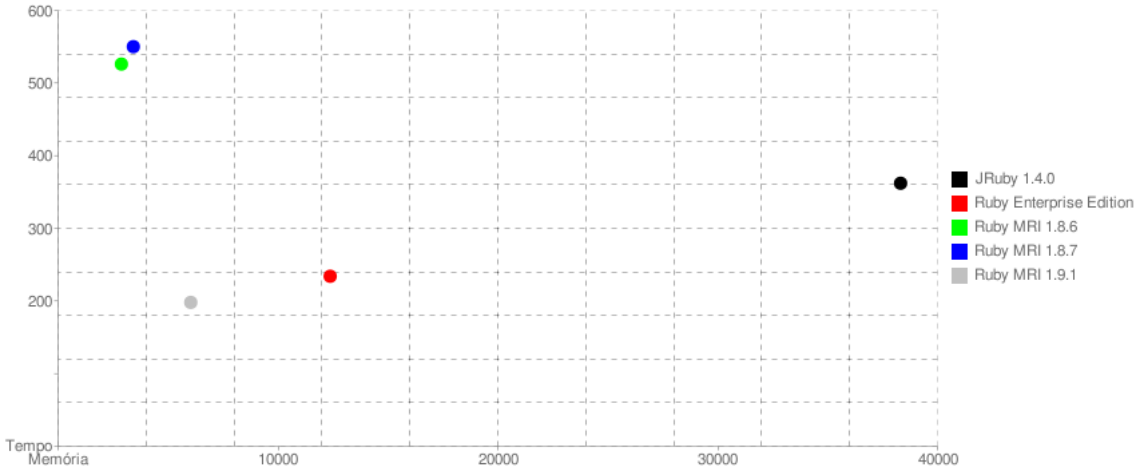

Figura 11.41: Algoritmo de leitura de arquivos - Média do Tempo do sistema (milisegundos) por Média da Memória RAM utilizada (kilobytes)

No gráfico da figura 11.43 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby Enterprise Edition.

No gráfico da figura 11.44 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.6.

No gráfico da figura 11.45 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.8.7.

No gráfico da figura 11.46 pode-se visualizar a utilização de memória RAM no decorrer da execução do algoritmo no interpretador Ruby MRI 1.9.1.

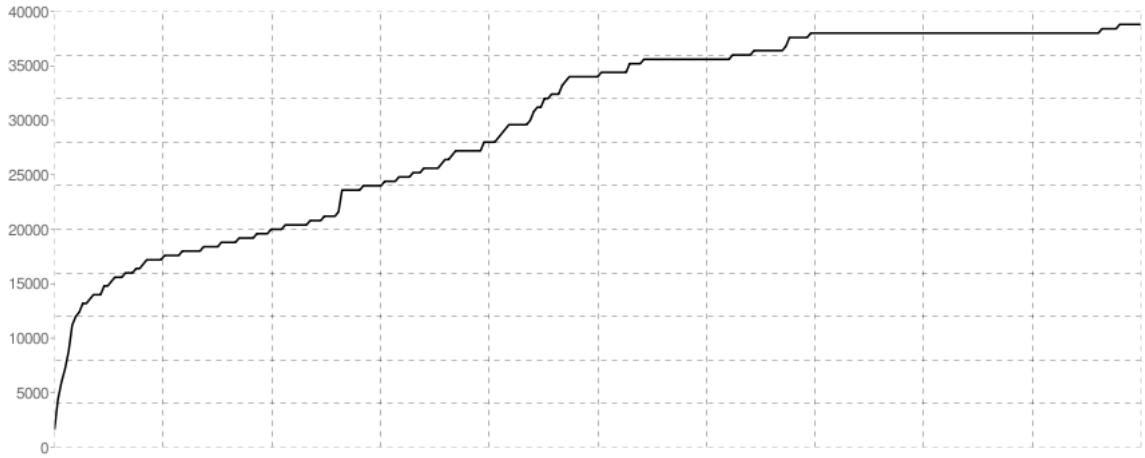

Figura 11.42: Algoritmo de leitura de arquivos - Linha do tempo do uso de memória RAM pelo JRuby em kilobytes

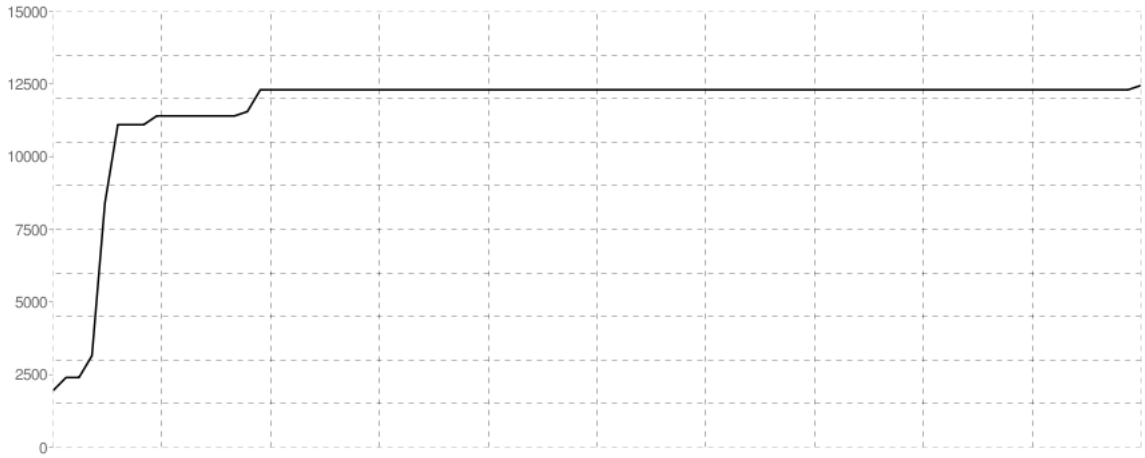

Figura 11.43: Algoritmo de leitura de arquivos - Linha do tempo do uso de memória RAM pelo Ruby Enterprise Edition em kilobytes

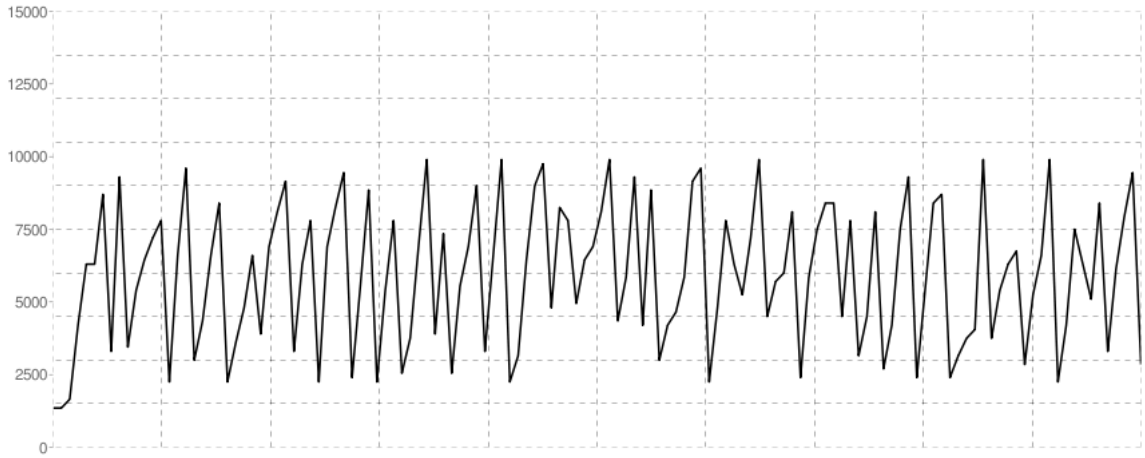

Figura 11.44: Algoritmo de leitura de arquivos - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.6 em kilobytes

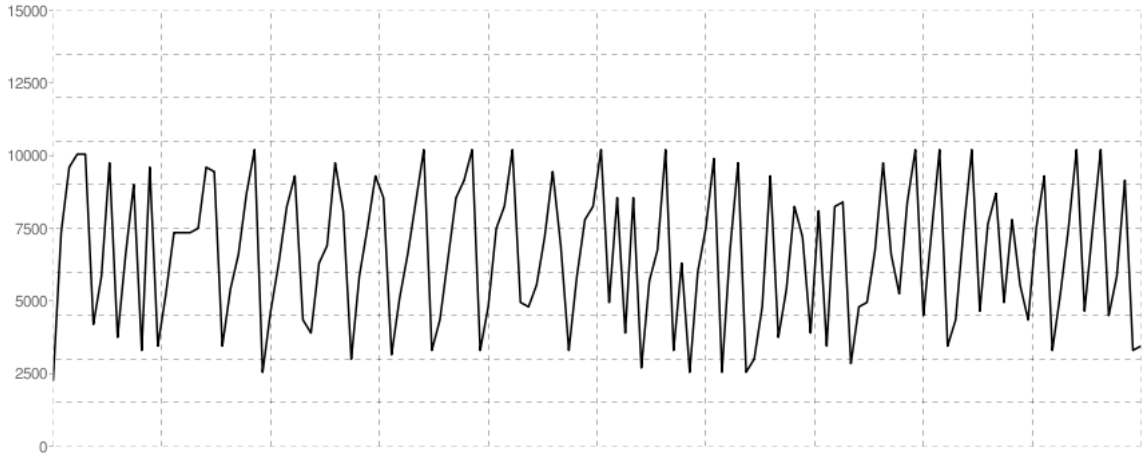

Figura 11.45: Algoritmo de leitura de arquivos - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.8.7 em kilobytes

Figura 11.46: Algoritmo de leitura de arquivos - Linha do tempo do uso de memória RAM pelo Ruby MRI 1.9.1 em kilobytes

12 Conclusão

Neste trabalho foi objetivado uma análise de alguns interpretadores da linguagem de programação Ruby. Foram escolhidos 5 algoritmos e 5 interpretadores para serem avaliados. Dentro de todo o processo saber o que testar e como testar foi relativamente simples. O passo mais complexo foi saber quais seriam os resultados a serem procurados. Ou seja, quais seriam os dados relevantes que deveriam ser coletados, e o fato de saber se esses mesmo dados tinham alguma importância. Outro passo que delegou mais pesquisa foi amadurecer a idéia de como medir o interpretador JRuby, que executa seu processo em uma máquina virtual, sendo assim diferente dos demais. Diante da pesquisa sobre metodologia de testes foi constatado que não existe uma fórmula única, ou uma lista de atributos a serem mensurados. Portanto, o caminho seguido foi o da busca da relação de tempo pelo uso de memória que cada interpretador utilizou.

Todos os algoritmos testados chegaram em resultados que foram analisados e expostos por meio de tabelas e gráficos. Portanto os objetivos de testar e demonstrar qualidades e limitações dos interpretadores foi alcançado com sucesso. O JRuby por ser executado na máquina virtual Java (JVM) teve o maior consumo de memória em praticamente 100% dos testes. O Ruby 1.9.1 na operações matemáticas obteve quase sempre uma ligeira vantagem sobre os demais. Os interpretadores restantes concorreram fortemente tanto no uso de memória, como nos tempos medidos, alcançando resultados bem parelhos.

A descoberta mais interessante do trabalho foi durante o estudo da metodologia que iria ser aplicada aos testes, onde viu-se que não existe um padrão fixo de medidas, e sim uma busca pelas medidas que convém para cada trabalho. Os resultados de cada teste estão bem explicitados e podem ser avaliados novamente num mesmo ambiente, ou mesmo em mais de um ambiente. Sobre o JRuby, um de seus líderes de projeto, Charles Nutter Nutter [2010], menciona uma biblioteca de apoio a monitoração do uso de memória pelo JRuby. Com isto pode-se modificar ou acrescentar algo mais às métricas de avaliação em trabalhos futuros. Uma outra dica do que pode ser feito para trabalhos futuros é o aumento no grau de complexidade dos algoritmos, em conjunto com diferentes máquinas e sistemas operacionais.

Referências Bibliográficas

Mother tongues of computer languages. http://www.digibarn.com/collections/posters/tongues/.

Ravi Sheti Alfred V. Aho and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison-Wesley, 1986.

Richard Bornat. *Understanding and Writing Compilers*. Macmillan Press Ltd., 1979.

Chad Fowler Dave Thomas and Andy Hunt. *Programming Ruby - The Pragmatic Programmers' Guide - Second Edition*. The Pragmatic Programmers LLC, 2004.

Obie Fernandez. *The Rails Way*. Addison-Wesley, 2008.

Michael Fitzgerald. *Ruby - Pocket Reference*. O'Reilly, 2007.

David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

Hal Fulton. *The Ruby Way: Solutions and Techniques in Ruby Programming, Second Edition*. Addison-Wesley, 2006.

Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. The MIT Press, 1985.

Paul Hudak. *Conception, Evolution, and Application of Functional Programming Languages*. Yale University, 1989.

linuxdevcenter, 2001. http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html.

Charles Oliver Nutter. Monitoring the jvm heap with jruby, 2010. http://www.engineyard.com/blog/2010/monitoring-the-jvm-heap-with-jruby/.

John K. Ousterhout. Scripting: Higher level programming for the 21st century, 1998. http://www.tcl.tk/doc/scripting.html.

Michael L. Scott. *Programming Language Pragmatics.* Morgan Kaufmann Publishers, 2006. ISBN 978-0-12-633951-2.

The Java Tutorials. Trail: Learning the java language, 2009. `http://java.sun.com/docs/books/tutorial/java/`.

David A. Watt and Deryck F. Brown. *Programming Language Processors in Java.* Prentice Hall, 2000. ISBN 0-130-25786-9.

Niklaus Wirth. *Compiler Construction.* Addison-Wesley, November 1981. ISBN 0-201-40353-6.

www.ruby lang.org, 2009. `http://www.ruby-lang.org/`.