Universidade Federal de Santa Catarina

INE - Departamento de Informática e Estatística

Sistemas de Informação

EDUARDO HOELLER

Estudo comparativo entre as plataformas .NET e Java Standard Edition com enfoque em desenvolvimento desktop

Florianópolis, outubro de 2010.

Universidade Federal de Santa Catarina

INE - Departamento de Informática e Estatística

Sistemas de Informação

EDUARDO HOELLER

Estudo comparativo entre as plataformas .NET e Java Standard Edition com enfoque em desenvolvimento desktop

Trabalho de conclusão de curso

apresentado como parte dos requisitos

para obtenção do grau de Bacharel em

Sistemas de Informação

Florianópolis, outubro de 2010.

EDUARDO HOELLER

Estudo comparativo entre as plataformas .NET e Java Standard Edition com enfoque em desenvolvimento desktop

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador:

_____

Prof. Vitório Bruno Mazzola
Universidade Federal de Santa Catarina

Banca Avaliadora:

_____

Prof. Frank Augusto Siqueira
Universidade Federal de Santa Catarina

_____

Prof. Ricardo Pereira e Silva
Universidade Federal de Santa Catarina

_____

Max Ricardo Benin
Bacharel em Sistemas de Informação, Faculdades Barddal

Florianópolis, outubro de 2010.

Sumário

Listas

Lista de Figuras

Lista de Tabelas

Lista de Trechos de Código

Lista de Gráficos

Lista de Reduções

API	Application Programming Interface
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CORBA	Common Object Request Broker Architecture
COM	Component Object Model
CTS	Common Type System
CUI	Character User Interface
DLL	Dynamic Link Library
GDI	Graphics Device Interface
GUI	Graphics User Interface
IDE	Integrated Development Environment
JDK	Java Development Kit
JEE	Java Enterprise Edition
JME	Java Micro Edition
JRE	Java Runtime Environment
JSE	Java Standard Edition
JVM	Java Virtual Machine
MSIL	Microsoft Intermediate Language

NOP	No-Operation
PE	Portable Executable
XAML	Extensive Application Markup Language
XHTML	Extensible HyperText Markup Language
XML	Extensive Markup Language

Resumo

Este trabalho se trata de uma comparação entre duas tecnologias muito utilizadas no panorama de desenvolvimento de software atual: as plataformas Java, da Sun Microsystems e o .NET, da Microsoft. O escopo do trabalho abrange as explicitar as principais características de cada arquitetura com um enfoque no desenvolvimento desktop; levantar diferenças entre as plataformas nos quesitos de portabilidade de código e interoperabilidade. São utilizadas as linguagens Java (plataforma Java) e C# (plataforma .NET) para dar apoio às explicações do estudo feito; não há, entretanto, pretensão de se entrar em detalhes da sintaxe, organização ou recursos de cada linguagem. Finalmente, tem como objetivo a aplicação de testes de desempenho de processamento através do uso de um conjunto de algoritmos apropriados para tais testes.

Palavras-chave: JSE, .NET, Java, C#, CLR, JVM.

1. INTRODUÇÃO

1.1. Apresentação

Desde seu surgimento em 1995, a plataforma Java, da Sun Microsystems (parte da Oracle Corporation desde janeiro de 2010), tem se mostrado muito popular no desenvolvimento de diversos tipos de sistemas. Durante todo o curso de graduação em Sistemas de Informação, foi a tecnologia utilizada pelos professores para a explanação dos mais variados conceitos e técnicas pertinentes ao desenvolvimento de software. Entretanto, nos últimos anos, a plataforma .NET da Microsoft, lançada em 2002, vem ganhando espaço na comunidade de desenvolvimento de software e muitos desenvolvedores estão migrando ou já migraram para ela.

Dentre um emaranhado de acrônimos, semelhanças superficiais e diferenças profundas, há muito do que se falar sobre ambas plataformas. Neste trabalho são dadas explicações a respeito de tais acrônimos; obviamente não se limitando ao seu significado, visa explicar o funcionamento de cada um dos componentes principais de cada uma delas além de comparar estas semelhanças e diferenças, porém, não antes de tentar desmistificar a confusão que tanto a Sun quanto a Microsoft fizeram na própria nomenclatura de suas tecnologias.

Por fim, é feita uma comparação levando em consideração as características mais destacadas e marcantes de cada plataforma: portabilidade e interoperabilidade, em conjunto a um teste de desempenho de processamento.

1.2. Motivação

Existem muitas razões para a comparação entre as plataformas Java e .NET. Uma rápida busca na internet por um termo como “Java C# diferenças” resulta em milhares de discussões baseadas em argumentos mal fundados envolvendo os defensores de cada uma das tecnologias. Obviamente existe material técnico sobre o assunto, mas é raro encontrar algo de qualidade, mais raro ainda encontrar material que distinga claramente as plataformas das principais linguagens de programação por elas suportadas em conjunto com uma análise de desempenho que não seja tendenciosa.

Além de tentar dar um maior apoio à comunidade a responder algumas dúvidas sobre esta questão, este trabalho também tem como motivação aprimorar o conhecimento nas plataformas e linguagens de programação escolhidas, além de ajudar na melhor compreensão e distinção dos conceitos de plataforma e linguagem.

Finalmente, o teste de desempenho em conjunto com a análise das tecnologias poderá ser esclarecedor no sentido de direcionar a uma resposta para a questão “.NET ou Java?” – ao menos dentro do escopo apresentado – sendo que esta questão vem perturbando desenvolvedores independentes e profissionais há algum tempo.

1.3. Escopo

O escopo deste trabalho delimita-se pelo estudo e comparação das arquiteturas das plataformas .NET 3.5 e Java 6 Standard Edition voltadas ao desenvolvimento desktop, pela aplicação das linguagens C# 3.0 e Java 1.6 na criação dos algoritmos utilizados na análise de desempenho. Foge ao escopo

deste trabalho, o detalhamento de aplicações ditas “Enterprise” e aplicações Web, sendo previsto, entretanto, menções ao longo do texto.

1.4. Objetivos

1.4.1. Objetivos Gerais

Este trabalho possui, como objetivo geral, a realização de um estudo sobre as plataformas .NET 3.5 e Java 6 Standard Edition, almejando melhor compreensão da arquitetura de ambas. Também visa fazer uma comparação da características que suas empresas desenvolvedoras (Microsoft e Sun/Oracle) põem em maior evidência: portabilidade (Java), interoperabilidade através do suporte multi-linguagem (.NET). Finalmente, apresenta a aplicação e resultados de testes de desempenho através da utilização de algoritmos de ordenação (Bubble Sort e Insert Sort), geração de números primos, e um algoritmo para benchmark conhecido como “Thread-Ring”, utilizando as linguagens C# 3.0 e Java 6 como suporte prático.

Secundariamente, este trabalho tem como objetivo fazer uma melhor distinção entre os conceitos de linguagem e plataforma, especificamente, plataformas Java e .NET e linguagens Java e C#.

1.4.2. Objetivos específicos

- Revisar os conceitos base para permitir a realização deste estudo;
- Estudar de maneira aprofundada as plataformas Java SE e .NET com enfoque no desenvolvimento desktop (isto é, descartando o suporte de ambas as plataformas para o desenvolvimento Web);

- Especificar o funcionamento das máquinas virtuais de cada plataforma (JVM e CLR);
- Comparar teoricamente as plataformas nos quesitos de portabilidade de código e interoperabilidade;
- Testar o desempenho em termos de tempo e custo de processamento de algoritmos clássicos de ordenação (Bubble Sort e Insert Sort), da geração de números primos e do algoritmo específico para benchmark Thread-Ring implementados em ambas as linguagens;
- Efetuar uma análise comparando o resultado dos testes de desempenho de cada algoritmo nas duas linguagens.

1.5. Organização do Trabalho

Os próximos capítulos deste trabalho estão organizados da seguinte maneira:

- Capítulo 2: Consiste em uma revisão dos conceitos chave para a compreensão do estudo realizado: plataforma, compiladores, linguagens intermediárias e de baixo nível, linguagem de máquina, máquinas virtuais, bem como uma revisão dos conceitos utilizados para a comparação das plataformas (portabilidade e interoperabilidade) e dos algoritmos utilizados no teste e comparação de desempenho das linguagens-alvo do estudo (Bubble Sort, Insert Sort, geração de números primos e Thread-Ring).
- Capítulo 3: Define o conceito de plataforma .NET, fornece uma visão geral da linguagem C# e analisa de forma aprofundada o núcleo da plataforma, a Common Language Runtime.

- Capítulo 4: Define os conceitos inerentes à plataforma Java: JDK, JRE, JVM, linguagem Java e API Java, analisando com mais especificidade o funcionamento da implementação HotSpot da JVM.
- Capítulo 5: Revisa comparativamente os principais conceitos de cada plataforma; realiza uma comparação teórica dos conceitos de portabilidade e interoperabilidade e realiza uma comparação de desempenho prática.
- Capítulo 6: A partir dos resultados obtidos no capítulo 5, formaliza a conclusão do estudo feito.

2. REVISÃO BIBLIOGRÁFICA

2.1. O Conceito de Plataforma

Plataforma é o hardware ou software no qual programas são executados. Dentre as plataformas mais populares, podem ser citadas o Microsoft Windows, o Linux, Solaris OS e o Mac OS. A maioria das plataformas podem ser descritas como a combinação de um sistema operacional e o hardware abaixo dele (Adaptado de Oracle: About the Java Technology). As duas plataformas aqui estudadas diferem da maioria das outras por se tratarem de apenas software executando sobre outras plataformas baseadas no híbrido hardware/software descrito acima.

Alternativamente, existe o conceito de “plataforma/ambiente de desenvolvimento”, que expande uma plataforma de execução. Neste trabalho, como o enfoque está em duas plataformas semelhantes baseadas puramente em software, é considerada plataforma de desenvolvimento o conjunto de ambiente de execução, suas bibliotecas e ferramentas voltadas ao desenvolvimento. Estabelece-se que o termo “ambiente de execução” será utilizado para tratar de plataformas de execução e que “plataforma” irá se referir a ambos os casos, a ser interpretado por contexto ou explicitado quando necessário (nota-se que, em algumas vezes, não faz diferença).

2.2. Máquinas Virtuais

Por definição, uma máquina virtual (por vezes chamadas de máquinas abstratas) é um software com o objetivo de simular a arquitetura de um

hardware físico. Dentro do escopo deste trabalho está o estudo de duas máquinas virtuais: a JVM (Java Virtual Machine) em sua implementação da Sun (conhecida como HotSpot), parte integrante da plataforma Java e a CLR (Common Language Runtime), parte da plataforma .NET da Microsoft. Estas se tratam dos **ambientes de execução** alvo deste estudo.

Alguns autores consideram que existe uma sutil diferença entre plataforma de execução e ambiente de execução. Estes vêem uma plataforma de execução como um conjunto da máquina virtual e as classes mínimas necessárias para dar suporte à execução de um programa ao passo que o ambiente de execução trata-se explicitamente da máquina virtual. Este trabalho não fará distinção entre os dois termos, utilizando de maneira ampla “ambiente de execução” (podendo significar qualquer um dos dois, também a ser analisado por contexto) e enfatizando a “máquina virtual” quando necessário.

2.3. Compiladores

O papel de um compilador é traduzir programas fonte em alguma linguagem de mais alto nível para um programa objeto em uma linguagem de baixo nível (assembly). O termo “tradutor” é adequado para situações quando esta tradução se dá de uma linguagem de alto nível para outra linguagem que não seja considerada de baixo nível. Mesmo assim, autores (especialmente autores de livros sobre .NET) utilizam levianamente o termo “compilar” quando estão se referindo a uma mera tradução. Para fins de praticidade, este trabalho utilizará da mesma terminologia que tais autores.

Um processo de compilação dito padrão é dividido em duas etapas, cada uma com três fases: a etapa de análise (fases: análise léxica, sintática e

semântica) e a etapa de síntese (fases: geração de código intermediário, otimização de código e geração de código final).

2.3.1. Compilação JIT

Um compilador Just-In-Time é aquele que compila o código a medida em que ele executa, ao contrario de uma compilação tradicional, onde todo o programa é compilado antes de ser executado. Assim como no caso da interpretação de código, geralmente há uma linguagem intermediária envolvida, onde o programa fonte original é traduzido para um programa em linguagem intermediária e este sim é compilado em tempo de execução.

Em um modelo comum de compilação JIT, os métodos são compilados a medida em que vão sendo invocados.

2.3.2. Interpretação de Código

Um interpretador pega um programa fonte e o executa instantaneamente. Urgência é a característica chave da interpretação: não há gasto de tempo prévio com a tradução do programa fonte em uma representação de mais baixo nível. (WATT, BROWN, 2000)

Em um ambiente interativo, a urgência é altamente vantajosa. Por exemplo, o usuário de uma linguagem de comandos espera uma resposta imediata para cada comando; de modo semelhante, o usuário de uma linguagem de query de banco de dados (como a linguagem SQL) espera uma resposta imediata para cada “pergunta” (query). Neste modo de funcionamento, o “programa” é executado uma vez e então, descartado. (WATT, BROWN, 2000)

O usuário de uma linguagem de programação, entretanto, tem muito mais propensão a guardar seu programa para desenvolvimento e uso futuro. Mesmo assim, tradução da linguagem de programação para uma linguagem intermediária seguido por interpretação desta linguagem pode ser uma boa alternativa à compilação completa, especialmente durante as primeiras etapas de desenvolvimento do programa. (WATT, BROWN, 2000)

De acordo com Watt e Brown, existem dois tipos de interpretação:

- Interpretação iterativa: Seu uso é adequado quando as instruções da linguagem fonte são todas primitivas. As instruções do programa fonte são buscadas, analisadas e executadas uma após a outra. Interpretação iterativa é ideal para código de maquinas reais e abstratas, para linguagens de programação de baixa complexidade e para linguagens de comando.
- Interpretação recursiva: Necessária se a linguagem fonte possui instruções compostas (declarações ou expressões). A interpretação de uma instrução pode desencadear a interpretação de instruções que a compõem. Um interpretador para uma linguagem de programação de alto nível ou linguagem de query deve ser recursivo. Contudo, interpretação recursiva é mais lenta e complexa que a interpretação iterativa; por este motivo é preferível compilar linguagens de alto-nível, ou ao menos traduzi-las para linguagens intermediarias de nível mais baixo que sejam propícias para interpretação iterativa.

Considera-se importante rever o conceito de interpretação para o escopo deste trabalho pois é uma das técnicas utilizadas pelo compilador da HotSpot – implementação da JVM aqui analisada.

2.5. Assembly, Opcodes e Linguagem de Máquina

O assembly ou linguagem de montagem é a linguagem de mais baixo nível considerada humanamente legível. Esta utiliza de mnemônicos para representar as instruções de máquina e é completamente dependente da arquitetura para a qual foi projetada. É considerada a linguagem ideal para desempenho, pois é legível e capaz de dar ao programador controle preciso sobre as instruções de máquina (em contrapartida, o programador deve conhecer intimamente a arquitetura do hardware com o qual está trabalhando).

Um programa em assembly também precisa ser compilado; esta, contudo, pode ser considerada uma compilação especial. O compilador (neste caso, chamado de montador ou assembler) traduz os mnemônicos para seus respectivos opcodes (*operation codes* ou “códigos de operação”), que como o próprio nome diz, são os códigos das operações a serem executadas pelo processador em questão, podendo ou não estarem associados à operandos, endereços de memória, entre outros. O conjunto de opcodes de uma arquitetura específica (geralmente expressados em valores hexadecimais nas listagens mas de fato executados pelo processador como valores binários) é exatamente o conjunto de instruções da linguagem de máquina da arquitetura em questão.

2.6. Linguagens Intermediárias

Entre outras utilidades, as linguagens intermediárias surgiram principalmente para suprir a necessidade de cumprir de uma forma mais eficaz

a etapa de otimização de código no processo de compilação. Elas não precisam, necessariamente, serem linguagens inacessíveis ao programador. A linguagem C por exemplo, é utilizada como linguagem intermediaria para outras linguagens de programação, como Eiffel, Sather, Esterel, etc.

O foco aqui está nas linguagens intermediárias MSIL (Microsoft Intermediate Language) e no Java bytecode. Estas linguagens possuem mnemônicos e opcodes, semelhantes às linguagens de montagem, entretanto, sua forma de execução é muito diferente por estar associada às suas respectivas máquinas virtuais e será explicada ao longo dos capítulos deste trabalho.

2.7. Critérios de Comparação

2.7.1. Portabilidade

No escopo deste trabalho, quando se fala em portabilidade, refere-se a portabilidade de código, ou seja, a habilidade de executar um mesmo programa em arquiteturas (físicas) diferentes sem a alteração do referido código. Portabilidade sempre foi o conceito-chave da plataforma e da linguagem Java, tornando-o um critério de comparação importante.

2.7.2. Interoperabilidade

Em se tratando de plataformas de execução gerenciadas (mais especificamente, Java e .NET), interoperabilidade e suporte a multi-linguagem são dois conceitos que quase se confundem. Ao passo que a interoperabilidade é a habilidade de dois componentes de software construídos

em linguagens diferentes se comunicarem, independente de ambiente de execução, o suporte a multi-linguagem diz respeito à capacidade de um ambiente de execução gerenciado executar software em diversas linguagens de programação, independente de sua interoperabilidade.

Assim como acontece com a portabilidade em Java, o suporte a multi-linguagem é uma das características principais da plataforma .NET, entretanto, é uma característica difícil de se mensurar. Portanto, decidiu-se comparar a capacidade de interoperabilidade das duas plataformas, sendo este um assunto mais interessante e intimamente ligado à característica principal, o suporte à multi-linguagem.

2.8. Algoritmos Utilizados

Segue a descrição dos algoritmos utilizados para o benchmarking. Suas respectivas implementações podem ser encontradas no Anexo I.

2.8.1. Bubble Sort

O Bubble Sort é um algoritmo de ordenação que incrementalmente avança em um array, comparando cada elemento adjacente e invertendo suas posições, se necessário. É um dos algoritmos de ordenação mais lentos, tornando-o uma ótima escolha para o uso em um benchmark de CPU.

2.8.2. Insert Sort

O Insertion Sort é um algoritmo simples aonde o array ou lista ordenada é construída uma posição por vez. É bem menos eficiente em listas grandes do que algoritmos mais avançados.

2.8.3. Geração de números primos

Um número primo é definido como qualquer número que possui divisão inteira apenas por um e por si mesmo. Há um número infinito de números primos, portanto, deve existir um limite superior ao se escrever um programa para calcular números primos. Este trabalho utiliza separadamente tanto a matemática de números inteiros quanto a matemática de ponto flutuante para a realização deste benchmark.

2.8.4. Thread-Ring

O Algoritmo Thread-Ring tem sua origem nas publicações *Performance Measurements of Threads in Java and Processes in Erlang* (1998) e *A Benchmark Test for BCPL Style Coroutines* (2004). Este especifica um programa que deve criar e manter vivas 503 threads preemptivas (threads que permitam ao sistema determinar quando uma troca de contexto pode acontecer), implicitamente ou explicitamente conectadas em um anel e que devam passar um token de uma thread à próxima no mínimo N vezes.

3. PLATAFORMA .NET

3.1. Visão Geral

Os desenvolvedores na Microsoft descrevem a plataforma .NET como sua chance de começar do zero através de um modelo de desenvolvimento novo e mais moderno (MacDonald, Szpuszta, 2009). O termo .NET consiste na combinação de um ambiente de execução e bibliotecas fornecidas pela Microsoft, incluindo também compiladores para programas em C# e VB.NET (Skeet, 2008). Introduzida no mercado em 2002, trata-se de uma plataforma para o desenvolvimento de software gerenciado, sendo “gerenciado” um termo chave – o conceito que difere o .NET da maioria dos outros ambientes de desenvolvimento (Mayo, 2008).

O outro termo-chave do .NET é “multi-linguagem”. O C#, desenvolvido em conjunto com a plataforma, é sua principal linguagem suportada, entretanto existe uma vasta gama de outras linguagens de programação compatíveis com a plataforma (como o Visual Basic, o C++/CLI e mais recentemente, o F#, entre outras). De qualquer forma, o C# é dito a única das linguagens suportadas capaz de fazer uso da maior parte de seus recursos (sendo a MSIL a única capaz de fazer uso de todos os seus recursos; isto será visto no tópico 4.5.1).

É importante ressaltar que, teoricamente, a linguagem de programação C# não faz parte da plataforma .NET, mas esta é uma linha tênue e causa bastante discórdia dentro da comunidade, já que na prática, o C# não existiria sem o .NET e vice-versa, inclusive por questões históricas. Por este se tratar de, em sua maior parte, um trabalho teórico, aqui é considerado que .NET e C# são tecnologias independentes.

De fato, nem termo “Plataforma .NET” é oficialmente utilizado pela Microsoft; esta o chama de .NET Framework, Microsoft .NET ou apenas .NET (pronunciado “*dot net*”). A própria definição de “Framework” é um tanto nebulosa sendo que autores e os próprios desenvolvedores parecem não conseguir chegar a um consenso a respeito das características que uma peça de software deve e não deve possuir para ser rotulado como tal. Apesar das divergências na definição, afirma-se, entretanto, que o .NET não se enquadra na categoria de Framework – uma declaração que também não é unânime dentro da comunidade .NET, apesar de ser a opinião da grande maioria. Por fim, ignorando definições formais, os termos .NET, plataforma .NET e .NET Framework acabam sendo utilizados como sinônimo pela comunidade e também neste trabalho.

O Framework versão 3.5 foi lançado no mercado em novembro de 2007 e pareceu a escolha certa quando o trabalho foi iniciado. Contudo, a Microsoft lançou em abril de 2010 a versão 4.0 do .NET, fato que não afeta este trabalho, considerando que muito já havia sido feito até então e que o processo de migração no mercado tem se mostrado extremamente lento, sendo que diversas empresas ainda utilizam o Framework 2.0.

Nota: De um ponto de vista comercial, o termo .NET Framework também possui o significado de plataforma de execução (equivalente ao Java Runtime Environment – JRE), sendo que o arquivo distribuído pela Microsoft que trás as ferramentas que dão suporte à execução de programas desenvolvidos em .NET é assim chamado (contendo apenas a máquina virtual e as bibliotecas base). Por não se tratar de terminologia técnica, este fato será ignorado, exceto quando ressaltado explicitamente.

3.2. Componentes da Plataforma .NET 3.5

Este item abrange alguns dos componentes da plataforma .NET 3.5, de forma mais detalhada resumida, dependendo da necessidade de uma compreensão mais aprofundada ou não para suprir as necessidades do escopo do trabalho. A figura abaixo representa os principais componentes da plataforma, separados de acordo com a versão em que foram introduzidos:

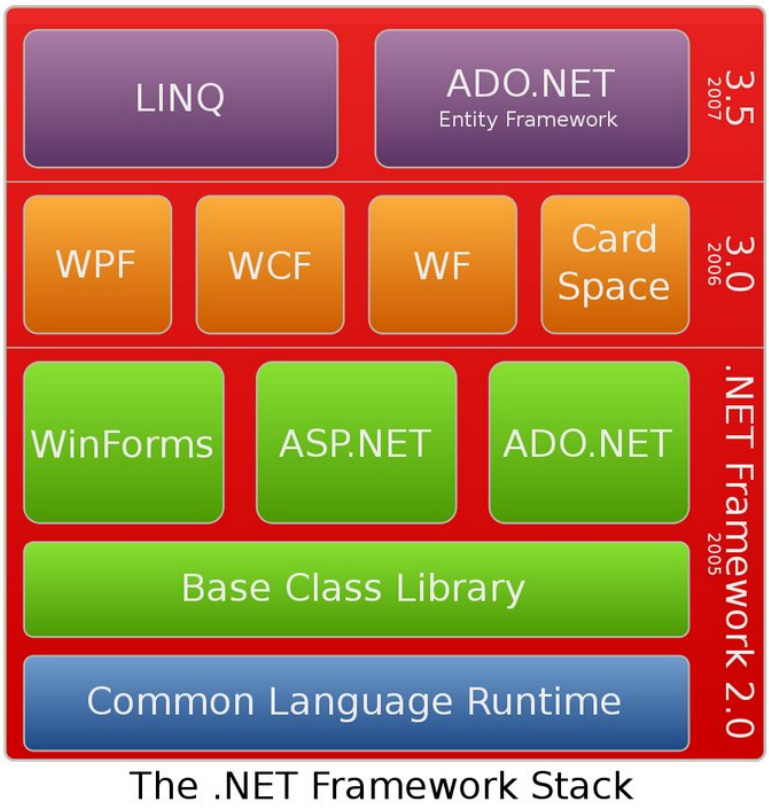

Figura 1: Componentes da plataforma .NET

Como pode ser observado, não há indicativos do que o Framework herdou de suas versões anteriores a 2005, pois este era constituído basicamente dos mesmos elementos que o Framework 2.0 mas sendo completamente reformulado para seu lançamento em 2005 em conjunto com o Visual Studio 2005.

Também nota-se a ausência de qualquer linguagem de programação inclusa como componente na “pilha” da plataforma; pode-se até encontrar diagramas que as incluam, mas há bastante polêmica e discórdia dentro da comunidade no que diz respeito a este assunto, como inferido anteriormente. Maiores detalhes sobre esta questão são dados no tópico 4.3: Linguagem de Programação C#.

3.2.1. Common Language Runtime (CLR): Visão Geral

A Common Language Runtime é o ambiente de execução sob o qual todas as aplicações .NET rodam – sua máquina virtual. Trata-se da implementação da Microsoft do padrão ECMA-335: Common Language Infrastrtucture - CLI (verificar Anexo II.2). Outras implementações deste padrão incluem o Mono da Novell, para plataformas Linux, e outras duas da Microsoft: uma para dispositivos móveis e para o Xbox 360 (incluso no .NET Compact Framework) e a outra trata-se de seu ambiente de execução “unificado” para o framework de aplicações web Silverlight. A Novell também está desenvolvendo o seu análogo ao Silverlight, chamado Moonlight.

A implementação completa da Microsoft da CLS tem como plataforma alvo ambientes executando Windows (a versão 2.0 da CLR, inclusa no Framework 3.5 é compatível com XP, Vista e Seven) rodando em processadores x86/x64.

O papel principal da CLR é localizar, carregar e gerenciar tipos .NET por conta própria. A CLR também cuida de uma quantidade de detalhes de baixo nível como gerenciamento de memória, criação de domínios de aplicação, threads e limites de contexto de objetos além de verificações de segurança.

(TROELSEN, 2007). A tabela 1 consiste em uma listagem de todas as características da CLR:

Característica	Descrição
Suporte à biblioteca de classes base do Framework (BCL)	Contem tipos[1] pré-construídos e bibliotecas para gerenciamento de assemblies, memória, segurança, threading e outros, para suporte ao sistema de execução
Depuração	Meios para facilitar a depuração de código
Gerenciamento de exceções	Permite a escrita de código para criar e tratar exceções
Gerenciamento de execução	Gerencia a execução do código
Interoperabilidade	Compatibilidade reversa com código COM e Win32
Compilação JIT	Característica que garante que a CLR compila o código exatamente antes de executá-lo.
Segurança	Suporte tradicional a segurança baseada em perfil, além do chamado “Code Access Security” (CAS)
Gerenciamento de Threads	Permite a execução de múltiplas threads
Carregamento de tipos	Encontra e carrega os assemblies e tipos
Verificações de segurança de tipos	Garante que as referencias são compatíveis com seus tipos
Garbage Collection	Gerenciamento automático de memória e coleta de lixo (garbage collection)

Tabela 1: Características da CLR (Adaptado de MAYO, 2008)

[1]Um “tipo” é o termo usado em .NET tanto para tipos primitivos como para classes (pré-construídos e criadas por usuários), strucs, interfaces e outras estruturas fundamentais para programação. Alguns autores o chamam de “tipo .NET”.

3.2.2. Base Class Library (BCL)

Também conhecida como Framework Class Library (FCL), a BCL é uma biblioteca abrangente de tipos reusáveis. As bibliotecas dentro da BCL estão organizadas em namespaces, contendo código para dar apoio a todas as tecnologias da plataforma .NET, como WinForms, ASP.NET, ADO.NET, etc

(ver figura 1). Como visto da tabela 1, a CLR suporta a BCL nos quesitos de tipos pré-construídos, tratamento de exceções, segurança e paralelismo (threading). A tabela 2 mostra alguns dos namespaces mais comuns da BCL:

System	Este namespace Inclui todos os tipos valorados (“value types”, a nomenclatura .NET para tipos primitivos, a classe String, funções matemáticas, classe DateTime e muitos outros tipos básicos.
System.Collections	Contem interfaces e classes que permitem a manipulação de coleções de objetos. Inclui estruturas de dados como listas, hashtables e dicionários
System.Configuration	Fornece a infra-estrutura para lidar com configuração de dados
System.Data	Este namespace representa a arquitetura ADO.NET
System.Diagnostics	Fornece a habilidade de diagnosticar o desempenho da aplicação
System.Drawing	Fornece acesso à API GDI+ (nativa do Windows) que inclui suporte para gráficos 2D e vetorados além de serviços de imagem, impressão e texto.
System.IO	Suporte a arquivos, console, portas seriais e interprocessos de entrada e saída
System.Linq	Define a interface IQuerable<T> e métodos relacionados, permitido que provedores LINQ sejam conectados
System.Net	Fornece uma interface para protocolos de rede como HTTP, FTP e SMTP. Oferece comunicação segura através de protocolos como o SSL.
System.Runtime	Permite o gerenciamento do comportamento de uma aplicação ou da CLR. Algumas das funcionalidades incluem interoperabilidade com COM ou outro tipo de código nativo, construção de aplicações distribuídas e serialização de objetos.
System.Security	Este namespace permite a construção de aplicações seguras baseadas em políticas e permissões. Fornece serviços como criptografia.
System.ServiceModel	
System.Text	Fornece a capacidade para manipular sequencias de caracteres de codificação ASCII, Unicode, UTF-7 e UTF-8
System.Threading	Fornece suporte para multi-thread.
System.Web	Fornece funcionalidades relacionadas à web. A maior parte das bibliotecas deste namespace definem a arquitetura ASP.NET
System.Windows	Namespace que fornece bibliotecas de suporte à criação de interfaces gráficas encapsulando a API do Windows e permitindo que rodem dentro de

	ambiente gerenciado. Contem as bibliotecas referentes ao WinForms e WPF.
System.Xml	Fornece suporte para processamento XML, incluindo leitura, escrita, schemas, serialização, busca e transformação.

Tabela 2: Namespaces comuns na BCL

Todas as classes .NET são derivadas da classe System.Object, incluída no assembly mscorlib.dll.

3.2.3. WinForms

Trata-se do módulo da plataforma destinado ao desenvolvimento de janelas e interfaces gráficas desktop. Existem muitos componentes de terceiros que o fazem de forma mais elegante que o WinForms e este mesmo está sendo substituído pela Microsoft a partir do Framework 3.0, com o Windows Presentation Foundation.

3.2.4. ASP.NET

O ASP.NET pode ser considerado a parte da plataforma .NET voltada ao desenvolvimento Web.

Aplicativos web criados através do ASP.NET são ditos aplicativos .NET completos, executados através de código compilado (diferente de paginas no ASP “clássico” ou do PHP, por exemplo, utilizando de scriptação interpretada) e gerenciados pelo ambiente de execução do .NET (a Common Language Runtime). Em essência, o ASP.NET torna tênue a distinção entre desenvolvimento de aplicações e desenvolvimento web através da inserção das ferramentas e tecnologias voltadas para o desenvolvedor desktop no mundo do desenvolvimento web (MacDonald, Szpuszta, 2009).

MacDonald e Szpuszta ainda ressaltam 7 fatos sobre o ASP.NET:

1) É integrado com o Framework .NET;
2) É compilado ao invés de interpretado;
3) É multi-linguagem;
4) Roda dentro da CLR;
5) É orientado a objetos;
6) É “multi-dispositivo” e “multi-navegador”;
7) É fácil de implantar e configurar (através do servidor IIS)

Resumindo, pode-se dizer que o ASP.NET herda todas as características da plataforma .NET e adiciona algumas próprias, tornando-o propício para o desenvolvimento Web; é errado dizer, entretanto, que o ASP.NET é uma extensão do .NET ou uma plataforma a parte (apesar do termo “plataforma ASP.NET” ser amplamente utilizado); trata-se, como dito no início, da tecnologia que constitui a parte da plataforma .NET voltada ao desenvolvimento web.

3.2.5. ADO.NET

O ADO.NET trata-se da tecnologia de acesso à dados pertencente ao Framework .NET, consistindo de classes gerenciadas que permitem que aplicações .NET se conectem a fontes de dados (geralmente bancos de dados relacionais), além de executar comandos e gerenciar dados desconectados. O ADO.NET possibilita que código muito semelhante seja escrito para aplicações desktop cliente-servidor, aplicações web e até mesmo aplicações que utilizem apenas uma base de dados local. Os conjuntos de classes que permite todas

essas operações são conhecidos como *data providers* ou provedores de dados (MacDonald, Szpuszta, 2009).

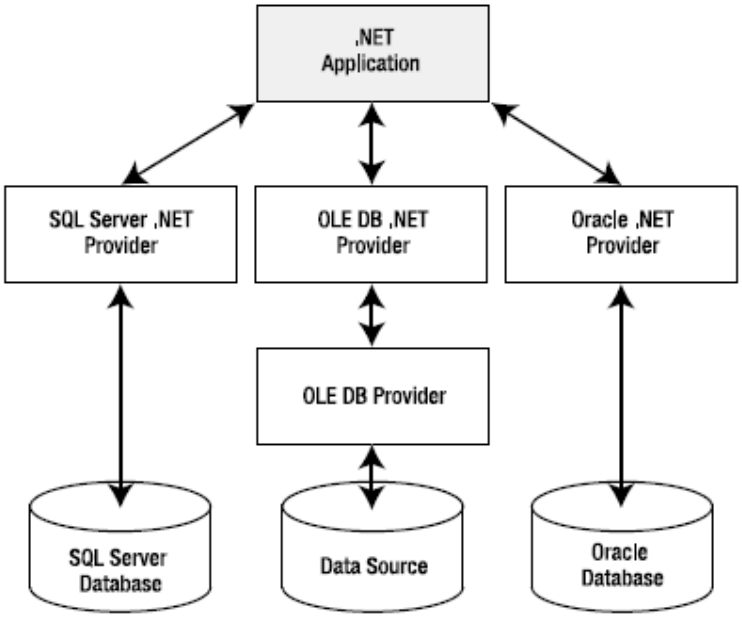

Figura 2: Arquitetura ADO.NET

3.2.6. Windows Presentation Foundation

Considerado o sucessor do WinForms, o WPF foi introduzido no Framework 3.0 em 2006 e apesar de ainda perder para componentes de terceiros no que diz respeito à popularidade em produção gráfica desktop dentro da plataforma .NET, o WPF é a ferramenta mais poderosa disponível para a tarefa, sendo que esta utiliza o DirectX para renderização, oposto aos componentes baseados no WinForms, que utiliza o GDI.

Grande parte da aversão dos desenvolvedores em adotar o WPF se dá pelo fato que o desenvolvimento é feito em XAML, uma linguagem declarativa derivada do XML e de fato bastante parecido com o uso dos User Controls para a implementação de interfaces gráficas em ASP.NET. Apesar da resistência da comunidade em adotar o WPF, os desenvolvedores estão cientes de suas

vantagens e percebe-se um crescente número de desenvolvedores buscando se aprimorar nesta “nova” tecnologia.

3.2.7. Microsoft Visual Studio 2008

Apesar de não fazer parte do .NET, sua popularização não seria possível sem esta ferramenta. Desde sua origem como Visual Studio .NET, em 2002, esta IDE vem sendo construída e atualizada conjuntamente com o .NET e a linguagem C#, quase pode ser considerado parte da plataforma. Apesar da existência de outras IDEs, o Visual Studio 2008, lançado em novembro de 2007, foi considerado o ambiente mais completo para o desenvolvimento e teste de aplicações .NET até o lançamento do Visual Studio 2010, em abril de 2010. De fato, grande parte da produtividade cunhada à plataforma .NET vem de recursos do Visual Studio, como o *IntelliSense* (ferramenta integrada ao VS extrapola o conceito de auto-completar código) ou seu depurador, de uso praticamente intuitivo.

A partir da versão 2005, o Visual Studio passou a ser distribuído gratuitamente através de versões chamadas “Express”, consistindo em IDE’s diferentes para linguagens/propósitos diferentes, mas com funcionalidades reduzidas e sem fornecer o nível de interoperabilidade que as versões Standard, Professional e Team System (adquiridas através de licença) são capazes de fornecer.

3.3. Linguagem de programação C# 3.0

C# é uma linguagem relativamente nova construída sobre conceitos encontrados em suas linguagens predecessoras, podendo ser dito que é uma

“descendente direta” de C e C++ e está intimamente relacionada ao Java, tornando-a imediatamente familiar para programadores experientes. (MICHAELIS, 2009; SCHILDT, 2009).

Mesmo se tratando de uma linguagem de programação que pode ser estudada isoladamente, o C# tem um relacionamento especial com a plataforma .NET, existindo duas razões para tal: 1) O C# foi criado, lançado e é atualizado pela Microsoft em conjunto com Framework .NET, com o objetivo de para gerar código para este e; 2) As bibliotecas utilizadas pelo C# são definidas pelo Framework. Então, mesmo sendo possível separar a linguagem C# do ambiente .NET, os dois tem uma ligação profunda (SCHILDT, 2009).

Apesar de herdar as características do C/C++ e “pegar emprestado” muitas idéias utilizadas no Java, o C# foi pioneiro na introdução de diversas características ao longo de suas versões, como laços foreach, generics, tipos delegate (uma forma simples de implementar o que em C++ é chamado de ponteiro para função), tipos parciais (a possibilidade de dividir uma mesma classe em vários arquivos para uma melhor organização de código) e com a versão 3.0, o LINQ (Language-integrated query) e expressões Lambda. Como explicado anteriormente, este trabalho não abrange as características introduzidas na versão 4.0 da linguagem.

De fato, o C# ser uma linguagem capaz de **gerar código** para a plataforma .NET é um dos argumentos utilizados pelos defensores da corrente que dita que o C# não é parte do .NET. Quando compilado, um código em C# gera código na linguagem intermediária da plataforma .NET, a MSIL (Microsoft Intermediate Language) – esta sim, parte da plataforma. Uma vez gerado, o código em MSIL é armazenado em um assembly .NET com a extensão .dll

(esta é diferente das dll's geradas pela compilação de programas em C ou C++). O assembly .NET é compilado em tempo de execução pelo compilador JIT (*just-in-time*) incluso na máquina virtual da plataforma, a Common Language Runtime (CLR), gerando código nativo que é executado normalmente pelo sistema operacional.

Este breve exemplo de como funciona todo o processo de compilação de um programa em C# não é válido apenas para ele: em linhas gerais, todas as linguagens que a plataforma .NET suporta passam por este mesmo processo. É este o significado de gerar código para a plataforma .NET – gerar o código em MSIL dentro de um assembly .NET – e como mencionado anteriormente, diversas linguagens tem essa capacidade, dado que seja usado o compilador/tradutor adequado.

3.4. O Assembly .NET

Em linhas gerais o Assembly .NET é um arquivo de extensão .dll que armazena o código gerado pelo compilador/tradutor de uma linguagem específica para MSIL. Entretanto, há mais no assembly do que apenas instruções. Um assembly .NET é composto de um manifesto (tabela de metadados), módulos gerenciados e arquivos de recursos (imagens, arquivos .html, etc). De acordo com Ritcher (2010), um módulo gerenciado é formado pelas seguintes partes:

1) **Cabeçalho PE32 ou PE32+:** O cabeçalho padrão de arquivos Windows PE. Caso esteja no formato PE32, o arquivo pode rodar em uma versão Windows 32 ou 64 bits, PE32+ é exclusivo para 64bits. O cabeçalho

também informa o tipo de arquivo: GUI, CUI ou DLL e possui um timestamp indicando quando o arquivo foi gerado.

2) **Cabeçalho CLR:** Contem a informação (interpretada pela CLR) que torna o assembly de fato um módulo gerenciado. Inclui a versão da CLR requerida3, o ponto de entrada do modulo (método Main) e a localização e tamanho dos metadados do módulo.
3) **Metadados:** Constituídos de dois conjuntos tabelas principais: um contendo tabelas que descrevem tipos e membros definidos no código do usuário e outro contendo tabelas definindo tipos e membros referenciados pelo código do usuário.
4) **Código intermediário:** O código em MSIL, produzido pela compilação do código fonte (seja este em C#, VB, etc). Em tempo de execução, a CLR compila o código intermediário em instruções nativas da CPU.

Um módulo gerenciado é criado ao compilar um único arquivo em uma linguagem fonte como C# ou VB dentro de um projeto .NET. As tabelas do manifesto (gerado junto com o assembly ao se compilar todo o projeto) descrevem os arquivos que compõem o assembly, os tipos públicos implementados pelos módulos do assembly e arquivos de recursos ou dados que estão associados ao assembly. Em muitos casos, um componente construído em .NET com diversas classes e recursos pode ser reduzido em um único arquivo .dll para sua utilização por outros componentes e projetos. A figura abaixo ilustra o processo de montagem do Assembly .NET:

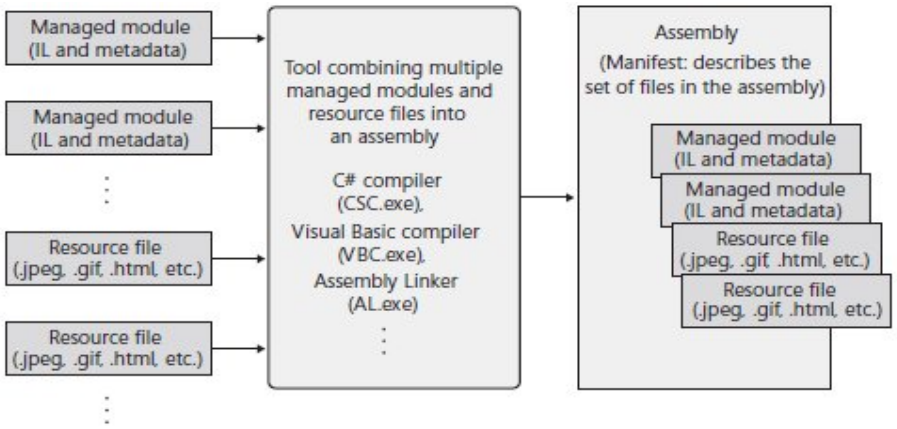

Figura 3: Montagem de um Assembly .NET

3.5. CLR: Visão Detalhada

Por se tratar do núcleo da plataforma .NET, há a necessidade de explicar a CLR mais detalhadamente. Os 3 componentes descritos abaixo (MSIL, CTS e CLS) fazem parte do padrão ECMA-335 e são as peças fundamentais para o funcionamento do ambiente de execução e da plataforma como um todo.

3.5.1. Microsoft Intermediate Language (MSIL)

A MSIL (também chamada de CIL – Common Intermediate Language, ou simplesmente IL) é a linguagem intermediária para a qual todo programa a ser executado na plataforma .NET é traduzida e armazenada em um assembly .NET. A MSIL é uma linguagem baseada em pilha, muito semelhante a linguagens de montagem, possuindo mnemônicos e opcodes para representar suas instruções. Entretanto, a IL também pode acessar e manipular tipos e tem instruções para criar e inicializar objetos, executar chamadas de métodos e manipular elementos de arrays diretamente. Ela até mesmo tem instruções

para tratamento de exceções, desta forma, podendo ser vista como uma linguagem de máquina orientada a objetos. Segue um simples código de um programa para impressão de string em C# e sua representação na IL para melhor esclarecimento:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Eduardo Hoeller, TCC 2010.2!");
    }
}
```

Trecho de Código 1: Impressão de string em C#

```
.class private auto ansi beforefieldinit Program
    extends [mscorlib]System.Object
{
    .method public hidebysig specialname rtspecialname instance void
.ctor() cil managed
    {
        .maxstack 8
        L_0000: ldarg.0
        L_0001: call instance void [mscorlib]System.Object::.ctor()
        L_0006: ret
    }

    .method private hidebysig static void Main(string[] args) cil
managed
    {
        .entrypoint
        .maxstack 8
        L_0000: nop
        L_0001: ldstr "Eduardo Hoeller, TCC 2010.2!"
        L_0006: call void [mscorlib]System.Console::WriteLine(string)
        L_000b: nop
        L_000c: ret
    }

}
```

Trecho de Código 2: Impressão de string em MSIL

Como pode ser observado, as instruções em pilha ficam dentro de classes e métodos. A MSIL é complexa o suficiente para a produção de um trabalho exclusivo, atingindo aqui o limite inferior do escopo deste trabalho. De fato, apesar de ser a linguagem intermediária dentro de todo o contexto .NET e parte da CLR, a IL é considerada uma linguagem de programação “legítima”, existindo literatura específica voltada para o ensino de sua utilização direta no

desenvolvimento de software. Além disso, é de conhecimento geral que uma linguagem de alto nível (incluindo C#) não expõe todos os recursos da CLR; apenas a programação direta com a IL o faz. Para ilustrar, o código acima foi gerado utilizando um disassembler a partir da .dll gerada pela compilação do código C#, mas um código mais enxuto e otimizado com a mesma função pode ser visto abaixo:

```
.assembly Hello {}
.assembly extern mscorlib {}
.method static void Main()
{
     .entrypoint
     .maxstack 1
     ldstr "Hello, world!"
     call void [mscorlib]System.Console::WriteLine(string)
     pop
     ret
}
```

Trecho de Código 3: Impressão de string em MSIL otimizada

Uma listagem completa dos Opcodes, mnemônicos e suas funções pode ser encontrada no Anexo III.

3.5.2. Common Type System (CTS)

A especificação Common Type System (CTS) descreve completamente todos os tipos de dados e estruturas de programação suportadas pela CLR, especifica como estas entidades interagem entre si e detalha como elas são representadas no formato de metadados do .NET. (TROELSEN, 2007)

É o CTS que dá o suporte multi-linguagem à programação voltada para a CLR; é ele que define esta característica do .NET. Ele estabelece uma compatibilidade binária entre os tipos .NET pré-construídos na BCL em cada uma das linguagens suportadas (MAYO, 2008). Para um melhor entendimento,

abaixo segue uma tabela listando alguns os tipos (valorados) em C#, Visual Basic e seu Tipo .NET correspondente, cuja relação é responsabilidade do CTS:

C# (Alias)	VB (Keyword)	Tipo .NET
object	Object	System.Object
string	String	System.String
bool	Boolean	System.Boolean
byte	Byte	System.Byte
sbyte	Sbyte	System.SByte
short	Short	System.Int16
ushort	Ushort	System.UInt16
int	Integer	System.Int32
uInt	UInteger	System.UInt32
long	Long	System.Int64
ulong	ULong	System.UInt64
float	Single	System.Single
double	Double	System.Double
decimal	Decimal	System.Decimal
char	Char	System.Char
n/d	Date	System.DateTime

Tabela 3: Tipos .NET e seus alias específicos de linguagem

Nota: O Tipo .NET System.DateTime não possui um alias associado em C# mas é definido como um struct, logo, trata-se de um tipo valorado.

3.5.3. Common Language Specification (CLS)

A Common Language Specification (CLS) é uma especificação relacionada à CTS que define um subconjunto de tipos e estruturas de programação compatíveis entre todas as linguagens de programação suportadas pela plataforma. Logo, compilar tipos .NET que utilizam apenas de recursos compatíveis com a CLS garante que qualquer linguagem suportada poderá fazer uso deles. Entretanto, ao se fazer uso de um tipo de dados ou estrutura de programação fora dos limites da CLS, não há garantias que todas

as linguagens suportadas poderão interagir com o componente .NET construído. (TROELSEN, 2007)

Por mais que uma linguagem seja suportada pela CLR ela também deve se manter fiel às suas origens. Os tipos unsigned, por exemplo (UInt16, UInt32 e UInt64) não existem em todas as linguagens, logo um código C# ou VB que fizesse uso deste tipo de dados não seria considerado compatível com a CLS.

Entretanto, o mero uso de um tipo .NET que não faça parte da CLS em um componente não bloqueia a interoperabilidade deste com um componente em outra linguagem que também suporte a CLR. Basta que estes componentes não sejam expostos na interface pública do componente em questão. Exemplificando, imagina-se um componente hipotético escrito em C# com o seguinte método público:

```
public uint metodo()
{
    //Faz algo
}
```

Trecho de Código 4: Não compatível com a CLS

A chamada do método a partir de um componente escrito em outra linguagem qualquer rodando em cima da CLR não poderá utilizar o componente C# se a linguagem não possuir suporte para números inteiros sem sinal, sendo que o tipo não compatível com a CLS é apresentado publicamente na interface a ser utilizada. Este é o significado de não ser compatível com a CLS. Entretanto, o seguinte código é compatível com a CLS, pois o tipo não compatível (uint) não é exposto publicamente:

```
public int metodo()
{
    uint inteiro = 9;
    //Faz algo
}
```

Trecho de Código 5: Compatível com a CLS

Em suma, é devido à CLS que se atinge a interoperabilidade, característica amplamente divulgada pela Microsoft como uma das grandes vantagens da plataforma .NET. As linguagens que suportam a CLR podem ser vistas como um subconjunto da própria CLR/CTS ao passo que a CLS é um subconjunto formado pela intersecção destas linguagens. Esta idéia pode ser melhor ilustrada pela figura abaixo:

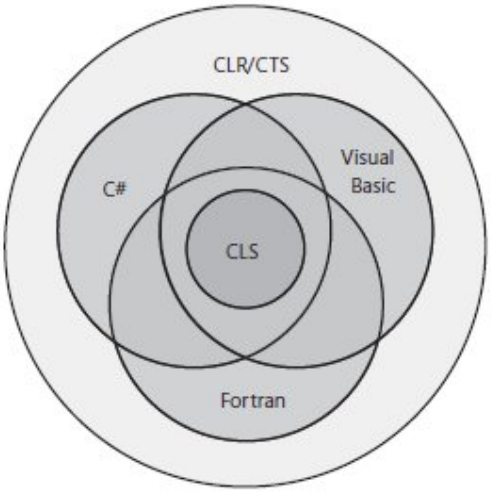

Figura 4: Relação entre CLR, CTS, CLS e linguagens suportadas

3.5.4. Compilação e Execução na CLR

O fluxograma abaixo descreve, de forma resumida, a execução de uma aplicação .NET em relação a uma aplicação não gerenciada no ambiente Windows:

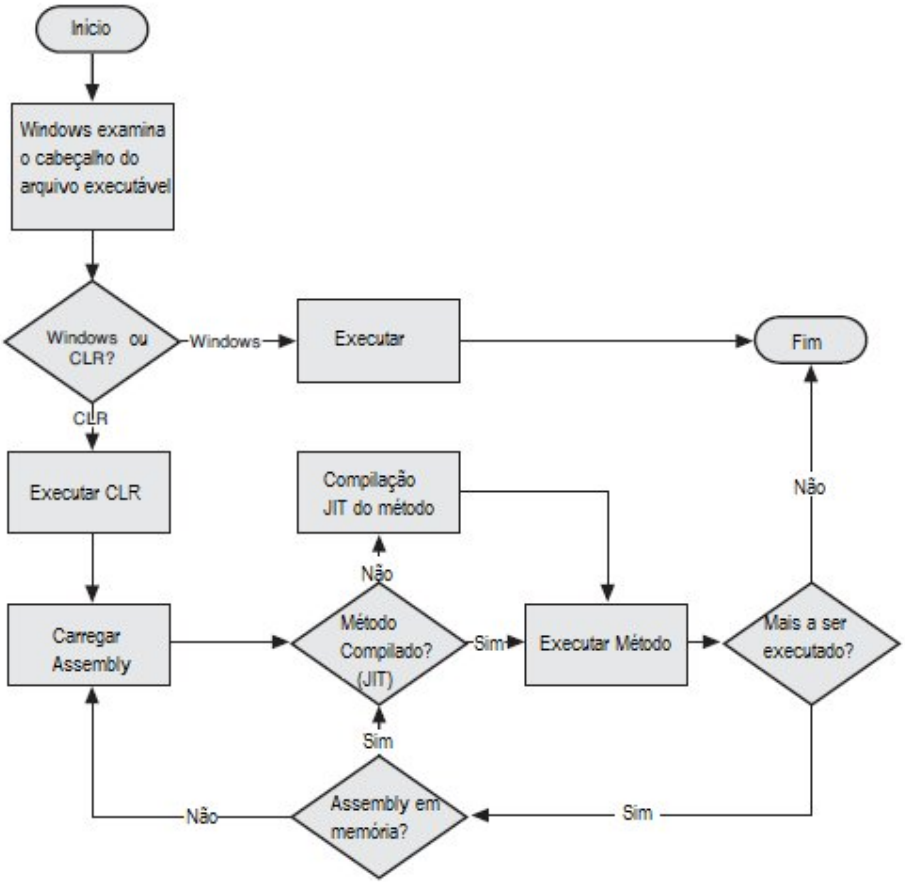

Figura 5: Resumo do processo de execução da CLR (Adaptado de Unleashed C# 3.0)

Após o Assembly .NET ser carregado, para um método ser executado, a IL deve ser convertida em instruções nativas para a CPU alvo. Esta tarefa é executada pelo compilador JIT da CLR:

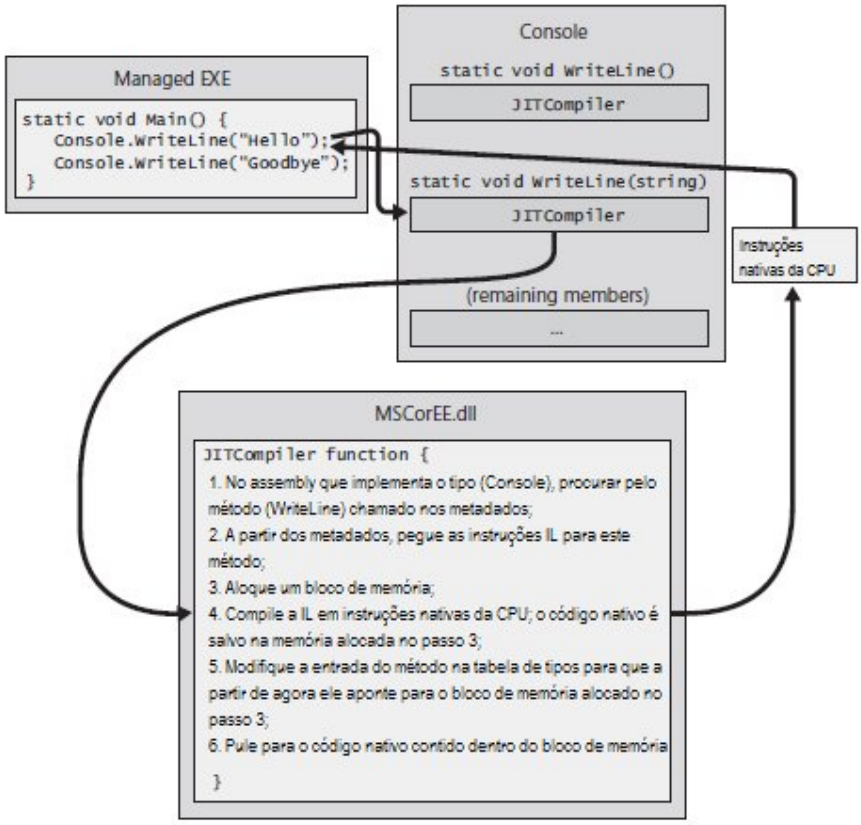

Figura 6: Primeira chamada de método (Adaptado de CLR via C#)

Antes da execução do método Main, a CLR detecta todos os tipos que são referenciados no código do método. Em seguida, a CLR aloca uma estrutura de dados interna que é usada para gerenciar o acesso aos tipos referenciados. No exemplo, é criada apenas a estrutura para o tipo Console, sendo este o único referenciado no método Main. Esta estrutura contém um campo para cada método definido no tipo Console. Cada campo contém o endereço onde a implementação do método está. Ao inicializar esta estrutura, a CLR define o valor deste campo como uma função interna e não-documentada contida dentro dela mesma, aqui chamada de JITCompiler.

Quando é feita a primeira chamada ao método WriteLine, a função JIT Compiler é invocada. Esta função é responsável por compilar o código IL de um método para instruções nativas da CPU.

Quando invocada, a função JITCompiler sabe qual método está sendo chamado e em que tipo este método é definido. A partir daí, a função procura nos metadados do assembly pelo código IL do método chamado. Em seguida, o JITCompiler verifica e compila o código IL em instruções nativas da CPU alvo, que são salvas em um bloco de memória alocado dinamicamente. Em seguida, tem-se o passo mais importante da compilação JIT na CLR: a função JITCompiler volta ao campo do método na estrutura de seu tipo (no caso, o método WriteLine da estrutura de Console) e substitui seu valor (que chamava a função JITCompiler) pelo endereço do bloco de memória contendo o código nativo da CPU. Finalmente, a função JITCompiler pula para o código no bloco de memória, retornando ao método Main em seguida.

Ao ser chamado pela segunda vez, o código para o método WriteLine já foi verificado e compilado. Desta forma, a chamada é feita diretamente ao bloco de memória, pulando a execução da função JITCompiler. A figura abaixo demonstra o processo:

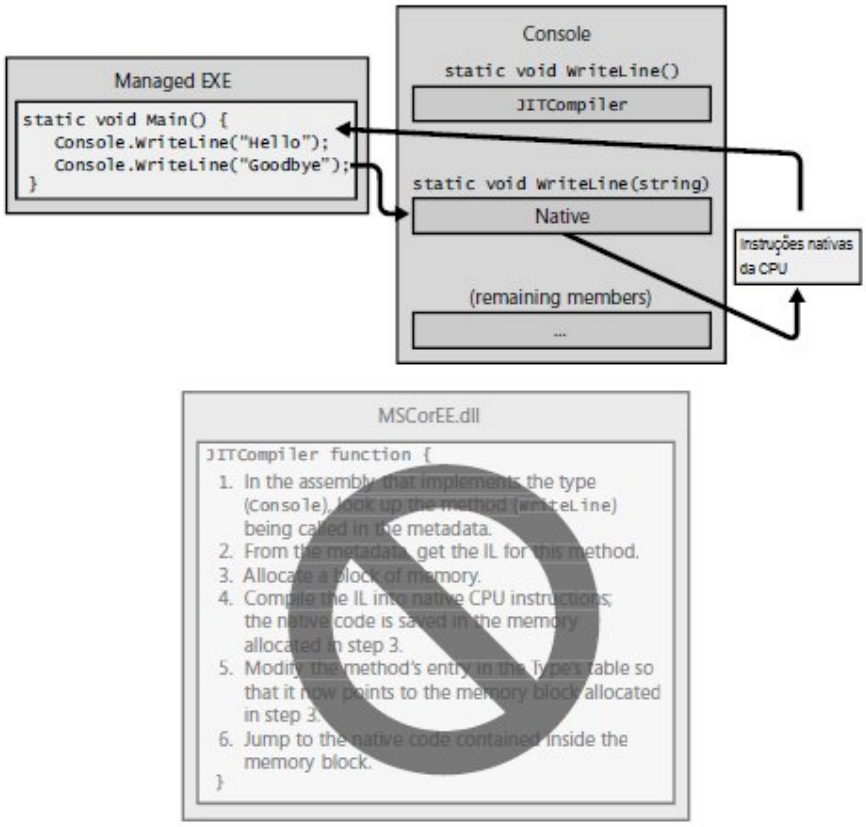

Figura 7: Segunda chamada de método (Adaptado de CLR via C#)

Assim nota-se que há perda de desempenho apenas na primeira vez que um método é invocado. Todas as chamadas subseqüentes ao método executam na velocidade total do código nativo. O compilador JIT armazena as instruções nativas dinamicamente na memória, ou seja, o código é descartado uma vez que a aplicação é finalizada.

3.5.5. Otimização de código na CLR

Existem dois parâmetros do compilador C# que têm impacto na otimização de código: ***/optimize*** e ***/debug***. A tabela a seguir mostra o impacto que estes parâmetros de compilador tem na qualidade do código IL gerado

pelo compilador C# e na qualidade do código nativo gerado pelo compilador JIT:

Parâmetros do Compilador	Qualidade do código IL	Qualidade do código nativo
/optimize- /debug- (padrão)	Não-Otimizado	Otimizado
/optimize- /debug(+/full/pdbonly)	Não-Otimizado	Não-Otimizado
/optimize+ /debug(-/+/full/pdbonly)	Otimizado	Otimizado

Tabela 4: Parâmetros do compilador C# para otimização de código

Dentro do escopo deste trabalho, o parâmetro **/optimize+** pode ser considerado o mais importante. Com ele instruções NOP (no-operation) – acrescentadas ao código para facilitar a depuração através do Visual Studio – não são inseridas, gerando um código IL otimizado e de leitura mais fácil, além de tornar menor o arquivo EXE/DLL resultante da compilação.

3.5.6. Linguagens suportadas pela CLR

Quando é falado em linguagens suportadas pela CLR, quer-se dizer, na realidade, linguagens de programação as quais foram criados compiladores/tradutores para gerar código objeto em MSIL a partir de um programa fonte em uma linguagem qualquer.

A Microsoft criou diversos compiladores que possuem como alvo o ambiente de execução: C++/CLI, C#, Visual Basic, F#, Iron Phyton, Iron Ruby e um assembler para a própria IL (ILasm). Além da Microsoft, diversas outras empresas, faculdades e universidades criaram compiladores capazes de produzir código para a CLR. Existem compiladores para Ada, APL, Caml, COBOL, Eiffel, Forth, Fortran, Haskell, Lexico, LISP, LOGO, Lua, Mercury, ML,

Mondrian, Oberon, Pascal, Perl, PHP, Prolog, RPG, Scheme, Smalltalk e Tcl/Tk. (RICHTER, 2010)

Obviamente não se pode esperar que todas estas linguagens tenham um grande nível de interoperabilidade através da CLS, entretanto é desejado algum nível mínimo existente entre elas, sendo que seria desmotivador desenvolver um compilador que vise a CLR como alvo ignorando uma das características principais do .NET.

Algo que se pode ser assertivo a respeito, contudo, é o fato que as características núcleo da CLR (gerenciamento de memória, carregamento de assemblies, segurança, tratamento de exceções e sincronização de threads) estão disponíveis para toda e qualquer linguagem de programação que a tenha como alvo. (RICHTER, 2010)

4. PLATAFORMA JAVA

4.1. Visão Geral

Antes da internet atingir o público geral, a maior parte dos programas eram escritos e compilados possuindo CPU e sistema operacional específicos. Enquanto é verdade que programadores sempre gostaram de reusabilidade, a capacidade de portar facilmente um programa de um ambiente para outro sempre era deixada de lado em detrimento à problemas mais urgentes. Entretanto, com a ascensão da internet, onde diversos tipos diferentes de CPU's e sistemas operacionais estão conectados, o antigo problema da portabilidade tornou-se substancialmente mais importante. Concorrentemente, havia um segundo problema fundamental impedindo que programação baseada na internet se tornasse realidade: falta de segurança. Os usuários não assumiriam o risco de executar um programa distribuído pela internet quando este poderia possuir código malicioso (Schildt, 2009). Foi neste contexto que em 1995, a Sun Microsystems lançou a plataforma Java.

Para ambos os problemas, a solução ideal foi implementada pela plataforma, na forma de seu ambiente de execução (a Java Virtual Machine ou JVM) e da linguagem intermediária projetada para rodar neste ambiente, o Java bytecode.

O problema da portabilidade foi resolvido pela tradução do código fonte na linguagem Java em código intermediário (o bytecode) executado pela JVM. Desta forma, um programa Java pode rodar em qualquer ambiente (real) que disponibilize a JVM e sendo esta de fácil implementação, tornou-se disponível para um grande número de ambientes. Além disso, já que a JVM executa o

bytecode, esta tem controle total do programa e pode preveni-lo de fazer algo que não deveria, resolvendo o problema da segurança (Schildt, 2009). Em sua maior parte, a implementação da JVM utilizada neste trabalho se trata da chamada HotSpot, da própria Sun, mas em alguns casos são levantadas comparações e utilizados exemplos da JRockit, implementação da Oracle.

É importante fazer a distinção do que significa o termo “Java”. A Oracle (atual subsidiaria da Sun) informa que Java é tanto uma linguagem de programação quanto uma plataforma. E, como plataforma, o conceito pode ser explorado de duas maneiras: o Java é uma plataforma de execução (constituído da JVM e do conjunto padrão de bibliotecas – sua API) distribuído gratuitamente sob o cunho de Java Runtime Environment (JRE) e também uma plataforma de desenvolvimento, encontrada nas versões Micro, Standard e Enterprise, que além da JRE, incluem conjuntos de bibliotecas e ferramentas com utilidade apenas para desenvolvedores, como bibliotecas para internacionalização e o compilador Java (também conhecido como “javac”), que faz a tradução do programa na linguagem Java (arquivos .java) para um programa em Java bytecode (arquivos .class). Estes são interpretados pela JVM, compilados em tempo de execução por um compilador JIT, ou ainda, é utilizado um híbrido de interpretação e compilação JIT, basicamente dependendo da implementação da JVM. Também podem ser encontrados compiladores ahead-of-time (AOT) para plataformas específicas que permitem pré-compilar programas em Java bytecode para código nativo, contornando o uso da JVM.

4.1.1. Edições da Plataforma Java

Uma edição da plataforma Java refere-se primeiramente a quais componentes estão inclusos em sua API e que ferramentas de desenvolvimento a acompanham. Segue abaixo uma lista das 3 edições da plataforma juntamente com suas características

- Java Platform Standard Edition (Java SE ou JSE): Kit necessário para o desenvolvimento de todas as aplicações, exceto por aquelas projetadas para aparelhos eletrônicos (ver Java Micro Edition). O Java SE é constituído de um compilador (o javac), um ambiente de execução (a JVM) e da API base. Esta é a edição selecionada para o desenvolvimento deste trabalho, em sua 6ª versão.
- Java Platform Enterprise Edition (Java EE ou JEE): Este pacote inclui um servidor de aplicações, um servidor web, APIs específicas da edição Enterprise, suporte para os chamados Enterprise Beans, API para Java Servlets e a tecnologia Java Server Pages (JSP). O JEE deve ser usado em conjunto com o JSE.
- Java Platform Micro Edition (Java ME ou JME): Edição direcionada ao desenvolvimento para aparelhos eletrônicos, como celulares e palm tops. Fornece ferramentas para desenvolvimento, implantação e configuração dos aparelhos, além de APIs especializadas para cada tipo de aparelho.

  (Adaptado de Sun: Introducing the Java Platform)

4.2. Componentes da Plataforma Java

A figura abaixo representa como a Plataforma Java está organizada, referente à versão 6:

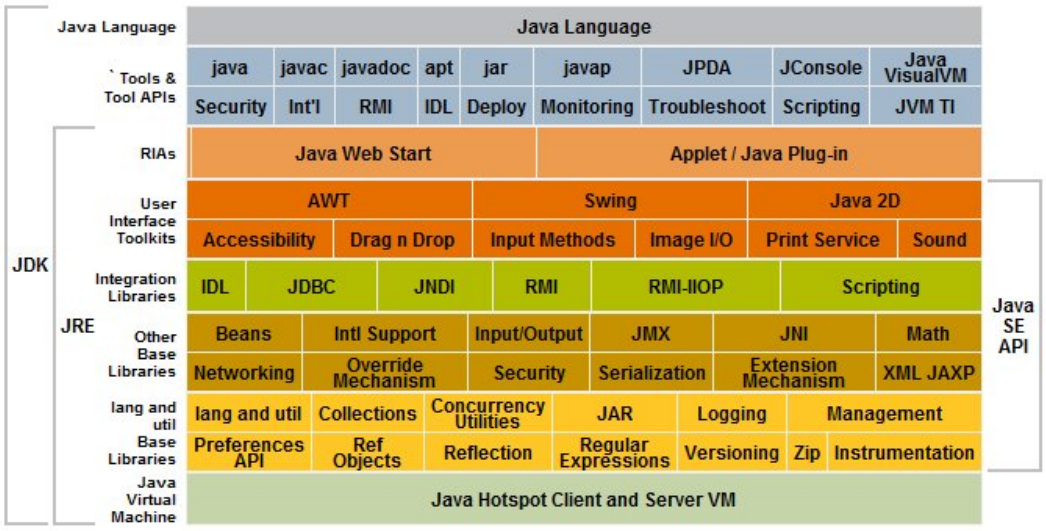

Figura 8: Organização da Plataforma Java

A primeira vista, nota-se uma fragmentação muito maior que a vista no capítulo 3.2 (Componentes da plataforma .NET). Entretanto, há uma relação de equivalência explicada nos tópicos a seguir.

4.2.1. Java SE API

Como pode ser visto na figura 8, a API do Java SE 6 possui mais de 40 componentes que encapsulam classes e bibliotecas, chamadas packages em Java. Segue abaixo a lista de todas as APIs e uma breve descrição de cada uma:

User Interface Toolkits

Nome da API	Descrição
Abstract Window Toolkit (AWT)	Suporte para programação de GUI. As classes Swing utilizam o AWT como base.
Swing	As classes Swing são parte do Java Foundation Classes

	(JFC) e implementam um conjunto de components para a construção de interfaces gráficas. Os componentes Swing são implementados inteiramente em Java e permite a criação de GUIs que sejam iguais entre diferente plataformas ou que assumam o aspecto visual do SO aonde estão rodando.
Java 2D	Conjunto de classes para criação de gráficos e imagens 2D avançadas.
Accessibility	API para criação de aplicações acessíveis à pessoas com deficiências.
Drag n Drop	Permite a transferência de dados entre aplicações construídas em Java e em outras linguagens de programação, bem como entre aplicações Java diferentes e entre uma mesma aplicação Java.
Input Methods	Suporte para métodos de entrada diferentes de um teclado padrão, tais como escrita a mão, voz, etc.
Image I/O	Fornece uma arquitetura para trabalhar com imagens armazenadas em arquivos e acessadas através de uma rede.
Print Service	Permite impressão em qualquer edição da plataforma Java.
Sound	API para captura, processamento e execução de áudio.

Tabela 5: APIs do subconjunto User Interface Toolkits

Integration Libraries

Nome da API	**Descrição**
Java Database Connectivity (JDBC)	API para acesso de dados a partir da linguagem de programação Java, desde bancos de dados relacionais à arquivos de texto.
Remote Method Invocation (RMI)	Permite o desenvolvimento de aplicações distribuídas permitindo com que objetos executando em uma JVM invoquem métodos de um objeto executando em outra JVM que pode estar localizada em uma máquina (física) diferente.
Java IDL	Adiciona as funcionalidades CORBA à plataforma Java, fornecendo interoperabilidade e conectividade baseada em padrões estabelecidos.
RMI-IIOP	RMI sobre Internet Inter-ORB Protocol. Permite a programação de servidores CORBA e aplicações através da API RMI.
Scripting	Framework que permite que aplicações Java possam hospedar engines de script.
Java Naming and Directory Interface (JDNI)	Fornece uma interface unificada para multiplos serviços de diretórios, permitindo que aplicações cliente escritas em Java descubram e obtenham dados através de um nome.

Tabela 6: APIs do subconjunto Integration Libraries

Other Base Libraries

Nome da API	**Descrição**

Intl Support	Permite o desenvolvimento de aplicações internacionalizadas.
Input/Output	Funcionalidade que inclui I/O de arquivos e dispositivos, serialização de objetos, gerenciamento de buffer e suporte a conjuntos de caracteres específicos. Adicionalmente oferece características adicionais para servidores escaláveis.
Java Management Extensions (JMX)	API padrão para o gerenciamento e monitoramento de recursos como aplicações, dispositivos, serviços e da própria JVM.
Java Native Interface (JNI)	Interface de programação padrão para escrita de métodos nativos em Java e embarcar a JVM em aplicações nativas. O objetivo principal da JNI é compatibilidade binária de bibliotecas de métodos nativos entre todas as implementações da JVM em uma dada plataforma.
JavaBeans	Contem classes relacionadas ao desenvolvimento de beans – componentes baseados na arquitetura JavaBeans que tem a característica de serem acoplados como parte do desenvolvimento de uma aplicação
Networking	Fornece classes para funcionalidades de rede, incluindo endereçamento, classes para uso de URLs e URIs, classes de socket para conexão com servidores, funcionalidades para segurança em redes, entre outros.
Endorsed Standards Override Mechanism	O mecanismo de sobrescrita de padrões reconhecidos provê meios para desenvolvedores fornecerem novas versões de padrões reconhecidos que não estão inclusos na Plataforma Java.
Security	API que fornece funcionalidades como controle de acesso configurável, assinatura digital, autenticação e autorização, criptografia, comunicação segura pela internet, etc.
Serialization	Extensão da serialização de objetos fornecida pela API de I/O.
Extension Mechanism	Permite o uso de pacotes de classes adicionais como se fossem classes nativas da Plataforma Java.
XML (JAXP)	Conjuntos de APIs para processamento de documentos e dados em XML.

Tabela 7: APIs do subconjunto Other Base Libraries

Lang and Util Base Libraries

Nome da API	**Descrição**
Lang and Util	Fornece classes fundamentais para a utilização da linguagem Java, como Object e Class, classes para encapsulamento de tipos primitivos, classe básica de matemática, etc.
Collections	O Framework Collections é uma arquitetura unificada para representação de coleções, permitindo que sejam manipuladas independentemente dos detalhes de sua representação.

Concurrency Utilities	Este pacote fornece um framework de manipulação de threads de alto desempenho, incluindo ferramentas como pools de threads e bloqueio de filas. Também fornece pacotes de primitivas de baixo-nível para programação paralela avançada.
Java Arquives (JAR) Files	O JAR é um formato de arquivo independente de plataforma que permite a agregação de múltiplos arquivos em um só, Também suporta compressão e assinatura digital.
Logging	API com o intuito de facilitar geração de logs e relatórios propícios para a análise por usuários finais, administradores de sistema, engenheiros de campo e equipes de desenvolvimento.
Monitoring and Management	APIs de monitoramento e gerenciamento para a JVM, para o Logging, jconsole e outras ferramentas de monitoramento, JMX e Sun's Platform Extension.
Preferences API	Fornece um meio para aplicações armazenarem e restaurarem preferências e dados de configuração do sistema e de usuários.
Ref Objects	Oferece um limitado nível de interação com o Garbage Collector
Reflection	Permite que o código Java discubra informações sobre campos, métodos e construtores de classes carregadas e use tais membros para operar em suas instâncias com restrições de segurança.
Versioning	Permite controle de versão em nível de pacote para que aplicações possam identificar em tempo de execução a versão de uma JRE, VM e pacote de classes específico.

Tabela 8: APIs do subconjunto Lang and Base Classes

Como dito anteriormente, o que distingue as edições da plataforma Java é o conjunto de APIs inclusas. As APIs descritas neste tópico são referentes ao Java SE 6 edição. Os subtópicos seguintes, cobrindo o Java Runtime Environment e o Java Development Kit, e o tópico 5.4. Java Virtual Machine podem ser considerados comuns à todas as edições da plataforma.

4.2.2. Java Runtime Environment (JRE)

O JRE é o produto da plataforma Java que permite os usuários executarem os aplicativos desenvolvidos com a plataforma. Como consta na figura 8, é composto pela API da edição da plataforma Java em questão, em

conjunto com a Java Virtual Machine (no caso, trata-se da implementação HotSpot da própria Sun) e o domínio RIA – Rich Internet Applications Development (desenvolvimento e implantação de aplicações “ricas” para internet). Assim como foi dada grande ênfase ao ambiente de execução da plataforma .NET – a CLR – o tópico 4.4 tem a finalidade de entrar em detalhes a respeito da arquitetura e funcionamento da JVM.

Em linhas gerais, a JVM interpreta ou compila de forma JIT o arquivo codificado em Java Bytecode (.class) gerado através da compilação/tradução de um programa com o código fonte em Java de extensão .java. A escolha entre interpretação e tradução depende da implementação da JVM utilizada bem como parâmetros de configuração. A implementação HotSpot, por exemplo, utiliza de um híbrido de interpretação e compilação, iniciando sempre com interpretação e após algum tempo de execução do programa, avalia quais métodos tem a tendência de serem chamados com maior freqüência (os ditos “hot spots”), os compilando de forma JIT para que a interpretação não seja mais necessária.

Até pouco tempo atrás (Java 1.4), o domínio RIA não fazia parte da estrutura da JRE. De fato, assim como ocorre com o arquivo distribuído pela Microsoft para a execução de programas em .NET cunhado .NET Framework, de maneira comercial, o JRE pode ser considerado o conjunto de APIs básica do Java e seu ambiente de execução – a JVM.

Nota: Por se tratar de um modelo de desenvolvimento voltado para applets e aplicações Java Web Start, o RIA não entra no escopo deste trabalho.

4.2.3. Java Development Kit (JDK)

O JDK pode ser visto como o macro-conjunto que engloba todas as outras tecnologias referentes a Plataforma Java SE 6. Incorpora ao JRE ferramentas de desenvolvimento como o compilador java (javac), debugger e outras ferramentas necessárias para o desenvolvimento de aplicações Java. Não inclui, entretanto, uma IDE, ficando ao desenvolvedor fazer a escolha de sua IDE de preferência dentre as diversas oferecidas no mercado.

Entre as IDE's mais famosas, pode-se citar o NetBeans, o Eclipse, JCreator, JBuilder, etc. Atualmente, o NetBeans é considerada a IDE mais adequada para iniciantes na tecnologia, ao passo que o Eclipse (e seus diversos plug-ins) é consagrada como a IDE preferida por desenvolvedores Java profissionais.

Além das ferramentas citadas, a Sun inclui a própria linguagem de programação Java como parte do JDK, o que é perfeitamente plausível. Ao se observar a figura 8, pode-se concluir que o JDK é a própria plataforma Java (no sentido de plataforma de desenvolvimento), com a linguagem Java inclusa, algo que, assim como na questão do .NET, gera certa discórdia dentro da comunidade, entretanto, diferente do .NET, a grande maioria dos desenvolvedores considera que neste caso, a linguagem de fato é parte da plataforma.

4.3. Linguagem de programação Java

Java é uma linguagem de programação estruturada, orientada a objetos com sintaxe e filosofia derivadas do C++. Os aspectos inovadores do Java não

foram guiados tanto pelos avanços na arte da programação quanto pelas mudanças no ambiente computacional.

Diferente do que ocorre com o C# e a plataforma .NET, é bem mais complexo distinguir a linguagem de programação Java da plataforma Java, sendo que a intersecção de suas características é quase total; as características para as quais a plataforma foi criada para dar suporte são implementadas pela linguagem de programação: robusta, segura, portável, distribuída. Como não houve preocupação com o suporte a programação multi-linguagem no projeto Java, conceituar a linguagem teoricamente separando-a da plataforma torna-se uma tarefa difícil e até certo ponto, desnecessária. Semelhante à tecnologia da Microsoft, a linguagem Java e a JVM são descritas em especificações separadas (estas, contudo, não são padronizadas), entretanto pode-se dizer com segurança que ambas implementações destas especificações são parte da Plataforma Java.

Em seu início, o lema da linguagem (e da plataforma como um todo) era “write once, run anywhere” (escreva uma vez, execute em qualquer lugar), enfatizando que Java é uma linguagem portável. Apesar de ainda ser válido para certas arquiteturas, hoje se deve levar em consideração a existência de centenas de máquinas virtuais diferentes para diversas arquiteturas diferentes – inclusive sistemas embarcados – e que a interpretação de código vem cada vez mais dando lugar para um esquema de compilação JIT, tornando a linguagem mais vinculada ao hardware do que inicialmente planejado.

4.4. Java Virtual Machine (JVM)

Apesar de seu uso geral, o termo Java Virtual Machine não diz respeito a uma máquina virtual em particular e sim a uma especificação de máquina virtual desenvolvida pela Sun para a plataforma Java. Existem várias implementações da JVM, desenvolvidas por diversas empresas e com objetivos específicos; este trabalho foca-se em uma implementação específica: a HotSpot, da própria Sun.

No geral, a JVM é o ambiente de execução para interpretação ou compilação JIT do Java bytecode, sendo esta a linguagem intermediária da arquitetura, resultante da compilação/tradução de programas fonte na linguagem Java pelo compilador javac. Entretanto, sendo que a especificação da JVM é aberta, ao longo dos anos mais linguagens surgiram que podem ser traduzidas para bytecode e serem interpretadas ou compiladas por alguma implementação da máquina virtual.

4.4.1. Java Bytecode

O Java bytecode é o formato de instruções executado pela JVM – a linguagem intermediária entre o programa fonte em Java (ou outra linguagem compatível com a JVM) e o programa objeto em instruções nativas. Semelhante à MSIL, o Java Bytecode é muito próximo a uma linguagem de montagem tradicional, com mnemônicos e opcodes correspondentes. É uma linguagem baseada em pilha onde cada opcode tem o tamanho de um byte, resultando em 256 possíveis instruções – algumas delas, contudo, necessitam de parâmetros, resultando em instruções de mais de um byte. Uma listagem completa das instruções, seus mnemônicos e opcodes pode ser encontrada no

Anexo IV – Lista de Instruções do Java Bytecode. Abaixo segue dois trechos de código: uma simples impressão de texto em tela na linguagem Java e seu correspondente em Java Bytecode.

```
public class HelloWorld {

	public static void main(String[] args)
	{
		System.out.println("TCC - Java vs .NET");
	}

}
```

Trecho de Código 6: Impressão de string em Java

```
// class version 50.0 (50)
// access flags 0x21
public class HelloWorld {

  // compiled from: HelloWorld.java

  // access flags 0x1
  public <init>()V
   L0
    LINENUMBER 2 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init>()V
    RETURN
   L1
    LOCALVARIABLE this LHelloWorld; L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

  // access flags 0x9
  public static main([Ljava/lang/String;)V
   L0
    LINENUMBER 6 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    LDC "TCC - Java vs .NET"
    INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V
   L1
    LINENUMBER 7 L1
    RETURN
   L2
    LOCALVARIABLE args [Ljava/lang/String; L0 L2 0
    MAXSTACK = 2
    MAXLOCALS = 1
}
```

Trecho de Código 7: Impressão de string em Java Bytecode

Apesar de não ser tão promovido quanto a MSIL, é de conhecimento geral que a compreensão do Bytecode é vantajosa para o programador, entretanto não é necessária para o desenvolvimento na plataforma. Diferente da MSIL, entretanto, não há relatos de que existam funcionalidades da

plataforma Java que possam ser acessadas apenas através do Bytecode; neste aspecto, o Java Bytecode é transparente entre a linguagem de programação Java e a plataforma de execução.

No caso de estudo apresentado neste trabalho – a HotSpot – o Java bytecode é visto de duas formas: High e Low Intermediate Representation. Maiores esclarecimentos são dados no tópico 4.5. HotSpot.

4.4.2. Implementações da JVM

Como mencionado anteriormente, existem diversas implementações da JVM. Apesar deste trabalho ter o enfoque em apenas uma – a mais popular – faz-se necessário fazer um breve comentário sobre as demais.

Implementações da JVM são diferentes na forma em que a compilação JIT, otimizações, garbage collector, plataformas e versões de Java suportados são implementados. Entretanto, todas devem satisfazer um conjunto de características e comportamentos definidos pela especificação para que o Bytecode seja executado corretamente.

Em se tratando de diferenças técnicas, ao longo dos anos, estas vêm diminuindo. Antigamente, a implementação J9 da IBM e a implementação JRockit (atualmente da Oracle) tinham um desempenho superior em relação à HotSpot. Isso se dava à diferenças significativas em otimizações em tempo de execução, garbage collection e na quantidade e qualidade do código nativo gerado. Hoje em dia, tais diferenças não são mais tão importantes.

Apenas para exemplificar, atualmente a JRockit não possui um interpretador, como a HotSpot. Seu comportamento se assemelha à um ambiente de execução como a CLR, onde o código intermediário é compilado

de forma JIT para código nativo na primeira chamada de cada método. Ver Anexo V – JRockit para uma visão mais detalhada de seu procedimento.

4.4.3. Linguagens suportadas por uma JVM

Apesar de não ser parte do projeto inicial da plataforma, compiladores/tradutores de várias linguagens para Java bytecode, permitindo que programas escritos nestas linguagens sejam executados por uma JVM. Algumas destas linguagens necessitam ser interpretadas por um programa Java, ao passo que outras são traduzidas para bytecode e executadas normalmente como um código escrito em Java se comportaria na JVM em questão.

As linguagens mais populares suportadas pela JVM são:

- AspectJ: extensão de Java orientada a aspectos (desenvolvido pela Eclipse Foundation);
- ColdFusion: linguagem de script traduzida para Java (Adobe);
- Clojure: um dialeto de Lisp (também é suportada pela CLR; criada pelo desenvolvedor independente Rich Hickey);
- Groovy: linguagem orientada a objetos, dinâmica, com sintaxe semelhante a Java e características de Phyton, Ruby, Perl e Smalltalk (desenvolvida por James Strachan);
- JavaFX Script: linguagem de script, desenvolvida pela própria Sun que tem como alvo o domínio RIA, competindo com o Flash/Flex da Adobe e com o Silverlight da Microsoft.
- JRuby: implementação de Ruby que visa a JVM;
- Jython: implementação de Phyton que visa a JVM;

- Rhino: implementação de JavaScript que visa a JVM, convertendo scripts em JavaScript em classes em Java (desenvolvida pela Mozilla Foundation);
- Scala: linguagem orientada a objetos e funcional.

As linguagens citadas acima podem ser consideradas totalmente interoperáveis com Java, apesar de algumas ainda não serem implementações completas de suas versões estáveis originais (JRuby, Jython). Implementações de outras linguagens visando a JVM incluem Ada, AWK, C, COBOL, Common Lisp, Component Pascal, Erlang, Forth, LOGO, Lua, Oberon2, Objective Caml, Pascal, PHP, Rexx, Scheme e Tcl. Além destas, existe ainda mais de uma dezena de novas linguagens criadas especialmente visando a JVM, onde sua maior parte tem propósitos puramente acadêmicos.

4.5. HotSpot JVM

As primeiras versões de JVM utilizavam exclusivamente de interpretação de código. Em seguida, surgiram versões com interpretador em adição à código gerado por modelos (templates) e finalmente vieram as JVMs com interpretador em conjunto com otimizadores de código (PALECZNY, 2001). Desde suas primeiras versões, a implementação HotSpot possui dois compiladores: o compilador para aplicações de uso geral, como aplicações interativas que possuam uma GUI (client) e o compilador para aplicações rodando em servidores (server). De acordo com Paleczny (2001), a versão client fornece um tempo compilação extremamente rápido e pequena utilização da memória principal, com níveis modestos de otimização. A versão server aplica otimizações mais agressivas com o fim de atingir um desempenho

melhorado, tornando a compilação mais custosa. Esta análise mantém o foco na versão Client da HotSpot, por se tratar, possivelmente, do compilador Java (Bytecode para código nativo) mais utilizado no mundo todo.

A versão 6 da máquina virtual Hotspot da Sun vêm com uma versão redesenhada do compilador JIT client, incluindo diversos resultados de pesquisas que precederam seu lançamento. O compilador client é o núcleo da configuração da máquina virtual usada por padrão por aplicações desktop interativas. Para tais aplicações, inicialização rápida e tempos de pausa são mais importante que desempenho total. (KOTZMANN, 2008)

Para melhor legibilidade, daqui em diante os compiladores client e server serão referidos como compiladores “cliente” e “servidor”

4.5.1. Interpretação e Gerenciamento de Memória na HotSpot

Portabilidade em Java é alcançada pela tradução do código fonte em bytecodes independentes de plataforma. Para executar programas Java em uma plataforma específica, uma JVM deve existir para tal plataforma. Ela executa os bytecodes após verificar se eles não comprometem a segurança e integridade da máquina sob a qual está rodando (KOTZMANN, 2008). A implementação da Sun Microsystems (atual subsidiária da Oracle) da JVM é a conhecida Java HotSpot VM.

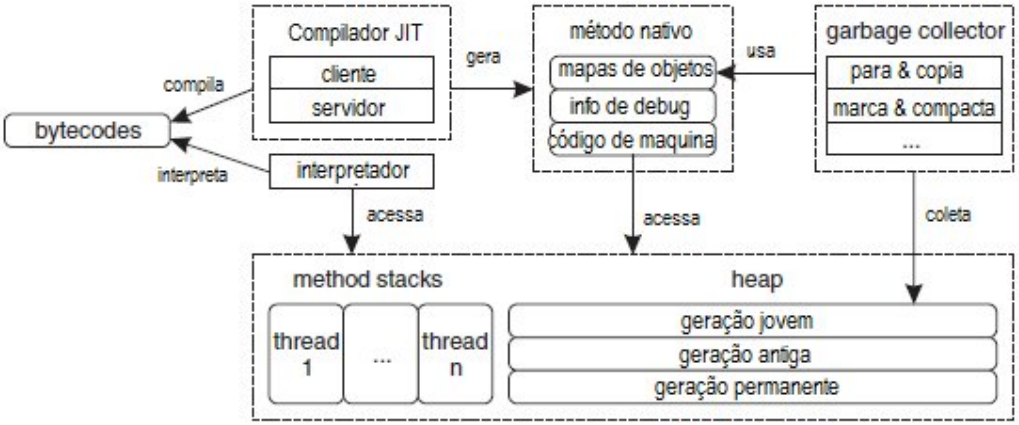

Figura 9: Arquitetura da Java HotSpot VM

A arquitetura geral é apresentada na figura 9. A execução de um programa Java inicia pelo interpretador, que varre os bytecodes de um método e executa um modelo de código (template) para cada instrução. Apenas os métodos invocados com maior freqüência, chamados de “hot spots” (pontos quentes) são agendados para compilação JIT. Esta informação permite que o compilador alinhe mais métodos (substitua a chamada do método pelo seu corpo) e gere um código de máquina com melhor otimização.

Se um método contém um loop extenso, ele pode ser compilado independente da freqüência de sua invocação. A máquina virtual conta o número de ramificações retroativas e quando um limite é alcançado, ela suspende a interpretação e compila o método em execução. Um novo quadro é configurado na pilha para o método nativo e inicializado para combinar com a pilha de quadros interpretados. Então a execução do método prossegue utilizando o código de máquina do método nativo. A mudança de código interpretado para compilado no meio da execução se chama “on-stack replacement” (OSR) ou substituição de pilha.

O garbage collector geracional da HotSpot gerencia memória alocada dinamicamente. Ele utiliza técnicas de garbage collection exatas, então todo objeto e todo ponteiro para um objeto deve ser precisamente conhecido no momento da “coleta”. A memória é dividida em três gerações: a geração jovem, para objetos recentemente alocados, a geração antiga para objetos de vida mais longa e uma geração permanente, para estruturas de dados internas.

Novos objetos são alocados seqüencialmente na geração jovem. Quando a geração jovem é preenchida, uma coleta “para-e-copia” (stop-and-copy) é iniciada. Quando objetos sobrevivem a um certo número de ciclos de coleta, estes são promovidos para a geração antiga, que é coletada por um algoritmo marca-e-compacta (mark-and-compact).

Garbage collection exata necessita de informações sobre ponteiros para objetos na heap (memória dinâmica). Para o código de máquina, esta informação está armazenada nos mapas de objetos criados pelo compilador JIT. Além desta, o compilador cria a informação de depuração, mapeando o estado de um método compilado de volta à seu estado interpretado. Isto permite otimizações mais agressivas, sendo que a máquina virtual pode *desotimizar* de volta à um estado seguro quando presunções sob as quais uma otimização ocorreu deixam de ser válidas. O código de máquina, os mapas de objeto e as informações de depuração são armazenadas na estrutura chamada *objeto de método nativo*. Garbage collection e desotimização podem ocorrer apenas em determinados pontos do programa (chamados “safepoints” ou pontos seguros), tais como ramificações retroativas, chamadas de métodos, instruções de retorno e operações que possam gerar exceção. (KOTZMANN, 2008)

A implementação HotSpot está disponível em versões 32 e 64-bit para o sistema operacional Solaris em plataformas SPARC e Intel, para Linux e Windows.

4.5.2. Geração de Código no Compilador Cliente da HotSpot

A figura abaixo representa a estrutura do compilador cliente da HotSpot:

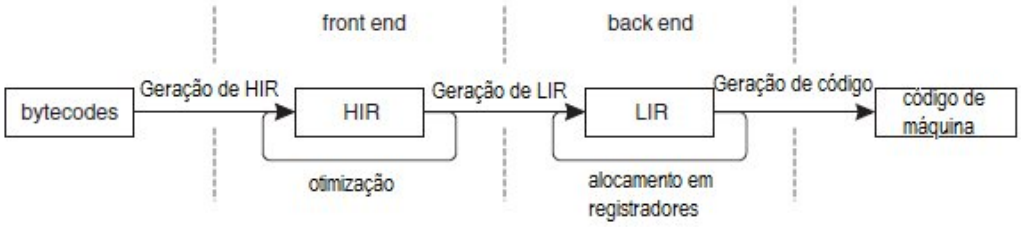

Figura 10: Estrutura do compilador cliente da Hotspot

O processo: Inicialmente uma representação intermediária de alto-nível (*high-level intermediate representation* – HIR) do método compilado é construída através de uma interpretação abstrata dos bytecodes. A HIR é uma representação de método baseada em grafos que utiliza a “forma simples de atribuição estática” (*Static Single-Assignment* ou SSA); é independente de plataforma e representa o método em um alto nível onde otimizações globais são fáceis de serem aplicadas.O fato da HIR estar na forma SSA significa que para cada variável existe apenas um ponto no programa aonde valor é atribuído a ela. Uma instrução que carrega e calcula um valor representa tanto a operação quanto seu resultado permitindo que operandos possam ser representados como ponteiros para instruções anteriores. Tanto durante quando após a geração da HIR, diversas otimizações são realizadas.

O *back-end* do compilador traduz o código otimizado em uma representação intermediária de baixo-nível (*low-level intermediate representation* – LIR). Conceitualmente a LIR é semelhante a código de máquina, mas de maneira geral, ainda é independente de plataforma. Em contraste às instruções em HIR, operações na LIR são feitas sobre registradores virtuais ao invés de referências a instruções anteriores. A LIR facilita várias otimizações de baixo nível e é a entrada para o *linear scan register allocator*, que faz o mapeamento de registradores virtuais para registradores físicos.

Após a alocação dos registradores, o código de máquina pode ser gerada de maneira simples e direta. O compilador atravessa a LIR, operação por operação e emite as instruções de máquina apropriadas em um buffer de código. Este processo também utiliza os mapas de objeto e a informação de depuração. (KOTZMANN, 2008)

4.5.3. Otimizações

Otimizações locais são executadas durante a geração da HIR: *Constant folding* (dobra constante) simplifica instruções aritméticas com operandos constantes assim como em ramificações onde a condição é sempre verdadeiro ou falso. *Local value numbering* (numeração de valor local) elimina sub-expressões comuns dentro de um bloco básico. *Method Inlining* (alinhamento de método) substitui chamadas de método por uma cópia do corpo do método.

Após gerada a HIR, otimizações globais são realizadas. *Null-check elimination* (eliminação de verificação de valor nulo) remove verificações de valor nulo se o compilador puder provar que o objeto acessado não é nulo.

Conditional expression elimination (eliminação de expressões condicionais) substitui o caminho comum de uma ramificação que carrega um de dois valores por uma instrução “move” condicional. Global value numbering (numeração de valor global) elimina sub-expressões comuns dentro dos limites de um bloco básico.

5. COMPARAÇÃO ENTRE AS PLATAFORMAS JAVA E .NET

Finalmente, chega-se ao resultado deste estudo. Como se trata de uma comparação entre as duas plataformas, o conceito é único entre todos os critérios de comparação: qual é a melhor no critério em questão. São 4 critérios de comparação: portabilidade de código, interoperabilidade e desempenho (custo processamento e consumo de memória) através do uso de 4 algoritmos para benchmark: BubbleSort, InsertSort, geração de números primos (utilizando tanto matemática de números inteiros quanto matemática de ponto flutuante) e Thread Ring. Os dois primeiros critérios são teóricos e ao longo do estudo, já foram mencionados. Este capítulo visa recapitular o que foi dito, reforçar o conceito, se necessário e concluir qual das plataformas se sobressai no critério. O 3º critério, desempenho, será testado através da execução de cada um dos algoritmos três vezes, utilizando diferentes cargas, comparar o resultado da execução de cada algoritmo isoladamente e ao final, comparar o resultado da execução de todos os algoritmos lado-a-lado, separados por carga.

Mas antes desta comparação que tenta eleger a melhor plataforma, é feito um apanhado geral das nomenclaturas e dos principais componentes vistos aqui, correlacionando-os entre as plataformas.

5.1. Nomenclaturas e Componentes

5.1.1. Ambientes de execução e bibliotecas base

Apesar de muitos autores utilizarem o termo JVM como um ambiente de execução implementado, ele não é: pode-se dizer que a JVM é uma mera

especificação, sendo assim, análoga à Common Language Infrastructure (CLI), ou seja, o padrão ECMA-335 que especifica a CLR. A CLR, portanto, não é análoga à JVM e sim à suas implementações, como a HotSpot e a JRockit.

Em se tratando de bibliotecas base, a BCL é análoga à API padrão Java. O conjunto API padrão e implementação da JVM resulta no JRE. Um análogo ao JRE seria o produto chamado Microsoft .NET ou Framework .NET que inclui apenas a CLR e a BCL (ver nota no tópico 4.1), distribuído gratuitamente para dar suporte à execução de programas desenvolvidos em .NET.

5.1.2. Linguagens

Durante o estudo, definiu-se que a linguagem C# não faz parte da plataforma .NET entretanto, a linguagem Java faz parte da plataforma Java. Esta separação de linguagem e plataforma no caso do .NET é feita meramente porque a “outra” implementação da CLI, o Mono, não é desenvolvida pela Microsoft, não é e nem faz parte da plataforma .NET mas suporta a linguagem C#. Além disso, por ser uma plataforma “multi-linguagem”, discute-se que, se o C# faz parte da plataforma .NET, também fazem o Visual Basic, o CLI/C++, o IronPython, o IronRuby, etc. Por outro lado, qualquer implementação da JVM é dita parte da plataforma Java e todas suportam a linguagem Java. Tecnologicamente falando, não há sentido em fazer essa separação de linguagem de programação e plataforma, mas teoricamente falando, ela pode ser feita.

Há entretanto, a linguagem de programação que não só faz parte da plataforma .NET como faz parte da CLR: a MSIL. A MSIL é uma implementação da CIL (Common Intermediate Language), que é especificada

dentro da CLI. Apesar da linguagem Java fazer parte da plataforma Java, de forma alguma ela faz parte da JVM; de forma análoga, entretanto, o Java bytecode é uma linguagem de programação que faz parte tanto da plataforma Java como da JVM (apesar de não ser promovida como uma linguagem "utilizável", diferente da MSIL).

5.1.3. Plataformas

Em linhas gerais, a plataforma de desenvolvimento Java consiste no JDK. De maneira análoga, a plataforma de desenvolvimento .NET é o .NET Framework em sua forma completa (figura 1), em conjunto com os compiladores para C# e VB. Nota-se que diferente da Sun, a Microsoft não faz distinção na nomenclatura de seus produtos para execução (JRE em Java) e desenvolvimento (JDK em Java). Nenhuma das plataformas inclui uma IDE, apesar de que o Visual Studio é considerado, de maneira extra-oficial, parte da plataforma .NET.

5.2. Portabilidade de Código

5.2.1 Em .NET

Levando em consideração que não existe um grande número de implementações da CLR, não há muito o que ser falado sobre a portabilidade de código dentro da plataforma .NET. A BCL é parcialmente padronizada; a Partição IV da CLI (Anexo II.2) define as bibliotecas padrão para sua implementação, entretanto, há muito mais na BCL do que apenas as bibliotecas padrão.

O mesmo pode ser dito sobre o Mono, já que este é uma implementação oficial da CLI, deve possuir o mesmo conjunto de bibliotecas padrão que a BCL/CLR, além de suas próprias implementações para as bibliotecas não padronizadas, como, por exemplo, o GTK#, que fornece suporte à construção de interfaces gráficas.

É dito, portanto, que se é escrito um programa que utiliza apenas as bibliotecas definidas na especificação da CLI, este deve rodar perfeitamente tanto na CLR quanto na plataforma Mono. O uso de qualquer outra biblioteca não padronizada pode – e muito possivelmente irá – gerar erros. Considerando a quantidade e a qualidade das bibliotecas não padronizadas no Framework .NET e sendo que este é altamente voltado à produtividade, deixar de usá-las em detrimento à portabilidade faz com que a plataforma seja fraca neste quesito.

5.2.2. Em Java

A linguagem Java (e sua plataforma como um todo) foi criada com o seguinte lema: "*write once, run anywhere*" (escreva uma vez, execute em qualquer lugar). Por questões históricas, o Java surgiu em um ambiente onde portabilidade deixou de ser uma característica meramente desejada para se tornar algo indispensável. E esta era a proposta da plataforma, de suprir a demanda este nicho que vinha crescendo com o advento da internet e da world wide web e exigindo portabilidade na criação de software.

A JVM nasceu como um ambiente de execução para interpretação de código independente de plataforma, o Java bytecode. Ao longo dos anos, a interpretação está perdendo espaço para a compilação JIT, algo que pode ter

um impacto na portabilidade do bytecode, mas não do código fonte escrito em Java.

Em linhas gerais, costuma-se considerar que contanto que a máquina virtual e a API padrão sejam portadas para uma arquitetura de hardware em específico, um código escrito em Java pode rodar nesta arquitetura, no máximo tendo que ser recompilado. Se não for feito uso de bibliotecas externas, esta afirmação pode ser dita verdadeira. Para termos de comparação, a API padrão do Java é muito mais abrangente que a parte padronizada da BCL do .NET, permitindo muito mais flexibilidade no desenvolvimento de software sem o auxílio de bibliotecas não-padronizadas.

5.2.3. Conclusão

Devido à própria natureza da plataforma Java, sua API padrão é muito melhor adaptada à portabilidade do que a BCL do .NET, e apesar da dependência de re-implementações da JVM e da própria API para permitir que a portabilidade de código seja possível, existem muitas destas para muitas arquiteturas diferentes, fazendo assim que a plataforma Java seja superior à plataforma .NET no quesito de portabilidade de código

5.3. Interoperabilidade

5.3.1. Em .NET

Com o advento do Framework .NET, a implementação de componentes interoperáveis tornou-se muito mais fácil devido à Common Language Specification. A grande limitação da interoperabilidade em .NET é que ela só

pode ser atingida de forma natural em componentes de software construídos sob as regras da CLS, entretanto, através de seu crescente suporte multi-linguagem, implementações de linguagens de programação que respeitam a CLS podem ser encontradas com cada vez mais freqüência.

De fato, o próprio uso de um componente escrito em uma linguagem em um projeto em outra linguagem se dá de forma muito simples: contanto que o componente respeite a CLS, basta importar seu assembly .NET para o projeto desejado através do Visual Studio e seus membros públicos tornam-se automaticamente acessíveis.

Além da CLS, há a tecnologia COM Interop, inclusa na CLR. Apesar da Microsoft ter a intenção de tornar a programação de objetos COM obsoleta com o próprio .NET, ela fornece esta tecnologia para prover acesso a componentes COM existentes sem a necessidade de modificá-los. Ela funciona ao tentar transformar tipos .NET em tipos COM equivalentes. Além disso, o COM Interop permite desenvolvedores COM acessarem objetos gerenciados com a mesma facilidade que acessariam objetos COM.

A aplicação do conceito de interoperabilidade é bastante complexa, a menos que o desenvolvedor esteja tentando usar um módulo escrito em C em seu programa escrito em C++, sempre haverá problemas. Apesar de limitada, as capacidades nativas de interoperabilidade do .NET podem simplificar bastante sua aplicação tanto agora quanto no futuro, o que torna o .NET uma boa plataforma neste quesito.

5.3.2. Em Java

Apesar do crescente número de linguagens que têm como alvo a JVM, sua interoperabilidade com Java não é de tão fácil uso como era de se esperar. Até o uso de versões de linguagens populares na JVM como JRuby e Jython necessitam de frameworks externos desenvolvidos por terceiros para tornar a interoperabilidade possível (JSR 233 ou Apache Bean Scripting Framework para JRuby ou Clamp para converter classes Python em classes Java no caso do Jython).

Interoperabilidade com linguagens fora da JVM (principalmente C/C++) também é possível através do uso do JNI para a chamada de métodos nativos, entretanto o estudo revelou não ser uma técnica prática ou eficiente. Ferramentas como Jace (para integração código C++ à JVM através de JNI) ou xFunction (permite interoperabilidade com módulos C/C++ e a API nativa do Windows contornando o uso do JNI) podem facilitar o desenvolvimento. Também existem ferramentas de terceiros para integração com objetos COM contornando o uso de JNI, como j-Interop ou com4j. No mundo de ferramentas para interoperabilidade, pode-se até encontrar aquelas que façam módulos Java se comunicar com módulos .NET, como a JNBridge, mas ainda assim, não passam de ferramentas de terceiros, com sua gama de problemas de compatibilidade e aumento de overhead.

5.3.3. Conclusão

Apesar de um suporte razoável para multi-linguagem, a JVM não tem as capacidades nativas de interoperabilidade da CLR, sendo que um grande número de ferramentas de terceiros surgem para suprir esta deficiência.

Mesmo que a interoperabilidade em .NET ainda seja um tanto limitada, a plataforma possui ferramentas nativas que podem torná-la praticamente transparente para o desenvolvedor, provando-se superior à Java neste critério.

5.4. Desempenho

O teste de desempenho através dos algoritmos mencionados foi feito no seguinte hardware:

Processador	Intel Pentium Dual CPU 2.16GHz
Memória	2GB
Sistema Operacional	Windows 7 Professional

5.4.1. Algoritmo BubbleSort

Para este teste, as três cargas são de 50000, 100000 e 250000 (ou seja, a quantidade de vezes que o método vai ser chamado). O principal objetivo deste algoritmo é testar a manipulação de arrays. Segue abaixo o gráfico com os resultados referentes ao tempo de processamento e consumo de CPU:

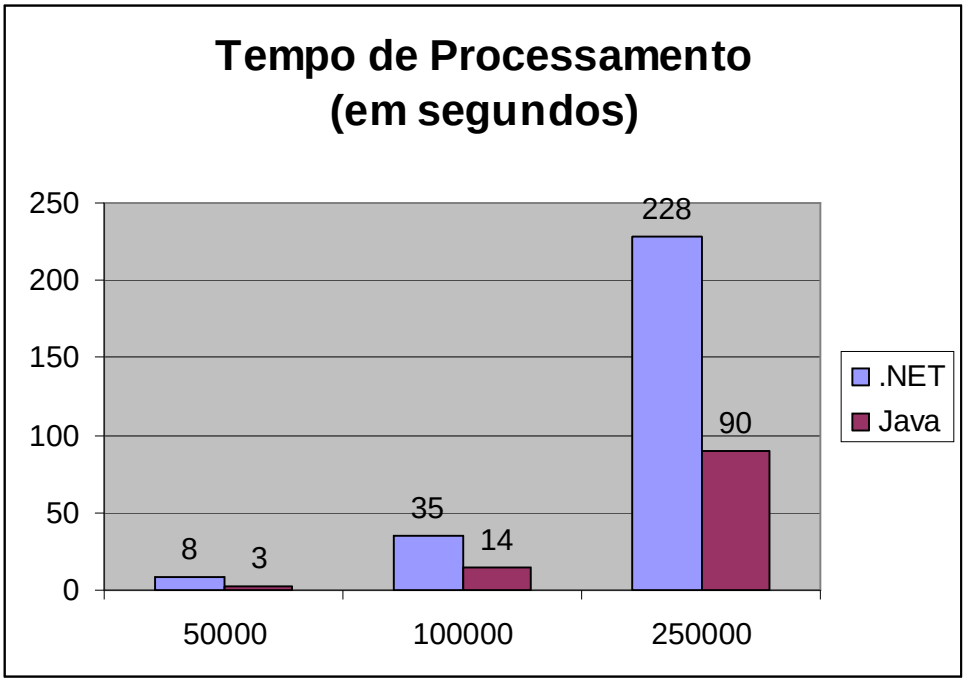

Gráfico 1: Tempo de processamento do BubbleSort

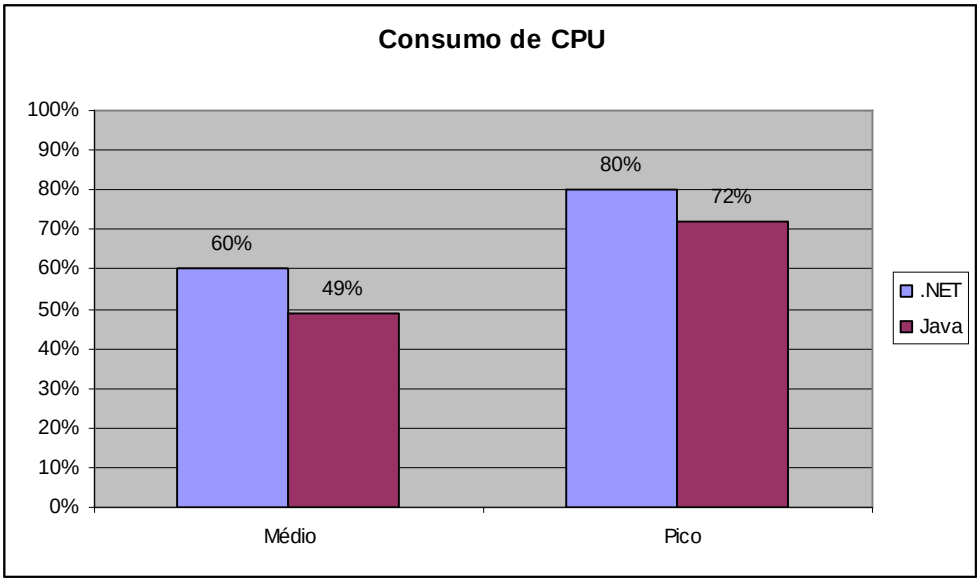

Gráfico 2: Consumo de CPU do BubbleSort (Carga máxima)

Como pode ser constatado, a HotSpot provou-se muito mais eficiente que a CLR no processamento do algoritmo BubbleSort, sendo que na carga

máxima, a CLR levou quase 150% de tempo a mais que a HotSpot para executar a ordenação.

5.4.2. Algoritmo InsertSort

Da mesma forma que foi feito com o BubbleSort, as mesmas três cargas serão utilizadas no InsertSort, de 50000, 100000 e 250000. Também tem como objetivo testar a manipulação de arrays.

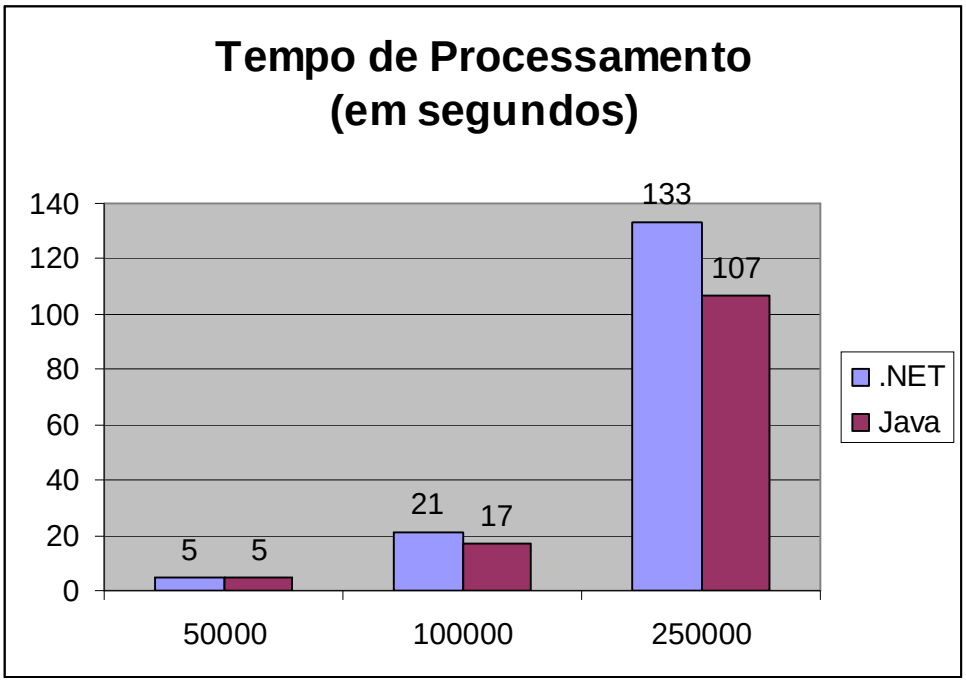

Gráfico 3: Tempo de processamento do InsertSort

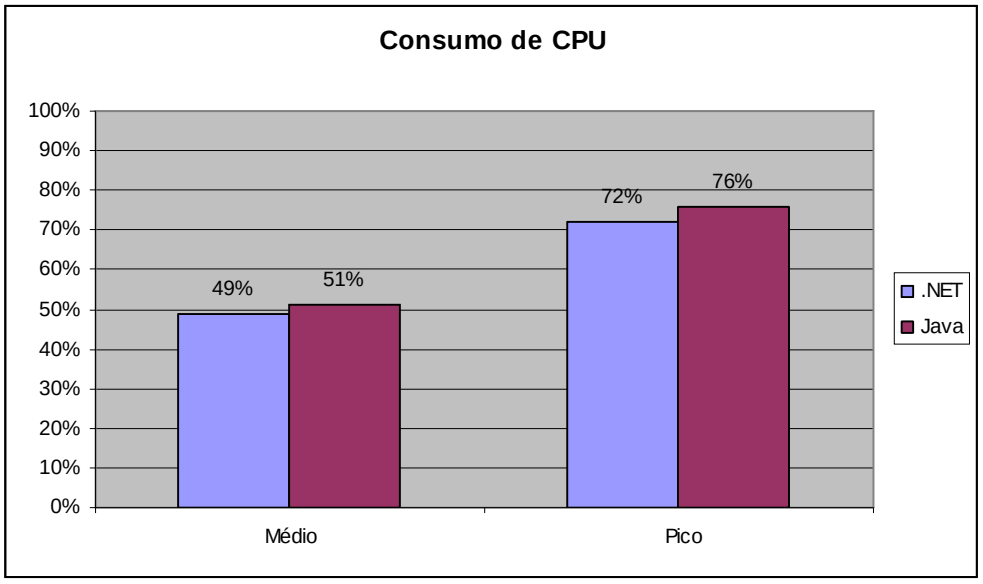

Gráfico 4: Consumo de CPU do InsertSort (Carga máxima)

Diferente do BubbleSort, onde a HotSpot teve uma grande vantagem sobre a CLR, no InsertSort a diferença no tempo de processamento não foi grande (mas a HotSpot ainda foi mais rápida) e a diferença no consumo de CPU foi mínima, com uma leve vantagem para a CLR.

5.4.3. Geração de Números Primos

Para geração de números primos, será utilizada tanto a matemática de números inteiros quanto a matemática de ponto flutuante a fim de ver como cada linguagem lida com os diferentes tipos primitivos. As cargas aqui se dão na geração dos primeiros 500000 números primos, em seguida do primeiro 1000000 e finalmente, 2500000. O objetivo deste teste é verificar como cada um dos ambientes de execução lida com elementos matemáticos.

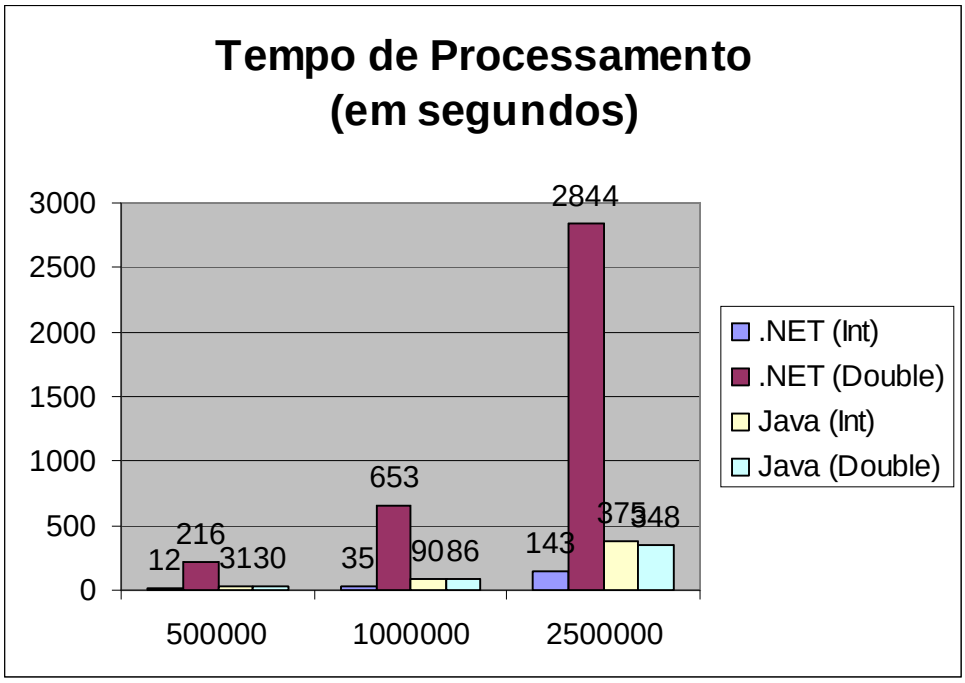

Gráfico 5: Tempo de processamento da geração de números primos

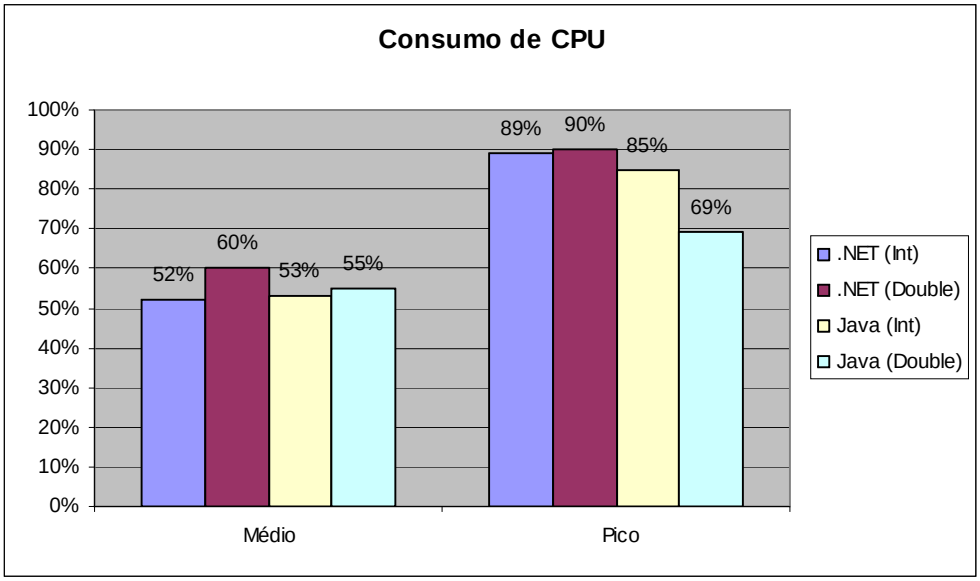

Gráfico 6: Consumo de CPU na geração de números primos

Este algoritmo mostrou o resultado mais curioso até agora: na manipulação de números inteiros, a CLR se mostrou ser mais de duas vezes mais rápida que a JVM. Entretanto, na manipulação de números em ponto flutuante, a CLR foi oito vezes mais lenta que a JVM e vinte vezes mais lenta

que ela mesma na manipulação de números inteiros, levando mais de 47 minutos para rodar o algoritmo na carga máxima em ponto flutuante. A JVM se mostrou constante na manipulação de números inteiros e de ponto flutuante.

5.4.4. Algoritmo Thread Ring

Como explicado anteriormente, o Thread Ring trata-se de um algoritmo específico para benchmark, que consiste na criação de 503 threads e na passagem de o token por cada uma das threads uma vez. Aqui se utilizará uma carga única, de 50000000, significando que o token passará 50000000 vezes em cada thread. Este teste foi feito com o auxílio do ANTS Profiler 5.1 (.NET) e do Netbeans Profiler, o que pode causar um pouco de overhead, mas não o suficiente para invalidar o teste.

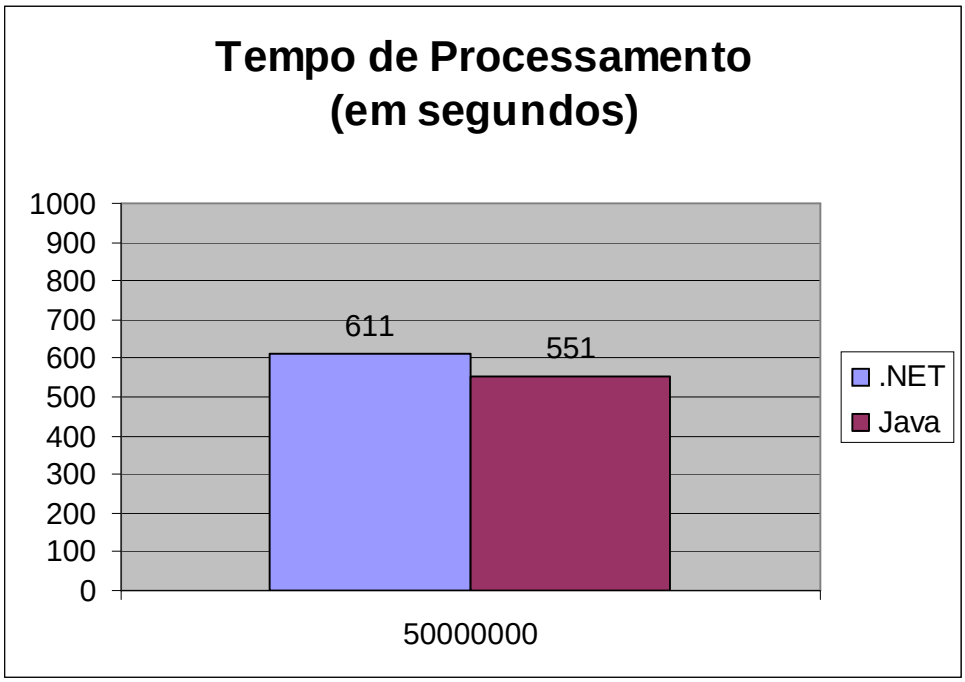

Gráfico 7: Tempo de processamento do algoritmo thread ring

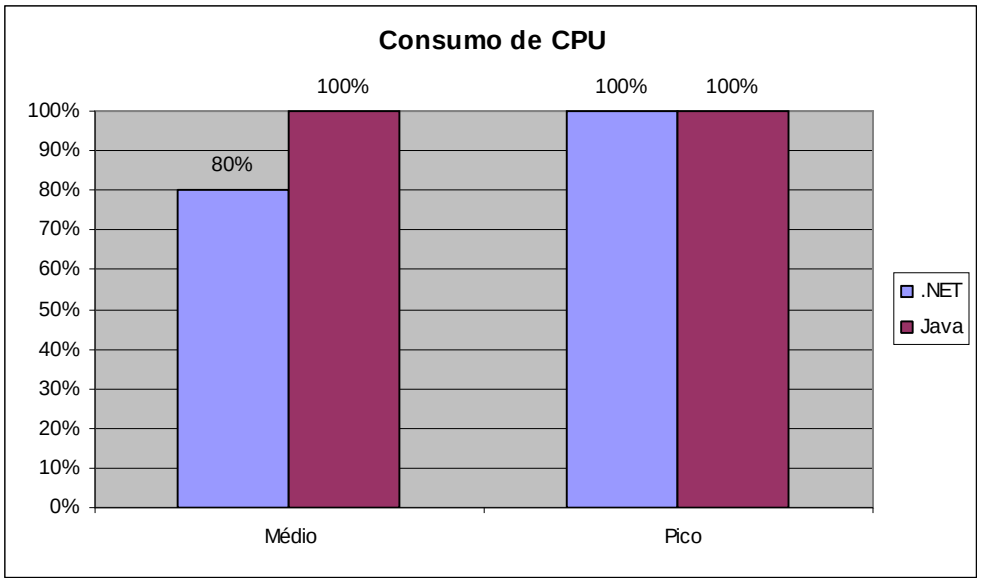

Gráfico 8: Consumo de CPU do algoritmo thread ring

O algoritmo Thread Ring serve de benchmark para criação e sincronização de threads. Como pode ser observado, não há muita diferença neste quesito entre a JVM e a CLR; apesar de ter levado um minuto a menos para executar na JVM, seu consumo de CPU ficou o tempo todo em 100%, ao passo que a média na CLR foi de 80%.

5.4.5. Conclusão do Teste de Desempenho

Em todos os algoritmos testados, percebeu-se a tendência de uma melhor performance da JVM. A grande surpresa foi o teste da geração de números primos, que mostrou que a matemática de números inteiros é bem superior na CLR e sua matemática de ponto flutuante é muito inferior. Outro resultado curioso foi o consumo de CPU da JVM no algoritmo Thread Ring: constantemente 100%. Apesar disso, com base nos testes realizados, conclui-se que a plataforma tem um melhor desempenho geral que a plataforma .NET.

5.5. As Comunidades Java e .NET

Apesar de não fazer parte dos critérios de comparação, não poderia se deixar de mencionar este assunto. Assim como toda plataforma e linguagem de programação, Java e .NET também possuem suas comunidades, grupos de organizações e desenvolvedores independentes espalhados por toda a internet com o objetivo de auxiliar outros desenvolvedores na utilização das tecnologias.

A grande diferença entre as comunidades Java e .NET reside no fato que bibliotecas, APIs, Frameworks e outras ferramentas desenvolvidas pela comunidade Java podem de fato vir a serem incorporadas em versões oficiais da plataforma, ao passo que todo o desenvolvimento de melhorias voltadas à plataforma .NET necessariamente partem de dentro da própria Microsoft.

6.CONCLUSÃO

Dentro do escopo deste trabalho, a conclusão é indiscutível: Java foi superior ao .NET. Entretanto, ao se analisar de uma maneira geral, o escopo aqui é um tanto reduzido, e não há como afirmar, assertivamente, que Java é uma plataforma superior em todos os casos. Existem várias outras características para análise as quais só poderiam ser trabalhadas de forma aprofundada através de trabalhos específicos sobre cada plataforma.

Ao se analisar uma plataforma de desenvolvimento, deve-se levar em consideração o domínio do problema. E mesmo que ambas as plataformas sejam comercializadas para um mesmo domínio de aplicação, elas tem suas vantagens e desvantagens entre si caracterizadas de forma tão distintas que não é possível dizer qual das duas é universalmente capaz de suprir toda e qualquer necessidade que um cliente venha a ter; é, contudo, possível dizer que nenhuma das duas o fará de forma completa e a escolha entre elas sempre dependerá dos fatores individuais de cada negócio.

6.1. Trabalhos Futuros

Apesar de ter ido a fundo, no núcleo de cada tecnologia, ainda há muito o que se explorar no assunto. Dois itens principais que foram descartados logo de início seriam um ótimo assunto para dar continuidade ao trabalho: comparação entre desenvolvimento web e entre desenvolvimento de aplicações “Enterprise” nas duas plataformas.

Outro item se trata de uma comparação aprofundada entre as linguagens Java e C#, sendo que o grande enfoque dado para plataformas no trabalho atual acabou fazendo com que este quesito ficasse de lado.

REFERÊNCIAS BIBLIOGRÁFICAS

A Benchmark Test for BCPL Style Coroutines. Disponível em <http://www.cl.cam.ac.uk/~mr10/Cobench.html>. Acesso em: Out. 2010.

A fast-track standards process. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=539730>. Acesso em: Ago. 2010.

C++/CLI. Disponível em: <http://en.wikipedia.org/wiki/C%2B%2B/CLI>. Acesso em: Ago. 2010

CHERRYSTONE SOFTWARE LABS. Algorithmic Performance Comparison Between C, C++, Java and C# Programming Languages, Massachusetts, 19 p. Ago. 2010.

ECMA International. Disponível em: <http://en.wikipedia.org/wiki/Ecma_International>. Acesso em: Ago. 2010.

ECMA International. Disponível em: <http://www.Ecma-international.org/>. Acesso em: Ago. 2010.

ECMA International: Standard ECMA-334 C# Language Specification, 4ª Edição, Junho de 2006, 553 p.

ECMA International: Standard ECMA-335 Common Language Infrastructure, 4ª Edição, Junho de 2006, 556 p.

ECMA International: Standard ECMA-372 C++/CLI Language Specification, 1ª Edição, Dezembro de 2005, 304 p.

Java SE Documentation. Disponível em: <http://download.oracle.com/javase/6/docs/>. Acesso em: Out. 2010.

Java SE Documentation. Disponível em <http://download.oracle.com/javase/6/docs/technotes/guides/index.html>. Acesso em: Out. 2010

KOTZMANN, Thomas, et al. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, New York, v. 5, n. 1, p. 1-13, mai. 2008.

MACDONALD, Matthew; SZPUSZTA, Mario. Pro ASP.NET 3.5 in C# 2008. 3ª Edição. Berkeley: Apress, 2009. cap 1, p. 4-14; cap 7, p. 257, 260; cap18, p. 741.

MAYO, Joe. C# 3.0 with the .NET Framework 3.5 Unleashed. 2ª Edição. Sams, 2008. 1022 p.

MICHAELIS, M. Microsoft .NET Development Series: Essential C# 3.0 For .NET Framework 3.5. Addison-Wesley, 2009. p. 1.

ORACLE. Oracle JRockit JVM Diagnosis Guide, R27.6. Redwood City, 2009. cap 2.

PALECZNY, M.; VICK, C.; CLICK, C. The Java HotSpot™ Server Compiler. *Proceedings of the Java Virtual Machine Research and Technology Symposium*, Monterey, p. 2, abr. 2001.

Performance Measurements of Threads in Java and Processes in Erlang. Disponível em <http://www.sics.se/~joe/ericsson/du98024.html>. Acesso em: Out. 2010.

RITCHER, Jeffrey. CLR via C#. 3ª Edição. Redmond: Microsoft Press, 2010. 861 p.

SKEET, Jon. C# in Depth. Greenwich: Manning Publications, 2008. cap 1, p. 3-31.

Standard ECMA-334 C# Language Specification. Disponível em: <http://www.Ecma-international.org/publications/standards/Ecma-334.htm>. Acesso em: Ago. 2010.

Standard ECMA-335 Common Language Infrastructure. Disponível em: <http://www.Ecma-international.org/publications/standards/Ecma-335.htm>. Acesso em: Ago. 2010.

SUN. Sun Developer Network. Disponível em <http://www.java.sun.com>. Acesso em: Nov. 2009.

TROELSEN, A. Pro C# 2008 and the .NET Platform. 4ª Edição. Berkeley: Apress, 2007. 1370 p.

WATT, David A.; BROWN, Deryck F. Programming Language Processors in Java: Compilers and Interpreters. Essex: Pearson Education, 2000. cap 8, p. 305.

Anexo I – Código Fonte

I.1. Bubble Sort

I.1.1. C#

```
using System;
using System.Collections.Generic;
using System.Text;

namespace BubbleSort
{
    class arraySort
    {
        private int[] a = new int[100];
        private int x;

        public void sortArray()
        {
            int i;
            int j;
            int temp;
            DateTime start;

            start = DateTime.Now;

            for (i = 0; i < x; i++)
            {
                for (j = 1; j < x; j++)
                {
                    if (a[j - 1] > a[j])
                    {
                        temp = a[j - 1];
                        a[j - 1] = a[j];
                        a[j] = temp;
                    }
                }
            }

            TimeSpan elapsed = DateTime.Now - start;

            String s;

            s = "Tempo Decorrido (C#): " + elapsed;
            Console.WriteLine(s);
            Console.ReadKey();

        }

        public static void Main(string[] args)
        {
            arraySort mySort = new arraySort();

            mySort.x = Int32.Parse(args[0]);
            mySort.a = new int[mySort.x];
```

```
            for (int i = 0; i < mySort.x; i++)
            {
                mySort.a[i] = mySort.x - i;
            }

            mySort.sortArray();
        }
    }
}
```

I.1.2. Java

```
package BS;
import java.util.*;

public class bubbleSort {
      public static void main(String[] a)
      {
      String      s;
      Date  d;
      int   i, n=10;
      int[]       array;
      long  l, start, end;


            if(a[0].equals("-n"))
                  n = Integer.parseInt(a[1]);

            array = new int[n];

            //
            // Configuração para o pior caso, onde todos os valores
devem ser trocados
            //

            for(i=0; i<n; i++)
                  array[i] = n - i;

            // "Aquece" o algoritmo (permitir a identificação do método
como "hot spot"
            BubbleSort(array, n/10);
            BubbleSort(array, n/10);

            // Desordena novamente
            for(i=0; i<n; i++)
                  array[i] = n - i;

            d = new Date();
            start = d.getTime();

            //Inicia o teste
            BubbleSort(array, array.length);

            d = new Date();
            end = d.getTime();

            s = "Tempo Decorrido (Java): ";
```

```
            s += (end - start) / 1000;
            System.out.println(s);
      }


      public static void BubbleSort( int a[], int n )
      {
      int i, j,t=0;

            for(i = 0; i < n; i++)
            {
                  for(j = 1; j < (n-i); j++)
                  {
                        if(a[j-1] > a[j])
                        {
                              t = a[j-1];
                              a[j-1] = a[j];
                              a[j] = t;
                        }
                  }
            }
      }
}
```

I.2. InsertSort

I.2.1. C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace sortBubble
{
  class CSLIShell
  {
    private int[] a = new int[100];

    public void insertionsort(int []data, int n)
    {
      int   i, j, value, done=0;


            for(i=1; i<n; i++)
            {
                  value = data[i];
                  j = i - 1;
                  done = 0;

                  do
                  {
                        if(data[j] > value)
                        {
```

```
                                data[j + 1] = data[j];
                                j--;

                                if(j < 0)
                                      done = 1;
                          }
                          else
                                done = 1;
                    }
                    while(done == 0);

                    data[j + 1] = value;
              }
        }


        public static void Main(String[] args)
        {
            CSLIShell cs = new CSLIShell();
            int []dataarray;
            int nvalues=10000, i, n=1;
            DateTime start;
            TimeSpan elapsed;
            String s;

          nvalues = Int32.Parse(args[0]);


              dataarray = new int[nvalues];

              start = DateTime.Now;

              for (i = 0; i<n; i++)
              {
              int   j;

                    for (j = 0; j < nvalues; j++)
                          dataarray[j] = nvalues - j;

                    cs.insertionsort(dataarray, nvalues);
              }

              elapsed = DateTime.Now - start;
              s = "Tempo Decorrido: " + elapsed;
              Console.WriteLine(s);
        }
    }
}
```

I.2.2. Java

```
import java.util.*;


public class insertion
```

```
{
	public static void main(String[] args)
	{
		String	s;
		Date	d;
		int	nargs=1, i, n=1, nvalues=10000;
		int[]	array;
		long	start, end;

		nvalues = Integer.parseInt(args[0]);

		array = new int[nvalues];
		d = new Date();
		start = d.getTime() / 1000;

		for(i=0; i<n; i++)
		{
		int	j;

			for(j=0; j<nvalues; j++)
				array[j] = nvalues - j;

			insertionsort(array, nvalues);
		}

		d = new Date();
		end = d.getTime() / 1000;

		s = "Tempo Decorrido: ";
		s += end - start;
		System.out.println(s);
	}


	public static void insertionsort(int data[], int n)
	{
		int	i, j, value, done=0;

		for(i=1; i<n; i++)
		{
			value = data[i];
			j = i - 1;
			done = 0;

			do
			{
				if(data[j] > value)
				{
					data[j + 1] = data[j];
					j--;

					if(j < 0)
						done = 1;
				}
				else
					done = 1;
			}
			while(done == 0);

			data[j + 1] = value;
		}
```

```
      }
}
```

I.3. Geração de Números Primos

I.3.1. C# (Integer)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class PrimesCS
{
      public  long[] computeprimes(int n)
    {

          long    divisor;
          double  squareroot;
          long    i;
          int     nprimes=0;
          long[]  primes;

            primes = new long[n];

            primes[nprimes++] = 2;

            for(i=3; nprimes<n; i+=2)
            {
            int   prime;

                  squareroot = Math.Sqrt(i);
                  divisor = 3;

                  prime = 1;

                  while(prime > 0 && divisor <= squareroot)
                  {
                        if(i % divisor == 0)
                              prime = 0;

                        divisor += 2;
                  }

                  if(prime != 0)
                        primes[nprimes++] = (long) i;
            }

            return primes;
      }


      public static void Main(String[] args)
      {
```

```
            String        debug="";
            DateTime      start, end;
            TimeSpan      elapsed;
            int           n=20;
            long[]        primes;

            n = Int32.Parse(args[0]);

          PrimesCS prime = new PrimesCS();

              start = DateTime.Now;

              primes = prime.computeprimes(n);

              end = DateTime.Now;

              elapsed = end - start;

              debug = "Tempo decorrido: ";
              debug += elapsed;
              Console.WriteLine(debug);

      }
}
```

I.3.2. Java (Integer)

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.lang.Math;
import java.util.*;

public class Primes
{
      public  long[] computeprimes(int n)
      {
            long  divisor, result, squareroot;
            long  i, half, tmp;
            int   nprimes=0;
            long[]      primes;

            primes = new long[n];

            primes[nprimes++] = 2;

            for(i=3; nprimes<n; i+=2)
            {
            int   prime;

                  squareroot = (long) Math.sqrt(i);
                  divisor = 3;

                  prime = 1;

                  while(prime > 0 && divisor <= squareroot)
                  {
                        if(i % divisor == 0)
                              prime = 0;
```

```
                        divisor += 2.00;
                  }

                  if(prime != 0)
                        primes[nprimes++] = (long) i;
            }

            return primes;
      }

      public static void main(String[] args)
      {
            String      debug, list;
            Date  d;
            long  start, end, elapsed;
            int       n=10;
            long[]      primes;

            n = Integer.parseInt(args[0]);

        Primes prime = new Primes();
            d = new Date();

            start = d.getTime();

            primes = prime.computeprimes(n);

            d = new Date();
            end = d.getTime();

            elapsed = (end - start) / 1000;

            debug = "Tempo decorrido: ";
            debug += elapsed;
            System.out.println(debug);
      }
}
```

I.3.3. C# (Double)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class PrimesCS
{
      public  long[]    computeprimes(int n)
      {
          double  divisor, result, squareroot;
          long    i, half, tmp;
          int     nprimes=0;
          long[]  primes;

            primes = new long[n];
```

```
            primes[nprimes++] = 2;

            for(i=3; nprimes<n; i+=2)
            {
            int   prime;

                  squareroot = Math.Sqrt(i);
                  divisor = 3;

                  prime = 1;

                  while(divisor <= squareroot)
                  {
                        result = i / divisor;
                        tmp = (long) result;
                        result -= tmp;

                        if(result == 0.00)
                              prime = 0;

                        divisor += 2.00;
                  }

                  if(prime != 0)
                        primes[nprimes++] = (long) i;
            }

            return primes;
      }

      public static void Main(String[] args)
      {
          String        debug="";
          DateTime      start, end;
          TimeSpan      elapsed;
          int               n=20;
          long[]        primes;

        n = Int32.Parse(args[0]);

        PrimesCS prime = new PrimesCS();

            start = DateTime.Now;

            primes = prime.computeprimes(n);

            end = DateTime.Now;

            elapsed = end - start;

            debug = "Tempo Decorrido: ";
            debug += elapsed;
            Console.WriteLine(debug);

      }
}
```

I.3.4. Java (Double)

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.lang.Math;
import java.util.*;

public class Primes
{
	public  long[]    computeprimes(int n)
	{
		double      divisor, result, squareroot;
		long  i, half, tmp;
		int   nprimes=0;
		long[]      primes;

		primes = new long[n];

		primes[nprimes++] = 2;

		for(i=3; nprimes<n; i+=2)
		{
		int   prime;

			squareroot = Math.sqrt(i);
			divisor = 3;
			prime = 1;

			while(prime > 0 && divisor <= squareroot)
			{
				result = i / divisor;
				tmp = (long) result;
				result -= tmp;

				if(result == 0.00)
					prime = 0;

				divisor += 2.00;
			}

			if(prime != 0)
				primes[nprimes++] = (long) i;
		}

		return primes;
	}

	public static void main(String[] args)
	{
		String      debug, list;
		Date  d;
		long  start, end, elapsed;
		int       n=10;
		long[]      primes;

		n = Integer.parseInt(args[0]);

	  Primes prime = new Primes();
		d = new Date();

		start = d.getTime();
```

```
            primes = prime.computeprimes(n);

            d = new Date();
            end = d.getTime();

            elapsed = (end - start) / 1000;

            debug = "Tempo Decorrido: ";
            debug += elapsed;
            System.out.println(debug);
      }
}
```

I.4. Algoritmo Thread-Ring

I.4.1. C#

O código apresentado abaixo não é otimizado para a versão 3.5 da plataforma .NET, mas trata-se de uma implementação do Thread-Ring desenvolvida por Isaac Gouy para rodar em Mono (a implementação da CLS para Linux), que possui capacidades equivalentes ao .NET 2.0. Não foi possível encontrar uma implementação do algoritmo otimizada para a plataforma .NET 3.5.

```
using System;
using System.Threading;

namespace ThreadRing
{
    internal class NamedThread
    {
        private int name;
        private AutoResetEvent signal = new AutoResetEvent(false);
        private int token = 0;

        internal NamedThread(int name)
        {
            this.name = name;
        }

        internal void Run()
        {
            while (TokenNotDone())
                NextThread().TakeToken(token - 1);

            if (token == 0) Console.WriteLine(name);
            NextThread().TakeToken(-1);
        }
```

```
        private bool TokenNotDone()
        {
            signal.WaitOne();
            return token > 0;
        }

        internal NamedThread NextThread()
        {
          return ThreadRing.threadRing[name % ThreadRing.numberOfThreads];
        }

        internal void TakeToken(int x)
        {
            token = x;
            signal.Set();
        }
    }


    public class ThreadRing
    {
        internal const int numberOfThreads = 503;
        internal static NamedThread[] threadRing = new NamedThread[503];

        public static void Main(string[] args)
        {
            for (int i = 0; i < numberOfThreads; i++)
            {
                threadRing[i] = new NamedThread(i + 1);
            }

            foreach (NamedThread t in threadRing)
                new Thread(new ThreadStart(t.Run)).Start();

            threadRing[0].TakeToken(int.Parse(args[0]));
        }
    }
}
```

Thread-Ring em C#, implementação de Isaac Gouy

I.4.2. Java

A implementação do Thread-Ring em Java por Klaus Friedel é bastante próxima da implementação em C# de Isaac Gouy e gerou resultados semelhantes.

```
package threadring;

import java.util.concurrent.locks.LockSupport;

public class ThreadRing {
  static final int THREAD_COUNT = 503;

  public static class MessageThread extends Thread {
    MessageThread nextThread;
    volatile Integer message;

    public MessageThread(MessageThread nextThread, int name) {
```

```
      super(""+name);
      this.nextThread = nextThread;
    }

    public void run() {
      while(true) nextThread.enqueue(dequeue());
    }

    public void enqueue(Integer hopsRemaining) {
      if(hopsRemaining == 0){
        System.out.println(getName());
        System.exit(0);
      }
      // as only one message populates the ring, it's impossible
      // that queue is not empty
      message = hopsRemaining - 1;
      LockSupport.unpark(this); // work waiting...
    }

    private Integer dequeue(){
      while(message == null){
        LockSupport.park();
      }
      Integer msg = message;
      message = null;
      return msg;
    }
  }

  public static void main(String args[]) throws Exception{
    int hopCount = Integer.parseInt(args[0]);

    MessageThread first = null;
    MessageThread last = null;
    for (int i = THREAD_COUNT; i >= 1 ; i--) {
      first = new MessageThread(first, i);
      if(i == THREAD_COUNT) last = first;
    }
    // close the ring:
    last.nextThread = first;

    // start all Threads
    MessageThread t = first;
    do{
      t.start();
      t = t.nextThread;
    }while(t != first);
    // inject message
    first.enqueue(hopCount);
    first.join(); // wait for System.exit
  }
}
```

Thread-Ring em Java, implementação de Klaus Friedel

Anexo II – Padrões ECMA

A ECMA International é uma organização para padronização existente desde 1961 e sediada em Genebra. É focada no desenvolvimento de padrões e relatórios técnicos a fim de facilitar e padronizar o uso de tecnologias de comunicação da informação e eletrodomésticos. É responsável pelo desenvolvimento de um procedimento de consolidação de padrões conhecido como fast-track o qual permite que um padrão seja adotado em um período de 6 meses - de acordo com a IEEE, geralmente se leva de 2 a 10 anos para adotar um padrão sem o uso deste procedimento. Desde 1987, a ISO/IEC utilizou este procedimento para a adoção de 464 padrões: 39% destes foram analisados pela ECMA.

Entre alguns dos padrões ECMA mais conhecidos, estão:

- ECMA-119: Estrutura de arquivos e volume do CD-ROM (ISO 9660);
- ECMA-262: ECMAScript - padronização de especificação de linguagem de scripting, amplamente usada em clientes na web na forma de dialetos como JavaScript, JScript e ActionScript (ISO/IEC 16262);
- ECMA-334: Especificação da linguagem C# (ISO/IEC 23270);
- ECMA-335: Common Language Infrastructure ou CLI (ISO/IEC 23271);
- ECMA-372: Especificação da linguagem C++/CLI (sucessor do Managed C++).

Neste trabalho viu-se a necessidade de ressaltar a importância da ECMA International e dos padrões ECMA-334 e ECMA-335, pois nenhuma discussão séria sobre .NET pode ignorar que a Common Language Runtime (o ambiente de execução da plataforma) é a implementação da Microsoft da

especificação ECMA-335 (conhecida como CLI – outra implementação famosa da CLI é o Mono tendo o Linux como plataforma alvo) e o C# é sua implementação do ECMA-334. Também é descrito o padrão ECMA-372, por se tratar da especificação do C++/CLI, que também tem relevância dentro do Framework .NET

II.1. ECMA-334: Especificação do Padrão C#

Padrão internacional baseado em documento enviado pela Hewlett-Packard, Intel e Microsoft, descrevendo a linguagem C#, desenvolvida internamente pela Microsoft. Os principais inventores desta linguagem foram Anders Hejlsberg, Scott Wiltamuth e Peter Golde. A primeira implementação amplamente distribuída do C# foi lançada pela Microsoft em julho de 2000 como parte da iniciativa de seu Framework .NET.

O Comitê Técnico Ecma 39 Grupo Tarefa 2 (Ecma Technical Commitee 39 Task Group 2 ou TC39-TG2) foi formado em setembro de 2000 a fim de produzir um padrão para o C#. Outro Grupo Tarefa, TG3, foi formado simultaneamente com o intuito de produzir um padrão para a biblioteca e ambiente de execução chamado Common Language Infrastructure (CLI). (A CLI é baseada em um subconjunto do Framework .NET) Apesar da implementação da Microsoft depender da CLI para biblioteca e suporte de tempo de execução, outras implementações do C# não possuem esta necessidade, dado que tenham suporte a uma forma alternativa de representar as características mínimas da CLI requeridas pelo padrão C#.

O padrão especifica a representação dos programas escritos em C#, a sintaxe e restrições da linguagem, as regras semânticas para a interpretação

dos programas em C# e as restrições e limites impostos por uma implementação ideal da linguagem C#. O padrão, entretanto, não especifica o mecanismo pelo qual programas escritos em C# devem ser transformados para seu uso por sistemas de processamento de dados, mecanismos pelos quais aplicações C# são invocadas para seu uso pelos mesmos sistemas; o mecanismo pelo qual dados de entrada são transformados para seu uso em uma aplicação C# bem como mecanismos que pelos quais dados de saída devem ser transformados após serem produzidos pela aplicação; se o tamanho ou a complexidade de um programa e seus dados excederá a capacidade de um sistema de processamento de dados ou processador em particular e, finalmente, não especifica os requisitos mínimos de um sistema de processamento de dados que é capaz de suportar uma implementação ideal.

Com a evolução da definição do C#, os objetivos utilizados em seu design são como segue:

- O C# tem o propósito de ser uma linguagem de programação orientada a objetos simples, moderna e de uso geral.
- A linguagem e suas implementações devem prover suporte para os princípios de engenharia de software tais como tipagem forte, verificação de limites de arrays, detecção de tentativas de uso de variáveis não inicializadas e coleta de lixo (garbage collection) automática. Robustez de software, durabilidade e produtividade para o programador são importantes.
- A linguagem tem o propósito de uso no desenvolvimento de componentes de software apropriados para implantação em ambientes distribuídos.

- Portabilidade do código fonte é muito importante, tal como portabilidade do programador, especialmente para programadores familiarizados com C e C++.
- Suporte para internacionalização é muito importante.
- O C# tem a intenção de ser apropriado para escrever aplicações tanto para sistemas hospedeiros quanto embarcados, com abrangência desde os maiores, que utilizam sistemas operacionais sofisticados até os menores, possuindo funções dedicadas.
- Apesar das aplicações em C# ter o propósito de serem econômicas, levando em consideração requerimentos de memória e poder de processamento, a linguagem não tem o propósito de competir diretamente em termos de desempenho e tamanho com a linguagem C ou linguagens assembly.

(Adaptado de Standard ECMA-334 C# Language Specification)

As seguintes organizações participaram no desenvolvimento do padrão: ActiveState, Borland, CSK Corp., HP, IBM, Intel, IT University of Copenhagen, Jaggersoft (Reino Unido), Microsoft, Mountain View Compiler, Monash University (Austrália), Netscape, Novell, Pixo, Plum Hall, Sun e a University of Canterbury (Nova Zelândia). O desenvolvimento da versão atual do padrão iniciou em janeiro de 2003 e foi adotada em junho de 2006. Este padrão é seguido pelo C# 2.0 entretanto, já existem as versões 3.0 e 4.0. Não foram encontradas informações sobre a atualização do padrão para a adequação das novas versões da linguagem.

II.2. ECMA-335: Common Language Infrastructure (CLI)

Como mencionado anteriormente, este padrão foi produzido pelo TC39-TG3, com inicio em setembro de 2000. Foi desenvolvido em conjunto com o padrão ECMA-334, sendo que a implementação C# da Microsoft do C# é dependente da Common Language Runtime (CLR), sua implementação do padrão ECMA-335 (CLI); sendo assim, a implementação da Microsoft do padrão ECMA-335 serve como camada inferior de sua implementação do ECMA-334, entretanto, como previsto na especificação do padrão ECMA-334, uma implementação C# qualquer não necessariamente requere uma implementação da CLI, contanto que exista uma forma alternativa de representar suas características mínimas.

O padrão internacional define a CLI como uma estrutura na qual aplicações escritas em múltiplas linguagens de alto-nível possam ser executadas em diferentes ambientes operacionais sem a necessidade de reescrever tais aplicações para levar em consideração as caracteristicas únicas destes ambientes. Este padrão consiste em 6 partes, denominadas "partições":

- Partição I: Conceitos e Arquitetura - Descreve a arquitetura geral da CLI e provê a descrição normativa do Common Type System (CTS), o Virtual Execution System (VES)[1] e da Common Language Specification (CLS). Também prove a informação descritiva dos meta-dados.
- Partição II: Definição de meta-dados e semântica - Provê a descrição normativa dos meta-dados, sua distribuição física (como formato de arquivo), seu conteúdo lógico (como um conjunto de tabelas e seus relacionamentos) e sua semântica (vista a partir de um compilador hipotético, o ilasm).

- Partição III: Conjunto de instruções da CIL - Descreve o conjunto de instruções da Common Intermediate Language (CIL).
- Partição IV: Perfis e bibliotecas - Provê uma visão geral das bibliotecas da CLI e uma especificação de sua fatoração em perfis[2] e bibliotecas.
- Partição V: Formato de intercâmbio de depuração.
- Partição VI: Anexos - Contem exemplos de programas escritos em CIL Assembly Language (ILAsm), informação sobre uma implementação específica de um assembler, uma descrição do conjunto de instruções da CIL para leitura de máquina que pode ser usada para derivar partes da gramatica utilizada por este assembler bem como ferramentas para manipular a CIL, um guia de instruções usado no design das bibliotecas da Partição IV e considerações sobre portabilidade.

(Adaptado de Standard ECMA-335 Common Language Infrastructure).

Assim como o padrão ECMA-334, a atual especificação ECMA-335 teve seu desenvolvimento iniciado janeiro de 2003 e foi adotado como padrão em junho de 2006. É fielmente implementada no Framework 2.0 (CLR 2.0), entretanto, as atualizações na CLR em versões posteriores do Framework (3.0, 3.5 e 4.0) não estão adequadas à especificação qualquer até o presente momento.

[1] Em linhas gerais, o propósito do VES é prover o suporte adequado para a execução do conjunto de instruções da CIL. O VES é um componente necessário em uma implementação da CLI, implementando e aplicando o Common Type System (CTS). Ele provê os serviços necessários para executar código em CIL e dados, usando metadados para conectar em tempo de

execução módulos gerados separadamente. Por exemplo, dado um endereço dentro do código para uma função, o VES deve ser capaz de localizar os metadados descrevendo a função, além de fazer buscas na pilha de execução, tratar exceções bem como armazenar e resgatar informações de segurança. (Adaptado de Standard ECMA-372 C++/CLI Language Specification)

[2] Um perfil simplesmente é um conjunto de bibliotecas agrupadas com o intuito de formar um todo que proveja um nível definido de funcionalidade [ECMA-335]. Existem dois tipos de perfil padrão explicitados na Partição IV da especificação: O perfil Kernel, que contem os tipos comumente encontrados nas bibliotecas de classes das linguagens de programação modernas e os tipos necessários para compiladores que tenham a CLI como alvo; e o perfil Compacto, que contém o perfil Kernel juntamente com um novo conjunto de bibliotecas, provendo maior funcionalidade mas ainda assim permitindo implementação em dispositivos com apenas uma quantidade modesta de memória física.

II.3. ECMA-372: C++/CLI

A especificação de linguagem C++/CLI (sucessor do Managed Extensions for C++; ambos popularmente conhecidos como Managed C++, antigamente conhecido como Visual C++) tornou-se um padrão ECMA em dezembro de 2005. É uma extensão do C++ padrão (ISO/IEC 14882:2003), provendo novas palavras reservadas, classes, excessões, namespaces e bibliotecas, bem como coleta de lixo (garbage collection).

Enquanto a especificação informa que o C++/CLI é um um conjunto que abrange o C++ padrão (como de fato era o caso com o Managed C++ original), popularmente é dito que este deve ser visto como uma linguagem de programação independente, devido a adição das novas palavras reservadas e de funcionalidades específicas do .NET - por exemplo, um ponteiro .NET deve ser declarado com a sintaxe NomeDaClasse^, especificando um ponteiro que deve ser destruído pelo Garbage Collector, ao passo que um ponteiro normal C++ (declarado da forma tradicional, ou seja NomeDaClasse*) deve ter seu espaço de memória desalocado manualmente, da mesma forma que no C++ padrão.

Atualmente, a única implementação do padrão ECMA-372 é a da própria Microsoft compatível com o Framework .NET 2.0 em diante.

Anexo III – Lista de Instruções da MSIL

Opcode	Instruction	Description
0x58	`add`	Add two values, returning a new value.
0xD6	`add.ovf`	Add signed integer values with overflow check.
0xD7	`add.ovf.un`	Add unsigned integer values with overflow check.
0x5F	`and`	Bitwise AND of two integral values, returns an integral value.
0xFE 0x00	`arglist`	Return argument list handle for the current method.
0x3B	`beq` <int32 (target)>	Branch to target if equal.
0x2E	`beq.s` <int8 (target)>	Branch to target if equal, short form.
0x3C	`bge` <int32 (target)>	Branch to target if greater than or equal to.
0x2F	`bge.s` <int8 (target)>	Branch to target if greater than or equal to, short form.
0x41	`bge.un` <int32 (target)>	Branch to target if greater than or equal to (unsigned or unordered).
0x34	`bge.un.s` <int8 (target)>	Branch to target if greater than or equal to (unsigned or unordered), short form
0x3D	`bgt` <int32 (target)>	Branch to target if greater than.
0x30	`bgt.s` <int8 (target)>	Branch to target if greater than, short form.
0x42	`bgt.un` <int32 (target)>	Branch to target if greater than (unsigned or unordered).
0x35	`bgt.un.s` <int8 (target)>	Branch to target if greater than (unsigned or unordered), short form.
0x3E	`ble` <int32 (target)>	Branch to target if less than or equal to.
0x31	`ble.s` <int8 (target)>	Branch to target if less than or equal to, short form.
0x43	`ble.un` <int32 (target)>	Branch to target if less than or equal to (unsigned or unordered).
0x36	`ble.un.s` <int8 (target)>	Branch to target if less than or equal to (unsigned or unordered), short form
0x3F	`blt` <int32 (target)>	Branch to target if less than.
0x32	`blt.s` <int8 (target)>	Branch to target if less than, short form.
0x44	`blt.un` <int32 (target)>	Branch to target if less than (unsigned or unordered).
0x37	`blt.un.s` <int8 (target)>	Branch to target if less than (unsigned or unordered), short form.
0x40	`bne.un` <int32 (target)>	Branch to target if unequal or unordered.
0x33	`bne.un.s` <int8 (target)>	Branch to target if unequal or unordered, short form.
0x8C	`box` <typeTok>	Convert a boxable value to its boxed form
0x38	`br` <int32 (target)>	Branch to target.
0x2B	`br.s` <int8 (target)>	Branch to target, short form.
0x01	`break`	Inform a debugger that a breakpoint has been reached.
0x39	`brfalse` <int32 (target)>	Branch to target if value is zero (false).

0x2C	`brfalse.s` <int8 (target)>	Branch to target if value is zero (false), short form.
0x3A	`brinst` <int32 (target)>	Branch to target if value is a non-null object reference (alias for brtrue).
0x2D	`brinst.s` <int8 (target)>	Branch to target if value is a non-null object reference, short form (alias for brtrue.s).
0x39	`brnull` <int32 (target)>	Branch to target if value is null (alias for brfalse).
0x2C	`brnull` <int8 (target)>	Branch to target if value is null (alias for brfalse.s), short form.
0x3A	`brtrue` <int32 (target)>	Branch to target if value is non-zero (true).
0x2D	`brtrue.s` <int8 (target)>	Branch to target if value is non-zero (true), short form.
0x39	`brzero` <int32 (target)>	Branch to target if value is zero (alias for brfalse).
0x2C	`brzero` <int8 (target)>	Branch to target if value is zero (alias for brfalse.s), short form.
0x28	`call` <method>	Call method described by method.
0x29	`calli` <callsitedescr>	Call method indicated on the stack with arguments described by callsitedescr.
0x6F	`callvirt` <method>	Call a method associated with an object.
0x74	`castclass` <class>	Cast obj to class.
0xFE 0x01	`ceq`	Push 1 (of type int32) if value1 equals value2, else push 0.
0xFE 0x02	`cgt`	Push 1 (of type int32) if value1 > value2, else push 0.
0xFE 0x03	`cgt.un`	Push 1 (of type int32) if value1 > value2, unsigned or unordered, else push 0.
0xC3	`ckfinite`	Throw ArithmeticException if value is not a finite number.
0xFE 0x04	`clt`	Push 1 (of type int32) if value1 < value2, else push 0.
0xFE 0x05	`clt.un`	Push 1 (of type int32) if value1 < value2, unsigned or unordered, else push 0.
0xFE 0x16	`constrained.` <thisType> [prefix]	Call a virtual method on a type constrained to be type T
0xD3	`conv.i`	Convert to native int, pushing native int on stack.
0x67	`conv.i1`	Convert to int8, pushing int32 on stack.
0x68	`conv.i2`	Convert to int16, pushing int32 on stack.
0x69	`conv.i4`	Convert to int32, pushing int32 on stack.
0x6A	`conv.i8`	Convert to int64, pushing int64 on stack.
0xD4	conv.ovf.i	Convert to a native int (on the stack as native int) and throw an exception on overflow.
0x8A	conv.ovf.i.un	Convert unsigned to a native int (on the stack as native int) and throw an exception on overflow.
0xB3	conv.ovf.i1	Convert to an int8 (on the stack as int32) and throw an exception on overflow.
0x82	conv.ovf.i1.un	Convert unsigned to an int8 (on the stack as int32) and throw an exception on overflow.

0xB5	conv.ovf.i2	Convert to an int16 (on the stack as int32) and throw an exception on overflow.
0x83	conv.ovf.i2.un	Convert unsigned to an int16 (on the stack as int32) and throw an exception on overflow.
0xB7	conv.ovf.i4	Convert to an int32 (on the stack as int32) and throw an exception on overflow.
0x84	conv.ovf.i4.un	Convert unsigned to an int32 (on the stack as int32) and throw an exception on overflow.
0xB9	conv.ovf.i8	Convert to an int64 (on the stack as int32) and throw an exception on overflow.
0x85	conv.ovf.i8.un	Convert unsigned to an int64 (on the stack as int32) and throw an exception on overflow.
0xD5	conv.ovf.u	Convert to a native unsigned int (on the stack as native int) and throw an exception on overflow.
0x8B	conv.ovf.u.un	Convert unsigned to a native unsigned int (on the stack as native int) and throw an exception on overflow.
0xB4	conv.ovf.u1	Convert to an unsigned int8 (on the stack as int32) and throw an exception on overflow.
0x86	conv.ovf.u1.un	Convert unsigned to an unsigned int8 (on the stack as int32) and throw an exception on overflow.
0xB6	conv.ovf.u2	Convert to an unsigned int16 (on the stack as int32) and throw an exception on overflow.
0x87	conv.ovf.u2.un	Convert unsigned to an unsigned int16 (on the stack as int32) and throw an exception on overflow.
0xB8	conv.ovf.u4	Convert to an unsigned int32 (on the stack as int32) and throw an exception on overflow.
0x88	conv.ovf.u4.un	Convert unsigned to an unsigned int32 (on the stack as int32) and throw an exception on overflow.
0xBA	conv.ovf.u8	Convert to an unsigned int64 (on the stack as int32) and throw an exception on overflow.
0x89	conv.ovf.u8.un	Convert unsigned to an unsigned int64 (on the stack as int32) and throw an exception on overflow.
0x76	conv.r.un	Convert unsigned integer to floating-point, pushing F on stack.
0x6B	conv.r4	Convert to float32, pushing F on stack.
0x6C	conv.r8	Convert to float64, pushing F on stack.
0xE0	conv.u	Convert to native unsigned int, pushing native int on stack.
0xD2	conv.u1	Convert to unsigned int8, pushing int32 on stack.
0xD1	conv.u2	Convert to unsigned int16, pushing int32 on stack.
0x6D	conv.u4	Convert to unsigned int32, pushing int32 on stack.
0x6E	conv.u8	Convert to unsigned int64, pushing int64 on stack.
0xFE 0x17	cpblk	Copy data from memory to memory.
0x70	cpobj <typeTok>	Copy a value type from src to dest.
0x5B	div	Divide two values to return a quotient or floating-point result.

0x5C	div.un	Divide two values, unsigned, returning a quotient.
0x25	dup	Duplicate the value on the top of the stack.
0xDC	endfault	End fault clause of an exception block.
0xFE 0X11	endfilter	End an exception handling filter clause.
0xDC	endfinally	End finally clause of an exception block.
0x4C	idind.u8	Indirect load value of type unsigned int64 as int64 on the stack (alias for ldind.i8).
0xFE 0x18	initblk	Set all bytes in a block of memory to a given byte value.
0xFE 0x15	initobj <typeTok>	Initialize the value at address dest.
0x75	isinst <class>	Test if obj is an instance of class, returning null or an instance of that class or interface.
0x27	jmp <method>	Exit current method and jump to the specified method.
0xFE 0x09	ldarg <uint16 (num)>	Load argument numbered num onto the stack.
0x02	ldarg.0	Load argument 0 onto the stack.
0x03	ldarg.1	Load argument 1 onto the stack.
0x04	ldarg.2	Load argument 2 onto the stack.
0x05	ldarg.3	Load argument 3 onto the stack.
0x0E	ldarg.s <uin8 (num)>	Load argument numbered num onto the stack, short form.
0xFE 0x0A	ldarga <uint16 (argNum)>	Fetch the address of argument argNum.
0x0F	ldarga.s <uint8 (argNum)>	Fetch the address of argument argNum, short form.
0x20	ldc.i4 <int32 (num)>	Push num of type int32 onto the stack as int32.
0x16	ldc.i4.0	Push 0 onto the stack as int32.
0x17	ldc.i4.1	Push 1 onto the stack as int32.
0x18	ldc.i4.2	Push 2 onto the stack as int32.
0x19	ldc.i4.3	Push 3 onto the stack as int32.
0x1A	ldc.i4.4	Push 4 onto the stack as int32.
0x1B	ldc.i4.5	Push 5 onto the stack as int32.
0x1C	ldc.i4.6	Push 6 onto the stack as int32.
0x1D	ldc.i4.7	Push 7 onto the stack as int32.
0x1E	ldc.i4.8	Push 8 onto the stack as int32.
0x15	ldc.i4.m1	Push -1 onto the stack as int32.
0x15	ldc.i4.M1	Push -1 of type int32 onto the stack as int32 (alias for ldc.i4.m1).
0x1F	ldc.i4.s <int8 (num)>	Push num onto the stack as int32, short form.
0x21	ldc.i8 <int64 (num)>	Push num of type int64 onto the stack as int64.
0x22	ldc.r4 <float32 (num)>	Push num of type float32 onto the stack as F.
0x23	ldc.r8 <float64 (num)>	Push num of type float64 onto the stack as F.

0xA3	ldelem <typeTok>	Load the element at index onto the top of the stack.
0x97	ldelem.i	Load the element with type native int at index onto the top of the stack as a native int.
0x90	ldelem.i1	Load the element with type int8 at index onto the top of the stack as an int32.
0x92	ldelem.i2	Load the element with type int16 at index onto the top of the stack as an int32.
0x94	ldelem.i4	Load the element with type int32 at index onto the top of the stack as an int32.
0x96	ldelem.i8	Load the element with type int64 at index onto the top of the stack as an int64.
0x98	ldelem.r4	Load the element with type float32 at index onto the top of the stack as an F
0x99	ldelem.r8	Load the element with type float64 at index onto the top of the stack as an F.
0x9A	ldelem.ref	Load the element at index onto the top of the stack as an O. The type of the O is the same as the element type of the array pushed on the CIL stack.
0x91	ldelem.u1	Load the element with type unsigned int8 at index onto the top of the stack as an int32.
0x93	ldelem.u2	Load the element with type unsigned int16 at index onto the top of the stack as an int32.
0x95	ldelem.u4	Load the element with type unsigned int32 at index onto the top of the stack as an int32.
0x96	ldelem.u8	Load the element with type unsigned int64 at index onto the top of the stack as an int64 (alias for ldelem.i8).
0x8F	ldelema <class>	Load the address of element at index onto the top of the stack.
0x7B	ldfld <field>	Push the value of field of object (or value type) obj, onto the stack.
0x7C	ldflda <field>	Push the address of field of object obj on the stack.
0xFE 0x06	ldftn <method>	Push a pointer to a method referenced by method, on the stack.
0x4D	ldind.i	Indirect load value of type native int as native int on the stack
0x46	ldind.i1	Indirect load value of type int8 as int32 on the stack.
0x48	ldind.i2	Indirect load value of type int16 as int32 on the stack.
0x4A	ldind.i4	Indirect load value of type int32 as int32 on the stack.
0x4C	ldind.i8	Indirect load value of type int64 as int64 on the stack.
0x4E	ldind.r4	Indirect load value of type float32 as F on the stack.
0x4F	ldind.r8	Indirect load value of type float64 as F on the stack.
0x50	ldind.ref	Indirect load value of type object ref as O on the stack.
0x47	ldind.u1	Indirect load value of type unsigned int8 as int32 on the stack

0x49	ldind.u2	Indirect load value of type unsigned int16 as int32 on the stack
0x4B	ldind.u4	Indirect load value of type unsigned int32 as int32 on the stack
0x8E	ldlen	Push the length (of type native unsigned int) of array on the stack.
0xFE 0x0C	ldloc <uint16 (indx)>	Load local variable of index indx onto stack.
0x06	ldloc.0	Load local variable 0 onto stack.
0x07	ldloc.1	Load local variable 1 onto stack.
0x08	ldloc.2	Load local variable 2 onto stack.
0x09	ldloc.3	Load local variable 3 onto stack.
0x11	ldloc.s <uint8 (indx)>	Load local variable of index indx onto stack, short form.
0xFE 0x0D	ldloca <uint16 (indx)>	Load address of local variable with index indx.
0x12	ldloca.s <uint8 (indx)>	Load address of local variable with index indx, short form.
0x14	ldnull	Push a null reference on the stack.
0x71	ldobj <typeTok>	Copy the value stored at address src to the stack.
0x7E	ldsfld <field>	Push the value of field on the stack.
0x7F	ldsflda <field>	Push the address of the static field, field, on the stack.
0x72	ldstr <string>	Push a string object for the literal string.
0xD0	ldtoken <token>	Convert metadata token to its runtime representation.
0xFE 0x07	ldvirtftn <method>	Push address of virtual method method on the stack.
0xDD	leave <int32 (target)>	Exit a protected region of code.
0xDE	leave.s <int8 (target)>	Exit a protected region of code, short form.
0xFE 0x0F	localloc	Allocate space from the local memory pool.
0xC6	mkrefany <class>	Push a typed reference to ptr of type class onto the stack.
0x5A	mul	Multiply values.
0xD8	mul.ovf.<type>	Multiply signed integer values. Signed result shall fit in same size
0xD9	mul.ovf.un	Multiply unsigned integer values. Unsigned result shall fit in same size
0x65	neg	Negate value.
0x8D	newarr <etype>	Create a new array with elements of type etype.
0x73	newobj <ctor>	Allocate an uninitialized object or value type and call ctor.
0xFE 0x19	no. { typecheck, rangecheck, nullcheck } [prefix]	The specified fault check(s) normally performed as part of the execution of the subsequent instruction can/shall be skipped.

0x00	nop	Do nothing.
0x66	not	Bitwise complement.
0x60	or	Bitwise OR of two integer values, returns an integer.
0x26	pop	Pop value from the stack.
0xFE 0x1E	readonly. [prefix]	Specify that the subsequent array address operation performs no type check at runtime, and that it returns a controlled- mutability managed pointer
0xFE 0x1D	refanytype	Push the type token stored in a typed reference.
0xC2	refanyval <type>	Push the address stored in a typed reference.
0x5D	rem	Remainder when dividing one value by another.
0x5E	rem.un	Remainder when dividing one unsigned value by another.
0x2A	ret	Return from method, possibly with a value.
0xFE 0x1A	rethrow	Rethrow the current exception.
0x62	shl	Shift an integer left (shifting in zeros), return an integer.
0x63	shr	Shift an integer right (shift in sign), return an integer.
0x64	shr.un	Shift an integer right (shift in zero), return an integer.
0xFE 0x1C	sizeof <typeTok>	Push the size, in bytes, of a type as an unsigned int32.
0xFE 0x0B	starg <uint16 (num)>	Store value to the argument numbered num.
0x10	starg.s <uint8 (num)>	Store value to the argument numbered num, short form.
0xA4	stelem <typeTok>	Replace array element at index with the value on the stack
0x9B	stelem.i	Replace array element at index with the i value on the stack.
0x9C	stelem.i1	Replace array element at index with the int8 value on the stack.
0x9D	stelem.i2	Replace array element at index with the int16 value on the stack.
0x9E	stelem.i4	Replace array element at index with the int32 value on the stack.
0x9F	stelem.i8	Replace array element at index with the int64 value on the stack.
0xA0	stelem.r4	Replace array element at index with the float32 value on the stack.
0xA1	stelem.r8	Replace array element at index with the float64 value on the stack.
0xA2	stelem.ref	Replace array element at index with the ref value on the stack.
0x7D	stfld <field>	Replace the value of field of the object obj with value.
0xDF	stind.i	Store value of type native int into memory at address

0x52	stind.i1	Store value of type int8 into memory at address
0x53	stind.i2	Store value of type int16 into memory at address
0x54	stind.i4	Store value of type int32 into memory at address
0x55	stind.i8	Store value of type int64 into memory at address
0x56	stind.r4	Store value of type float32 into memory at address
0x57	stind.r8	Store value of type float64 into memory at address
0x51	stind.ref	Store value of type object ref (type O) into memory at address
0xFE 0x0E	stloc <uint16 (indx)>	Pop a value from stack into local variable indx.
0x0A	stloc.0	Pop a value from stack into local variable 0.
0x0B	stloc.1	Pop a value from stack into local variable 1.
0x0C	stloc.2	Pop a value from stack into local variable 2.
0x0D	stloc.3	Pop a value from stack into local variable 3.
0x13	stloc.s <uint8 (indx)>	Pop a value from stack into local variable indx, short form.
0x81	stobj <typeTok>	Store a value of type typeTok at an address.
0x80	stsfld <field>	Replace the value of field with val.
0x59	sub	Subtract value2 from value1, returning a new value.
0xDA	sub.ovf	Subtract native int from a native int. Signed result shall fit in same size
0xDB	sub.ovf.un	Subtract native unsigned int from a native unsigned int. Unsigned result shall fit in same size.
0x45	switch <uint32, int32,int32 (t1..tN)>	Jump to one of n values.
0xFE 0x14	tail. [prefix]	Subsequent call terminates current method
0x7A	throw	Throw an exception.
0xFE 0x12	unaligned. (alignment) [prefix]	Subsequent pointer instruction might be unaligned.
0x79	unbox <valuetype>	Extract a value-type from obj, its boxed representation.
0xA5	unbox.any <typeTok>	Extract a value-type from obj, its boxed representation
0xFE 0x13	volatile. [prefix]	Subsequent pointer reference is volatile.
0x61	xor	Bitwise XOR of integer values, returns an integer.

Anexo IV – Lista de instruções do Java Bytecode

Mnemonic	Opcode (in hex)	Other bytes	Description
aaload	32		loads onto the stack a reference from an array
aastore	53		stores into a reference to an array
aconst_null	01		pushes a null reference onto the stack
aload	19	index	loads a reference onto the stack from a local variable #index
aload_0	2a		loads a reference onto the stack from local variable 0
aload_1	2b		loads a reference onto the stack from local variable 1
aload_2	2c		loads a reference onto the stack from local variable 2
aload_3	2d		loads a reference onto the stack from local variable 3
anewarray	bd	indexbyte1, indexbyte2	creates a new array of references of length count and component type identified by the class reference index (indexbyte1 << 8 + indexbyte2) in the constant pool
areturn	b0		returns a reference from a method
arraylength	be		gets the length of an array
astore	3a	index	stores a reference into a local variable #index
astore_0	4b		stores a reference into local variable 0
astore_1	4c		stores a reference into local variable 1
astore_2	4d		stores a reference into local variable 2
astore_3	4e		stores a reference into local variable 3
athrow	bf		throws an error or exception (notice that the rest of the stack is cleared, leaving only a reference to the Throwable)
baload	33		loads a byte or Boolean value from an array
bastore	54		stores a byte or Boolean value into an array
bipush	10	byte	pushes a byte onto the stack as an integer value
caload	34		loads a char from an array
castore	55		stores a char into an array
checkcast	c0	indexbyte1, indexbyte2	checks whether an objectref is of a certain type, the class reference of which is in the constant pool at index (indexbyte1 << 8 + indexbyte2)
d2f	90		converts a double to a float
d2i	8e		converts a double to an int
d2l	8f		converts a double to a long
dadd	63		adds two doubles

daload	31		loads a double from an array
dastore	52		stores a double into an array
dcmpg	98		compares two doubles
dcmpl	97		compares two doubles
dconst_0	0e		pushes the constant 0.0 onto the stack
dconst_1	0f		pushes the constant 1.0 onto the stack
ddiv	6f		divides two doubles
dload	18	index	loads a double value from a local variable #index
dload_0	26		loads a double from local variable 0
dload_1	27		loads a double from local variable 1
dload_2	28		loads a double from local variable 2
dload_3	29		loads a double from local variable 3
dmul	6b		multiplies two doubles
dneg	77		negates a double
drem	73		gets the remainder from a division between two doubles
dreturn	af		returns a double from a method
dstore	39	index	stores a double value into a local variable #index
dstore_0	47		stores a double into local variable 0
dstore_1	48		stores a double into local variable 1
dstore_2	49		stores a double into local variable 2
dstore_3	4a		stores a double into local variable 3
dsub	67		subtracts a double from another
dup	59		duplicates the value on top of the stack
dup_x1	5a		inserts a copy of the top value into the stack two values from the top
dup_x2	5b		inserts a copy of the top value into the stack two (if value2 is double or long it takes up the entry of value3, too) or three values (if value2 is neither double nor long) from the top
dup2	5c		duplicate top two stack words (two values, if value1 is not double nor long; a single value, if value1 is double or long)
dup2_x1	5d		duplicate two words and insert beneath third word (see explanation above)
dup2_x2	5e		duplicate two words and insert beneath fourth word
f2d	8d		converts a float to a double
f2i	8b		converts a float to an int
f2l	8c		converts a float to a long
fadd	62		adds two floats
faload	30		loads a float from an array
fastore	51		stores a float in an array

fcmpg	96		compares two floats
fcmpl	95		compares two floats
fconst_0	0b		pushes 0.0f on the stack
fconst_1	0c		pushes 1.0f on the stack
fconst_2	0d		pushes 2.0f on the stack
fdiv	6e		divides two floats
fload	17	index	loads a float value from a local variable #index
fload_0	22		loads a float value from local variable 0
fload_1	23		loads a float value from local variable 1
fload_2	24		loads a float value from local variable 2
fload_3	25		loads a float value from local variable 3
fmul	6a		multiplies two floats
fneg	76		negates a float
frem	72		gets the remainder from a division between two floats
freturn	ae		returns a float
fstore	38	index	stores a float value into a local variable #index
fstore_0	43		stores a float value into local variable 0
fstore_1	44		stores a float value into local variable 1
fstore_2	45		stores a float value into local variable 2
fstore_3	46		stores a float value into local variable 3
fsub	66		subtracts two floats
getfield	b4	index1, index2	gets a field value of an object objectref, where the field is identified by field reference in the constant pool index (index1 << 8 + index2)
getstatic	b2	index1, index2	gets a static field value of a class, where the field is identified by field reference in the constant pool index (index1 << 8 + index2)
goto	a7	branchbyte1, branchbyte2	goes to another instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
goto_w	c8	branchbyte1, branchbyte2, branchbyte3, branchbyte4	goes to another instruction at branchoffset (signed int constructed from unsigned bytes branchbyte1 << 24 + branchbyte2 << 16 + branchbyte3 << 8 + branchbyte4)
i2b	91		converts an int into a byte
i2c	92		converts an int into a character
i2d	87		converts an int into a double
i2f	86		converts an int into a float
i2l	85		converts an int into a long
i2s	93		converts an int into a short
iadd	60		adds two ints together
iaload	2e		loads an int from an array
iand	7e		performs a bitwise and on two integers

iastore	4f		stores an int into an array
iconst_m1	02		loads the int value -1 onto the stack
iconst_0	03		loads the int value 0 onto the stack
iconst_1	04		loads the int value 1 onto the stack
iconst_2	05		loads the int value 2 onto the stack
iconst_3	06		loads the int value 3 onto the stack
iconst_4	07		loads the int value 4 onto the stack
iconst_5	08		loads the int value 5 onto the stack
idiv	6c		divides two integers
if_acmpeq	a5	branchbyte1, branchbyte2	if references are equal, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_acmpne	a6	branchbyte1, branchbyte2	if references are not equal, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_icmpeq	9f	branchbyte1, branchbyte2	if ints are equal, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_icmpne	a0	branchbyte1, branchbyte2	if ints are not equal, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_icmplt	a1	branchbyte1, branchbyte2	if value1 is less than value2, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_icmpge	a2	branchbyte1, branchbyte2	if value1 is greater than or equal to value2, branch to instruction atbranchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_icmpgt	a3	branchbyte1, branchbyte2	if value1 is greater than value2, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
if_icmple	a4	branchbyte1, branchbyte2	if value1 is less than or equal to value2, branch to instruction atbranchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifeq	99	branchbyte1, branchbyte2	if value is 0, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifne	9a	branchbyte1, branchbyte2	if value is not 0, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
iflt	9b	branchbyte1, branchbyte2	if value is less than 0, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifge	9c	branchbyte1, branchbyte2	if value is greater than or equal to 0, branch to instruction at branchoffset(signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifgt	9d	branchbyte1, branchbyte2	if value is greater than 0, branch to instruction

			at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifle	9e	branchbyte1, branchbyte2	if value is less than or equal to 0, branch to instruction at branchoffset(signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifnonnull	c7	branchbyte1, branchbyte2	if value is not null, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
ifnull	c6	branchbyte1, branchbyte2	if value is null, branch to instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2)
iinc	84	index, const	increment local variable #index by signed byte const
iload	15	index	loads an int value from a local variable #index
iload_0	1a		loads an int value from local variable 0
iload_1	1b		loads an int value from local variable 1
iload_2	1c		loads an int value from local variable 2
iload_3	1d		loads an int value from local variable 3
imul	68		multiply two integers
ineg	74		negate int
instanceof	c1	indexbyte1, indexbyte2	determines if an object objectref is of a given type, identified by class reference index in constant pool (indexbyte1 << 8 + indexbyte2)
invokeinterface	b9	indexbyte1, indexbyte2, count, 0	invokes an interface method on object objectref, where the interface method is identified by method reference index in constant pool (indexbyte1 << 8 + indexbyte2)
invokespecial	b7	indexbyte1, indexbyte2	invoke instance method on object objectref, where the method is identified by method reference index in constant pool (indexbyte1 << 8 + indexbyte2)
invokestatic	b8	indexbyte1, indexbyte2	invoke a static method, where the method is identified by method referenceindex in constant pool (indexbyte1 << 8 + indexbyte2)
invokevirtual	b6	indexbyte1, indexbyte2	invoke virtual method on object objectref, where the method is identified by method reference index in constant pool (indexbyte1 << 8 + indexbyte2)
ior	80		bitwise int or
irem	70		logical int remainder
ireturn	ac		returns an integer from a method
ishl	78		int shift left
ishr	7a		int arithmetic shift right
istore	36	index	store int value into variable #index
istore_0	3b		store int value into variable 0
istore_1	3c		store int value into variable 1
istore_2	3d		store int value into variable 2

istore_3	3e		store int value into variable 3
isub	64		int subtract
iushr	7c		int logical shift right
ixor	82		int xor
jsr	a8	branchbyte1, branchbyte2	jump to subroutine at branchoffset (signed short constructed from unsigned bytes branchbyte1 << 8 + branchbyte2) and place the return address on the stack
jsr_w	c9	branchbyte1, branchbyte2, branchbyte3, branchbyte4	jump to subroutine at branchoffset (signed int constructed from unsigned bytes branchbyte1 << 24 + branchbyte2 << 16 + branchbyte3 << 8 + branchbyte4) and place the return address on the stack
l2d	8a		converts a long to a double
l2f	89		converts a long to a float
l2i	88		converts a long to a int
ladd	61		add two longs
laload	2f		load a long from an array
land	7f		bitwise and of two longs
lastore	50		store a long to an array
lcmp	94		compares two longs values
lconst_0	09		pushes the long 0 onto the stack
lconst_1	0a		pushes the long 1 onto the stack
ldc	12	index	pushes a constant #index from a constant pool (String, int or float) onto the stack
ldc_w	13	indexbyte1, indexbyte2	pushes a constant #index from a constant pool (String, int or float) onto the stack (wide index is constructed as indexbyte1 << 8 + indexbyte2)
ldc2_w	14	indexbyte1, indexbyte2	pushes a constant #index from a constant pool (double or long) onto the stack (wide index is constructed as indexbyte1 << 8 + indexbyte2)
ldiv	6d		divide two longs
lload	16	index	load a long value from a local variable #index
lload_0	1e		load a long value from a local variable 0
lload_1	1f		load a long value from a local variable 1
lload_2	20		load a long value from a local variable 2
lload_3	21		load a long value from a local variable 3
lmul	69		multiplies two longs
lneg	75		negates a long
lookupswitch	ab	<0-3 bytes padding>, defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, npairs1, npairs2, npairs3, npairs4, match-offset pairs...	a target address is looked up from a table using a key and execution continues from the instruction at that address
lor	81		bitwise or of two longs

lrem	71		remainder of division of two longs
lreturn	ad		returns a long value
lshl	79		bitwise shift left of a long value1 by value2 positions
lshr	7b		bitwise shift right of a long value1 by value2 positions
lstore	37	index	store a long value in a local variable #index
lstore_0	3f		store a long value in a local variable 0
lstore_1	40		store a long value in a local variable 1
lstore_2	41		store a long value in a local variable 2
lstore_3	42		store a long value in a local variable 3
lsub	65		subtract two longs
lushr	7d		bitwise shift right of a long value1 by value2 positions, unsigned
lxor	83		bitwise exclusive or of two longs
monitorenter	c2		enter monitor for object ("grab the lock" - start of synchronized() section)
monitorexit	c3		exit monitor for object ("release the lock" - end of synchronized() section)
multianewarray	c5	indexbyte1, indexbyte2, dimensions	create a new array of dimensions dimensions with elements of type identified by class reference in constant pool index (indexbyte1 << 8 + indexbyte2); the sizes of each dimension is identified by count1, [count2, etc]
new	bb	indexbyte1, indexbyte2	creates new object of type identified by class reference in constant poolindex (indexbyte1 << 8 + indexbyte2)
newarray	bc	atype	creates new array with count elements of primitive type identified by atype
nop	00		performs no operation
pop	57		discards the top value on the stack
pop2	58		discards the top two values on the stack (or one value, if it is a double or long)
putfield	b5	indexbyte1, indexbyte2	set field to value in an object objectref, where the field is identified by a field reference index in constant pool (indexbyte1 << 8 + indexbyte2)
putstatic	b3	indexbyte1, indexbyte2	set static field to value in a class, where the field is identified by a field reference index in constant pool (indexbyte1 << 8 + indexbyte2)
ret	a9	index	continue execution from address taken from a local variable #index (the asymmetry with jsr is intentional)
return	b1		return void from method
saload	35		load short from array
sastore	56		store short to array
sipush	11	byte1, byte2	pushes a signed integer (byte1 << 8 + byte2) onto the stack

swap	5f		swaps two top words on the stack (note that value1 and value2 must not be double or long)
tableswitch	aa	[0-3 bytes padding], defaultbyte1, defaultbyte2, defaultbyte3, defaultbyte4, lowbyte1, lowbyte2, lowbyte3, lowbyte4, highbyte1, highbyte2, highbyte3, highbyte4, jump offsets...	continue execution from an address in the table at offset index
wide	c4	opcode, indexbyte1, indexbyte2 or iinc, indexbyte1, indexbyte2, countbyte1, countbyte2	execute opcode, where opcode is either iload, fload, aload, lload, dload, istore, fstore, astore, lstore, dstore, or ret, but assume the index is 16 bit; or execute iinc, where the index is 16 bits and the constant to increment by is a signed 16 bit short
breakpoint	ca		reserved for breakpoints in Java debuggers; should not appear in any class file
impdep1	fe		reserved for implementation-dependent operations within debuggers; should not appear in any class file
impdep2	ff		reserved for implementation-dependent operations within debuggers; should not appear in any class file
(no name)	cb-fd		these values are currently unassigned for opcodes and are reserved for future use
xxxunusedxxx	ba		this opcode is reserved "for historical reasons"

Anexo V – JRockit JVM

Apesar de parecer for a do escopo proposto para o trabalho, achou-se interessante apresentar o funcionamento de uma implementação da JVM diferente da HotSpot a fim de buscar uma comparação mais aprofundada. Para tal, apresenta-se a seguir uma adaptação da segunda parte do Diagnosis Guide R27.6 da JRockit, “Understanding JIT Compilation and Optimizations”, publicado pela Oracle em abril de 2009:

V.1. Compilação de código com a JRockit JVM

O gerador de código na JRockit executa em background durante toda a execução de uma aplicação Java, automaticamente adaptando o código para executar da melhor forma. O gerador de código funciona em três passos, apresentados abaixo:

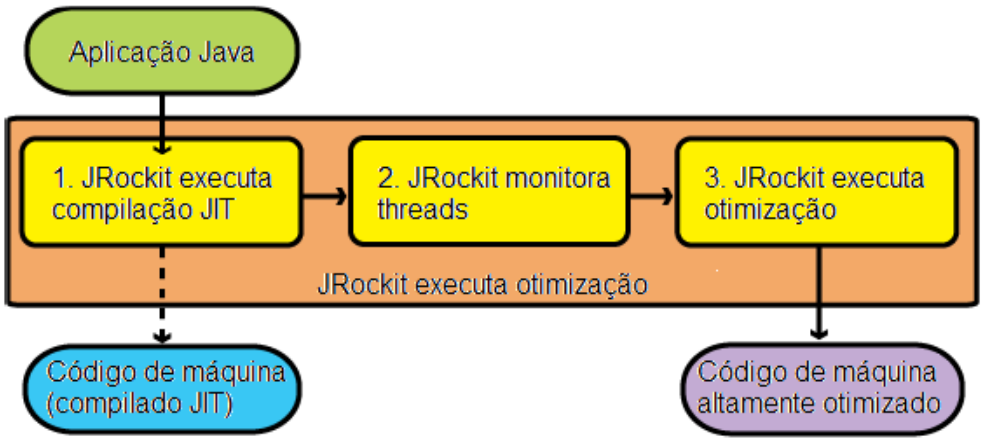

Figura 11: Execução da JRockit

V.1.1. JRockit executa compilação JIT

O primeiro passo da geração de código é a compilação JIT. Esta compilação permite que a aplicação Java seja iniciada e execute enquanto o

código que está sendo gerado não está altamente otimizado para a plataforma. Apesar do compilador JIT não ser parte da especificação da JVM, ele é, contudo, um componente essencial na plataforma Java. Na teoria, o compilador JIT é acionado sempre que um método é invocado e compila o bytecode do referido método para código nativo.

Depois que um método é compilado, a JRockit usa o código compilado diretamente ao invés de tentar interpretá-lo, tornando a execução da aplicação mais rápida. Entretanto, durante o início da execução, milhares de novos métodos são invocados, o que pode tornar a inicialização da JRockit mais lenta em comparação a outras JVMs. Isto é causado pelo processamento excessivo (overhead) no compilador JIT. Então, se uma JVM é executada sem um compilador JIT, tal JVM tem uma inicialização rápida mas executa mais lentamente. Se a JRockit é executada, esta pode ter uma inicialização lenta, mas terá uma execução mais rápida ao longo do tempo. Em um determinado momento, o usuário pode achar que se demora mais para inicializar a JVM do que para executar a aplicação.

Compilar todos os métodos com todas as otimizações disponíveis na inicialização teria um impacto negativo em seu tempo. Portanto o compilador JIT não otimiza completamente todos os métodos na inicialização.

V.1.2. JRockit monitora Threads

Durante a segunda fase, a JRockit utiliza uma técnica de baixo custo, baseada em amostragem para identificar quais funções mostrariam vantagem em ser otimizadas: uma thread de “coleta de amostragem” é acionada em intervalos periódicos e verifica o estado de diversas threads da aplicação. Ela

identifica o que cada thread está executando e toma notas do histórico de execução. Esta informação é levantada para todos os métodos e quando percebe-se que um método está sendo usado muito – em outras palavras, é “quente” – este método é marcado para otimização. Geralmente, um turbilhão de oportunidades de otimização ocorrem nos estágios iniciais da execução da aplicação, com sua taxa diminuindo na medida que a execução prossegue.

V.1.3. JRockit executa otimização

Na terceira fase, a JVM executa uma rodada de otimização dos métodos que ela percebe serem os mais usados. Esta otimização é executada no background e não perturba a execução da aplicação.

V.2. Diferenças entre a JRockit e a HotSpot

Como pode ser constatado, a principal diferença entre as duas JVM’s é que ao passo que a HotSpot utiliza um híbrido de interpretação e compilação JIT, a JRockit utiliza exclusivamente compilação JIT. Um fato curioso de se notar é que a JRockit utiliza uma forma que se assemelha à identificação de “pontos quentes” da HotSpot para encontrar métodos que precisem de otimização mais pesada, ao passo que estes são os escolhidos na HotSpot para serem compilados.

Outra diferença grande está na própria otimização: enquanto a JRockit tem apenas uma fase de otimização por ciclo (a otimização dos métodos identificados como “quentes” pela thread de coleta de amostragem), a HotSpot possui duas otimizações, uma referente à HIR e outra referente a LIR, um conceito que nem se quer existe no compilador da JRockit.