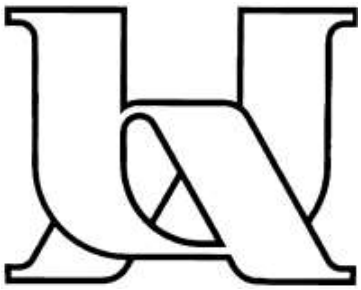

UNIVERSITEIT ANTWERPEN
Departement Wiskunde en Informatica

2003-2004

Java and .NET: A look into today's virtual machine technology

Bart Van Rompaey

Proefschrift ingediend tot het behalen van
de graad Licentiaat in de Wetenschappen

Begeleider: Filip Van Rysselberghe
Promotor: Prof. Dr. S. Demeyer
Copromotor: Prof. Dr. B. Watson

Acknowledgements

First of all, I'd like to thank Prof. Dr. Demeyer for letting me start with a thesis on this subject. At the time of the proposal I had barely an idea in which direction I wanted to research.

Many thanks goes out to LORE-assistants Hans, Andy and Filip, who were always open for questions and helped me pushing the research in the right direction.

Benny helped me with his more thorough knowledge of the English language to limit the amount of mistakes against spelling and grammar. I also enjoyed discussing some matters with Steven.

A special loving thank you goes to my girlfriend An, who supported me along the way with her continuous patience while the thesis got priority.

In times when I was walking across the house murmuring all kinds of weird terms, my family let it quietly happen and gave me the tranquillity necessary to create sound theories. Thank you for that!

Nederlandstalige Samenvatting

Hoewel het begrip virtuele machine (VM) niet nieuw is binnen de informatica, merkt men dat ze maar echt populair zijn geworden zo'n tien jaar geleden. De komst van het Internet, kleine draagbare toestellen, de nood aan *portable* en robuuste software en zeker ook Java hebben hiertoe de aanleiding gegeven. Virtuele machine technologie wordt vandaag de dag gebruikt op vele plaatsen waar software nodig is, van de kleinste GSM tot de zwaarste server.

De twee meest gebruikte virtuele machine platformen zijn Java en .NET. Aan de hand van deze twee zullen de begrippen die verbonden zijn aan een VM praktisch worden toegelicht. Deze twee platformen kennen veel gelijkenissen, maar verschillen op bepaalde vakken ook grondig. Er wordt ingegaan op deze verschillen en hun gevolgen.

De criteria aan da hand waarvan Java en .NET vergeleken worden, zijn zo gekozen dat ze beantwoorden aan de noden van een diversiteit software domeinen.

- Functionaliteit toont de kracht van de virtuele machine en de toegevoegde waarde die ze kan bieden aan zowel de ontwikkelaars als het te bekomen product.
- Portabiliteit is een belangrijk criterium gezien de brede waaier van hardware en besturingssystemen waarvoor virtuele machine technologie wordt ingezet.
- Met het criterium *resources* wordt de hoeveelheid geheugen, harde schijf en processortijd bedoelt die nodig is voor de uitvoering van applicaties.

Aan de hand van theoretische beschrijving van de technologieën achter virtuele machines enerzijds, en metingen aan de hand van een *case study* anderzijds wordt getest hoe goed virtuele machines, en Java en .NET in het bijzonder, aan deze criteria kunnen voldoen.

De resultaten zijn zeer divers en het blijkt dat de criteria elkaar zeer sterk beïnvloeden. Hieruit kan afgeleid worden dat Java en .NET schaalbaar zijn naar de criteria die belangrijk zijn in een bepaald domein door middel van de juiste configuratie van de componenten van deze VMs. Er moet wel rekening mee gehouden worden dat die vaak ten nadele is van de andere criteria.

Deze thesis kan keuzes tussen beide platformen, of het gebruik van virtuele machine technologie in het algemeen, vergemakkelijken, zonder dat hier gepretendeerd wordt voor elke mogelijk situatie een antwoord te kunnen bieden.

Abstract

Virtual machine technology is growing in popularity in a lot of software domains. In particular the Java and .NET platforms are used from the smallest handheld up to the biggest server.

This thesis compares virtual machines execution with traditional native execution and tests the concepts of virtual machines, and more particular the Java and .NET implementations, for suitability in environments with varying needs.

Those needs are formulated as a set of criteria containing functionality, portability, performance and resources usage.

The mix of theoretical background and a case study concludes how well Java and .NET virtual machine implementations are able to respond to one or more of these criteria and reveal interesting benefits as well as fundamental weaknesses.

Contents

List of Figures

List of Tables

Chapter 1

Contents

Problem Statement Virtual machine technology is becoming widely used in an increasingly growing set of domains within software development. Each domain carries its own specific needs, which seem able to be answered by virtual machines.

The particular questions that this thesis tries to answer are:

- Why is executing applications under control of a virtual machine (VM) a valuable alternative to the classical native execution? How is it different, and what are the implications?
- How do Java and .NET, two widespread virtual machine based platforms compare to each other? What are the major differences between these two that one should watch out for when choosing between them? Where are they applicable, and in which configuration?

Criteria The following criteria form the guidelines throughout the thesis to judge virtual machines in general, and Java and .NET in particular.

- **Functionality** Which functionality or services can a virtual machine offer in addition to native execution? Functionality goes together with complexity. More functionality probably leads to better support towards the developer and the applications, but will increase complexity. Better support for the developer means VM properties and tools to help in creating software of higher quality in less time. Note that a more complex system will hinder portability, the next criterion.
- **Portability** With a range of hardware platforms and operating systems in mind, portability is an important factor for a virtual machine in order to be widely usable. It comes in two sorts: at one side a virtual machine implementation must be designed in such a way that porting it to a new platform must be doable in a limited period of time. At the other side, code

that is meant to be run under a VM is preferably transferable to another platform without recompilation. In the case of Java and .NET, portability also means scalability, as they are both used from the embedded software market until the biggest server.

- **Resources** The amount of resources needed, both space and time, is an important factor when deciding to go for virtual machine technology or not. And then again, how do Java and .NET differ in those domains.
  - **Space** resources are expressed in disk space needed for the VM and the applications at one side, and main memory usage at runtime at the other side.
  - **Time** resources are expressed in the time a VM needs to execute a certain application. This time consists of the start-up and initialization of the VM, execution of the application, and shutdown of the VM. Some of the functionality of the VM will also influence the execution and take extra resources. This criterion will further be called performance.

Approach and construction In order to equal the readers' foreknowledge in the domain of virtual machines, and more specific the knowledge about the Java and .NET platforms, chapter 2 will explain those environments in general. The origin of virtual machines is briefly discussed along with its early usage and some well known examples. Next, the Java and .NET terms are unravelled and the relevant parts for this thesis indicated. Readers familiar with the origin and the general concepts can skip this chapter.

Chapter 3 concentrates on the additional functionality that VM can offer. In practice Java and .NET will be compared against standard C++, which is, once compiled, an example of classic execution.

The intermediate language (IL), which is the language expressing the power of the virtual machine, is the subject of chapter 4, followed by a chapter dealing with the file format the IL is stored into after compile time. These two are rather descriptive by nature and do not directly make use of the criteria. However, the case study in chapter 6 and the performance tests in chapter 8 are founded on the information contained in these chapters.

Using the criteria mentioned above the strengths and weaknesses of Java and .NET as a platform, but also of their particular implementations will be revealed. The insight in VM technology created here not leave the reader with an impression of a VM as a black box, but will sharpen his/her with the necessary knowledge to investigate a certain VM for aspects mentioned here before deciding to use it in a project.

Chapter 2

Introduction

2.1 What are virtual machines?

There is no unique definition for *virtual machine*. Rather, a couple of definitions circle around containing some factors, which together cover the concept of a virtual machine.

- A VM is a virtual, abstract, simulated computer.
- It runs on a host computer.
- It isolates and manages a part of the memory it got from the host system.
- It provides a new, virtual architecture for applications that run on top of the VM without those applications having to be aware of the underlying system.

Several benefits come to play when making use of VMs. When new hardware architectures are being developed, it can be practical to test the platform in software before the hardware will effectively being produced. It can also be used as a test bed for software, operating systems for example can be debugged via a VM. Applications that are written for an architecture that does not correspond to the hardware of the machine it will be run on can be run under a VM that mimics the intended hardware. A VM allows running multiple operating systems at the same time too, each in its own separated part of memory but all sharing the same underlying processor.

Perhaps the first principles originated when IBM built a system with virtual memory in 1967, the IBM 360 model 67 (figure 2.1). A program for this machine was written simulating the behaviour of several standard 360s without virtual memory. This program was called CP67 (Control Program), it allowed IBM to run other operating systems written for the 360 concurrently.

Figure 2.1: IBM 360 model 67 at the University of Newcastle

One of the more famous VMs in the late 70s was the P-Code machine of UCSD Pascal. A Pascal compiler compiled source code into a set of machine-language-alike p-code instructions, which were then interpreted on various hardware platforms of that time.

Another product of the 70s is Smalltalk. Developed by Xerox, this platform is object-oriented and portable across various hardware and operating systems.

2.2 What is Java?

2.2.1 Java History

Once there was a project at Sun Microsystems, started in 1990, that did research in software for smart consumer electronics and set-top boxes. It was called the Green Project. One of the first prototypes of the Java virtual machine was created by this project and ran on a handheld device. In 1994, with the rise of the World Wide Web, the developers discovered that the requirements for web software were the same as that for their previous targets: small, platform independent, secure and reliable code. Because the programming languages at that time, mainly C and C++, couldn't respond to those requirements, a new language named Oak was invented. This would eventually become the Java programming language. The first product that resulted from this efforts was the HotJava browser.

Since then, Java has become quite popular, and is available from the smallest handheld device to the largest server. Not only does the availability varies over such a large range, but Java is also used for a variety of application domains. Java is used as operating system in small devices, as small internet applications embedded in web pages (called applets), as desktop applications but also as server software.

2.2.2 Java Platform

Java is said to be a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multi-threaded and dynamic language [Sun95]. The idea behind Java is that the same software should run on many different systems, devices and operating systems throughout networks. To achieve this, a virtual machine is deployed on these systems. The Java 2 Platform (figure 2.2), as it is called as a whole, consists of a virtual machine, the Java programming language compiler, The Java Class Libraries (JCL) and a couple of tools.

- The **VM** is the layer that translates the application's code, which is contained in a format that is not hardware specific, to a format that gets understood by the hardware. Hence Sun's "Write Once, Run Everywhere" slogan. The intermediate format is called byte code. It is thus completely portable over all hardware platforms where a Java Virtual Machine (JVM) exists for. The virtual machine is that part of the platform that will get the interest in this thesis.

- The **Java programming language** is Sun's language for the programmer targeted towards the Java Platform. Its syntax largely resembles the C and C++ style.

- The API (Application Programming Interface) provides interfaces to the **JCL** in order to be used by programmers. The JCL contains functionality that is often needed in applications and does not need to be created again and again by application programmers. Moreover, the libraries are extensible. The content varies over core libraries as strings and data collections over graphical user interfaces (AWT, Swing) to database connectivity (JDBC), communication with native code (JNI), multimedia, etc.

- The Java 2 Platform is not only a runtime platform, but it contains also some **tools** in order to function as a development platform. Besides the compiler it contains tools for packaging, debugging, automatic documentation creation, security, etc.

2.2.3 Distribution

When considering the variety of targeted platforms (hardware as well as operating systems) and application domains, it may be clear that Java ships in

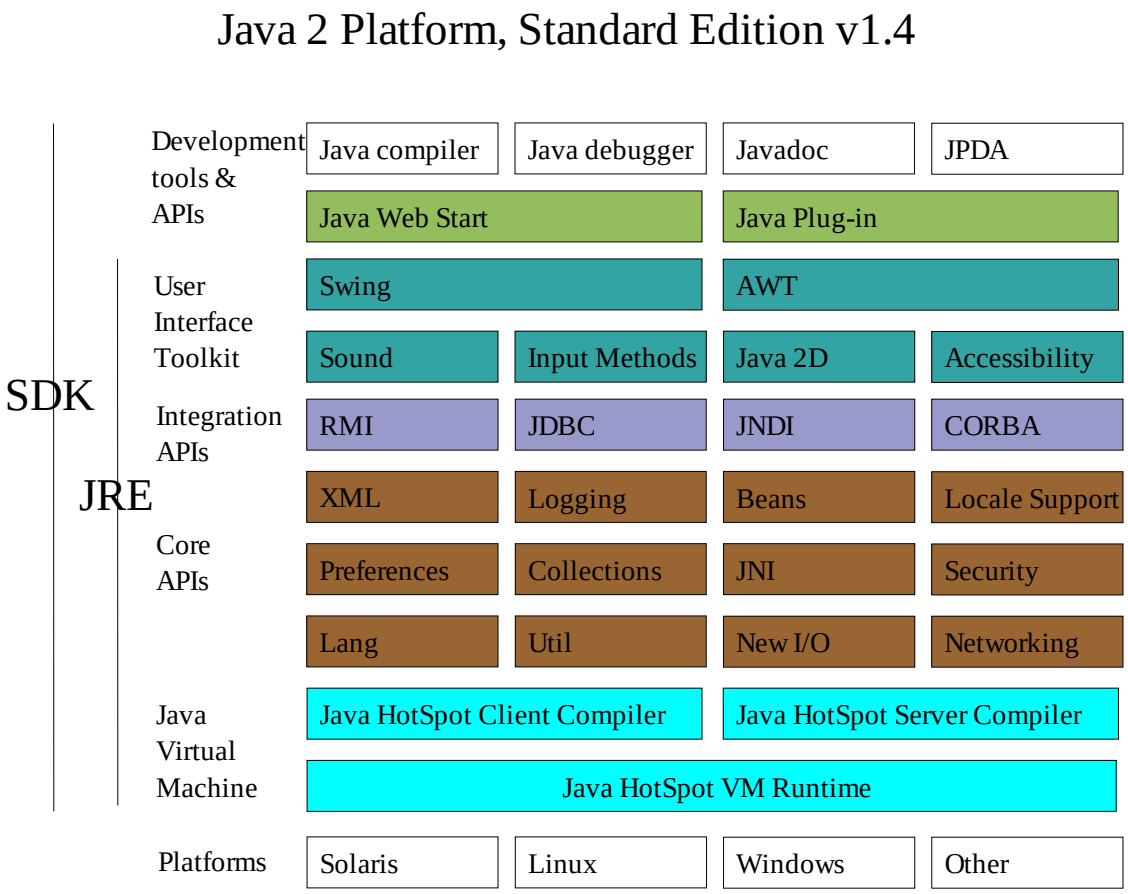

Figure 2.2: Java 2 Platform (image from [Sun03])

different forms, depending on the needs of a specific target. It becomes even more troubling when different parties provide Java versions. Indeed, The Java Virtual Machine Specification [LY99] describes the characteristics of the JVM. Any VM following those rules will be able to read and execute the portable byte code, even if this code is compiled by a compiler belonging to another JVM implementation.

Sun itself splits its distributions in three parts, all containing a VM but different in size, target and possibilities.

- Java 2 Platform, Standard Edition (J2SE). This is the standard Java 2 distribution. This platform is available in two deliverables, one being the Java 2 Software Development Kit (SDK), the other one the Java Runtime Environment (JRE). Developers should download the SDK, as it comes with the compiler, tools and documentation. The JRE only allows to run applications.
- Java 2 Platform, Enterprise Edition (J2EE) is an extended Java platform, meant for developing component-based multi-tier enterprise applications. It comes with an application server and additional APIs.
- Java 2 Platform, Micro Edition (J2ME) targets mobile devices such as PDAs, smart cards, pagers etc. The J2ME platform is adapted to the limited resources available on such devices.

Whereas the specification of the JVM remains the same, concrete implementations will differ in their construction internals depending on the purpose. Also, the libraries and software that comes with the distribution vary in extensiveness.

Besides Sun, other companies ship JVMs too. Examples are IBM's Websphere and BEA's Weblogic, which compete with Sun and others on the enterprise level. They differentiate in the software they run on their application server and in their database connectivity support. The JVM is also a popular platform in the academic world for research concerning performance, extensibility, etc. Kaffe is such a VM, JBoss is an Open Source application server with full J2EE support. Throughout the thesis commercial, research as well as open source JVMs will be used as examples.

2.3 .NET

2.3.1 Microsoft in the 90s

In 1992, Microsoft released Windows 3.1 together with OLE (Object Linking and Embedding). It was originally used for passing data and compound documents. Gradually it grew into a technology for component-based software developing, and together with the Internet-related ActiveX it was renamed to

COM (Common Object Model) in 1997. In the year 2000, COM was again renamed to COM+. The Microsoft Transaction Server (MTS) offered services at runtime: components could now be reused while still in memory and used in distributed systems.

COM had serious problems with dependencies and versioning of components. To address these problems, people at Microsoft started developing a new component platform. While trying to offer solutions for the problems experienced in COM, they reused and evolved the valuable parts of it. COM was usable by multiple programming languages, but each language still had different frameworks.

Apart from guaranteeing backwards compatibility and integration of the technology at that time (COM, DCOM, the common programming and scripting languages at that time), it also had to support new and rising technologies like Web Services, distributed systems and target other devices than the personal computer.

2.3.2 .NET Framework, General Overview

When the developers finished their new platform, the marketing guys came and called the new Microsoft core business strategy .NET, covering clients, servers, tools, services, experience and solutions. Starting from small devices running Windows CE over the MSN services to the Office and Visual Studio suites, everything is covered within .NET.

Now that part of .NET that serves as software platform and thus being comparable to the Java Platform is the .NET Framework, from here abbreviated to .NET. It includes the Common Language Infrastructure (which can be compared to the JVM), a set of libraries, compilers and tools.

The CLI consists of the Virtual Execution System (VES), the Common Type System (CTS), the Common Language Specification (CLS), meta data and a custom file format. Its specifications were submitted to ECMA (European Computer Manufacturers Association) and was standardized in 2001 as ECMA-335, together with the C# programming language (ECMA-334). The Common Language Runtime (CLR) is an implementation of the CLI by Microsoft.

- VES is an environment for executing managed code. Compilers for .NET compile source code into an intermediate form called the Common Intermediate Language (CIL - but Microsoft tends to call it MSIL). This form is then loaded, linked and executed by the VES, which implements the CTS.

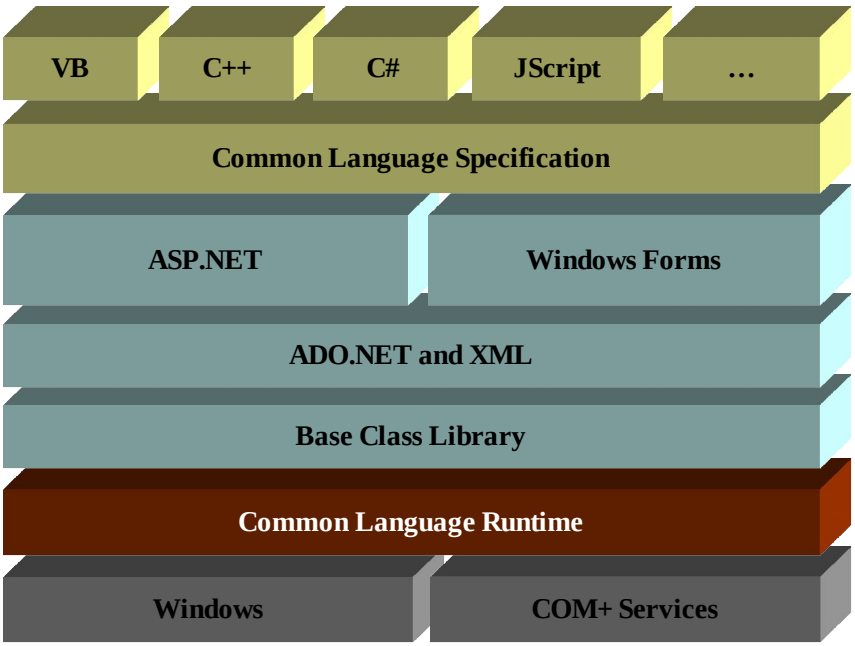

Figure 2.3: .NET Framework

- CTS defines how types are declared, used and managed in the runtime. This universal type system is used by all programming languages that make use of the .NET platform.
- CLS specifies stricter rules than and is a subset of the CTS. Languages with compilers that follow these rules benefit from the full language interoperability the .NET platform offers. These stricter rules allow for example that a class written in C# can inherit from a class written in Visual Basic.NET.

The core libraries are called the Base Class Libraries (BCL) and consist of fundamental building blocks that any application needs: base types, collection, I/O, etc. The libraries are organized in a tree hierarchy based on the CTS. The core resides in the *System namespace*. It is standardized together with the XML libraries (*System.XML*). Other libraries cover the necessary technologies for graphical user interfaces (Windows.Forms), database access (ADO.NET), native code API (P/Invoke), web applications (ASP.NET) etcetera.

Unless Sun, that provides the Java Platform only with a Java compiler (although the platform itself isn't restricted to only that language - see chapter 3), Microsoft ships .NET with compilers for the languages they supported in the past and a new one: C# (pronounce as "cee sharp").

.NET allows you to develop basically three different kinds of applications:

- Standard console or GUI applications, running local and standalone.
- Web Forms: web-based client side applications using the Active Server Page (ASP) .NET web technology.
- Web Service: web-based server side applications also relying on ASP.NET.

Some books that introduce the .NET framework are .NET Framework Essentials [TQ01] and Essential .NET: The CLR [Box02].

2.3.3 .NET, implementations

Despite of crucial parts being standardized and thus freely consultable by anyone, there are only five known implementations of .NET, which will all be briefly introduced below. Again they can be divided in three groups: the commercial ones, the research ones and the open source ones.

- Microsoft .NET Framework, the "original" implementation. While the redistributable version is meant to only execute managed executables, the SDK version adds tools, documentation and development support (you can compare this with JRE and JDK distributions for Java). The standard SDK comes with C#, C++ VB.NET and JScript compilers.

- Microsoft .NET Compact Framework. This framework is targeted for small devices that run on Windows CE. It takes into account the limited memory and storage of those devices.
- Microsoft Shared Source CLI (code name Rotor). This is a educational implementation of the framework by Microsoft which comes with source code. It also runs on other platforms than Windows: Mac OS X and FreeBSD. Rotor contains those parts of the framework that are ECMA standardized plus a remoting implementation, JScript compiler and some tools. The standard license that comes with it is for non-commercial use.
- Mono is an Open Source project sponsored and developed mainly by Ximian [Mon], which is now a part of Novell. One of their primary targets is creating a free and portable .NET framework. Mono targets a series of operating systems (Windows, Linux, FreeBSD,...) and hardware platforms (x86, PPC, S390, SPARC,...). Additionally to the standardized parts, it also implements other parts of the framework that Microsoft has, and even completely new parts. Examples of the former are ASP.NET, ADO.NET, Windows.Forms, VB.NET; the latter one including GTK#, an alternative to the Windows.Forms graphical user interface libraries, Apache, CORBA, ...
- DotGNU, part of the GNU Project. This project aims to be a Free Software solution for Web Services and C# programming. DotGNU exist of three parts:
  - Portable.NET is their CLI implementation [Dot], targeting a whole range of operating systems and hardware platforms too.
  - phpGroupWare is a multi-user web-based GroupWare suite.
  - Web service server DotGNU Execution Environment (DGEE).

  The class library consists of the standardized and some graphical user interfacing libraries.

A summary of which implementations are available on which operating system is presented in table 2.1 , but note that the SSCLI is not usable in a production environment and that the open source implementations are under constant development.

	Windows	*Windows CE*	*MacOS X*	*Linux*	*FreeBSD*	*Solaris*
MS.NET	v					
MS.NET Compact		v				
SSCLI	v		v		v	
Mono	v		v	v	v	v
Portable.NET	v		v	v	v	v

Table 2.1: .NET on different OSs

2.4 Summary

This general introduction already shows that Java and .NET contain quite some similar concepts and that it is justified to compare them. Table 2.2 summarizes the names and concepts introduced in this chapter. From here on, the various parts contained within a VM system will be further researched by means of the criteria presented earlier.

The important part of the platforms is thus the JVM and the CLI here.

part	subpart	*Java part*	*.NET part*
General Name		Java 2 Platform	.NET Framework
Virtual Machine		JVM	CLI
Libraries	Core	JCL	BCL
	GUI	AWT, Swing	Windows.Forms
	Database	JDBC	ADO.NET
	Native	JNI	P/Invoke
Distribution	Runtime	JRE	Redistributable
	Development	SDK	SDK
Description		specified in The Java Virtual Machine Specification [LY99]	ECMA standard [ECM02a][ECM02b][ECM02c]

Table 2.2: Comparing Java and .NET

Chapter 3

Virtual Machine Services

Why would one make use of a virtual machine? Besides some reasons that mostly apply to special occasions already mentioned in the introduction, VMs can be used for most everyday applications too. In this chapter a range of services are discussed which the VM can provide to the developer and applications.

Surely, the presence of these services implies the need of additional resources. The cost of these resources must be weighed out against the profit they can yield. While the former situates in disk space, memory and run-time performance, the latter are benefits that will lead to software that is safer, of higher quality, easier to integrate and portable among others.

3.1 Language Complexity

At the time that Java was designed, C++ was (and still is) very popular. C++ is an object oriented programming language (OOPL) and a superset of C. The language is quite complex [Str00]. C++ served as an example for Java, for language syntax decisions as well as on the conceptual domain. The Java developers restricted the features of their VM because of the simplicity of the language platform they wanted to create.

This simplicity omitted several concepts that were common in other languages and which all had their advocates. Nevertheless, those concepts are not essential and their behaviour can be simulated or replaced by Java structures.

.NET gave many of those concepts back to the programmer. Microsoft wanted to continue supporting existing programming languages they previously had compilers for such as C++, Visual Basic and J++. Thus they kept most of the concepts omitted in Java and created a more complex VM.

The conclusion is that while Java chose for simplicity, .NET included more of the set of C++ language features, partially because of the large part of Windows legacy software written in C++ that still needs to be supported under .NET.

3.2 Language Interoperability

When Microsoft started developing .NET, they not only wanted to continue support for those existing languages they previously shipped, but they wanted a platform that could seamlessly integrate those languages with each other. Language conversion tools, upgrade tools and new compilers take care of old code that can now be executed under the .NET runtime. The result is a platform that supports a lot of the wide range of C++ features (which can be considered as the most complex language among the .NET languages), that can easily integrate managed and unmanaged code and that allows excessive interoperability between those languages (figure 3.1).

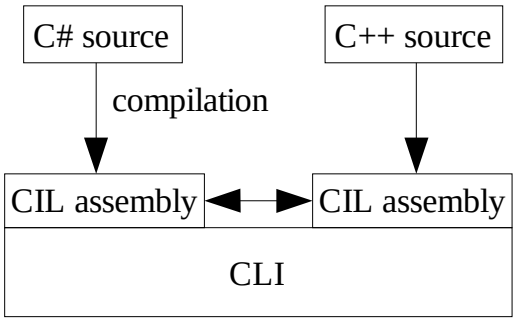

Figure 3.1: .NET language interoperability

Therefore, all of the programming languages are compiled into an intermediate language (IL) format at compile time. Those components written in different languages can directly work together through the IL. Even more, concepts such as inheritance and polymorphism can be used cross-language. While Microsoft ships a couple of languages with its platform, at the other side the Java programming language has always been the sole language that came with the Sun Java Platform.

Nevertheless, Java byte code is targeted too by a whole range of third party programming languages: [VMl] contains a list. There are examples of functional, logical, procedural as well as object oriented languages in this list. Most known are COBOL, Smalltalk, Lisp and Python. .NET too has some success stories with functional and logical programming languages [DHR]. The advantage of .NET is that other language styles than the object-oriented one are supported by the CIL and don't need to be simulated or transformed as happens in Java byte code [J.S03].

3.3 Portable & Architecture Neutral

The nature of the working of the platforms implies that the code compiled to the corresponding intermediate language formats is portable across all VMs created according to the rules. It's the duty of the VM implementation developers to get the VM to work on a variety of hardware and operating systems.

One issue raises here. Different versions of each platform exist, mainly differing in two parts. First, the internal implementation of the VM can change. Because Chapter 7 will dig deeper into this, only a small example will be presented here: over the years the execution engine implementation for Java has been changed from interpretation to JIT-compilation for performance reasons. This factor clearly doesn't restrict portability of the byte code much, as the system will still execute the same IL code.

Second, the libraries. Through the years, new technology is invented and becomes popular. New Java versions include some of this technology in their distributions. Applications developed for newer VMs will not run on older VMs, but need to be compiled again first.

3.4 Modularity

Modules are the building blocks for an application. Those modules are separated entities that can be reused in other applications. A new version of a certain module doesn't require the other modules to be rebuilt.

Imagine for example an application that automatically updates itself via the Internet. Temporary omit the important security issues and concentrate on the ease of use of that application. When a new update is available, only that parts that have been changed are downloaded and installed, without recompilation of the application.

Modularity in Java can take place on the level of classes, but classes can be grouped together in archives too. The same applies to .NET, where assemblies contain one or more types.

3.5 Robust and Security

Robustness and security are two related topics. Indeed, when there is lack of necessary security, then malicious code can try and break down a robust system. While robustness needs to be situated as a property of the IL and the VM, security is rather a control mechanism of the whole platform.

3.5.1 Robustness

Automatic Memory Management The Java developers wanted to get rid of the most common mistakes made by programmers in C++: memory leaks and abnormal behaviour caused by the programmer forgetting to free memory and accessing already freed memory. In order to let the VM manage the memory, it has to keep track of objects, their references and their accessibility. Objects that are no longer accessible must be removed. This concept is generally called garbage collection (GC), and will be treated extensively in Chapter 7 when the various parts of a virtual machine are discussed.

Pointer Arithmetic Pointers allow code to directly access memory. This leads to the programmer having a lot of responsibilities. In C and C++, pointer arithmetic is part of the language and a very welcome feature as it allows programming on a lower level. Moreover, it is a key feature in system software and software where speed really does matter (for instance games and real time software). Garbage collection in combination with pointer arithmetic is a lot more difficult, but not impossible. Other techniques need to be used in that case [JD93].

Java's IL however has absolutely no way for the programmer to access raw memory, no programming language for the Java Platform will thus be able to do it. Pointing randomly in memory hasn't any use either, because the memory allocation and reference model aren't decided until run-time and can be different per VM implementation.

.NET's CIL does provide the necessary instructions, keeping in mind that it is a target for C++ among others. It depends on the programming language to decide whether it allows pointer arithmetic or not. For example VB.NET and J# don't, Managed C++ and C# do. This is probably the right moment to explain the different flavours of C++ involved in .NET.

Intermezzo: C++ in .NET

First, there is the distinction between managed and unmanaged code. Unmanaged code (or native code) is all code that is not compiled to IL and

is not aware or makes use of the .NET Framework. Managed code is code that is under the control of the CLI.

The proper name for C++ controlled by .NET is Managed Extensions for C++ (also called Managed C++). It combines the features of the .NET Framework with the power of the C++ language. It allows the programmer to mix managed and unmanaged code, even in one single source file. This makes it possible to reuse existing C++, but also to gradually port legacy code to Managed C++ and add .NET support to it.

Managed C++ has additional keywords on top of standard C++, meant for instructing the CLI. For example, the programmers have to indicate which classes must be managed by the CLI and which will reside in native and unmanaged memory. Moreover, not all C++ code can be compiled to managed code because of differences in the object model between standard C++ and .NET. For example, .NET doesn't have multiple inheritance or templates. There is also the memory model: .NET doesn't allow types to be allocatable on both stack and heap.

At run-time, the CLI will switch between unmanaged and managed mode when executing mixed applications. Data will be spread in the memory, and the code will alternately run native and JIT-compiled. An introduction to Managed C++ can be found at [Man].

Type-safety Related to the paragraphs above, type-safety means that the programmer can only manipulate objects via the operators that are defined on the type the object is associated with. Both Java and .NET are type safe, no illegal casts can be done. This type-safety can be checked by the verifier before run-time. Not allowing pointer arithmetic is one of the factors to obtain a type-safe language.

Exceptions Another concept that adds to the robustness is exception handling. It is a type of flow control and error handling that passes control and information to a handler when a certain events happen. The actions that happen in this context are the *throwing* and the *catching* of exceptions. An exception is thrown from the point where it occurred and is caught at the point to which control is transferred. Although exceptions can be defined, thrown and caught by the programmer, the focus lies here on exceptions thrown by the virtual machine itself when an erroneous application is being executed.

Those exceptions can be foreseen by the application and caught after which it can handle the exception and continue execution. Else, if the exception is unhandled, the runtime will abort the application's execution and shut down

itself in a clean way. The exception concept is a feature that exists in both the Java and .NET platforms, providing an elegant alternative for the well known C and C++ segmentation faults and core dumps.

3.5.2 Security

Because both platforms are used in distributed environments where applications are transferred through networks (in particular the Internet), they need clear and configurable security measures.

.NET The security in .NET can be separated into two parts ([FT00] focusses on more security aspects): a set of policies that restrict applications depending on their properties and embedded security requirements. The latter are embedded in an application and tell the CLI which rights the application need in order to execute correctly.

Security policies are set by the application's user. They restrict the rights of applications based on the physical location of the code, the publisher, the URL, etc . File I/0, GUI, web, network or database access, reflection are all protected and can only be used if the application satisfies the policies.

Java A good example of the security in Java are the precautions taken for applets. When someone browses the Internet and accesses a web page with a Java applet, the origin and creator (and more important: his intentions) of the applet are unknown. The risks involved when accessing this page would be huge when the applet could do anything a normal local application is able to do.

For this reason, the applet is executed in a sandbox, which limits its rights. An applet will not be able to directly interoperate with some parts of the virtual machine, in order to prevent the applet to load its own custom VM parts. A verifier will check the class file for malicious code. Finally Java's security manager comes into play. It can be compared to the security policies of .NET: it allows or disallows specific operations to the applet, such as I/O, network access, external program execution or shutdown of the JVM.

Of course, this scenario for an applet counts for Java applications too, but applets are considered as very dangerous and have the most strict security measures. Normal applications will run as regular applications without a sand boxed environment.

Security goes further than preventing attacks: the security measures can ensure confidentiality of data, integrity of applications and authentication of the origin. Applets can be digitally signed using a system of public and private keys. Sun wrote a white paper about Java and security [S.F96].

3.5.3 Reflection and Dynamic Class Loading

Reflection is the ability to access internal information about types loaded into the JVM at run-time. It not only allows to get information, but also to use this information to call methods for example and to create new types. The reflection API is an interesting tool when dealing with information that is not known until run-time. The information needed for reflection is present in the types' meta data. Related to reflection, dynamic class loading allows loading a class file from disk at run-time and instantiate and using its contained type.

3.5.4 Thread and Synchronization Support

The Java and .NET VM thread support is part of the language with keywords and APIs. This stands in contrast with C++, where threads are no standard language feature and additional libraries are needed in order to use them. In these VMs however,

The thread constructs are rather simple, the VM is responsible to handle the real switch and lock work. It must map those high level threads onto system specific ones. Threaded VM code is thus immediately portable too.

3.6 Summary

- Both Java and .NET are mostly inspired upon C++, with .NET retaining more of the complex language structures C++ contains. The more complex structure of .NET originates from the continuous support that Microsoft wants to offer to existing software.
- The concept of an intermediate language is related to a VM, resulting in interoperability between languages that are compiled to that IL. On top, the use of an IL makes an application immediately portable.
- A VM can provide security, both in terms of protection against malicious software from the outside as well as mechanisms that avoid the most common bugs made by developers, bugs that are memory related and can corrupt an application's code and data.
- Another aspect is the possibility to load, create and change classes at run-time on demand of instruction by the running application.

As seen in this chapter, VM technology certainly offers additional benefits in comparison with native execution.

Chapter 4

Intermediate Languages

In order to make the source code suitable for execution in a virtual machine, it will be compiled into an intermediate language (IL). This language is the only one the VM understands. The concepts and properties of this language and its instruction set are the subject of this chapter. Sometimes the link will be made with the Java and C# programming languages, to clarify the impact of the intermediate design decisions. C# has been chosen because it is standardized, the other languages provided by Microsoft aren't. Second, as C# is the native and new language that comes with .NET, it's the language that profits the most of the CIL functionality.

From here on the .NET Common Intermediate Language will be called CIL, the Java counterpart Java IL will be called JIL. These languages determine the expressiveness of the whole platform: languages that target the platform must be compilable to this IL. The CIL's syntax and semantics are strictly defined as partition III of the ECMA-335 standard [ECM02c]. In contrast, there has not been defined a syntax for the Java Intermediate Language, except for some mnemonics for the instructions mentioned in the Java Virtual Machine Specification [LY99].

In Java, the relationship between the programming language Java and JIL is actually a mapping: each language construct corresponds with one or more JIL instructions and the other way around. Compilation of other languages than the Java programming language to JIL was initially not foreseen. Besides, compilers for the other languages are created by third parties and not supported by Sun.

The CIL instruction set is richer, and no programming language for .NET uses the full set of instructions, as shown in figure 4.1.

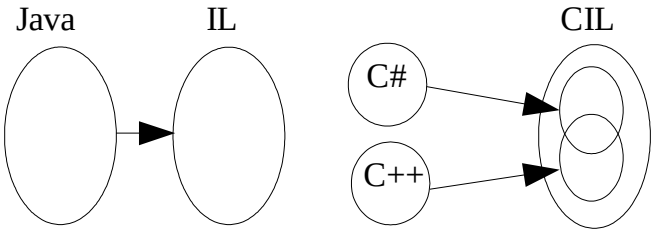

Figure 4.1: Usage of the IL instruction set by the programming languages

4.1 Object-Oriented

Both platforms support the object-oriented paradigm: the concepts of classes, objects, abstraction, inheritance and polymorphism are known in the IL. C++ is a multi-paradigm programming language, which allows the programmer to circumvent the provisions for object-oriented programming. It can then be used as an imperative programming language and constants, variables and functions can be declared global. Java and .NET only support the object-oriented paradigm. But, with other paradigms in mind, .NET provides some structures that help, together with a lot of simulation work, logical and functional languages to be compiled to CIL.

Pure object-oriented languages are languages that fully and uniquely support the object-oriented paradigm. Each author seems to express the requirements for a pure OOPL differently, but the following description by Hunt, together with the more traditional object-oriented concepts will clarify the idea. [Hun97, A pure object-oriented language only supports the concept of an object. Any program is made up solely of interacting objects which exchange information with each other and request operations or data from each other.]

Examples of pure OOPLs are Eiffel, Smalltalk and Ruby. Table 4.1 denotes why C++, Java and C# can not be qualified as pure object-oriented. They are thus all three hybrid languages.

Property	*C++*	*Java*	*.NET*	*Smalltalk*
Encapsulation	Yes	Yes	Yes	Yes
Inheritance	Yes	Yes	Yes	Yes
Polymorphism	Yes	Yes	Yes	Yes
All pre-defined types are Objects	No	No	No	Yes
All operations are messages to Objects	No	No	No	Yes
All user-defined types are Objects	No	Yes	No	Yes

Table 4.1: Pure Object Orientation: Properties

4.2 Types

4.2.1 Java type system and hierarchy

Figure 4.2 shows the different types of the Java IL. The value types are the set of primitive types consisting of the integral types *byte* (8 bit), *short* (16 bit), *integer* (32 bit), *long* (64 bit), the floating-point types *float* (32 bit) and *double* (64 bit), and the char type representing Unicode characters. The latter ones are encoded according to the IEEE 754 single and double precision standard.

In most cases instances of the *byte*, *short* and *char* type are mapped onto an *int* before computations. The *boolean* type of the Java programming languages is always represented as an integer 0 or 1. The only notion of boolean in the instruction set is the possibility to create an array of booleans, but even those entries are accessed by using instructions for shorts.

Another primitive type is the internal JVM type used for storing pointers to opcodes in the code: the *returnAddress* type. This type is only used in instructions that deal with exception handling.

At the other side there are three kinds of reference types: *class* types, *interface* types and *array* types. The value of those types is a reference to class instances and arrays, or class instances that implement interfaces.

java.lang.Object forms the root of all objects. All classes inherit from this class and it can thus be used as polymorphic class for all objects. This hierarchy (figure 4.3) is the basis of standard libraries, but also a starting point for user created types.

The primitive types are not part of the object hierarchy, which causes problems when variables of primitive types need to be used in the Java collection libraries or in other places where polymorphic behaviour on primitive types is desired. Therefore, each primitive type has a corresponding wrapper that is part of the object hierarchy.

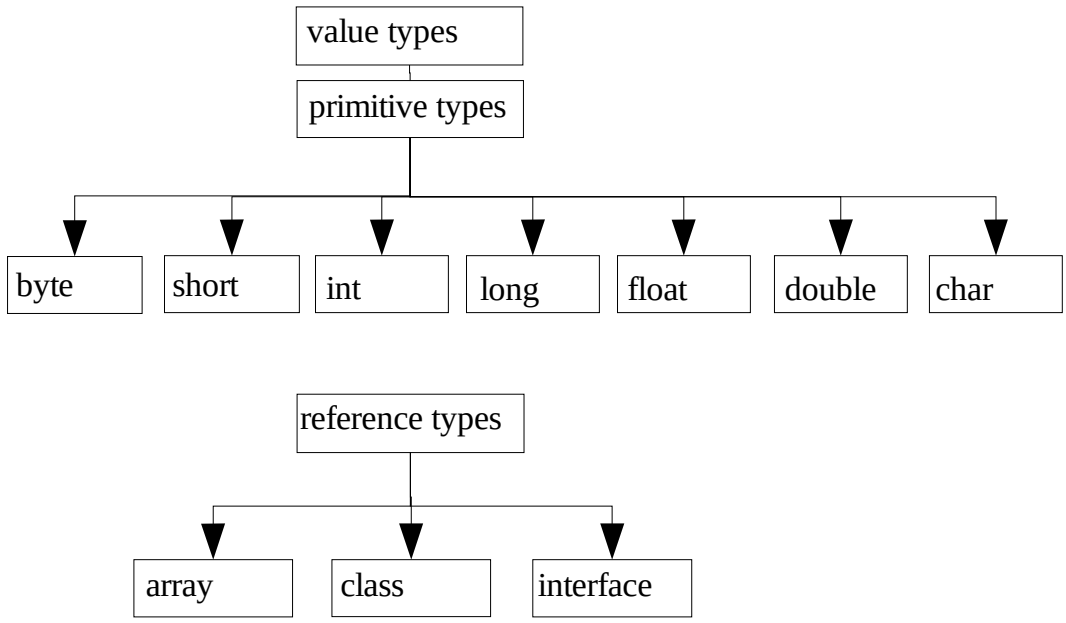

Figure 4.2: Java Types

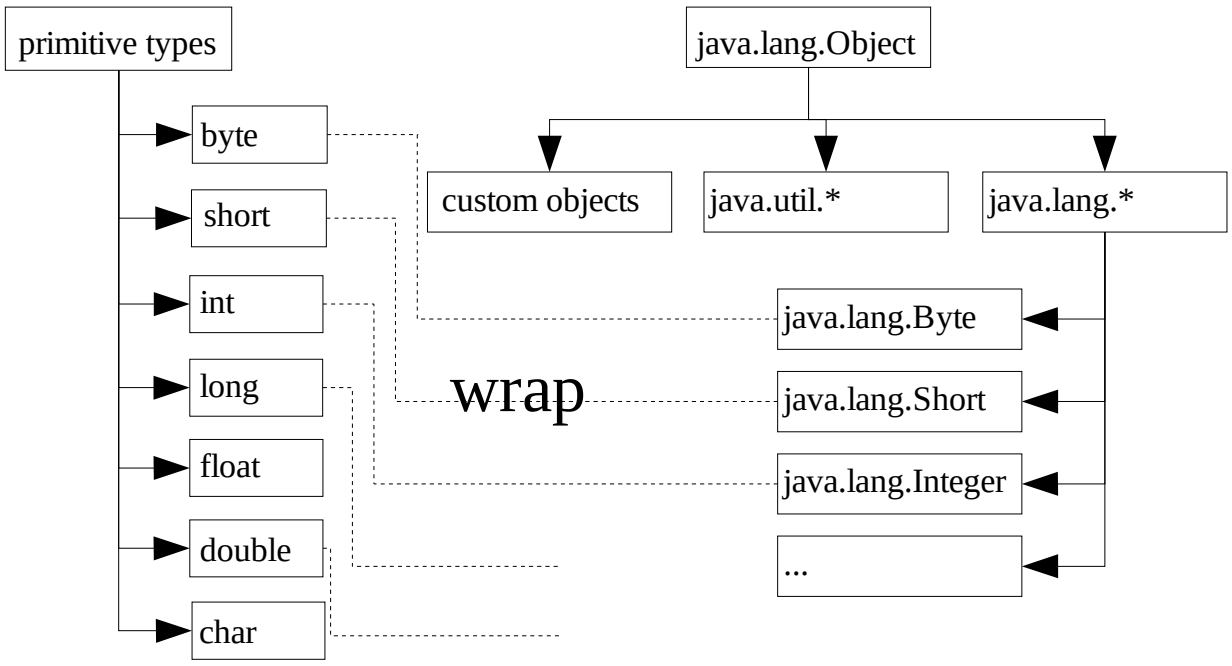

Figure 4.3: Java Object Hierarchy

4.2.2 .NET type system and hierarchy

The specification of the type system in .NET is the task of the CTS: how are types declared, what is their semantics and usage, and how are they managed by the CLI.

Primitive Types .NET has the same set of integer types as Java, but each type has both a signed and an unsigned version. The floating point types are the same in size and layout as the Java ones, the *char* type is also in Unicode format. Besides these, there is a boolean type, a type reference and a pointer-size integer, which represents the size of a native integer. Important to notice is the fact that those primitive type names are in fact aliases for some *structs* in the type hierarchy.

.NET contains two pointer types too: a managed one and an unmanaged one. The former will be guarded by the garbage collector, the latter not. Furthermore there are user defined value types in .NET: *structs* are a kind of lightweight classes and *enumerations* are types containing named values.

Reference types are divided into *interfaces*, *pointer types* and *self-describing types*. Interfaces are work the same way as they do in Java, pointer types are all types which are referenced by a pointer - the original type can be either a value type or a reference type. Finally, a self-describing type is called that way because its type can be determined from its values. Arrays and classes are self-describing. An overview of the .NET type hierarchy is presented in figure 4.4.

As opposed to Java, the primitive types are part of the Object hierarchy in .NET (figure 4.5). Or rather: when needed, they will be made part. As already briefly mentioned in Chapter 3 those primitive types, which are in fact already structs, will be *boxed* into objects when needed. This procedure of converting a value type into a reference type is defined for every value type and takes three steps:

- First, memory is allocated from the heap, with a size large enough to hold the value type data and additional information for objects.
- The value type's data is copied to the heap.
- The object's address is returned and can now be accessed via a reference

Every value type that is boxed into a reference type is a descendant of *System.Valuetype*. More specifically, enumerations will be descendants of *System.Enumeration*. At the reference side, all delegates have *System.Delegate* as parent node.

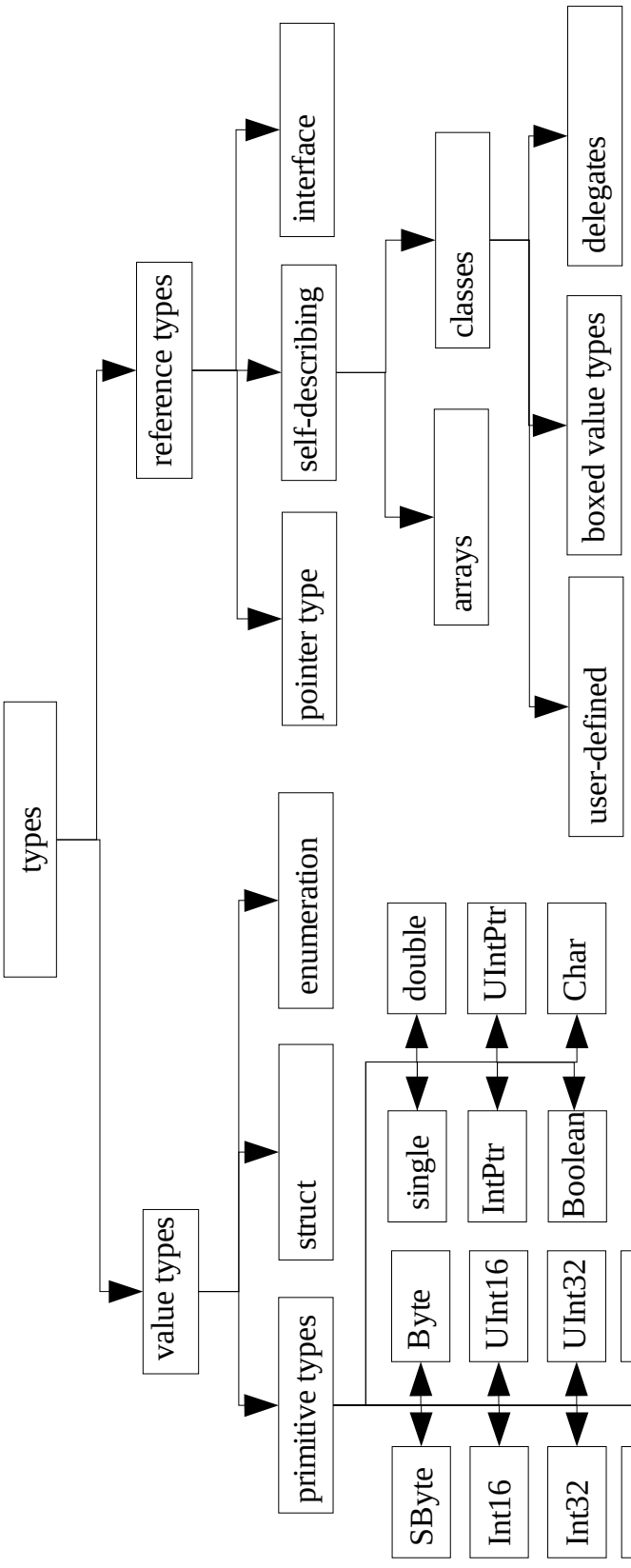

Figure 4.4: .NET Types

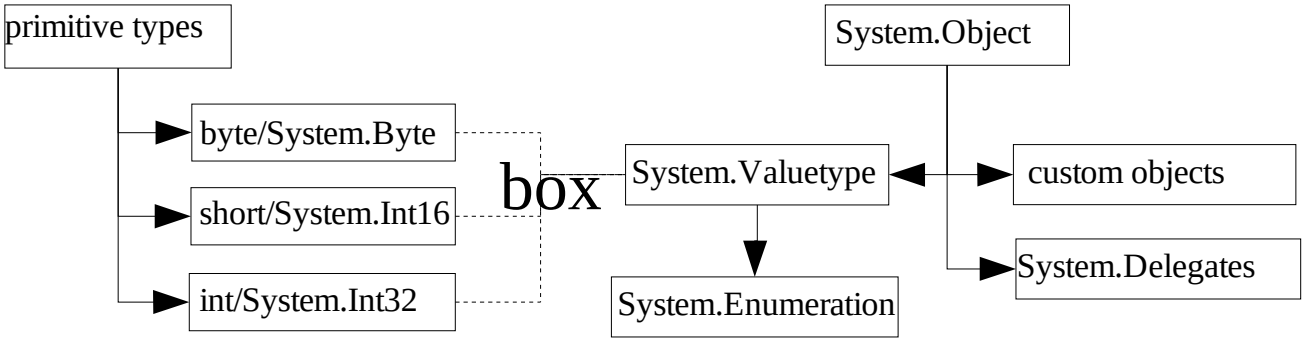

Figure 4.5: .NET Object Hierarchy

All types except for interfaces are derived via some way through the hierarchy from *System.Object*. This implies that in .NET the primitive types can directly be used in for example the collection libraries. *System.Object* forms the root of all types.

4.3 Intermediate Language Properties

Strongly typed Both Java and .NET are strongly typed. This property allows the compiler to statically check for many mistakes and because of the additional knowledge also to better optimize the code.

Array boundary checking Checking the array boundaries help in creating a secure and robust system. Corrupting memory by passing the allocated memory for the array will not happen, and the exception mechanism will start working when trying to access a non-existing array index.

Extra CIL features Features that Java IL lacks are for example unboxed structures and unions, enums, reference parameters, variable argument lists, multiple return values, function pointers, overflow sensitive arithmetic, operator overloading, tail calls, etc [MG00]. .NET contains all of those but multiple return values.

Some of this interesting CIL functionality is described below.

- **Delegates** are type safe function pointers.
- **Attributes** allows to store random information about classes, methods, etc. into assemblies

- **Reference parameters** allow variables of primitive types to hold the values they are assigned in the called method when the program returns back to the calling method.
- **Operator Overloading** is a mechanism that lets the programmer add new data types to those that the language operators are designed to handle. The overloading is only available for class types. Java does not allow operator overloading, in contrast with C++ and C#.
- **Reference parameters**: parameters of a value type can be passed by value or by reference.
- **Tail calls** optimize calls of a certain method at the end of another method. Instead of double allocation and deallocation of stack frames, the subroutine call is replaced by a goto. This is especially usefull in case of functional programming.

CIL contains clearly more features, but the fact that platforms don't offer those doesn't stop languages which do have some of the features from targeting the platforms (e.g. C++, Eiffel). Source-to-IL compilers can transform the source code into a suitable form, or an extension to the platform can be used. But, with language interoperability in mind, those features in the IL would allow to share their potential between all the languages.

4.4 Instruction Set

Intermediate language instructions consist of an opcode identifying the instruction, followed by zero or more operands (figure 4.6). A Java instruction is composed of a one byte opcode with optionally one or more operands. The operands can be generated at compile time and embedded in the instruction, or they can be loaded at execution time.

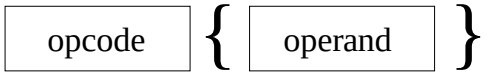

Figure 4.6: IL instruction composition

CIL provides a similar amount of instructions, but the opcode is not restricted to one byte: some instructions have a two byte opcode. A system with only one byte opcodes can theoretically provide up to 256 instructions. Since not all one-byte opcodes are in use in .NET, the allowance of two-byte instructions implies the possibility to extend the instruction set in the future. The VM can reserve some opcodes for internal use only too. The set is divided into a couple of categories containing related instructions. It can be clearly seen in Table 4.2 that the number of CIL instructions per category generally agrees with the number of JIL instructions.

Instructions	*Java*	*.NET*
Arithmetical	42	28
Conversion	15	34
Flow Control	20	27
Load and Store	66	52
Stack operations	9	2
Fields and Methods	14	12
Addressing Classes and Value Types	7	20
Array Instructions	20	22
Other Instructions	11	16
Total	204	213

Table 4.2: Number of instructions per category

Arithmetical instructions Arithmetical instructions are the instructions that work on numerical data: basic mathematical operations, bitwise and shift operations. The Java instructions have the type they work on embedded in the instruction (contained in the opcode), while .NET provides a single instruction and type-variable operands.

An example is shown in Table 4.3. While the operands of the Java instructions must be of integer type, the operands of the .NET instruction must adhere to the type rules for binary numeric operations, which is described in Partition III: CIL Instruction Set of the ECMA standard [ECM02c]. Note that neither of those two instructions is checked for overflow. .NET provides extra instructions that do perform the check and throw an exception in case of.

Platform	*Instruction name*	*Opcode*	*Operand(s)*	*Result*
Java	*iadd*	0x60	ivalue1 ivalue2	iresult
.NET	*add*	0x58	value1 value2	result

Table 4.3: an example instruction

Conversion Instructions When considering the instructions that convert values of simple types to other simple types, it's once again .NET that has a lot more facilities. One can convert between 10 types of integers and 2 types of floats, nearly without restrictions. Converting to a narrower type means applying truncation, converting to a wider type will lead to sign- or zero-extension. For all the instructions with an integer resulting type, there are equivalent instructions that check for overflow, just like in the case of the mathematical instructions. Java follows the same style as mentioned before: type-embedded instructions.

Flow control instructions Flow control covers instructions that jump to various places in the instruction list. Those are the instructions that loops and conditional blocks are compiled into. Java has branch instructions that branch after comparing between a value and zero and between two values. .NET omits the comparison with zero, but provides short forms. Those forms contain a one byte branch value instead of four bytes, reducing memory usage but limiting the possible jump distance too. The switch instruction is special in both languages as it carries, as the only instruction in the set, a variable amount of arguments in the instruction itself.

Load and Store Both Java and .NET make use of a per method stack based operation system. There are memory areas for local variables and parameters. Those variables are loaded on the stack, a Last In First Out (LIFO) memory structure, when needed. Figure 4.7 shows a simple Java example that makes use of local variables and the stack.

It starts with an empty stack and two local variables. Both are loaded on the stack using the *iload_1* and *iload_2* instructions, which are shortcuts for the more generic *iload* instruction which expects an operand containing an index in the local variable list. Next the two values are popped from the stack and the result of the addition is pushed on the stack again, after which the *istore_1* instruction stores the result back in the first local variable.

The example showed here was in Java, but an equivalent .NET example would be very similar.

Stack operations Duplication, swapping and popping the top stack variables are the pure stack instructions of JIL. CIL only allows duplication and popping. These instructions differ from the previous ones in the fact that they work on the stack only, and not on local variables.

Fields and Methods Which instructions do exist for method calling and field access? Are there different instructions for static members, virtual methods and interface methods?

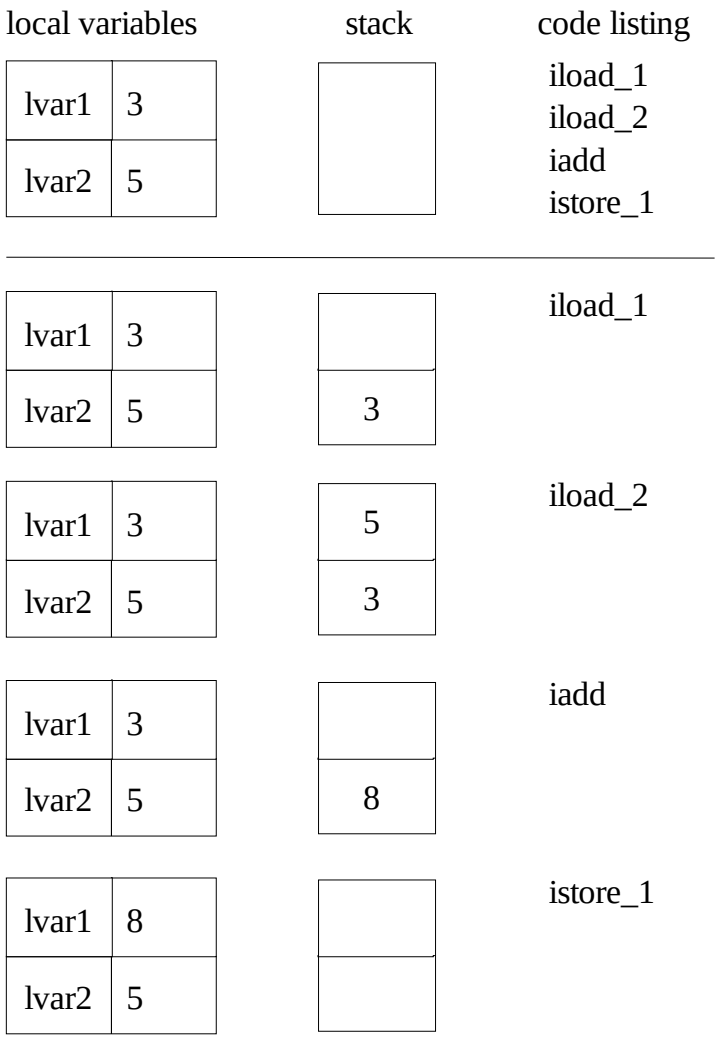

Figure 4.7: instruction stack usage

Java has 4 method call instructions.

- *invokevirtual* is used for normal methods. Remember that all methods are virtual in Java by default. JIL separates static and non-static members.
- *invokespecial* is only used in some special cases: for calling the instance initialization method, calling private methods and methods in super classes.
- *invokeinterface* calls methods defined in interfaces. An additional type check is needed to see whether the given object reference does implement the given interface.
- *invokestatic* calls static methods.

The field access instructions separate between static and instance fields.

call and *callvirt* are the instructions used in .NET. Because the method signature contains information whether the method is static or not, there are no different instructions for class and instance methods in CIL. However, there do exist two instructions for virtual and non-virtual methods. One of the arguments for this instruction is a class name, but it doesn't necessary represent the class where the method is defined. It is rather an indication to search in this class and its base classes.

Addressing Classes and Value Types These instructions manipulate class and value type instances. The classes and types itself are typically defined as tokens in CIL and as entries in the Constant Pool in Java. Both methods refer thus to meta data. Some of the typical instructions present here are instructions for creating, casting and checking objects.

The .NET instructions for storing and loading from raw memory are put here too, something Java does not have.

Array Instructions Instructions manipulating arrays are subdivided here. JIL has again per type-embedded instructions for loading onto the operand stack and storing into array. As seen in the previous section, arrays are objects in Java and .NET. They are type safe and information about name, rank, and length can be retrieved.

Other Instructions Exception handling instructions, breakpoints, thread lock instructions are the remaining instructions not assigned to any other partition.

It may be clear that although CIL doesn't have a lot more instructions, it does have more instructional power. It has also support for native types, pointers and references in unsafe mode, raw memory access, etc . At the other side almost all JIL instructions have a matching CIL instruction with the same semantics.

A complete overview for the Java instruction set can be found in The Java Virtual Machine Specification [LY99]. For the .NET instruction set, both the ECMA CIL standard [ECM02c] and Lidin's Inside Microsoft IL Assembler book [S.L02] provide all details. Other authors compared both the Java and .NET instruction sets to each other too [Gou01].

4.5 Summary

- Both Java and .NET support the object-oriented paradigm, but not pure.
- .NET allows the programmer to create custom value and reference types, Java only the latter.
- Value types are part of the object hierarchy in .NET by a technique called *boxing*. This can be seen as an improvement on the wrapper classes Java offers for its primitive types.
- The instruction sets of both platforms are quite similar. .NET has additional features like raw memory manipulation, a tail call instruction, native integer instructions, etc.

Chapter 5

File Formats

When the compilation process from source code to IL has been done, the IL and the meta data belonging to it are stored in file on disk, waiting for an execution of this code at run-time. Both Java and .NET work component-based. The division in components is always per class in Java, while .NET allows creating larger components named *assemblies*. The meta data describe the content of the component where they belong to, a concept called self-description.

5.1 Java Class File Format

Each class file represents one Java class, in conformity with Java source where each public class has its own file. Inner classes and classes that are not public will be compiled into their own class file too. Class files are mainly divided into four parts: initial bytes, the constant pool, flags & "pointers", and the class properties. A schematic representation is given in figure 5.1.

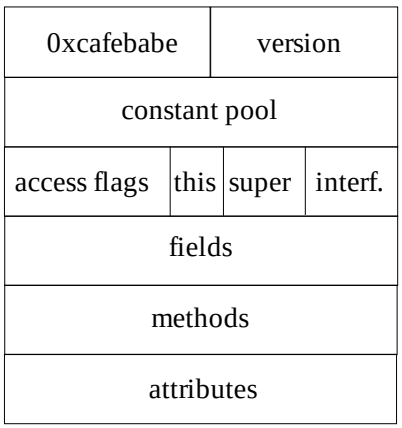

Figure 5.1: Class File Format

5.1.1 General Structure

- The first 4 bytes of a class file always are Oxcafebabe. These magic bytes represent a class file. Next are bytes that describe the compiler version.
- The constant pool contains a list of constants: names of classes, methods, variables, but also names of attributes present in parts of the class file. Instructions refer to the information in this constant pool. Each entry consists of a tag, which defines the entries' type, the length and finally the content itself. Example tags are Class, Fieldref, Integer, Utf8, ...
- The next part contains the classes' access modifiers and the fields *super class*, *this* and *interfaces* which point into the constant pool, to the constants containing the name of the super class, the name of the class itself, and the names of the interfaces this class implements.
- Next are the field and method information of this class, which both have a list of attributes. An attribute is an additional variable length structure belonging to the method or member. A method has a Code attribute, wherein the instruction list for that method is stored. Other attributes examples are the exception attribute, indicating which exceptions a method may throw.

5.1.2 An example class file

```
1 public class HelloWorld {
2         public static void main(String argv[]) {
3                 System.out.println("hello, world!");
4         }
5 }
```

Figure 5.2: Java source example

An example of a Java source file is shown in figure 5.2, the corresponding class file is dumped in hexadecimal form in figure 5.3. This file is the representation of the simple program writing "hello, world!" to the command line. On the left is the hexadecimal representation, on the right the ASCII equivalent.

The first few hexadecimal numbers are recognizable as the magic word. The constant pool counts 29 ("00 1d") entries. The first entry ("0a 00 06 00 0f") is a Method Reference which points to the sixth entry of the constant pool for the class to which it belongs and to the fifteenth entry for its signature. Those point on their turn to entries that contain constants. The method described here is thus the contructor of *java.lang.Object*, the superclass of this class file's HelloWorld class. This example is schematically represented in figure 5.4. The

```
000  ca fe ba be 00 00 00 2e  00 1d 0a 00 06 00 0f 09  ................
010  00 10 00 11 08 00 12 0a  00 13 00 14 07 00 15 07  ................
020  00 16 01 00 06 3c 69 6e  69 74 3e 01 00 03 28 29  .....<init>...()
030  56 01 00 04 43 6f 64 65  01 00 0f 4c 69 6e 65 4e  V...Code...LineN
040  75 6d 62 65 72 54 61 62  6c 65 01 00 04 6d 61 69  umberTable...mai
050  6e 01 00 16 28 5b 4c 6a  61 76 61 2f 6c 61 6e 67  n...([Ljava/lang
060  2f 53 74 72 69 6e 67 3b  29 56 01 00 0a 53 6f 75  /String;)V...Sou
070  72 63 65 46 69 6c 65 01  00 0f 48 65 6c 6c 6f 57  rceFile...HelloW
080  6f 72 6c 64 2e 6a 61 76  61 0c 00 07 00 08 07 00  orld.java.......
090  17 0c 00 18 00 19 01 00  0d 68 65 6c 6c 6f 2c 20  .........hello,
0a0  77 6f 72 6c 64 21 07 00  1a 0c 00 1b 00 1c 01 00  world!..........
0b0  0a 48 65 6c 6c 6f 57 6f  72 6c 64 01 00 10 6a 61  .HelloWorld...ja
0c0  76 61 2f 6c 61 6e 67 2f  4f 62 6a 65 63 74 01 00  va/lang/Object..
0d0  10 6a 61 76 61 2f 6c 61  6e 67 2f 53 79 73 74 65  .java/lang/Syste
0e0  6d 01 00 03 6f 75 74 01  00 15 4c 6a 61 76 61 2f  m...out...Ljava/
0f0  69 6f 2f 50 72 69 6e 74  53 74 72 65 61 6d 3b 01  io/PrintStream;.
100  00 13 6a 61 76 61 2f 69  6f 2f 50 72 69 6e 74 53  ..java/io/PrintS
110  74 72 65 61 6d 01 00 07  70 72 69 6e 74 6c 6e 01  tream...println.
120  00 15 28 4c 6a 61 76 61  2f 6c 61 6e 67 2f 53 74  ..(Ljava/lang/St
130  72 69 6e 67 3b 29 56 00  21 00 05 00 06 00 00 00  ring;)V.!.......
140  00 00 02 00 01 00 07 00  08 00 01 00 09 00 00 00  ................
150  1d 00 01 00 01 00 00 00  05 2a b7 00 01 b1 00 00  .........*......
160  00 01 00 0a 00 00 00 06  00 01 00 00 00 01 00 09  ................
170  00 0b 00 0c 00 01 00 09  00 00 00 25 00 02 00 01  ...........%....
180  00 00 00 09 b2 00 02 12  03 b6 00 04 b1 00 00 00  ................
190  01 00 0a 00 00 00 0a 00  02 00 00 00 03 00 08 00  ................
1a0  04 00 01 00 0d 00 00 00  02 00 0e                ...........
```

Figure 5.3: class file dump

whole constant pool is made out of entries referring to each other or to constant entries.

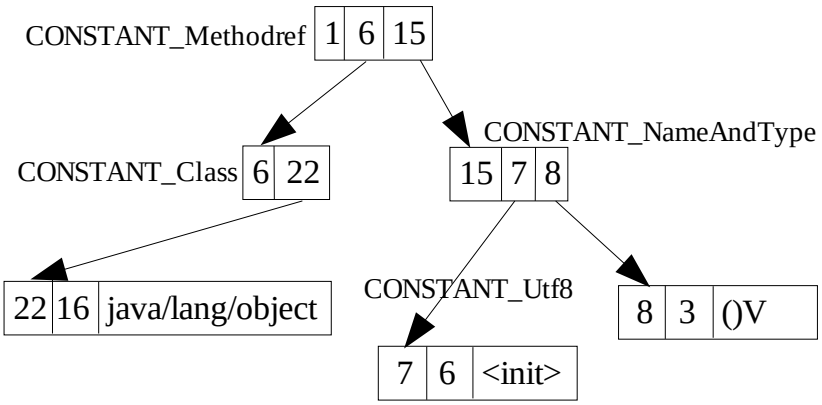

Figure 5.4: Constant Pool Example

Next the two methods in this class are give, the constructor en the main method. Their most important attribute is the Code attribute. Method main has the code array "b2 00 02 12 03 b6 00 04 b1", which translates to the instructions in figure 5.5.

```
0:   getstatic       #2; //Field java/lang/System.out:
                                 //Ljava/io/PrintStream;
3:   ldc             #3; //String hello, world!
5:   invokevirtual   #4; //Method java/io/PrintStream.println:
                                 //(Ljava/lang/String;)V
8:   return
```

Figure 5.5: an method's code array

Note that classes and interfaces are always being represented in their fully qualified form, the namespace separator being '/' and not '.'. Field and method descriptors describe the type of a class, instance, local variable or method. A grammer describes which sequences of characters can form a valid descriptor. The base types are represented using a simple type, reference types must be named after an 'L', Arrays are denoted with a [. An example can be seen above: the "java/io/PrintStream.println:(Ljava/lang/String;)V" string represents the Java "println" method containing one argument of type String and returning nothing (void). The complete syntax for those descriptors can be again be found in The Java Virtual Machine Specification [LY99].

5.2 .NET PE File Format

The Microsoft Portable Executable and Common Object File Format (PE and COFF), described in [Cor99] was a file format introduced by Microsoft with Windows NT. It's meant to be portable across all 32-bit Windows operating systems. The format is almost identical on the hard disk as when it is loaded in memory for execution.

The .NET file format forms an extension [Cor00] on that file format, with a description of how the CIL and meta data must be stored. A PE file contains various sections, the managed module that the .NET compilers create forms an extra section. The operating systems treats the file as a normal executable, but while reading the file, the OS will find out that this is a managed module, and pass control to the CLR. Three main parts are distinguished: operating system-specific headers, the CLI part and native sections.

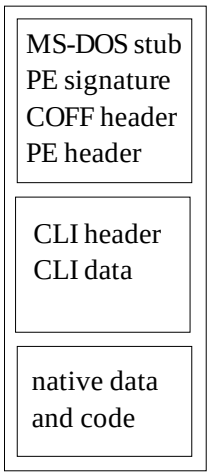

Figure 5.6: PE File Format

The operating system specific parts start with a MS-DOS stub (remember that exe's already existed in the MS-DOS era), which points the control to the next header, the Common Object File Format (COFF) header. When trying to execute a Win32 executable in DOS, an error message, contained in the PE file, will be printed. Otherwise, the operating system will pass control to the .NET implementation on the system. The COFF header contains general

information, most fields not used in combination with a managed module. The characteristics field indicates the presence of an image file and whether the file is pure-IL managed or contains native code too.

Next header is the PE header, providing information for the OS loader. One of the entries in this header contains the relative address and size of the runtime header in the image file. This header is put in a readable, sharable section of the image file, the .text section. This .text section furthermore contains meta data tables and the IL method code, which are parts of the CLR data section of a PE file. The general classification of the PE file format is shown in figure 5.6.

In the native section unmanaged code can be stored. This way a PE file can contain a mix of managed IL code and unmanaged native code. This is usefull in conditions where legacy code is used in combination with new code, as mentioned earlier when discussing the possibilities of C++ in .NET.

A module contains one or more types, meta data and code. In contradiction to Java class files, .NET allows multiple modules to be included into one assembly. An assembly forms thus a logical collection of modules. Assemblies are described by Box as "atoms of deployment" in [Box02]. Indeed, .NET distributes, versions, packages and loads applications and libraries in assemblies.

5.2.1 General Structure of the CLI part in the PE file format

This part starts with the CLI header, containing the application's entry point, the size and location of the meta data, the strong name signature, etc. The meta data part describes the managed contents of this assembly: its modules, its classes, references and methods. Again the IL code is ordered as code arrays per method.

The String heap section can be compared to the Java constant pool: constants containing the names of methods, classes, interfaces. The meta data makes use of these constants. Behind that comes the user string section, which gathers user defined string constants.

Meta data tables refer to the heaps: for example a method table entry contains a reference to the String heap for its name and to the Blob (binary large object) heap for its method signature.

5.2.2 Example PE File

Figure 5.7 contains the C# code corresponding with the Java example. The IL assembler equivalent for the Main method is showed in figure 5.8.

```
using System;
public class HelloWorld {
        public static void Main(String[] argv) {
                Console.WriteLine("Hello, World!");
        }
}
```

Figure 5.7: C# source example

```
.method public static void Main(class System.String[] argv)
                cil managed
        {
                // Start of method header: 20f4
                .entrypoint
                .maxstack  8

                ldstr      "Hello, World!"
                call       void [mscorlib]System.Console::WriteLine
                                        (class System.String)
                ret
        }
```

Figure 5.8: ilasm code example

The dump of the PE file is a lot longer than the corresponding Java class file, which is due to two reasons. First, there are 580 bytes of operating system specific headers. Second, the native data section at the end of the file is empty and denoted with zeroes. In fact, the parts that must be compared to the class file are the CLI header and data.

The dump of a PE file containing the "hello world" code example is showed in figure 5.9. Note that lines containing a single * denote that in this place in the file there are a certain amount of sections containing only zeroes. Those are thus omitted. The dump starts with the MS-DOS stub, wherein the embedded error code is clearly seen in the ASCII translation. The PE signature resides at 0x080, immediately followed by the COFF header at 0x84.

The most interesting part starts at 0x264, where the CLI header resides. The IL code for the Main method resides in the area between 0x2e5 and 0x2ef ("72 01 00 00 70 28 02 00 00 0a 2a"). Figure 5.8 shows this method in dissassembled form.

The String heap is situated at 0x435, followed by the user string section which contains only one entry in this case: the "hello, world!" string.

```
000  4d 5a 00 00 06 00 00 00  04 00 00 00 ff ff 00 00  MZ..............
010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@.......
020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  ................
040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  ........!..L.!Th
050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  is program canno
060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  t be run in DOS
070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  mode....$.......
080  50 45 00 00 4c 01 03 00  aa 20 52 40 00 00 00 00  PE..L.... R@....
090  00 00 00 00 e0 00 0e 01  0b 01 06 00 00 04 00 00  ................
0a0  00 04 00 00 00 00 00 00  00 20 00 00 00 20 00 00  ......... ... ..
0b0  00 40 00 00 00 00 40 00  00 20 00 00 00 02 00 00  .@....@.. ......
0c0  04 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00  ................
0d0  00 80 00 00 00 02 00 00  00 00 00 00 03 00 00 00  ................
0e0  00 00 10 00 00 10 00 00  00 00 10 00 00 10 00 00  ................
0f0  00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  ................
100  18 20 00 00 4f 00 00 00  00 40 00 00 e4 02 00 00  . ..O....@......
110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
120  00 60 00 00 0c 00 00 00  00 00 00 00 00 00 00 00  .`..............
130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
*
150  00 00 00 00 00 00 00 00  10 20 00 00 08 00 00 00  ......... ......
160  00 00 00 00 00 00 00 00  64 20 00 00 48 00 00 00  ........d ..H...
170  00 00 00 00 00 00 00 00  2e 74 65 78 74 00 00 00  .........text...
180  e8 02 00 00 00 20 00 00  00 04 00 00 00 02 00 00  ..... ..........
190  00 00 00 00 00 00 00 00  00 00 00 00 20 00 00 60  ............ ..`
1a0  2e 72 73 72 63 00 00 00  e4 02 00 00 00 40 00 00  .rsrc........@..
1b0  00 04 00 00 00 06 00 00  00 00 00 00 00 00 00 00  ................
1c0  00 00 00 00 40 00 00 40  2e 72 65 6c 6f 63 00 00  ....@..@.reloc..
1d0  0c 00 00 00 00 60 00 00  00 02 00 00 00 0a 00 00  .....`..........
1e0  00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 42  ............@..B
1f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
200  ff 25 10 20 40 00 00 00  00 00 00 00 00 00 00 00  .%. @...........
210  42 20 00 00 00 00 00 00  5a 20 00 00 00 00 00 00  B ......Z ......
220  00 00 00 00 4e 20 00 00  10 20 00 00 00 00 00 00  ....N ... ......
230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
240  00 00 5f 43 6f 72 45 78  65 4d 61 69 6e 00 6d 73  .._CorExeMain.ms
250  63 6f 72 65 65 2e 64 6c  6c 00 40 20 00 00 00 00  coree.dll.@ ....
260  00 00 00 00 48 00 00 00  02 00 00 00 00 00 21 00 00  ....H........!..
270  e8 01 00 00 01 00 00 00  02 00 00 06 00 21 00 00  .............!..
280  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
*
2e0  00 00 00 00 00 00 00 00  00 00 00 00 1e 02 28 01  ..............(.
2f0  00 00 0a 2a 2e 72 01 00  00 70 28 02 00 00 0a 2a  ...*.r...p(....*
300  42 53 4a 42 01 00 01 00  00 00 00 00 0c 00 00 00  BSJB............
310  76 31 2e 30 2e 33 37 30  35 00 37 40 00 00 05 00  v1.0.3705.7@....
320  70 00 00 00 c2 00 00 00  23 7e 00 00 34 01 00 00  p.......#~..4...
330  68 00 00 00 23 53 74 72  69 6e 67 73 00 00 00 00  h...#Strings....
```

```
340  9c 01 00 00 1d 00 00 00  23 55 53 00 bc 01 00 00  ........#US.....
350  19 00 00 00 23 42 6c 6f  62 00 00 00 d8 01 00 00  ....#Blob.......
360  10 00 00 00 23 47 55 49  44 00 00 00 00 00 00 00  ....#GUID.......
370  00 00 00 00 01 00 00 00  47 05 00 00 09 00 00 00  ........G.......
380  00 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00  ................
390  02 00 00 00 02 00 00 00  01 00 00 00 02 00 00 00  ................
3a0  01 00 00 00 01 00 00 00  00 00 44 00 01 00 00 00  ..........D.....
3b0  00 00 06 00 0a 00 11 00  06 00 1e 00 11 00 00 00  ................
3c0  00 00 3b 00 00 00 00 00  01 00 01 00 01 00 10 00  ..;.............
3d0  53 00 00 00 05 00 01 00  01 00 ec 20 00 00 00 00  S.......... ....
3e0  86 18 18 00 01 00 01 00  f4 20 00 00 00 00 16 00  ......... ......
3f0  5e 00 13 00 01 00 00 00  01 00 63 00 09 00 18 00  ^.........c.....
400  01 00 11 00 26 00 0e 00  04 80 00 00 00 00 00 00  ....&...........
410  00 00 00 00 00 00 00 00  00 00 30 00 00 00 01 00  ..........0.....
420  00 00 e4 0c 00 00 00 00  00 00 05 00 01 00 00 00  ................
430  00 00 00 00 00 6d 73 63  6f 72 6c 69 62 00 4f 62  .....mscorlib.Ob
440  6a 65 63 74 00 53 79 73  74 65 6d 00 2e 63 74 6f  ject.System..cto
450  72 00 43 6f 6e 73 6f 6c  65 00 57 72 69 74 65 4c  r.Console.WriteL
460  69 6e 65 00 68 65 6c 6c  6f 77 6f 72 6c 64 00 3c  ine.helloworld.<
470  4d 6f 64 75 6c 65 3e 00  68 65 6c 6c 6f 77 6f 72  Module>.hellowor
480  6c 64 2e 65 78 65 00 48  65 6c 6c 6f 57 6f 72 6c  ld.exe.HelloWorl
490  64 00 4d 61 69 6e 00 61  72 67 76 00 00 1b 48 00  d.Main.argv...H.
4a0  65 00 6c 00 6c 00 6f 00  2c 00 20 00 57 00 6f 00  e.l.l.o.,. .W.o.
4b0  72 00 6c 00 64 00 21 00  00 00 00 00 00 03 20 00  r.l.d.!....... .
4c0  01 08 b7 7a 5c 56 19 34  e0 89 04 00 01 01 0e 05  ...z\V.4........
4d0  00 01 01 1d 0e 00 00 00  8a 2a 46 e9 0f 9a ad 43  .........*F....C
4e0  af b7 d4 4b 7f e7 84 7a  00 00 00 00 00 00 00 00  ...K...z........
4f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
*
600  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00  ................
610  10 00 00 00 18 00 00 80  00 00 00 00 00 00 00 00  ................
620  00 00 00 00 00 00 01 00  01 00 00 00 30 00 00 80  ............0...
630  00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 00  ................
640  00 00 00 00 48 00 00 00  58 40 00 00 8c 02 00 00  ....H...X@......
650  00 00 00 00 00 00 00 00  8c 02 34 00 00 00 56 00  ..........4...V.
660  53 00 5f 00 56 00 45 00  52 00 53 00 49 00 4f 00  S._.V.E.R.S.I.O.
670  4e 00 5f 00 49 00 4e 00  46 00 4f 00 00 00 00 00  N._.I.N.F.O.....
680  bd 04 ef fe 00 00 01 00  00 00 00 00 00 00 00 00  ................
690  00 00 00 00 00 00 00 00  3f 00 00 00 00 00 00 00  ........?.......
6a0  04 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  ................
6b0  00 00 00 00 44 00 00 00  01 00 56 00 61 00 72 00  ....D.....V.a.r.
6c0  46 00 69 00 6c 00 65 00  49 00 6e 00 66 00 6f 00  F.i.l.e.I.n.f.o.
6d0  00 00 00 00 24 00 04 00  00 00 54 00 72 00 61 00  ....$.....T.r.a.
6e0  6e 00 73 00 6c 00 61 00  74 00 69 00 6f 00 6e 00  n.s.l.a.t.i.o.n.
6f0  00 00 00 00 7f 00 b0 04  ec 01 00 00 01 00 53 00  ..............S.
700  74 00 72 00 69 00 6e 00  67 00 46 00 69 00 6c 00  t.r.i.n.g.F.i.l.
710  65 00 49 00 6e 00 66 00  6f 00 00 00 c8 01 00 00  e.I.n.f.o.......
720  01 00 30 00 30 00 37 00  66 00 30 00 34 00 62 00  ..0.0.7.f.0.4.b.
```

```
730  30 00 00 00 28 00 02 00  01 00 50 00 72 00 6f 00  0...(.....P.r.o.
740  64 00 75 00 63 00 74 00  56 00 65 00 72 00 73 00  d.u.c.t.V.e.r.s.
750  69 00 6f 00 6e 00 00 00  20 00 00 00 24 00 02 00  i.o.n... ...$...
760  01 00 43 00 6f 00 6d 00  70 00 61 00 6e 00 79 00  ..C.o.m.p.a.n.y.
770  4e 00 61 00 6d 00 65 00  00 00 00 00 20 00 00 00  N.a.m.e..... ...
780  24 00 02 00 01 00 50 00  72 00 6f 00 64 00 75 00  $.....P.r.o.d.u.
790  63 00 74 00 4e 00 61 00  6d 00 65 00 00 00 00 00  c.t.N.a.m.e.....
7a0  20 00 00 00 28 00 02 00  01 00 4c 00 65 00 67 00   ...(.....L.e.g.
7b0  61 00 6c 00 43 00 6f 00  70 00 79 00 72 00 69 00  a.l.C.o.p.y.r.i.
7c0  67 00 68 00 74 00 00 00  20 00 00 00 38 00 0b 00  g.h.t... ...8...
7d0  01 00 49 00 6e 00 74 00  65 00 72 00 6e 00 61 00  ..I.n.t.e.r.n.a.
7e0  6c 00 4e 00 61 00 6d 00  65 00 00 00 68 00 65 00  l.N.a.m.e...h.e.
7f0  6c 00 6c 00 6f 00 77 00  6f 00 72 00 6c 00 64 00  l.l.o.w.o.r.l.d.
800  00 00 00 00 2c 00 02 00  01 00 46 00 69 00 6c 00  ....,.....F.i.l.
810  65 00 44 00 65 00 73 00  63 00 72 00 69 00 70 00  e.D.e.s.c.r.i.p.
820  74 00 69 00 6f 00 6e 00  00 00 00 00 20 00 00 00  t.i.o.n..... ...
830  1c 00 02 00 01 00 43 00  6f 00 6d 00 6d 00 65 00  ......C.o.m.m.e.
840  6e 00 74 00 73 00 00 00  20 00 00 00 24 00 02 00  n.t.s... ...$...
850  01 00 46 00 69 00 6c 00  65 00 56 00 65 00 72 00  ..F.i.l.e.V.e.r.
860  73 00 69 00 6f 00 6e 00  00 00 00 00 20 00 00 00  s.i.o.n..... ...
870  48 00 0f 00 01 00 4f 00  72 00 69 00 67 00 69 00  H.....O.r.i.g.i.
880  6e 00 61 00 6c 00 46 00  69 00 6c 00 65 00 6e 00  n.a.l.F.i.l.e.n.
890  61 00 6d 00 65 00 00 00  68 00 65 00 6c 00 6c 00  a.m.e...h.e.l.l.
8a0  6f 00 77 00 6f 00 72 00  6c 00 64 00 2e 00 65 00  o.w.o.r.l.d...e.
8b0  78 00 65 00 00 00 00 00  2c 00 02 00 01 00 4c 00  x.e.....,.....L.
8c0  65 00 67 00 61 00 6c 00  54 00 72 00 61 00 64 00  e.g.a.l.T.r.a.d.
8d0  65 00 6d 00 61 00 72 00  6b 00 73 00 00 00 00 00  e.m.a.r.k.s.....
8e0  20 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ...............
8f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
*
a00  00 20 00 00 0c 00 00 00  02 30 00 00 00 00 00 00  . .......0......
a10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
*
c00
```

Figure 5.9: class file method descriptor

5.3 Storage Variants

Now that the basic file format has been discussed, a few possible storage methods are presented. As said earlier, Java class files contain one type per class file. However, those separately compiled units can be gathered together into an archive file. Such a file, which is called a Java ARchive (JAR), basically contains a sequence of class files. Additionally, a manifest will provide info about the archive, as shown in figure 5.10.

By default JAR files will also be compressed using the ZIP file format. While compression makes the file size smaller, and thus taking less time to transfer through a network and taking less space on disk, it comes at the price of additional processing time for decompression at application start-up time.

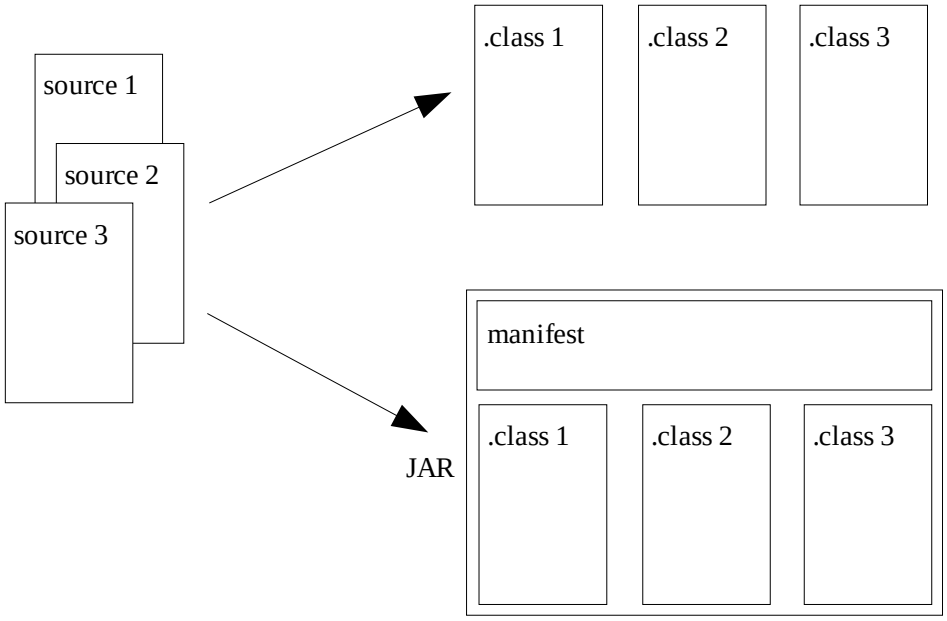

Figure 5.10: Java storage possibilities

In .NET more is possible on the storage area. One extreme is compiling each reference type into its own assembly. At the other side, .NET allows to include all source code into one big assembly (figure 5.11).

Normal distribution of .NET applications tends to be in the middle of those two methods. Applications are distributed into components containing clearly separated code, for example per package. This allows the replacement of a component by a newer version without the need to modify anything to the

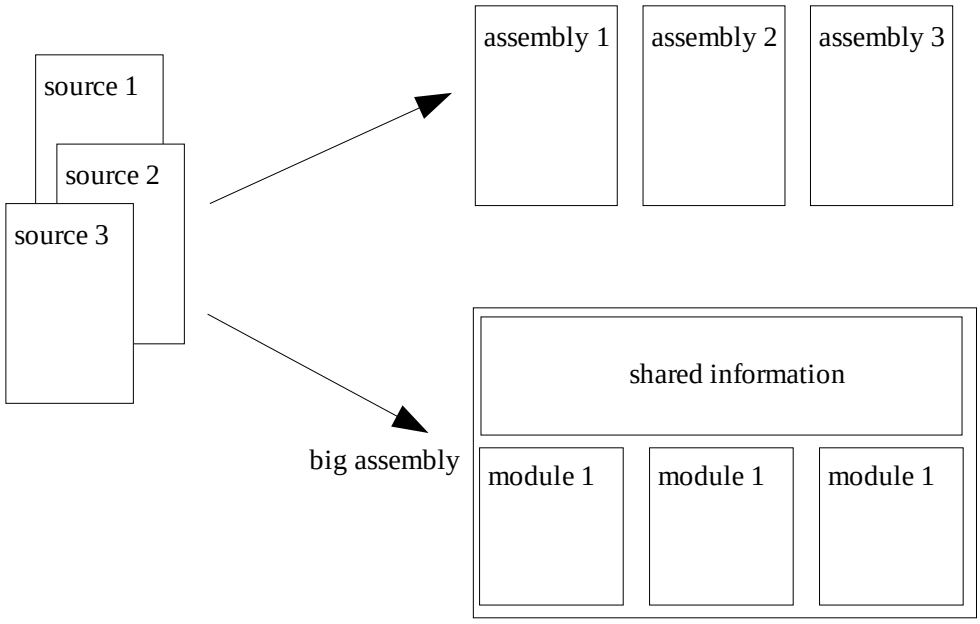

Figure 5.11: .NET storage possibilities

other parts. Bringing more modules together into one assembly allows to share common data such as constants.

5.4 Summary

- Each platform has its own file format. The Java one is very compact, but doesn't allow to bring together duplicate information. .NET's file format is an extension on the existing PE file format, containing quite some legacy operating system specific content. Native sections allow storing managed and unmanaged code in the same file.
- On the conceptual domain, Java and .NET storage is similar: IL code and meta data are stored in a file format at compile time, waiting to be loaded for execution. IL code makes use of meta data contents for referring to modules, classes, constants, etc. Those meta data can be approached at runtime with the reflection API.

Chapter 6

Case Study

After two chapters of theoretical descriptions, it is time to apply them to practice and situate concrete usage's influence on the announced criteria. This chapter will investigate statistics about IL instruction usage, storage methods and file size on some source code. Java class files have already been the point of interest in different research projects. There has been research to the procentual composition of the class files [DM98], to methods to make the files smaller in order to decrease memory usage [VP03] and byte code obfuscation.

6.1 Grande benchmarking suite

The origin of this so called Grande suite is the JavaGrande benchmark suite created by the Java Grande Forum community [MCH99]. This forum promotes the use of Java for /em grande applications. A *grande* application is defined as an application which has large requirements for any or all of memory, bandwith and processing power. The benchmark suite developed by them tests the power and the suitability of different systems for grande applications.

Figure 6.1 shows the origin of the Grande benchmarks prepared for this thesis. In order to be able to execute the same tests under the .NET platform, this suite was translated using the Microsoft Java Language Conversion Assistant [JLC]. This tool translates Java source code to C# source code. Additional manual manipulation is required to translate libraries that the tool is not able to convert. For certain libraries that where not available or differed in their behavior under .NET, Java class The Java and .NET suite are together called the Grande benchmarking suite.

The original JavaGrande benchmarks contain sections for microbenchmarking, but also shorter and longer typical algorithms up to parallel and distributed tests. Because later in this thesis the code analysed here will be effectively run for benchmarking too, the translation effort and analysis upon it are restricted

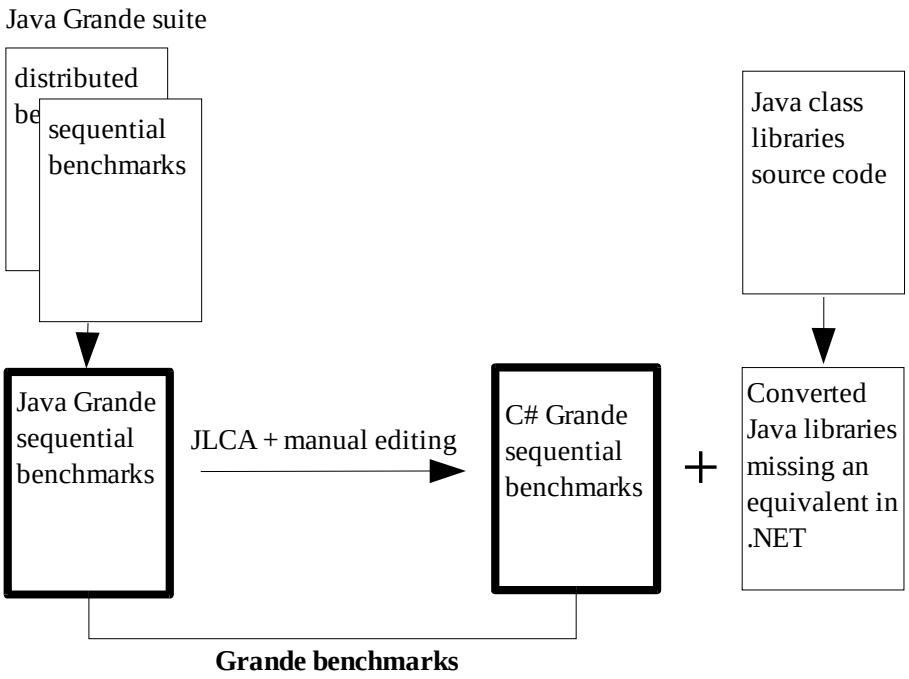

Figure 6.1: Grande benchmarks origin

to those parts that will be used later on. Those parts are the sequential benchmarks part 2 and 3. Singer has also been experimenting with section 2 of this benchmarks on both the JVM and the CLR in [J.S03].

6.2 IL Statistics

Something quite noticeable is the fact that the intermediate language produced by compilers of the same platform is very similar. Almost all the Java code is equivalent, whether it is compiled by Sun or by Kaffe. This fits in the concept of the platform where the programming language is mapped onto the whole IL set. Between Mono and MS.NET, there are more differences, mainly in the tackle of loops, usage of short instructions, etc. Choosing not to make use of short form instructions implies a slightly larger assembly, but probably in favour of the compilation time.

The influence on performance is harder to estimate. Mono sometimes generates inefficient byte code by the use of temporary loads and stores, but it's not always possible to blame less performance on this. It depends on whether and how the code will be transformed and optimized at runtime before execution.

Using the byte code of the Grande benchmark suite, the appearance of the individual instructions are count. First, the instructions are separated into the 9 partitions described in Chapter 4. Figure 6.2 shows an equal usage in similar partitions. Most used are the "Load and Store" and the "Fields and Methods" instructions. The first can be explained by the fact that both intermediate languages are stack-oriented, the second by the object-oriented nature of the programming languages compiled to the IL.

When looking at the five most frequent instructions in Table 6.1, those are indeed instructions belonging to the two most popular partitions.

Instruction	*Java*		*.NET*	
	name	*%*	*name*	*%*
1.	aload_0	15	ldarg.0	13
2.	getfield	10	ldfld	10
3.	invokevirtual	6	ldloc.s	7
4.	putfield	4	ldelem.ref	5
5.	iload	4	stfld	4

Table 6.1: Most used instructions

More particularly, the most popular instruction is loading local variables (remember that method arguments are part of the local variables in Java, in .NET

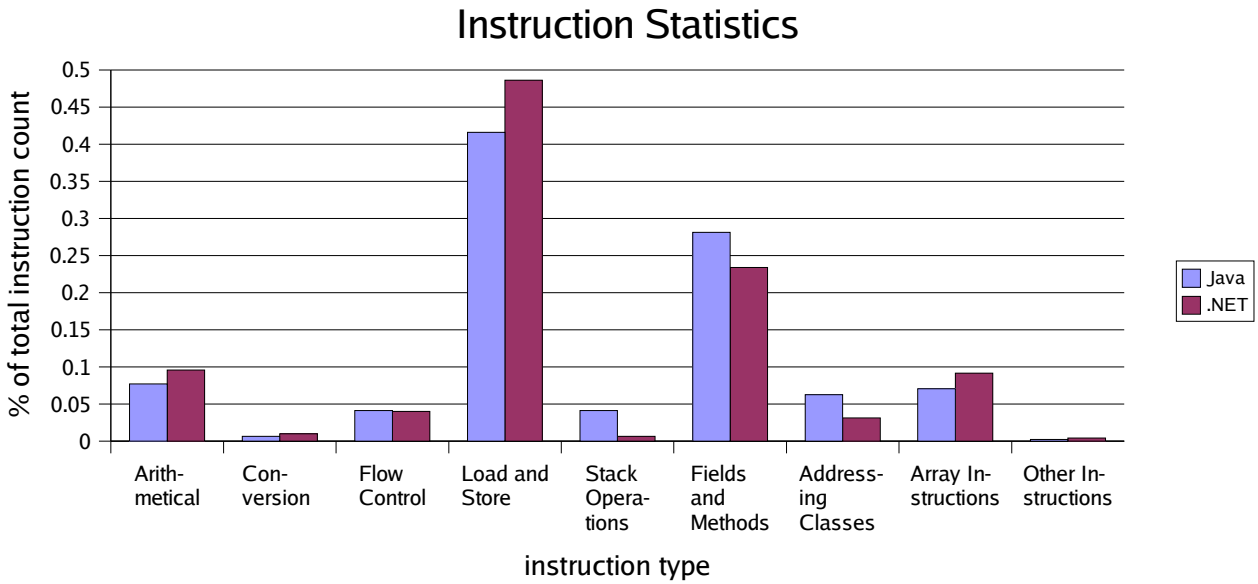

Figure 6.2: Instruction Distribution

they are separated). With the Java instructions having the type build in into the instruction, we see that most used instruction is the short form for loading a reference stored in the first local variable on the stack. In instance methods, the first local variable contains a reference to the object on which the method is called (instruction *aload_0*). In .NET, the object reference appears as an invisible first parameter to the instance method and can thus also be loaded with the *ldarg.0* instruction.

The Java statistics show that invoking a virtual method is the third most used instruction. This is the consequence of all methods being virtual by default in Java. In .NET, no similar instruction is found in the top five. The calling of methods will be more distributed over default non-virtual methods and explicitly declared virtual methods.

The other top five instructions are the loading and storing of fields (*getfield*, *ldfld*, *putfield*, *stfld*) and the loading of an array object entry onto the stack (*ldelem.ref*).

Although the statistics acquired here are certainly a good reference, they are no more than a best effort. The results depend partially on the type of program, the programming style and the source code compiler. For these reasons, the statistics obtained here are not completely similar to the results in other research [VP03].

6.3 File Size Statistics

The file format description tends to be in the advantage of Java. The file format is simpler and does not contain operating system specific headers that are only for Windows after all. The Grande suite is used again to discover the expected overhead in file size that comes with the PE files after the lengthy example in chapter 5.

The first thing that can be determined is the size of the HelloWorld example. The corresponding Java class file is more than seven times smaller than the .NET one. This simple example teaches that the overhead that can be expected for PE files with few CLI content is quite large. Examples of such files are interfaces and classes with many data members and small methods in the first place.

The question is how Java and .NET relate over larger pieces of code. First, the .NET source is compiled like the Java one: one class per assembly. Note that creating a JAR file doesn't change the individual class layouts as it is just an archive collecting all the different class files.

Unfortunately, the Grande suite code is rather small in size. The total class and interface count results in 59 classes containing about 10.000 lines of code. Since there is no larger example available, statistics will need to be performed on this code and generalized predictions must be expressed carefully.

	Java	*.NET*
one class per file	161 Kb	288 Kb
one assembly/JAR	161 Kb	98 Kb
zipped JAR	89 Kb	X

Table 6.2: Java and .NET storage size for Grande suite, per storage method

Table 6.2 shows different methods for storing the Grande suite. When making the sum over all the class files and all the assemblies, the .NET total size is 179% of the Java one. At the other side, if all the classes of this benchmark suite are gathered together into one assembly, the .NET file size is only 61% of the Java files' size. There are several reasons for this decrease in size:

- First, the operating system specific sections and the native parts are only included only once in the whole file instead of once for each class.
- Second, the string heap and the user string sections of the various classes are brought together and the duplicates are removed. This duplicates are voluminous, because most assemblies contain for example the *System.Object* and *mscorlib* constants. The latter is a reference to the core .NET libraries, which are evidently used in every assembly.

The use of the GCJ Java-to-native compiler results in a file size (299 Kb) that is nearly two times the size of an uncompressed Java archive. It is thus obvious that by nature of the IL and the file format the needed disk space is smaller than in case of native code.

However, when space is really important, it is interesting to remember that compiling a .NET application into one single assembly generates the smallest possible file size.

6.4 Case Study experiences

- The Grande suite is a set of sequential benchmarks in both Java and C#. They provide an excellent source base to perform deduce statistics from and to perform benchmarks on.
- The instruction usage of CIL and JIL is quite the same, except for parts where fundamental design decisions occur (example: default virtual methods in Java, not in .NET). This leads to the conclusion that starting with

similar ILs, with an equivalent occurrence of corresponding instructions, differences in runtime performance will not be caused by the IL.

- The initial assumption of a relatively big .NET files does not hold: when bringing multiple modules together into one assembly, only a zipped Java archive can go below this file size. Larger .NET units result in less overhead but have an increasingly negative effect on modularity. The choice depends on the purpose and target device of the resulting IL program.

Chapter 7

Runtime Environments

Now that the concepts of the VMs, the intermediate language and the corresponding file format are understood, the focus will be shifted towards the runtime details. At one side there is the sequence of actions that happen from the loading of the application over execution until shutdown of the VM, at the other side there are the internals of the VM that allow this sequence to function.

Because the application code in IL form is portable across VMs, the VM's interface (APIs, tools) towards applications is always the same. The part of the VM that talks to the operating system changes as it makes use of the services a particular OS can offer (for example thread management, GUI libraries, etc.). The terms *runtime (environment)* and *virtual machine* are used interchangeably here.

To end this chapter with, a couple of concrete Java and .NET implementations are treated, with their particular details concerning garbage collection and execution model.

7.1 The Java Runtime Environment

A runtime instance of the JVM is started whenever an application for it is called to be executed. The runtime starts, loads the application, executes its code and dies after program execution has ended. Each application will be executed in its own runtime instance. The following paragraphs are loosely based upon information from [LY99] and [B.V02].

The Java runtime is composed of three large parts: the *class loader*, the *runtime data areas* and the *execution engine*. Figure 7.1 shows their interrelationship and their internal details. The figure is based on figures and information in [KCSL00] and [Li97].

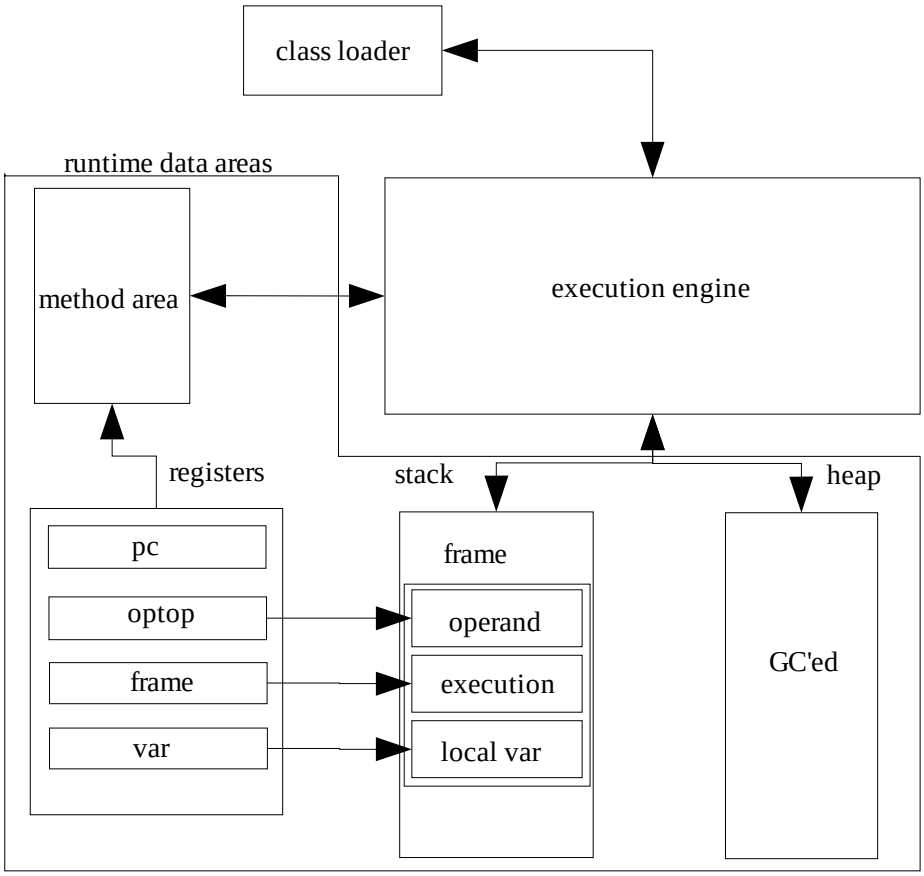

Figure 7.1: JVM Structure

7.1.1 Class Loader Subsystem

The class loader is that part of the runtime that prepares code for execution when needed. Classes and interfaces are dynamically loaded, linked and initialized by the class loader at the moment they are first referenced. Moreover during these activities the correctness of the imported files is gradually checked.

- **Loading** The JVM searches and opens the class file corresponding to the referenced class. Basic file format checks will be performed, where after meta data will be checked without looking into the byte code itself. Next, the class or interface will be loaded into memory.
- **Linking**
  - Verification: ensures the correctness of the imported type, types are verified when methods and fields are accessed. Instruction opcodes are checked for having suitable arguments.
  - Preparation: allocates memory for class variables and initializing the memory to default values.
  - Resolution: transforms symbolic references from the type into direct references to the place where the type is effectively stored in the memory.
- **Initialization** A class or interface is initialized the first time it is referenced by the program by invoking the static initializers and the initializers for static fields. This can happen when certain instructions are executed, when invoking reflective calls on the class, initializing of one of the subclasses or being the initial class at VM start-up.

Another code validation check will happen at runtime, where correct type referencing, access rights and descriptors will be checked the first time a method is called. A more detailled explanation of the class loading architecture can be found in [B.V97].

7.1.2 Runtime Data Areas

Stacks & Registers

The Java virtual machine can run many threads at once. Each thread has its own program counter (PC), pointing to the instruction currently being executed. The stack contains frames; for each method that is called a new frame is created. Three registers point to three parts of the current stack frame:

- The variable register points to the frame part containing the local variables.
- The frame register points to the execution environment section, which contains data to allow constant pool resolution, and information to restore the previous stack frame for both normal as exception method return.

- The operation top register points to the top of the operand stack. This is the stack used by the IL: variables are pushed on the stack, instructions take their arguments from it, and values can be stored back into variables when popped.

The sizes of the variable and operation part of the frame are determined at compile time by analyzing the behaviour of the generated IL code.

There are no other registers for computations. If a method is successfully executed a value may be returned. The current stack frame will then be destroyed and the new current frame will be the frame of the invoking method. Registers will be restored in their state at the time the previous method was called. When an exception that is thrown inside a method's code is not handled by that method, the method will be terminated unsuccessfully, the current frame will change and the exceptions will be thrown again and again until there is a method that catches the exception. The stack is aligned on words, which is at least 32 bit, but can be larger too.

Method Area

Somewhere in the memory addressable by the JVM the method area resides. When a type is imported, the meta data is taken out of it and stored in the method area. The area will be consulted under execution by all running threads. When a certain type is not available in the method area, it must be loaded first before execution can continue. The following information is stored:

- Type Information: kind of type, name, super class name, modifiers.
- Constant Pool: the constant pool that comes with the class file loaded.
- Field Information: name, type and modifiers of the types' fields.
- Method Information: contains the method's name, return type, number and types of parameters and its modifiers. Besides that, it also holds the method's byte code and exception table. The program counter of each thread points into this byte code.
- Class variables are shared among all instances of a class and thus available in the method area.

Garbage Collected Heap

The heap is the storage place for all instances of reference types the programmer initializes. The garbage collector will clean memory that is occupied by objects which are no longer being referenced (and as such can't be used anymore). The heap is aligned on words, and shared across all the threads running in the JVM.

7.1.3 Execution Engine

The execution engine is that part of the VM that reads the IL code, makes use of the data areas to gather information and converts this information into such a way that the program can be executed on the underlying hardware. It can thus be seen as the heart of the VM. A wide range of methods to implement an execution engine exist. Since this is not typical to either Java or .NET, execution engine implementations will be discussed in a separate section later in this chapter.

7.2 The Common Language Interface

Like the JVM, the CLI can be represented as a three part structure too (figure 7.2).

7.2.1 Platform Adaptation Layer

The platform adaptation layer (PAL) is that part of the VM that directly talks to the operating system and hardware. By grouping these system specific parts together, the rest of the VM is completely portable. A new system can be targeted by the VM by only changing the PAL to the new system specifications.

7.2.2 Assembly Loader

Loading assemblies into the CLI forms a two step procedure:

- When executing .NET applications the VM first seeks the assembly containing the main entry point depending on manifest information, versioning and namespaces when the assembly is fully specified. Otherwise, it will just lookup the assembly by name.

- The class will then be opened and cached. Every method in the class will be provided with a stub. Other classes that are being referenced by the original class will be loaded too, but only on demand at the time they are needed. The static parts of the class will be initialized. The verifier will then check for valid meta data and type safety, before the execution engine is allowed to execute the code. This process always happens per method.

7.2.3 Runtime Data Area

Just like the JVM, the CLI has a per-thread stack-based structure which contains method states, the method state of the method currently being executed marked as the active one.

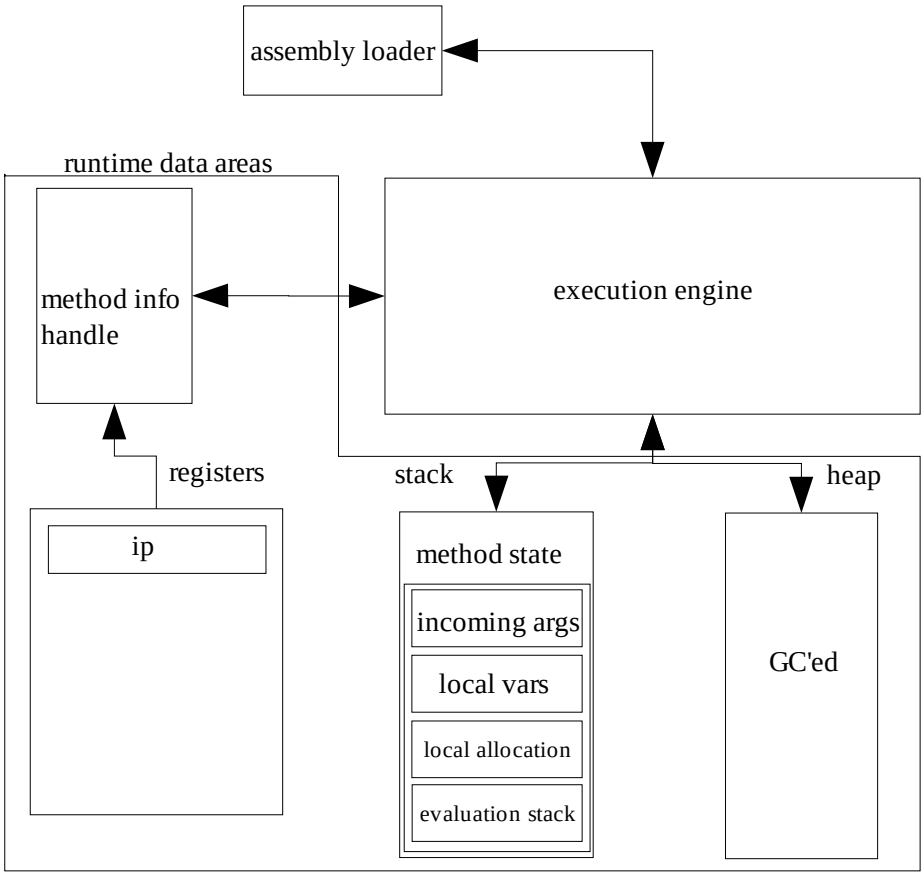

Figure 7.2: .NET Runtime Structure

7.2.4 Method State

A method state is basically a frame on the stack which represents the current state of the method. When a method is called, a new frame will be created and initialized. On return, the previous method will regain the current method status and its state must be the same as it was at the time the new method was called. In case of exceptions, method frames will be popped until a method catches the exception. Note how similar this works as the JVM. A method state consists of four parts:

- **Array of Local Variables** This array holds the local value type instances of the current method. Instructions load their values from here on the operand stack, apply instructions, and store the result back here.
- **Array of Incoming Arguments** The difference between local variables and arguments is that simple type arguments can be used as reference parameters, returning back a value to the calling method.
- **Evaluation Stack** The operand stack is local to a method and contains arguments and results of instructions. This includes arguments for method calls.
- **Local Allocation** This local memory pool allows allocating raw, addressable memory. The CIL contains instructions to make use of this memory. Upon method termination the allocated memory is reclaimed.

The instruction pointer points to the next CIL instruction that will be executed. In order to be able to return to a previous method and restore its information, a return state handle contains the state of the method's caller.

7.2.5 Method Info Handle

This part holds read-only information about the model. It holds the method signature, types of local variables and exception information.

7.2.6 Garbage Collected Heap

Like in the JVM, the garbage collected heap is used to allocate reference types from. How garbage collection works in order to take care of the memory management is described further in this chapter.

7.3 Execution Modes

The core of the virtual machine is the execution environment, which makes use of the structures above to run programs provided to the VM. The typical idea behind the execution is the loop in 7.3, as described in [LY99].

```
do {
     fetch an opcode;
     if (operands) fetch operands;
     execute the action for the opcode;
} while (there is more to do);
```

Figure 7.3: Execution Loop

Different methods with corresponding pros and contras exist to implement an execution environment. Depending on the environmental constraints one solution can be preferred to others. The most important factors when choosing are contained within the criteria presented back in Chapter 1: start-up time, runtime performance, memory usage and portability. In this section, the various possibilities are reviewed and advantages and disadvantages are discussed.

7.3.1 Interpreter

An interpreter is a piece of software that reads the intermediate language instructions, and executes them one by one. This process is quite slow in comparison with native executables, because of the overhead of reading the IL instructions one by one, getting the opcode out of it and execute the code needed to simulate that instruction.

An interpreter is usually implemented as a large switch table containing entries for the different instructions. For each instruction, a separate function is then called. Speedups for this method are discussed in [DW03]. The traditional switch table can be replaced by more efficient structures: direct threading replaces the byte code with the address of the corresponding implementation code. The advantages of an interpreter are the easiness to implement, the portability and little memory usage. In then next chapter this memory advantage will effectively be proved.

7.3.2 Just-In-Time Compiler

A Just-In-Time Compiler (JIT) will generate machine code on-the-fly at runtime. More specifically: for every first encounter of a method, the JIT-compiler will generate machine code, store them in a cache, and execute them. So while a JIT-compiler might be slower than an interpreter at the first stages of a program's execution, it will speed up when reusing the cached machine code for methods it has already compiled.

The JIT-compiler doesn't start to work until the first method call on a class's instance. Before that time, information about types is handled by meta data. Figure 7.4 shows several actions in the JIT process.

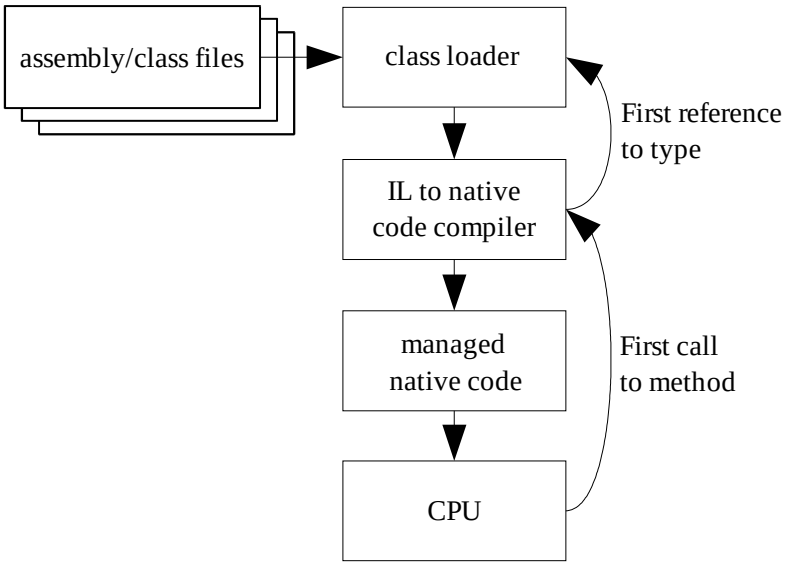

Figure 7.4: JIT compilation

Furthermore, some JIT-compilers are able to dynamically evaluate the generated code and adapt it at later stages when non-optimized parts are detected. A JIT compiler often works with a couple of intermediate representations on which various optimizations can be performed before emitting machine code. The engine must decide between further optimization investigation and the speedup it may generate, a decision that can be made with the help of some profiling data.

A disadvantage of a JIT-compiler is the slow start-up time: at the early stages of an application's execution, all methods which are called for the first time must be compiled to native code and stored in memory. This native code will reside in memory alongside meta data, consuming more memory than an interpreter.

7.3.3 Ahead-Of-Time (AOT) compilation

Instead of generating machine code at runtime, the application's methods are compiled to machine code and stored at compile time or between compile time and runtime (sometimes called *install time*). The execution engine can then make use of this stored native code at runtime, avoiding the extra cost of JIT-compilation. The advantage is situated at start-up time, where the JIT-ing of methods doesn't need to be done.

7.3.4 Compilation to native code

One could call this AOT too, but there's a difference between the former approach and this one. There is no virtual machine anymore in this approach. One can for example translate Java byte code to C. A standard C compiler can then be used to generate native code.

Another approach is compiling the Java code into an executable assembler binary. However all the meta data must be contained, and the compiler being able to do this must link against a natively compiled class library too.

Note that in either way the services provided by the VM must be simulated or integrated in the native code. At one side there can be a small runtime environment that takes over the responsabilities of the VM, mainly memory management. At the other side, memory management can be directly integrated in the code, for example by adding a reference couting system.

7.3.5 Java processors

A Java processor is a physical processor whose instruction set is the set of Java IL instructions. This execution model is interesting for embedded systems running only Java applications.

execution method	*performance*	*start-up*	*memory*	*VM portability*	*code portability*
Interpretation	-	+	+	+	+
JIT-compiler	+	-	-	-	+
AOT-compiler	+	+	-	-	+
Native code	+	+	+	-	-

Table 7.1: Execution method pros and contras

7.3.6 Choosing the right execution method

After reviewing the most common execution methods, you might ask which the best one is. Well, it all depends on the environment in which it needs to be used. Each of the execution techniques has its own advantages and consequences because of the design decisions. The pro and contras of each method are summarized in Table 7.1.

If performance is top priority, you should choose for JIT- or AOT-compilation. But then again, does start-up time matter? AOT solves the slow start-up times inherent to JIT-compilers, but there are additional portability issues: not only is the runtime difficult to port, but the applications are compiled to a non-portable format. When portability is important or there are memory constraints, you should rather go for interpretation. It may be clear that the execution engine is of large influence on the announced criteria.

Those techniques, and a number of others, including optimizations and transformations are discussed by Kazi, Stanley et Al. [KCSL00]. A number of research VMs experiment with these techniques, such as Harissa [MS99] and Sable VM [EL01].

7.4 Intermediate Representations and Optimizations

In the conversion process between the IL and native code, a JIT-compiler can make use of one or more intermediate representations that increasingly go more low level and are suited to perform optimization techniques on.

Typical optimizations include constant expression evaluation, dead code elimination, inline method calls, etc. A well known intermediate representation (IR) is static single assignment (SSA).

An example of an SSA representation of a simple source program (figure 7.5) is given in figure 7.6. Every variable will only be assigned once in an SSA representation. This way all useless assignments are discovered and thrown away.

The resulting IR shows five local variables created from 2 source code variables. The first two variables are the result from the local variable initializations. The assignment $x = y$ is replaced with an assignment of the value of y of that moment to x.

```
1 public class ssa_ex {
2         public static void Main(string[] test) {
3                 int y, x;
4                 y = 1;
5                 y = 2;
6                 x = y;
7         }
8 }
```

Figure 7.5: Typical code handled by SSA-based IR optimizations

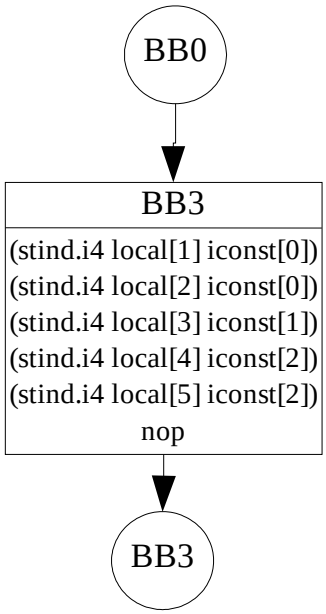

Figure 7.6: SSA representation

SSA representations not only allow removing useless assignments, but with the use of constant propagation it is also possible to remove conditional branches. The next step is to compute register allocation and conversion to native instructions.

Intermediate representations allow JIT-compilers to generate faster and smaller pieces of code out of IL, but the optimizations themselves will take time too. In

particular, these optimizations will lead to a longer start-up time.

7.5 Garbage Collection

Every computer program uses memory to store data and code. In object oriented programming languages, objects are typically components that require some memory. Their life cycle can be described in five points:

1. **Allocation of memory for the type of the object** This happens when an application creates a new object. Its size is determined and space is allocated.
2. **Initialization of the memory** The object's constructor will initialize the allocated space with some values.
3. **Usage of the object in the program** The object is used in the application as long as there is a reference to the object. GC cycles will leave the object in memory.
4. **Cleanup** The object is no longer reachable and will be marked for deletion at the next GC cycle.
5. **Freeing of the memory** The memory is cleared for new usage.

From the programmer's point of view, steps one and two are atomic: he/she calls a constructor on a variable of a suitable type. In C++, cleanup happens in the destructor, at the moment the programmer deletes the object. In languages with automatic memory management, the memory is not manually freed. A garbage collector will keep track of objects no longer in use and free the memory automatically. This way common programming errors like forgetting to free objects after usage or trying to access objects that are already freed are prevented.

Note that GC isn't by any means a new concept: it has been studied since the early sixties and many programming languages make use of it: Lisp, Smalltalk, Eiffel, Haskell, etc. In the following sections, some common GC algorithms are described to show how this technology evolved into the current implementations. Wilson [Wil92] provides a neat overview of wide spread garbage collection techniques.

7.5.1 Garbage Collection Types

Numerous approaches exist to implement the concept of garbage collection. Choosing between different implementations always includes having a trade off between:

- **Precise or Conservative** Can the GC precisely define what objects are in memory and which ones are no longer being referenced?

- **Moving/compacting or Non-moving** Are the surviving objects moved and brought together in memory after a reclamation phase?
- **Explicit or Implicit reclamation phase** Does the GC need to traverse through the unreachable objects in order to reclaim them (explicit) or not (implicit).
- **Stopping or Incremental or Concurrent** When does the garbage collector starts to work? Does it stop the program's execution for some time and starts collecting, or does it work concurrently with program execution? Another option is incremental collection, where the GC can be interrupted at any time in the collection process. The work happens thus in tiny time slices between program execution. This method is frequently used in programs where there mustn't be a user perceptible pause such as games.
- **Generational or Non-generational** Does the GC distinguish between short-living and long-living objects?

Automatic Reference Counting For each object managed by the collector, a counter containing the number of references is maintained. For each pointer assignment the counters are incremented and decremented. If a counter reaches zero, the object is not referenced anymore and can be reclaimed. Problems can arise in the case of cycles. Furthermore it produces a considerable overhead on the application. An advantage is the determinism of memory reclamation: as soon as a reference counter reaches zero, the object will be reclaimed.

Tracing Collector Rather than having this continues overhead at runtime, a technique called tracing collection is used. This technique stops the application from time to time, and starts searching for unreferenced objects. The runtime maintains a list of storage locations which refer to objects on the heap or to null. Such locations are called *roots*. Examples of roots are global and static object pointers, variable and parameter object pointers. The GC starts from the roots and creates a graph of accessible objects. Objects are followed recursively until all reachable objects have been examined.

Copying GC A copying garbage collector copies all the live objects into a certain area of the heap. The rest of the heap is then ready to be allocated again since it will only contain garbage.

Mark-Sweep Algorithm The mark-sweep collector is a tracing collector. The first phase is the *mark* phase, using a tracing algorithm. After all the references are visited, the second phase starts. Every object on the heap will be checked against this set and those that aren't referenced anymore will be swept.

Mark-Compact Algorithm A problem with Mark-Sweep is that it will be difficult to allocate new objects in the gaps that are cleaned. There must be a check whether the new object fits in the gap, and if it does there will most likely remain an even smaller part of memory. Allocatable memory will be fragmented after a while.

The Mark-Compact algorithm will compact the used memory space by moving most of the live objects. The rest of the memory will then be a contiguous free space. It differs from a Mark-Compact algorithm because the marking and the copying happens at the same time instead of in two phases.

Incremental GC In order to avoid pausing the application for too long, an incremental GC interleaves collection with the actual program execution.

Generational GC This approach is based on the difference in lifetime between objects [LH83]. The heap is divided into different generations, whereby younger generations are more often collected. However, pointers from older generations to newer ones must be identified and stored into a data structure: the remembered set. Analysis on object lifetime showed that two kinds of objects exist: those used for temporary storage and those containing data that continues to exist throughout program execution. In this algorithm, the time to free memory is in relation to the object's lifetime: older objects are garbage collected more slowly than newer ones.

Conservative GC Languages that don't have runtime pointer information can be provided with automatic memory management by applying a conservative garbage collection technique. Every space in memory that looks like a pointer is considered as a potential one, for example an integer can be confused for a pointer. When such a GC is not sure whether the pointer is real, it is dangerous to relocate the object. This impossibility to move objects leads to fragmentation. Conservative GC allows code that was not written to be garbage collected to get collected anyway.

7.6 Memory Model

Something unspecified in both the CLI standard and the Java VM specification is the memory model. An implementation of a VM is free to choose how data is stored in memory, how classes and objects reference each other and so on. A good object model will try to optimize the most occurring operations, but also minimize memory usage.

7.7 Implementation Details

7.7.1 Sun SDK 1.4

This Sun release is an implementation of the Java 2 Platform.

Garbage Collection Because of the impossibility to directly access the memory or to perform random object casting, a fully accurate GC can be deployed. The Sun JVM makes use of different GCs, depending on the kind of environment and objects.

- **Generational Copying Collection** This GC results in the decrease of the frequency and duration of the GC's execution time. New objects are allocated into an object nursery, the amount of living objects of which is already small at collection time. The living objects are moved elsewhere, after which the nursery is considered empty again.
- **Parallel Copying Collector** For hardware with multiple processors, this GC will search and move the living objects of the young generations in parallel.
- **Mark-Compact Old Object Collector** The memory area containing the old objects doesn't need to be collected often. However, some occasions do require this, e.g. on request of the running program. The VM will make use of a Mark-Compact algorithm to eliminate memory fragmentation and thus speed up of allocation.
- **Incremental Low-Pause Garbage Collector** The previous GC can lead to pauses of arbitrarily length, which is unacceptable for some applications. This collector guarantees smaller, constant pauses.
- **Mostly Concurrent Mark-Sweep Collector** This algorithm reduces GC pause times by doing most of the tracing and sweeping work concurrent with programming execution. Spare processors and processor cycles are used to collect large heaps.

Execution features The Java HotSpot VM combines interpretation and JIT-compilation while tracking *hot spots* in the code. These performance critical parts will be optimized and inlined, which reduces the amount of method invocations. Dynamic deoptimization can be performed as a result of dynamic class loading or new hot spot analysis. It comes with two execution engines that use the same other parts of the VM. Depending on the kind of program the right compiler must be chosen.

The **client** compiler is targeted for client applications. Typical problems for those applications were the slow start-up time and the memory usage, those issues have been improved in this compiler. Three phases are used to transform

byte code into native code. The first phase, platform-independent, constructs a high-level intermediate representation from the byte codes. This is followed by a platform-dependent low-level form. The final phase generates machine code. The focus lays on local code optimization, since global ones cost too much time.

The **server** compiler is meant for long-running server applications. The start-up time and memory footprint matter less in those environments, more time is thus put in analysis en optimal compilation of code parts. A static single assignment-based intermediate representation is used to perform the optimizations on. The time to fully optimize the code is paid back by lengthened execution times typical for server applications. A detailled description of the server compiler's internals is given in [MCC01].

The performance curve deduced from the theoretical descriptions in [Sun02] throughout time is graphically represented in figure 7.7, with the server compiler being slower at initial stage because of the heavy optimizations performed at that time.

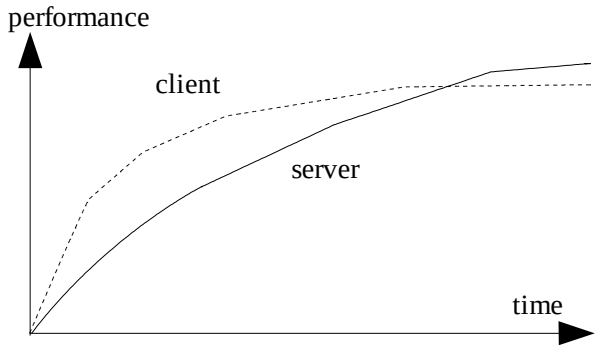

Figure 7.7: Compiler Performance

7.7.2 Microsoft .NET

Generational Garbage Collection MS.NET has a generational garbage collector. Newly created objects will be put into the part of memory containing objects of the first generation, which have not been through a GC cycle yet.

However, if the new object is large, it will directly be placed in a higher generation because it is assumed that large objects will live longer. When the amount of memory allocated in the first generation exceeds a certain number, the GC will start working. Objects still alive in the first generation will be moved to the second and the first will be cleared. Then, the first generation is again ready for new objects. Memory in the other generations is compacted.

The CLR contains three generations, whose size's can be altered at runtime. When collection starts, all threads are stopped temporary at a suitable place in order to avoid some thread pointing to an object that has been moved by the GC. When moving objects the order is remained in order to have a good locality.

Garbage collection in the Microsoft .NET Framework is discussed in Microsoft's Developer Network [NET].

JIT-compilers Three JIT-compilers compile the IL code into native executable code.

- The standard JIT-compiler compiles to optimized code on a per-method base. This native code is cached for later use.
- The EconoJIT compiler is meant for fast compilation and little memory usage. It will not store the compiled code, but compile the code again when it is later used again. The performance is less optimal as less time is put into compilation.
- The third variant is the preJIT. The compilation to native code is done at the so called install time. When the application is deployed on a certain computer, the install procedure includes a compilation process. The native code is stored and will be used at runtime. This reduces the start-up time as mentioned earlier.

Native Image Generator Assemblies that are often used can be precompiled into a native image by the Native Image Generator (NGEN). This image will be used at runtime, preventing dynamic compilation. Typically the base class libraries are precompiled. MS.NET makes thus use of AOT-techniques by default.

7.7.3 Mono

Boehm's GC Mono makes use of the garbage collector made by Boehm and Demers [HA]. This GC provides conservative, incremental and generational collection. It is conservative because it works on C and C++ code, Mono is written in C and uses C structures to represent objects and classes in memory.

Three Execution Methods Mono provides no less than three execution methods. The first one is the classic interpreter, which is mainly advised to be used on platforms where it doesn't have a JIT for yet (e.g. s390, StrongARM).

The JIT compiler (currently for x86, Solaris/SPARC and PowerPC) and an AOT compiler complete the choice. The JIT compiler makes use of three IRs, which help to optimize the code. One of them is an SSA representation. The AOT compiler allows to precompile the assemblies which will lead to a reduced start-up time. Mono still requires the assembly to be available at runtime: it contains the meta data needed to understand the generated native code.

7.7.4 Kaffe

Kaffe is an open source implementation of the Java virtual machine provided with a JIT-compiler. Although it's not up-to-date with the latest commercial VMs, it is used as a base for a lot of research. Kaffe's code is freely available and can thus be adapted to suite the needs of certain research projects.

It comes with a conservative mark-sweep garbage collector. It is conservative, again because Kaffe is written in C++.

7.7.5 GCJ

Native GCJ is a Java-to-native compiler that compiles both Java source and Java class files to native shared and static libraries. Those applications are linked with the GCJ runtime, containing class libraries, a garbage collector and a byte code interpreter. GCJ is part of the GNU Compiler Collection (GCC) [Gnu].

GCJ compiles both code and meta data into a native binary. The transformation of the byte codes and the representation of the meta-data in the same file are described by Bothner [Bot97]. Just like Mono, GCJ makes use of Boehm's conservative garbage collector.

The additional advantages of GCJ are the link-compatibility with C and C++ object files, modest memory usage and fast start-up time .

7.8 Conclusions

- From an internal point of view, the Java and .NET VM contain similar structures to store and execute the application at runtime. Methods are represented as frames on a stack, containing on their turn stacks for operations and storage. Objects reside in a garbage collected heap, meta data is gathered in a separated structure.

- Garbage collection is a well studied domain, with a variety of methods available depending on the specific application domain.
- A broad range of execution techniques exist. Which one to choose depends on criteria one finds most important in a certain situation.
- The commercial VMs contain a lot more technology: multiple JIT compilers and garbage collectors, between which they can switch depending on the situation.
- A number of implementations of Java and .NET have been reviewed, the different execution techniques of them are schematically brought together in figure 7.8. This figure is a simplified version of a figure in [KCSL00] and focusses on execution techniques handled here.

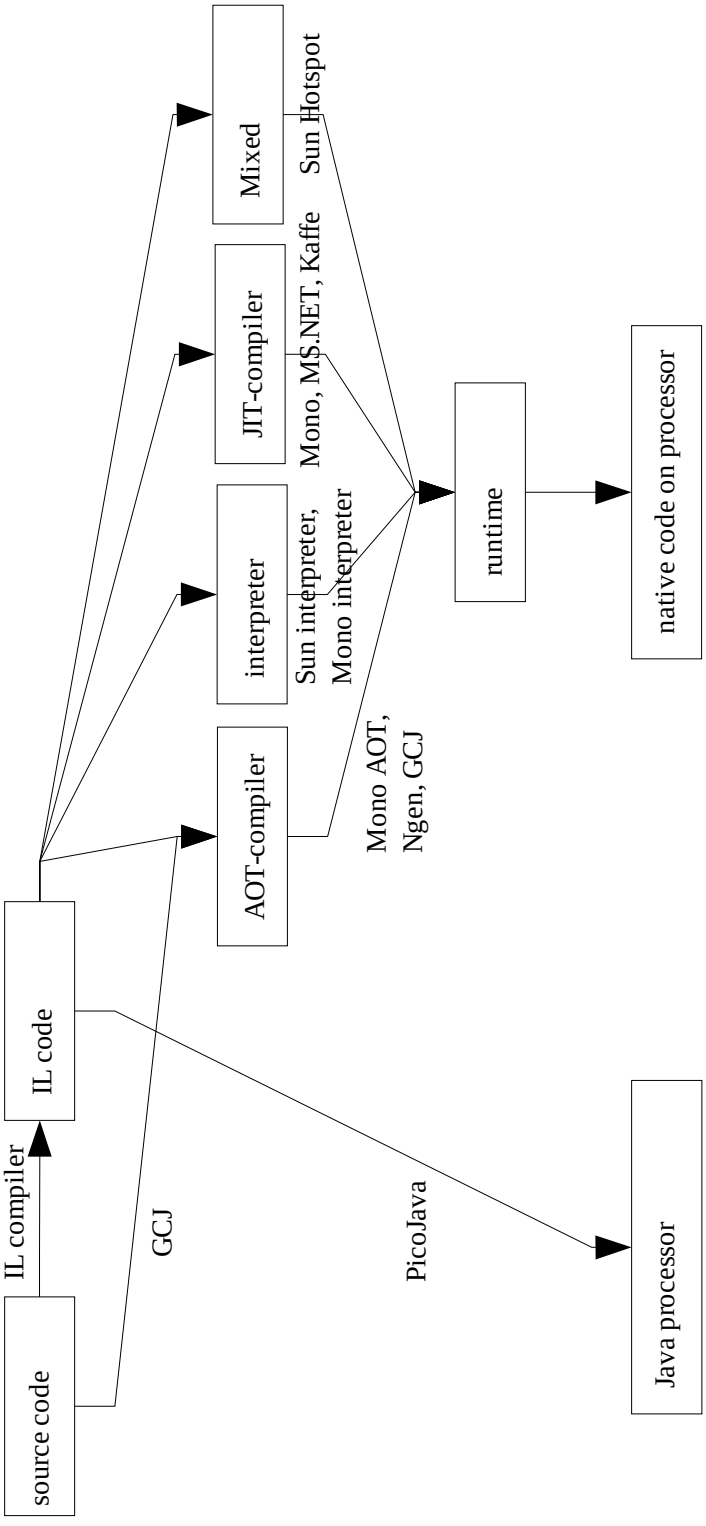

Figure 7.8: Execution Engine possibilities

Chapter 8

Performance of Java and .NET

8.1 Goals and preparation of benchmarks

Benchmarking needs some preparation as it is a difficult task. A detailed explanation is needed to describe what exactly is going to be benchmarked and how that can be done in a way certifies that the tests measure what is intended to be measured and that they are not influenced too much by side effects.

The goal of this chapter is to measure difference in performance across operating systems, Java and .NET virtual machine implementations as well as specific VM internal configurations. All those variants will be installed on a typical x86 desktop computer.

Which questions will be answered? The results should be reviewed from many sides:

- First, performance differences between Java and .NET platforms can be seen. The code presented to both platforms is as identical as possible, the tests are thus suited to draw conclusions. Commercial as well as research and open source implementations of both platforms will be tested and compared. Linked with the detailed information about the implementation of the example VM components acquired in chapter 7, the requirements for a VM to satisfy as a performant machine will be found. When other criteria than performance are important, the implication of those on the performance will be found too.

- Additional to the previous point, the often asked question which of either Java or .NET is the fastest will be answered, but still under the constraints of the tests and the environment as presented here. If all implementation

of a certain platform perform notably on a certain test, this can point to a platform specific weakness or strength.

- Is the operating system an important factor? Two of the tested VMs: Java Sun SDK as well as Mono are deployed on both Windows and Linux. Because portability is one of the praised features of those platforms, it is interesting to investigate if portability also implicates the preservation of speed across operating systems on the same hardware.

- How does the performance compare to natively executed code acquired with GCJ? The performance penalty that comes with the start-up of a VM and the features that it provide at runtime has always been a disadvantage in the acceptance of an application controlled by a VM as alternative to natively compiled applications.

Two major parts So, what exactly needs to be tested? The benchmarks will be divided into two major parts. First, based on the instruction sets, source files will be written targeted to measure one or a couple of instructions at a time. This way, striking differences for similar instructions on both platforms will be noticed. In order to limit the amount of tests, only tests for instructions that frequently occur in typical code will be run. The statistics of chapter 6 help in choosing the right ones.

These kind of tests are called micro-benchmarks or synthetic benchmarks. They do not directly represent real programs and have no particular computational target. From these results only, you can't judge a certain execution environment, but rather it is hoped to uncover certain oddities in some VMs design. These results have to be placed in a context too.

For the second part of the benchmarks presented here, the Grande benchmark suite will be used. As explained earlier, they can be seen as performance-critical parts of real applications, containing algorithms that occur in lots of software.

Timing For the micro-benchmarks, the time measured does not count the time to start-up the virtual machine nor initializing the test class. In this way just the time for the execution engine to execute the piece code is timed. A warming up round in the test framework will provide already JIT-compiled code before the actual timing takes place.

In the Grande benchmarks, the interest lies in the complete execution time, including start-up, initialization and shutdown. To test the execution time of the whole program, one can use the *time* command, commonly found in UNIX operating systems. Other operating systems with equivalent shell software, such as Linux, also have this command. Unfortunately in Windows no such command exists, however installing Cygwin [Cyg] in Windows can provide a

solution, providing a similar shell. This *time* command provides real time, user time and system time. The user time is the desirable value then.

8.2 Hardware, Operating Systems, Virtual Machines and Versions

The test machine is an AMD Duron 600Mhz (i686) with 512 MB RAM. Two different operating systems are used: Redhat Linux 9 and Windows XP Pro. For Linux, the following VMs are used: Sun SDK 1.4.2_04, Kaffe 1.0.7 (Java version 1.1), GCJ 3.2.2 and Mono 0.31. Windows has Sun SDK 1.4.2_04, Mono 0.31 and Microsoft .NET 1.1.4322.

In order to be able to reply the benchmarks, the configurations of the VMs are shown in table 8.1. Because some of the tests use large sets of data, the JVMs which don't automatically enlarge their heap size when necessary are given a large enough start-up heap size to avoid the abortion of tests.

8.3 Micro-benchmarks

8.3.1 Test Construction

These tests are mostly newly created, with inspiration coming from the Mono and the PNetMark benchmarks [PNe]. All the tests share the same framework as showed in figure 8.1, similar to the tests method in [Wil03]. Each test code is wrapped inside two loops, one of which is customizable in the number of iterations. These loops enlarge the time of the few lines of code from the test itself, in order to get a significant difference in absolute execution time between the various test runs.

In order to give JIT-compilers the chance to compile and cache the test method, a warm-up call is executed before the start of the actual test. This happens under the assumption that in normal applications methods will typically be executed more than once. The first method call would take more time than the following because of the compilation, although this is not the case when the compiler dynamically recompiles methods throughout program execution. The start-up time is thus not measured in these tests.

The execution time is timed with two facilities in C# and Java, specifically designed to provide the current time in milliseconds. By calling these methods just before and just after the real test, and then subtracting these two values, the elapsed time during the execution of the test code is given. The language facilities in C# and Java are respectively *System.DateTime.Now.Ticks* and *System.currentTimeMillis()* .

OS	*VM*	*Abbreviations*	*Command Line*	*Comment*
Windows	Sun SDK 1.4.2_04	SunSDKClient	java	-Xms256m -Xmx256m for heap size
	Sun SDK 1.4.2_04	SunSDKServer	java	-Xms256m -Xmx256m for heap size
	Sun SDK 1.4.2_04	SunSDKInt	java -Xint	-Xms256m -Xmx256m for heap size
	Mono 0.31	MonoJIT	mono -O=all	
	Microsoft .NET 1.1.4322	MS.NET		csc /O at compile time, will be passed to JIT
Linux	Sun SDK 1.4.2_04	SunSDKClient	java	-Xms256m -Xmx256m for heap size
	Sun SDK 1.4.2_04	SunSDKServer	java	-Xms256m -Xmx256m for heap size
	Sun SDK 1.4.2_04	SunSDKInt	java -Xint	-Xms256m -Xmx256m for heap size
	Kaffe 1.0.7	Kaffe	java	-ms256m -mx256m
	GCJ 3.2.2	GCJ		gcj -O2 at compile time
	Mono 0.31	MonoJIT	mono -O=all	

Table 8.1: VM settings

```
1  public class classname {
2          // some possible class members needed in the test
3          ...
4          public long testname(int repeat) {
5                  long start = 0, end = 0;
6                  // some initializations
7                  ...
8                  // start of measure
9                  start = System.currentTimeMillis();
10
11                 for(int j = 0; j < repeat; j++) {
12                         for (int i = 0 ; i < 500000; i++) {
13                            // actual code to be tested
14                            ...
15                         }
16                 }
17
18                 // end of measure
19                 end = System.currentTimeMillis();
20
21                 return (end-start);
22         }
23
24         public static void main (String args[]) {
25                 int repeat = 500;
26                 long time = 0;
27                 classname b = new classname();
28
29                 time = b.testname(1); // warmup
30                 time = testname(repeat);
31
32                 // print test time
33                 ...
34         }
35 }
```

Figure 8.1: Micro-benchmark test framework

OS	*VM*	*sumsub*	*muldiv*	*logic*
Windows	SunSDKClient	100.0	100.0	100.0
	SunSDKServer	100.0	79.8	0.0
	SunSDKInt	2769.9	796.7	1511.7
	MonoJIT	571.3	50.4	23.9
	MS.NET	55.2	60.0	24.7
Linux	SunSDKClient	106.1	81.7	104.6
	SunSDKServer	120.4	53.1	0.0
	SunSDKInt	3186.8	862.7	1699.6
	Kaffe	664.6	63.0	60.8
	gcj	75.2	62.4	6.0
	MonoJIT	594.9	73.1	28.6

Table 8.2: results of arithmetical micro-benchmarks (in % relative to the result of SunSDKClient on Windows)

8.3.2 Micro-benchmark execution

The tests and their results will be grouped together and discussed in the same manner as the underlying byte codes are partitioned in chapter 2. The Mono interpreter, which was initially foreseen to be included, is too slow to execute all tests. On the few tests performed with it, it was a factor 70 to 350 slower than the Sun Java VM.

Arithmetical tests

sumsubtest, muldivtest and logictest These tests exist solely of basic mathematical operations addition, subtraction, multiplication, division (integer division) and logical operations. Arithmetical instructions represent 8% to 10% of the total instruction count.

The results are presented in table 8.2. Some facts that strike:

- The SunSDKServer compiler is able to optimize the *logictest* in such a way that there is no noticeable execution time. This means that the JIT-compiler can detect that the code inside the loop is useless and doesn't need to be executed. The time needed to find this out isn't measured, since this happens before the actual method and thus before the start of the timing.

- The VMs with quite some optimization facilities put too much time in optimizing in the *sumsubtest*. Mono, Kaffe and MS.NET too perform the best.

- The native code generated by GCJ can be beaten in performance by JIT generated code once the compilation process is done.

OS	*VM*	*load and store*
Windows	SunSDKClient	100.0
	SunSDKServer	0.0
	SunSDKInt	4044.3
	MonoJIT	33.2
	MS.NET	33.2
Linux	SunSDKClient	103.8
	SunSDKServer	98.9
	SunSDKInt	4142.5
	Kaffe	85.9
	GCJ	52.1
	MonoJIT	34.3

Table 8.3: results of micro-benchmarks testing load and store instructions (in % relative to the result of SunSDKClient test)

- When comparing execution times of VMs that run twice, it is clear that the *logictest* benefits the most of optimizations.

Load and Store

These tests make maximal usage of the instructions that load and store between local variables and the operand stack. Table 8.3 learns that .NET is in favour here, being three times as fast as Java, for MS.NET as well as for Mono. Even GCJ is thereby surpassed. This test is actually the most important one of all the micro-benchmarks, as load and store instructions count for no less than 40% to 50% of all instructions.

Addressing Classes and Value Types

Instructions interacting with classes are important in object oriented languages. The execution of these tests heavily relies on the object memory model used in the different implementations.

Field Access First, fields will get the focus (table 8.4). The tests measure the time needed to access both an instance and a static field. The first one is the second most used instruction in both Java IL and CIL (10% of instructions). The latter one appears less frequently, only counting for 1% to 2% of the instructions. The fact whether access to an instance field is faster than to a static field depends on the kind of object memory model. In optimal VMs more effort should be put in making instance field access faster rather than static access.

Basically, two kinds of memory models are observed. One type has an equally fast system for accessing both static and instance fields. Mono and GCJ fall

OS	*VM*	*instance field access*	*static field access*
Windows	SunSDKClient	100.0	73.0
	SunSDKServer	0.0	0.0
	SunSDKInt	1214.0	1098.0
	MonoJIT	43.3	43.0
	MS.NET	29.0	44.0
Linux	SunSDKClient	73.9	76.0
	SunSDKServer	60.2	70.0
	SunSDKInt	1306.5	1372.0
	Kaffe	68.2	49.0
	GCJ	52.0	52.0
	MonoJIT	45.2	45.0

Table 8.4: results of micro-benchmarks testing field access (in % relative to the result of SunSDKClient test for instance field access on Windows)

in this category. The other type has faster access for instance fields, for example MS.NET. Sun's VM shows strange behaviour as the performance seems to change depending on the operating system.

Object creation and method calls Table 8.7 shows that Sun Java and Microsoft's .NET are more or less equally fast in creating new object instances. The other machines lag behind and are between 6% and 40% times slower. As a general trend, except for GCJ, it is observed that static method calls are faster than non-virtual instance calls.

When comparing the cost of a virtual method call to a non-virtual method call, it is observed that for the Java VMs, the cost is more or less the same (table 8.6). In .NET however, large differences in favour of non-virtual method calls are noticed. The time to call a non-virtual method in Ms.net is only 7.7% of the time to call a virtual method!

This fits perfectly with the programming languages Java and C#. As mentioned earlier, in Java all methods are virtual by default, while in C# they are not. Because methods in C# are non-virtual by default, it's unlikely that there will be more virtual than non-virtual methods in an application, hence the focus on non-virtual ones.

Two .NET languages supported by Microsoft have virtual methods by default: J# and the scripting language JScript. They will suffer the most under the slow processing of virtual method calls.

Other OO tests The instructions tested in the following micro-benchmarks appear less frequently in real code, but are nevertheless interesting after the IL

OS	*VM*	*new instance*	*non-virtual method*	*static method*
Windows	SunSDKClient	100.0	100.0	100.0
	SunSDKServer	82.7	0.0	0.0
	SunSDKInt	895.2	3552.1	2273.7
	MonoJIT	908.7	99.6	47.8
	MS.NET	102.4	66.7	48.1
Linux	SunSDKClient	102.5	105.0	98.4
	SunSDKServer	89.5	138.5	115.4
	SunSDKInt	967.1	3739.9	2472.8
	Kaffe	4094.9	519.3	224.8
	GCJ	581.0	366.4	574.2
	MonoJIT	634.0	107.0	49.9

Table 8.5: results of object orientation related micro-benchmarks (in % relative to the result of SunSDKClient on Windows): creating a new instance, virtual versus non-virtual method calls and static calls

OS	*VM*	*Non-virtual / Virtual*
Windows	SunSDKClient	100.0
	SunSDKServer	X
	SunSDKInt	96.4
	MonoJIT	31.6
	MS.NET	7.7
Linux	SunSDKClient	100.0
	SunSDKServer	100.0
	SunSDKInt	94.7
	Kaffe	88.2
	GCJ	96.3
	MonoJIT	32.6

Table 8.6: Non-virtual method call relative to virtual method call

OS	*VM*	*cast class*	*box*	*isinst*
Windows	SunSDKClient	100.0	100.0	100.0
	SunSDKServer	62.3	59.9	28.5
	SunSDKInt	1424.1	1173.1	1196.9
	MonoJIT	124.3	366.8	109.19
	MS.NET	499.5	116.9	71.4
Linux	SunSDKClient	105.1	112.6	105.1
	SunSDKServer	55.1	80.7	29.63
	SunSDKInt	1519.7	740.0	1232.6
	Kaffe	999.4	1259.0	652.2
	GCJ	559.25	156.4	295.61
	MonoJIT	129.96	336.6	107.19

Table 8.7: results of object orientation related micro-benchmarks (in % relative to the result of SunSDKClient on Windows): creating a new instance, virtual versus non-virtual method calls and static calls

explanation in chapter 4.

- **castclasstest** this test measures the time it takes to cast from an instance of a super class, instantiated as one of the subclass, back to the subclass and change a member field.
- **boxtest** tests the time it takes to assign an instance of a subclass to an instance of a super class. In this case, an instance of the test class itself is assigned to the root of the object hierarchy, *System.Object* in C#, and *java.lang.Object* in Java.
- **isinsttest**: This instruction checks if the given object can behave as an object of the presented type.

In this test, the Sun VM is clearly the fastest. The server compiler is even 1.25 to 3.5 times faster than the client compiler. Mono and MS.NET come relative close, but Mono is up to 6 times slower for the *boxtest* and MS.NET 8 times slower for the *castclasstest*. However, these instructions don't appear as frequently as the previous ones.

Array Instructions

Array related instructions, which count for 6% in the Java Grande tests, and for 9% in the .NET Grande tests, are tested here in table 8.8 using computations with integer arrays.

Performance of SunSDKClient, MonoJIT and MS.NET all lie in a 20% range, the SunSDKServer however is more or less 3 times and GCJ 4 times faster.

OS	*VM*	*array test*
Windows	SunSDKClient	100.0
	SunSDKServer	34.5
	SunSDKInt	559.6
	MonoJIT	104.0
	MS.NET	94.9
Linux	SunSDKClient	103.0
	SunSDKServer	33.8
	SunSDKInt	579.0
	Kaffe	121.4
	GCJ	24.5
	MonoJIT	108.4

Table 8.8: results of micro-benchmarks testing array usage (in % relative to the result of SunSDKClient test)

8.4 Grande Benchmarks

8.4.1 Contents

The sequential part of the Grande benchmark suite contains typical applications demanding a serious amount of CPU cycles and memory. Parts of this benchmark are derived from the SciMark Benchmarks [Sci], which try to measure performance of algorithms used in scientific and engineering applications. The sections of the original javaGrande suite used are section 2 and 3.

Section 2 contains short pieces of computation intensive code:

- **Fourier coefficient analysis** The coefficients of an exponential function are computed, thereby relying on trigonometric libraries.
- **LU Factorisation** This technique deals with matrices using partial pivoting. Lots of problems result in a set of linear equations, for example computations in electric circuits.
- **Jacobi Successive over-relaxation** This test contains a method to solve a linear system of equations.
- **Heap sort** An array of integers is sorted using the heap sort technique. Sorting data is frequently used in lots of programs.
- **IDEA encryption** This test performs an encryption and decryption of an International Data Encryption Algorithm (IDEA)
- **Fast Fourier Transformation** This technique is often used in signal processing. The test relies on random numbers and trigonometric functions.

- **Sparse Matrix multiplication** Matrices with lots of zeroes can be more efficiently (in time and space) computed than using traditional matrix multiplication methods. These algorithms are heavily used in signal and imaging processing, but also in document retrieval.

Section 3 contains larger pieces of code with a presumably longer running time. They contain while simulations and software to solve problems, rather than just an algorithm. The following parts are contained:

- **Search** A connect-4 game is solved an alpha-beta pruned search.
- **Euler** A fluid dynamics problem is solved here.
- **Molecular Dynamics** This test contains a simulation for particles interacting under certain constraints.
- **Monte Carlo** This simulates computes a financial situation using Monte Carlo techniques: price products derived from the price of an underlying asset.
- **Ray Tracer** A 3D scene is rendered by this ray tracer.

These tests will be run with two datasets differing in size, to control behaviour of the VMs with different data sets. Some of the tests validate the computed result against a reference value. If the difference is too large, this will be reported.

8.4.2 Relevance

Why are these tests relevant? Finding or creating relevant benchmark tests is not easy. Such tests are created to simulate real (that is, a program that performs something useful) program behaviour. But, in order to behave as benchmark, they need to be adapted in order to be measured. In this case, tests are needed that are sufficiently large to test the suitability of the VM, but those tests need to be available in two languages: one time in Java (or another language that compiles to Java byte Code) and another time in a language that targets .NET. Real applications come in different forms: client applications with a graphical user interface, server applications which need to answer client requests, command line applications performing long-running background jobs, ... What is of interest here, are those pieces of code that take quite some time to compute: long running pieces of code which are required to perform as fast as possible. In general, this means that the code that is benchmarked must contain as less as possible functions of the class libraries that come with the VM because these are replaceable if it turns out that they contain inefficient algorithms. Graphical user interfaces, network libraries, I/O are thus kept out of the tests.

Why are these tests not so relevant after all? Although these tests reasonably represent the VMs power, they can be criticized because real programs do make use of the libraries belonging to the VM. Users will complain when a GUI is not responding enough, server applications will not be used if the networking support doesn't provide enough throughput etc.

8.4.3 Normal Run

In order to be able to separate the influence of the various VM parts, a couple of test runs with different VM configuration are held. The first run is the normal run, containing the normal VM settings as they would be used in production environments. That means that possible optimizations are enabled.

The results are presented in table 8.9. First thing to be noticed is the gap between interpretation and compilation. However, between interpretation engines too, there are major differences: where the Sun SDK interpreter is 1.6-22 times slower than its JIT compiler, the GCJ interpreter already is 2.5-23 times slower ending with Mono's interpreter being 4-68 times slower. Because of these enormous differences monitored, results for interpreters will no further be handled. If performance is important, then an interpreter is most likely no option. In order to still have a reference, the Sun interpreter will still be tested.

For the setion 3 tests, it is noticed that the difference between JIT compilation and interpretation has increased in comparison with section 2.

Some tests fail the result verification: GCJ and Mono. Most of the time these are near misses, still this raises questions since the size of the primitive types is standardized and the mathematical and random number libraries (those libraries are used by the tests) should return the same results across all Java implementations and all .NET implementations.

- Generally speaking, MS.NET is the better performer from all the competitors, followed by the natively executing GCJ. MS.NET is between 2% and 73% faster than the reference Sun Client benchmark.
- The Sun Java VM is faster on Windows than on Linux (most tests are between 2% and 12% faster). Presuming that they contain exactly the same components, the performance gap can either be a problem in the operating system transformation layer of the VM or a property of the operating system itself. Given that the Mono distribution tends to perform better under Linux than under Windows, the other side around, the former is suspected.
- In one test, Mono fails to produce a result under Windows and aborts without any error. Multiple problems with the installation have been observed too, it seems that the distribution for Windows isn't quite ready yet. The very slow *Seriestest* seems to confirm this.

	Windows			Linux							
	Sun Client	Sun Server	Sun Int	Mono	MS.NET	Sun Client	Sun Server	Sun int	Kaffe	GCJ	Mono
Series	100.0	41.8	169.3	336.6	27.3	141.9	52.8	201.9	99.6	108.5	54.8
LUFact	100.0	99.9	903.1	147.4	88.3	102.7	111.1	891.1	194.8	85.2	139.0
HeapSort	100.0	87.1	867.1	141.1	76.3	103.1	99.7	895.7	96.6	81.7	134.9
Crypt	100.0	93.4	382.5	X	43.8	111.6	105.9	390.1	119.0	85.9	197.4
FFT	100.0	89.1	492.0	105.8	87.5	102.3	89.3	503.5	152.0	99.4	110.3
SOR	100.0	118.3	1349.9	121.5	93.5	103.2	116.2	1562.1	413.1	98.1	117.1
SparseMatmult	100.0	118.0	351.9	113.5	97.8	103.8	104.6	365.0	159.3	101.5	109.9
Euler	100.0	80.3	1629.6	231.3	116.9	110.0	167.3	1710.7	475.5	185.8	220.3
Moldyn	100.0	113.1	1312.1	110.4	85.4	105.2	133.9	1277.6	1008.5	102.7	103.9
Montecarlo	100.0	137.0	636.4	X	101.8	128.3	146.6	575.8	181.1	136.9	228.3
Raytracer	100.0	80.5	2125.1	155.2	76.9	104.8	89.9	2192.4	608.9	146.7	149.9
Search	100.0	93.0	905.1	157.4	91.7	107.3	103.4	974.6	194.0	112.1	150.6

Table 8.9: results of grande tests (in % relative to the result of SunSDKClient test)

- Java server versus Java client compiler: in general, the server compiler seems faster, sometimes up to twice as fast. This has been noticed in the micro-benchmarks too.

8.4.4 More Memory

The same tests as before will be run, but the data sizes will be larger, between 2 and 10 times as large. Results are presented relative to the times of SunJavaClient compiler, but also the relative difference between these tests results and the results from the normal run are given. These need to be interpreted as follows: all numbers below zero represent runs that are relatively faster in this run than in the previous run, compared to the SunJavaClient. That does not mean that they effectively run faster than that reference. table 8.10 summarizes these results.

- Overall, MS.NET keeps setting the pace, but relatively seen in comparison with the normal test run, the Sun Server compiler comes closer.
- With the increased memory sets, some VMs start to suffer and don't manage to compute a result for the section 3 tests. In particular Kaffe, but also Mono and even the Sun VM.
- It's not only the ability to deal with the large amount of data, but also choosing the right optimizations that cause some tests to be executed fast. Sometimes it seems that the more sophisticated VMs try to optimize too much or the wrong parts, as shown in the section 2 *seriestest*, where the Sun VM is slower than the simpler Kaffe.
- The Montecarlo test with larger dataset seems to be a VM killer as several VMs couldn't manage to produce a result. Surprisingly, it is Kaffe which is foremost the fastest. In the end all of the VMs seem to handle the larger data sets quite well with no excessive results, except for this one test.

8.4.5 No Verification

The verification of class files before their usage is an important addition to the security of the platform. But, if the code is known to be trustful, it should be possible to omit the verification process. As explained in chapter 7, verification is done at class load time, which happens the first time the class is being referenced. In case of large applications, e.g. an applications with a GUI divided over a lot of classes, this verification process can seriously slow down the application's start-up time. Some large Java applications are known to execute without verification in order to limit start-up time.

Unfortunately, the benchmarks presented here don't consist of many classes. The tests of section 2 are mostly contained within one class, the ones of section 3 consist of between 5 and 14 classes. Still there is some distinction in execution without verification.

	Windows				Linux				
	Sun Client	Sun Server	Mono	MS.NET	Sun Client	Sun Server	Kaffe	GCJ	Mono
Series	100.0	42.9	358.0	26.1	140.9	112.4	95.7	106.2	52.2
LUFact	100.0	86.2	166.5	87.4	102.6	89.7	218.9	95.0	141.7
HeapSort	100.0	83.0	130.2	81.8	102.8	91.7	101.1	89.2	127.6
Crypt	100.0	89.6	X	44.0	113.4	96.6	124.0	90.1	206.4
FFT	100.0	98.0	111.3	100.0	103.9	100.7	X	109.6	116.9
SOR	100.0	117.5	137.4	88.7	103.5	109.8	430.4	100.3	120.6
SparseMatmult	100.0	104.1	123.0	95.1	104.6	97.8	155.5	104.3	116.4
	0.0	1.1	21.4	-1.1	-1.1	59.5	-1.1	-2.3	-2.6
	0.0	-13.7	19.2	-0.9	-0.1	-21.4	-0.9	9.9	2.7
	0.0	-4.0	-10.8	5.5	-0.3	-8.0	5.5	7.5	-7.3
	0.0	-3.8	X	0.2	1.8	-9.3	0.2	4.2	9.1
	0.0	8.9	5.5	12.5	1.5	11.4	X	10.2	6.6
	0.0	-0.8	16.0	-4.7	0.3	-6.4	-4.7	2.2	3.5
	0.0	-14.0	9.5	-2.8	0.8	-6.9	-2.8	2.8	6.5
Euler	100.0	117.7	225.6	120.9	110.9	140.3	491.2	185.6	192.9
Moldyn	100.0	104.1	92.0	89.0	102.5	109.8	X	101.5	102.9
Montecarlo	100.0	584.8	X	268.0	X	X	56.7	1035.9	1954.8
Raytracer	100.0	66.1	136.7	70.5	104.6	74.3	X	129.4	128.9
Search	100.0	93.1	165.4	96.9	108.5	97.2	210.9	118.2	158.1
	0.0	37.4	-5.7	4.0	0.9	-27.1	15.7	-0.2	-27.5
	0.0	-8.9	-18.4	3.6	-2.7	-24.1	X	-1.2	-1.0
	0.0	447.7	X	166.1	X	X	-124.5	899.0	1726.5
	0.0	-14.4	-18.5	-6.5	-0.3	-15.6	X	-17.3	-21.0
	0.0	0.1	8.0	5.2	1.2	-6.2	16.9	6.1	7.5

Table 8.10: results of grande tests running with larger data sets (in % relative to the result of SunSDKClient test)

VM	*mean memory surplus*
Kaffe MonoInt	-1%
MonoJIT	+5%
MS.NET	+20%
GCJ	+28%
SunSDKInt	+41%
SunSDKClient	+56%
SunSDKServer	+73%

Table 8.11: Mean memory usage relative to Kaffe

From the competitors in these tests only the Sun and the Kaffe VMs have provisions for running without verification. The results that have collected from runs without verification are only a little better than normal runs, with gains starting from nearly nothing up to 4% for the tests containing the most classes and the biggest size.

8.4.6 Memory Usage

One of the criteria introduced in the beginning of the thesis is the memory usage. By measuring the memory used for the Grande benchmarks, it will get clear if there are large differences between VMs. The memory usage is deduced from the VMs' process memory as displayed by a system monitoring tool.

The results reveal consistent relative memory usage over the set of tests. Kaffe and Mono are clearly the VMs with the least needs. As expected based on the IL and the VM internals, neither Java nor .NET has any special needs for extra memory over the other. Table 8.11 shows the mean memory surplus over the Grande section 2 and 3 tests.

Compared to Sun Java, MS.NET consumes little memory. Keep in mind that the Sun VM runs on multiple platforms, while MS.NET is probably optimized for Windows only and is more integrated in the system, probably sharing resources with other processes. The interpretation mode of the Sun VM and the Mono VM show indeed that interpretation uses less memory than JIT-compilation. The Server compiler with more optimization features needs more than the Client compiler. The native GCJ execution is in the middle of the table, which shows that applications executed under a VM don't necessarily use more memory.

8.5 Performance Conclusions

- A large part of the micro-benchmark tests for the Sun Server VM return zero. It seems that this VM can very intelligently discover the useless computation performed in the tests and optimize them.
- In general MS.NET and second Sun Java are the fastest VMs, which was expected because they are the commercial ones. They even surpass GCJ native execution in many of the tests. All of the VMs seem to work well with large data sets.
- Efforts to measure the time needed to verify IL code and meta data resulted in observed 0% to 4% of total execution time.
- Interpretation indeed takes less memory than JIT-compilation, but the VM in its totality is the most significant factor. Kaffe en Mono are lightweight VMs, MS.NET does fairly well and Sun VM takes 40% to 70% more memory than the smallest one.

Chapter 9

Conclusions and future work

9.1 Thesis Conclusions

Now that the criteria stated in the beginning of the thesis have been researched in detail, it is time to come back to them and see how they have been answered and how they are related. Choosing one of the criteria as most important one almost always has consequences for the other ones.

9.1.1 Criteria revision

Functionality .NET has some more functionality in the IL instruction set, having raw memory instructions and some instructions for functional programming. Another benefit is the possibility to mix managed and unmanaged mode, for example to extend unmanaged code with new components.

Performance It is clear that when performance is the most important criterion, then complex and memory-consuming JIT-compilers are the right execution engine to choose. Start-up time reduction can be reached by AOT-compilation and disabling IL verification. The best ratio between performance and few memory usage is brought by Microsoft .NET. Performance is most needed for long-running server side applications, however Microsoft .NET is not availabe for UNIX and Linux operating systems typically running on those big servers.

Portability Portability is a criteria that is more a consequence of using virtual machine technology. However, for example the same Sun Java virtual machine can be used on quite some platforms, while Microsoft .NET only runs on Windows. An alternative here is Mono, however this .NET implementation is still in beta phase and although performance is increasing with every release, it's not

on the same level as Microsoft .NET (yet). Portability is important in domains where new platforms arise fast, such as the handheld and small devices domain.

Interpretation execution engines are easier to port to new platforms than JIT-compilers. Keep in mind however that the performance of the former is often weak: interpreters are up to 20 times as slow as JIT-compilers, the Mono interpreter even up to 70 times.

Memory and disk resources The Java virtual machine goes a little further in supporting tiny devices like smart cards. There exists a Java processor in silicon, allowing Java to be deployed in really small memory environments. The JAR-distribution form allows zipping of an application, resulting in less disk space than .NET applications and faster network transfers. The class file format however carries a lot of duplicated information.

For the storage and runtime of the VM, more sophisticated (and thus likely more performant) VMs are bigger in size than more simple ones. Performance comes with a price for space resources.

The .NET PE file format carries a lot of legacy parts that are nothing but overhead, especially in non-Windows environments. It does a better job in removing duplication when larger parts of code are brought together into one assembly.

9.1.2 Web of choice

Figure 9.1 provides a graphical overview of the relationships between criteria, VM components and implementations. The figure should be interpreted as follows:

- Criteria that appear near each other are closer related, those that are further are mostly opposite ones.
- VM components that are mentioned in the neighbourhood of some criteria are components that are beneficial for them.
- VM implementations appear near criteria where they are better in.

9.2 Future work

9.2.1 New Java and .NET versions

While research for this thesis was going on, already new versions of both platforms were presented and will be available to the large public within a short period of time.

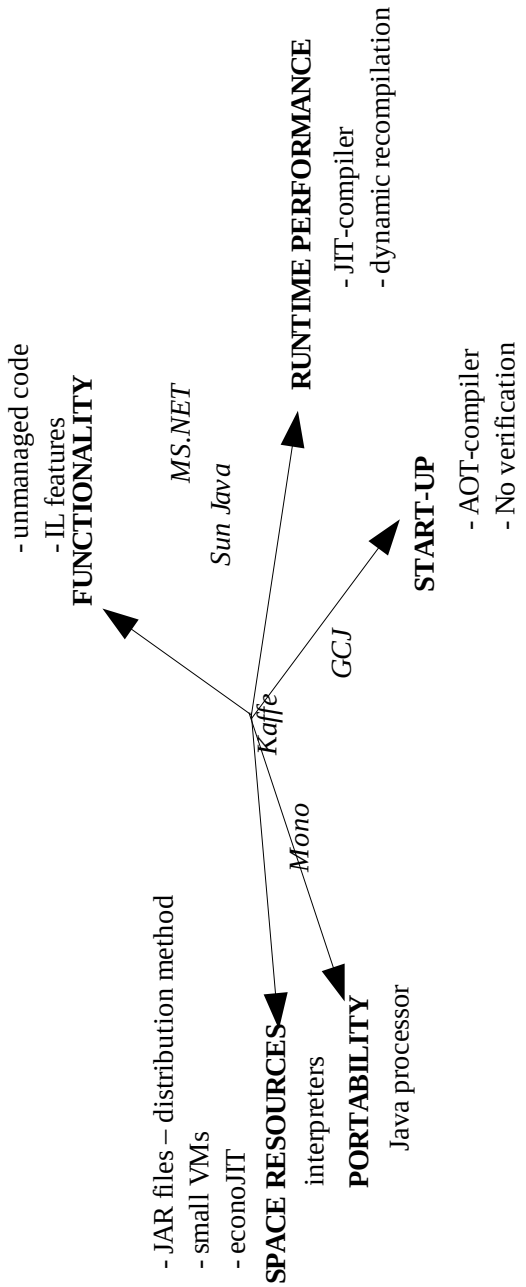

Figure 9.1: Web of choice

The new Java 1.5 series provide the platform with additional language functionality, alongside other innovations such as new APIs, improved performance and changes in graphical user interfaces. The most important IL innovations are the introduction of generics, boxing, an enumeration type and variable argument list. The three last ones of these are already in .NET.

.NET developers too are looking to put generics into CIL as an attempt to prevent lots of boxing and casting.

In the end, there will always be a constant evolution towards more functionality and better performance at one side, and scalability to all kinds of devices at the other side.

9.3 Java and .NET conversion and interoperation

With Java and .NET both being wide spread platforms, it's unlikely that one of them will disappear shortly. Research in the interoperation between them and automatic conversion from one platform to the other is valuable as those problems arise with the increased use of the two platforms.

Automatic conversion tools exist already, but are incomplete. Communication between Java and .NET can be done using middleware such as SOAP, CORBA or DCOM [Cor]. Another way is creating a bridge into native code, as both platform haves APIs to interact with C or C++ code. Some research in that direction has already been done [JRB02], but the performance of these techniques is still to be tested.

List of Abbreviations

ADO.NET	ActiveX Data Objects for .NET
AOT	Ahead-Of-Time
API	Application Programming Interface
ASP	Active Server Page
BCL	Base Class Libraries
CIL	Common Intermediate Language
CLI	Common Language Interface
CLR	Common Language Runtime
CLS	Common Language Specifications
CTS	Common Type System
COFF	Common Object File Format
COM	Common Object Model
GAC	Global Assembly Cache
GC	Garbage Collection
IR	Intermediate Representation
J2EE	Java 2 Platform, Enterprise Edition (J2EE)
J2ME	Java 2 Platform, Micro Edition (J2ME)
J2SE	Java 2 Platform, Standard Edition (J2SE)
JAR	Java Archive
JCL	Java Class Libraries
JDBC	Java Database Connectivity
JIT	Just-In-Time
JLCA	Java Language Conversion Assistant
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
MSIL	Microsoft Intermediate Language
OLE	Object Linking and Embedding
OOPL	Object Oriented Programming Language
OS	Operating System
PAL	Platform Adaptation Layer
PE	Portable Executable
SDK	Software Development Kit
VES	Virtual Execution System
VM	Virtual Machine

Bibliography

[Bot97] P. Bothner. A gcc-based java implementation, 1997.

[Box02] D. Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley, 2002.

[B.V97] B.Venners. Security and the class loading architecture. 1997. http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.html.

[B.V02] B.Venners. *Inside The Java Virtual Machine 2nd Edition*. McGraw-Hill Osborne Media, 2002.

[Cor] Borland Softwrae Corporation. Borland janeva programmer's reference. Technical report.

[Cor99] Microsoft Corporation. Microsoft portable executable and commonobjectfileformat specification, 1999.

[Cor00] Microsoft Corporation. Common language runtime file format spec, 2000.

[Cyg] Cygwin, Linux-like environment for Windows. http://www.cygwin.com, accessed May 2nd, 2004.

[DHR] T. Dowd, F. Henderson, and P. Ross. Compiling mercury to the .net common language runtime.

[DM98] D.Antonioli and M.Pilz. Analysis of the java class file format. Technical report, Departement of Computer Science, University of Zurich, 1998.

[Dot] DotGNU Portable.NET, suite of Free Software tools to compile and execute applications for the Common Language Infrastructure (CLI). http://www.gnu.org/projects/dotgnu/pnet.html, accessed May 2nd, 2004.

[DW03] B. Davis and J. Waldron. A survey of optimalisations for the java virtual machine, 2003.

[ECM02a] ECMA. *ECMA-335: Common Language Infrastructure (CLI) Partition I: Concepts and Architecture*. ECMA (European Association for Standardizing Information and Communication Systems), October 2002.

[ECM02b] ECMA. *ECMA-335: Common Language Infrastructure (CLI) Partition II: Metadata Definition and Semantics*. ECMA (European Association for Standardizing Information and Communication Systems), October 2002.

[ECM02c] ECMA. *ECMA-335: Common Language Infrastructure (CLI) Partition III: CIL Instruction Set*. ECMA (European Association for Standardizing Information and Communication Systems), October 2002.

[EL01] E.Gagnon and L.Hendren. Sablevm: a research framework for the efficient execution of java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[FT00] Inc Foundstone and CORE Security Technologies. Security in the microsoft .net framework. 2000.

[Gnu] Free Software Foundation, GNU Compiler Collection. http://gcc.gnu.org, accessed May 2nd, 2004.

[Gou01] K John Gough. Stacking them up: a comparison of virtual machines. In *Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 55–61. IEEE Computer Society, 2001.

[HA] H.Boehm and A.Demers. A garbage collector for c and c++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/ accessed March 25th, 2004.

[Hun97] J. Hunt. *Smalltalk and Object Orientation: an introduction*. Springer-Verlag, 1997.

[JD93] J.Ellis and D.Detlefs. Safe, efficient garbage collection for c++, 1993.

[JLC] Microsoft's Java Language Conversion Assistant 2.0. http://msdn.microsoft.com/vstudio/downloads/tools/jlca/, accessed May 25th, 2004.

[JRB02] J.Bishop, R.N.Horspool, and B.Worrall. Experience with integrating java with new technologies: C#, xml and web services. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 225–225. ACM Press, 2002.

[J.S03] J.Singer. Jvm versus clr: A comparative study. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169, Proceedings of the 2nd international

conference on Principles and practice of programming in Java, 2003, Kilkenny City, Ireland, 2003.

[KCSL00] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.

[LH83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM, 26(6):419-429*, 1983.

[Li97] Qiaoyun Li. Java Virtual Machine - Present and Near Future. *IEEE Micro*, pages 14–19, May 1997.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition*. Addison Wesley, 1999.

[Man] S.Gentile, Introduction to Managed C++, 01/13/2003. http://www.ondotnet.com/pub/a/dotnet/2003/01/13/intromcpp.html, accessed May 25th, 2004.

[MCC01] M.Paleczny, C.Vick, and C.Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[MCH99] J.A. Mathew, P.D. Coddington, and K.A. Hawick. Analysis and development of java grande benchmarks. Technical report, Advanced Computational Systems Cooperative Research Centre, 1999.

[MG00] E. Meijer and J. Gough. Technical overview of the Common Language Runtime, 2000.

[Mon] Mono, a free implementation of the .NET Development Framework. http://www.go-mono.org, accessed May 2nd, 2004.

[MS99] G. Muller and U.P. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[NET] Garbage Collector Basics and Performance Hints, R.Mariani, Microsoft Corporation. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetgcbasics.asp, accessed May 26th, 2004.

[PNe] Southern Storm Software, benchmarking tool for Common Language Runtime environments. http://www.southern-storm.com.au/doc/pm_faq.html, accessed May 2nd, 2004.

[Sci] National Institute of Standards and Technology, Java Scimark Benchmark for Scientific Computing. http://math.nist.gov/scimark, accessed April 7th, 2004.

[S.F96] S.Fritzinger and M.Mueller. Java security. White paper, 1996.

[S.L02] S.Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[Str00] B. Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000.

[Sun95] Sun Microsystems. The Java Language: An overview. White paper, Sun Microsystems, Inc., 1995.

[Sun02] Sun Microsystems. The Java HotSpot Virtual Machine, v1.4.1, d2. White paper, Sun Microsystems, Inc., 2002.

[Sun03] Sun Microsystems. Java 2 SDK, Standard Edition Documentation. Technical report, Sun Microsystems, Inc., 2003.

[TQ01] T.Thai and Q.Lam. *.NET Framework Essentials*. O'Reilly, 2001.

[VMl] Programming Languages for the Java Virtual Machine. http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html, accessed May 2nd, 2004.

[VP03] V.Kotrajaras and P.Chongstitvatana. Nibbling byte code for resource-critical devices. Technical report, Department of Computer Engineering Chulalongkorn University Thailand, 2003.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[Wil03] M. Wilson. Does C# measure up? *Windows Developer Network*, 2003.