Universiteit Leiden
Opleiding Informatica

A Framework for Cross-Platform Dynamically Loaded Libraries

Name:	Christian Veenman
Studentnr:	1265563
Date:	15/8/2015
1st supervisor:	Kristian Rietveld
2nd supervisor:	Harry Wijshoff

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

There are many applications that make use of plugins. Plugins are extensions to an application that can be attached at run-time and mainly consist of an object file with executable code. Object files that are loaded at run-time are more commonly called Dynamically Loaded Libraries or DLL for short. The current problem is that every operating system only provides support for its own object file format. Developers of cross-platform applications need to manage different plugin binaries for every operating system. They are also limited in their choice of compilers because not every compiler can produce every object file format. One potential solution to this problem is to use a programming language which is compiled to cross-platform bytecode. In turn, this bytecode is translated into machine code at run-time, for instance using a Just-In-Time compiler. However, a disadvantage of this approach is that for computational kernels, which are often used in for example audio and video plugins, JIT compiled codes are often outperformed by code that is compiled to machine language directly. This thesis explores a solution to this problem which offers the performance of native plugins while allowing developers to use the object file format of their choice. This is done by creating a cross-platform framework that can dynamically load plugins on different operating systems without having native support for that object file format.

Acknowledgements

I would like to thank all my friends, family and colleague students that supported me throughout the process of creating this thesis. Most of all I would like to thank my supervisor Kristian Rietveld for providing me with useful new insights and for his continuous support during the project. Special thanks should be given to the second reader, Harry Wijshoff, for his criticism which allowed this thesis to improve.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many software is extensible using plugins. Examples of software that support plugins are Adobe Photoshop, Libreoffice and Mozilla Firefox. A plugin is a code that can be loaded by the application at run-time to provide new or enhanced functionality. There are many advantages when using plugins such as third-parties being able to provide extensions to the application without having to recompile the whole program. Another benefit is that plugins can be hot swapped at run-time, meaning that another or newer plugin can be loaded while the program is still running.

Software that uses plugins with a native binary format (so not JVM or CLR bytecode) can currently only be used if the operating system supports that particular binary format. All major operating systems have their own binary file formats: Windows uses PE-COFF, Unix-based operating systems use ELF while OS X uses Mach-O as binary file format. This limitation particularly limits cross-platform development because developers are not free to use whatever compiler they want to use, but instead are limited to the compilers that can produce the right format. Some compilers might be better at optimizing code or might provide features that are not present on other compilers. This would also limit the usage of compiler and language features to a set that all compilers involved support. Another reason is that for each operating system different binary plugins would need to be compiled and distributed. This makes makes it harder to manage the software and to distribute new plugins to the clients.

In this thesis we investigate a cross-platform framework that can be used by applications to load differently formatted plugins. This means that Windows can load plugins in ELF format which can normally only be used within a Unix/Linux environment. The framework itself should be cross-platform in such a way that it can be compiled for all different operating systems. This way we do no longer need multiple versions of a plugin, but instead only have one plugin for all operating systems that run on the same architecture. A plugin can then be compiled and optimized using a compiler of the developer's choice. The reason to prefer native plugins rather then JVM-based code is that native plugins outperform JVM code in most cases, as will be shown in Chapter 7, which makes them a good choice for applications that require outstanding performance. A schematic overview of the problem and the solution can be seen in Figure 1.1.

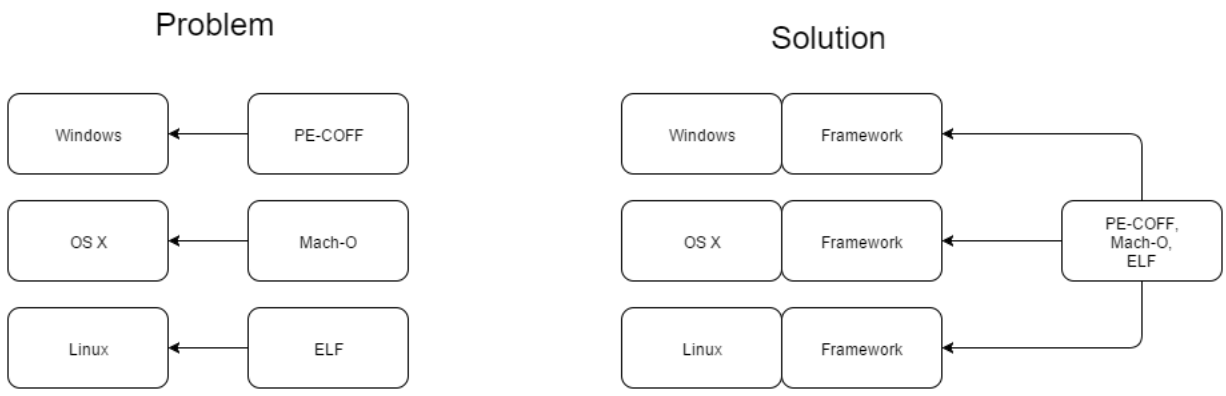

Figure 1.1: The problem and its solution.

As can be seen in Figure 1.2 the plugin and the application can both be compiled with different compilers. The application produced by the compiler is operating system dependent but as long as the plugins are not the different versions of the application on each operating system can all use the same plugin.

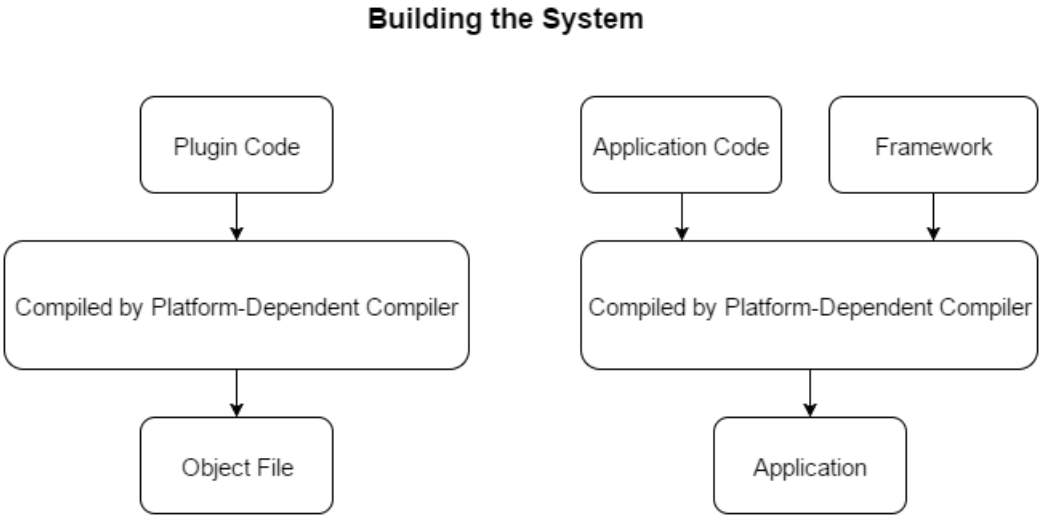

Figure 1.2: Global overview of building the application.

The development of such a framework requires a good understanding of binary formats and of dynamic linking. Therefore the first part of this thesis will go in detail about the necessary background information, such as the ELF format and an explanation about Position Independent Code and Relocation. Since the duration of this project was limited we decided to only support ELF since it is the simplest object format. The framework is designed so that it can be extended with multiple binary file formats and operating systems in the future. The second part of this thesis is devoted to the implementation of the framework and the trade-offs that had to be made. The third part presents an experimental evaluation of the performance of native plugins loaded by our framework compared to JVM bytecode. Currently a solution like the one proposed in this thesis has not been implemented yet. We suspect that this is because of the limitations imposed by the C and C++ programming language that will explained in Chapter 6.

Chapter 2

Related Work

The use of plugins to add extra functionality to a program is not new. However efforts to make a native-code plugin framework that is truly cross-platform without having to recompile the plugin have not been made. A number of plugin systems will be briefly discussed to provide insight into the current state of plugin technologies.

A framework that appears somewhat similar to the framework described in this thesis is Lib Sourcey's Pluga [1]. Pluga is framework that makes it easier to load binary plugins and to use their functionality. The framework does require to compile the plugin separately for each operating system and is therefore limited in compiler choice unlike the framework that is proposed in this thesis. The framework does allows to reuse the plugin code on multiple operating systems provided that it has been written in a cross-platform way. Another notable example of a plugin framework written in C/C++ is described by Gigi Sayfan [2], which provides roughly the same functionality as Pluga but gives the option to write the plugins in C for better compatibility between different compilers producing the same output, but just like Pluga, it does not support the ability to run the same plugin on multiple platforms without recompiling.

Java and C# provide built-in support for executing new code dynamically. Java and C# are both languages that are Just-In-Time compiled and have the ability to adjust code and data at run-time allowing to load classes at run-time. Because code is converted to bytecode before execution, Java and C# can be run on every operating system that has a Virtual Machine (VM) that can run the bytecode. The downside of this approach is that the bytecode needs to be translated before it can be executed resulting in a lower performance. Scripting languages such as Python also have built-in dynamic loading which allows to import modules at run-time, however just like Java and C#, they are also limited in performance because of the translation that need to occur at run-time.

NDISWrapper [3] enables the use of Windows Network Drivers on Linux operating systems. This is done by implementing the Windows Kernel and NDIS APIs on Linux and dynamically linking the Windows Network Drivers which are stored inside a Dynamic Link Library. This way Linux users can use most network cards since Windows Drivers are made for virtually all network cards. So in fact windows drivers are being loaded and executed on Linux operating systems. Our framework is more generic in nature in the ability to load different binary formats on different platforms.

Wine [4] is a free and open source software that allows Windows applications to run on a Unix-like environment. It duplicates functions of Windows by providing alternative implementations of the DLLs that Windows programs call. Similar to NDISWrapper, Wine implements the loading of Windows binaries on Unix-like systems. Our framework is more generic in nature and does not allow any of the operating-specific APIs to be called. Instead the executable must provide an environment and a set of API functions for the plugins to operate in.

Google's PNaCl [5] allows for the performance of native code inside web applications. Web developers can write code that takes advantage of the hardware capabilities of a computer. This can then be compiled to PNaCL bytecode and can be distributed to web clients which will then run the code in a sand-boxed environment. Unlike Java and C# the PNaCL bytecode is only translated to machine instructions once at load-time. Instructions that are not supported by the hardware of a web client can then be substituted with

instructions that can be executed on the web clients hardware. Using a sand-boxed environment the native code is executed in isolation. Because our framework is only meant to be used with plugins from trusted sources, security features are not implemented. Plugins are directly loaded into the executables address space for optimal performance.

Chapter 3

Executable and Linkable Format

To be able to write a framework that can parse ELF files and load them into memory, we first need to gain a thorough understanding on the ELF format. The ELF format stands for Executable and Linkable Format and is mainly used on UNIX-like systems including Linux, Solaris and BSD. The format is used for executables, object files and both statically and dynamically loaded libraries. Figure 3.1 shows the global structure of an ELF file.

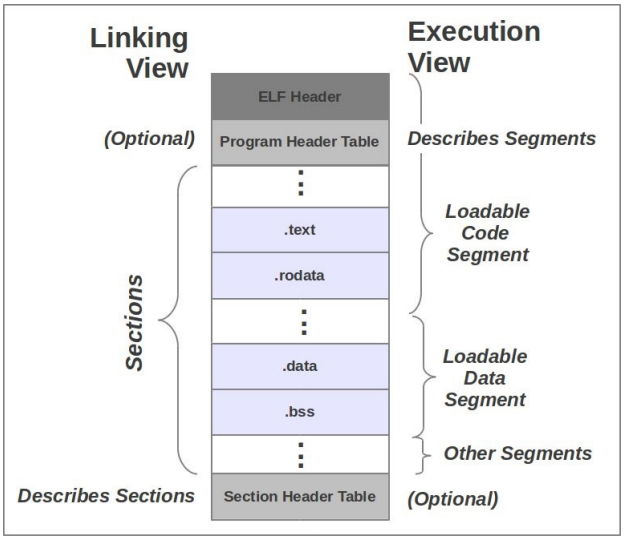

Figure 3.1: Global structure of an ELF file. [6]

As can be seen in Figure 3.1, an ELF file mainly consists of 4 parts: the ELF Header, the Section Header Table, the Program Header Table and the actual sections which groups data of the same type together. In the remainder of this chapter these will be described in turn.

3.1 ELF Header

The ELF Header is 64 bytes long and describes how the file should be parsed. The first 4 bytes of the ELF header contain the magic numbers 0x7F, E, L, F which are used to identify the file as an ELF formatted file. Some of the most important properties are listed in Table 3.1. Note that the description focuses on the properties of main importance to this thesis, for the full ELF specification the reader is referred to [7].

Field	Description
Magic Number	A magic number consisting of 0x7F, 'E', 'L', 'F' to indicate that this is indeed an ELF file.
Class	Describes whenever the file uses a 64 or 32 bit ELF format.
Endianess	Describes the byte-order in the rest of the file. This can be either big-endian or little-endian.
Version	Contains the version of the ELF specification that this file conforms to.
Architecture	Contains a unique number that represents the architecture that the code compiled for.
Type	Specifies if this file is an executable, reallocatable object or shared library.
Section Header Table Offset	Contains the offset to the Section Header Table in bytes relative to the start of the file.
Program Header Table Offset	Contains the offset to the Program Header Table in bytes relative to the start of the file.
Section Header Table Size	Contains the number of entries inside the Section Header Table.
Program Header Table Size	Contains the number of entries inside the Program Header Table.
Offset to the String Table for Sections	Contains the offset to the String Table relative to the start of the file that contains the strings that represent the names for sections.

Table 3.1: ELF Header file

The ELF Header also contains the offset into the file where the Program Header Table and the Section Header Table can be found. This is important because in the ELF file format there is no predefined location for anything aside the ELF Header itself, thus we need the ELF Header to provide us with the locations of these two tables that will give us more information about the file's contents. Furthermore, the ELF Header contains the offset to a String Table relative to the start of the file where the names of all sections are stored. The String Table is necessary to identify the different sections.

3.2 Section Header Table

As stated before, an ELF file is divided into multiple so called sections. Each section represents a group of data which can be anything from static data, executable code or metadata about the compiled code. The Section Header Table is a table that contains metadata about each section. Each section header has a predefined size and format, so the metadata of every section can be easily parsed and displayed. Some of the important properties stored in the section header are listed below:

Field	Description
Section name	This is an offset into the String Table at which the section name can be found. The offset always points to the first character of the string and is terminated with a ASCII \0 character. The section name is not always a unique name because sometimes sections are split up into two sections with the same name when they become too large. The reason that this is an offset, instead of a variable-length string is that otherwise each entry in the Section Header Table would not have a fixed size anymore which would complicate parsing.
Section Type	Describes the type of content in the section, examples are: String Table, Symbol Table, Executable Code, Initialized Data, Uninitialized Data and metadata about the code or compiler used.
Location	This is the location of the section inside the file. It is expressed as an offset from the start of the file.
Amount of Entries	If the section is a table, this field holds the amount of entries that the table contains. If the section does not contain a table this field is ignored.
Section Size	This holds the size in bytes of the section which is used when loading sections into memory.

Table 3.2: Section Header Table

The Section Header Table provides an entry point to every section. There are many different sections and compilers might add sections to hold meta-data or debug information. The most important sections are described below to give a general impression about data is contained in each of the sections. Compiler-specific sections are not considered in this thesis.

3.3 Section Types

.text

Text sections contain all the executable code that the library contains. These sections are typically mapped directly into memory and might undergo some relocations to make them ready to run. When loading these sections it is important to make sure to load them at right virtual address or, in case of using relocatable Position Dependent Code, to adjust all references when relocating the program. Relocation will be further explained in Chapter 5. To avoid run-time relocation Position Independent Code can be used since can be directly mapped into memory without a need to be altered.

.data

Data sections contain data that is already initialized such as static strings. This data can be copied directly into memory and since it is data, it does not have to undergo any transformations.

.bss

BSS stands for Block Started by Symbol Sections but is now almost exclusively used to contain uninitialized data. Where the name originated from is beyond the scope of this thesis. The primary difference with the .data sections is that .bss sections do not take any space in the file. The Program Header that contains this section typically contains two attributes: The size of the segment in the file and the size of the segment when it is loaded into memory. Where for other sections these values are set to the same size in the file of a .bss sections is 0 and the size in memory is set to the amount of uninitialized bytes that need to be reserved. This

way we do not have to make the file larger than necessary. Important to note is that after reserving memory for .bss sections the memory is set to only contain zeros.

.comment

Typically one of these sections is included in the file and contains data about the compiler and linker that produced this file.

.shrstrtab

Only one section with this name may exist in each file. .shrstrtab stands for Section Header String Table and is the table that contain the names for each section. This is the only String Table that is referred to by the ELF Header.

.symtab and .strtab

The .symtab sections contain all symbols used in the program. This is necessary when the library is dynamically linked. The symbols need to be looked up to find the addresses of functions or data that resides in the dynamic library. Each symbol has a name and that name is stored in another section called the .strtab section which stands for String Table.

.rela and .rel

These sections contain information about relocations. Each entry in the table stores the location that needs to be transformed, and the value which should be added to it. The difference between the .rel and the .rela sections is that .rel sections assume that the location already contains a value, and that the value needs to be added to it, while the .rela section can assume the location consist of only zeros and can therefore simply replace the contents by the value. Relocations will be discussed in depth in Chapter 5.

.eh_frame_hdr and .eh_frame

These frames contain information about language specific features such as Exception and Call stack unwinding. For simple programs written in C these sections can simply be ignored.

3.4 Program Header Table

Just as the Section Header Table contains Section Headers, the Program Header Table contains Program Headers. A Program Header describes the different segments in the file. A segment is basically a group of sections that share common properties, for example a group of sections that need to be mapped into memory, or metadata about the program. Segments do not have a name, but the Program Headers contain information that is not present in the Section Header Table. Some of these properties are shown in Table 3.3.

Field	Description
Segment Type	This field contains the type of the segment. This can range from being a loadable segment, a segment containing metadata, or information about the interpreter (if necessary) that needs to run the code.
Virtual Address	This is the address that the segment prefers to be loaded on. Code that uses absolute addresses to data or functions need to be allocated at the right place in memory, otherwise they would crash or the behavior will be undefined. This field is used to do manual relocation or to set up Virtual Memory.
Flags	This field contains data about the volatility of the segment. Executable code is mapped at a memory region with only execute access, while data sections typically have read-write access.

Table 3.3: Program Header Table

Chapter 4

Position Independent Code

Early operating systems were written in a way that programs could only run at fixed addresses. Multiple programs could only run at the same time as long as they were compiled for different addresses and did not have an overlapping address space. Of course this was hardly flexible and makes it very hard to load different sets of multiple programs at once (such as is required for Interactive Systems). Because of operating systems being implemented this way compilers at that time produced programs with Position Dependent Code (i.e. Absolute machine code). Later on this was slowly replaced by PDC that could be relocated at load-time which enabled multiple programs to run at the same time. A small part of the operating system called the dynamic loader is then used to load the program in memory and to adjust the absolute addresses used in the program.

Modern operating systems use Virtual Memory (VM) which implements a mapping between virtual addresses and actual physical addresses. This way programs appear to be loaded in a separate continuous address spaces starting at address 0 while the actual physical addresses of the loaded program might be completely different. The biggest benefit is that this eliminates the process of manually having to adjusting all the absolute references, which saves time when the program is loaded. Instead the mapping from logical to physical addresses is done by hardware. Later on there was a shift to use using Position Independent Code (PIC) for Dynamically Loaded Libraries[1]. DLLs using PIC code typically have the advantage that they can be loaded in memory only once and can be used by multiple processes. This is why this type of dynamic library is also called a shared library. It does not matter where PIC code is loaded in memory and no adjustments are needed at run-time.

4.1 Sharing code with Multiple Processes

The biggest obstacle to achieve sharing of code is that every process runs in its own logical address space and cannot access the address space of another process by default. To solve this a physical page containing the shared code is mapped in multiple logical address spaces, allowing multiple processes to access that code. Note that it is not possible to load the library at a predefined logical address, because a process might already be using that logical address for its own data or functions. This is why the library is loaded at a different logical address for every process. Position Independent Code does not require relocation of code which allows it to be mapped at any free logical address space of a process. This is impossible to realize with Position Dependent Code, because the addresses used inside the code are different depending on which address it is loaded, it is therefore impossible to share PDC with multiple processes. Because it is not known till run-time where variables are loaded a shared library uses a Global Offset Table to look up the addresses of the variables it uses. Obtaining the addresses of functions that are loaded is handled by the Procedure Linkage Table.

[1]This should not be confused with Dynamic Link Library which an implementation of a Dynamically Loaded Library on Windows platforms. The difference between a Dynamically Loaded Library and a Dynamic Link Library is that a Dynamically Loaded Library is a general term for all libraries that are loaded at run-time while a Dynamic Link Library is a specific type of dynamically loaded library used on Windows. In the remainder of this thesis DLL refers to Dynamically Loaded Libraries, unless it is explicitly stated that Microsoft's implementation is meant.

4.1.1 Global Offset Table

Since it is not known till run-time where the variables and functions used by the shared library are loaded, a special table called the Global Offset Table, or GOT for short, is used. The GOT is basically a table where each entry represents a specific external symbol which can be a function or variable. The index of each symbol in the table is already known at compile time. Instead of trying to reference an external symbol directly, a shared library instead looks up the symbol's address in the GOT. The GOT can be created on the fly when the library is loaded because at that point all addresses of external symbol are known. After the GOT has been loaded the library can access all symbols by using the predefined indexes for each symbol. It is important to know that each process has its own GOT table because each process works on its own copy of data used by the shared library. Since the GOT is referenced a lot by the shared library its address is typically stored in a register for quick access. This way the libraries functions just have to use this address and they will automatically refer to the GOT of the calling process. So no extra management has to be done to find the GOT of the process.

4.1.2 Procedure Linkage Table

Similar to the addresses of data that are not known till run-time, the addresses of functions are also not known till run-time. The Procedure Linkage Table (PLT) is similar to the GOT. When the shared library is loaded into memory the addresses of functions are known and the PLT can be built. Since the logical addresses might be different for each process, each process has its own PLT. Instead of storing the addresses of each function inside the PLT, each entry is a small code stub which calls the actual function. The PLT is useful for implementing a technique called Lazy Evaluation [8]. Lazy Evaluation is a technique to delay the loading of functions until they are actually used. When a function is not yet loaded its corresponding GOT entry will contain the address of a function that loads the function and fills the GOT with the right address. When the function is successfully loaded the GOT is filled with the correct address, the function can then be called by the PLT stub. Lazy Evaluation is useful if an application needs to start up quickly but at the cost of lower run-time performance because of the function call overhead. Figure 4.1 shows an example PLT implementation where the address stored in GOT can either refer to the resolver, which is the function that loads the function, or jumps to the instruction that calls the function.

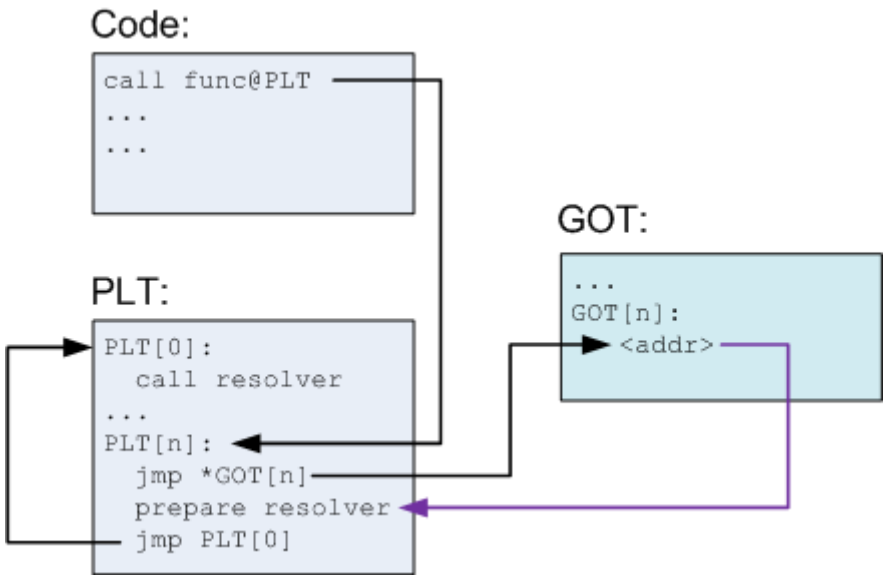

Figure 4.1: The process of calling a function in a shared library. [9]

Before a call is a made, PIC first calls special PLT code. The PLT code will in turn look up the address of the function in the GOT and jumps to that address. Initially the GOT is filled with the address of a resolver function, which is a small piece of code that loads the function we would like to call in memory. After the function is loaded in memory the value inside the GOT will be updated so that all future calls will correctly call the function without the need of the resolver code.

4.2 Comparing PIC and PDC

When comparing both PIC and PDC we see that PIC sacrifices some performance to be able to be shared by multiple processes. In PIC every call or data reference to an external symbol requires looking up its address in the GOT, which makes it slower than PDC which directly references the variable. To share a dynamic library with multiple processes, specific OS system calls are used.

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-0x4], edi

mov     edx, DWORD PTR [rip+0x0]

mov     eax, DWORD PTR [rbp-0x4]
add     eax, edx
pop     rbp
ret
```

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-0x4], edi

mov     rax, QWORD PTR [rip+0x2008fa]
mov     edx, DWORD PTR [rax]

mov     eax, DWORD PTR [rbp-0x4]
add     eax, edx
pop     rbp
ret
```

Figure 4.2: Position Dependent Code vs Position Indepedent Code

As you can see in Figure 4.2 the bold lines show the difference between PDC and PIC code. PDC directly loads the variable from the the .data segment while PIC first needs to load the address of the variable from the GOT and can then load the contents stored at that address. Note that the PDC code is not relocated yet (the offset added to RIP is zero) so the addresses must still be adjusted to their final value.

Chapter 5

Relocations

Before we can execute the plugin code we first need to load the code in memory. The virtual address where the code will be loaded will vary between each run and cannot be known at compile-time. Object files based on Position Dependent Code therefore assume they will be loaded on a fixed address (often zero) but require modification when they are loaded on different addresses. All the references used inside the code need to be modified such that they refer to the right location at run-time. This process of modification is called relocation. In this Chapter we will first describe the process of loading sections into memory. After that we will provide an overview of the relocation of references to symbols.

5.1 Relocation of Sections

As is explained in Chapter 3 an ELF file contains multiple sections which need to be loaded into memory in order to be able to execute the program code. The order in which these sections are loaded can be freely chosen by the implementation. This way a large program does not need a continuous amount of memory, but can instead consist of multiple smaller memory regions. Each of these sections is assigned a virtual address, this is a preferred address on which the code could be loaded in memory.

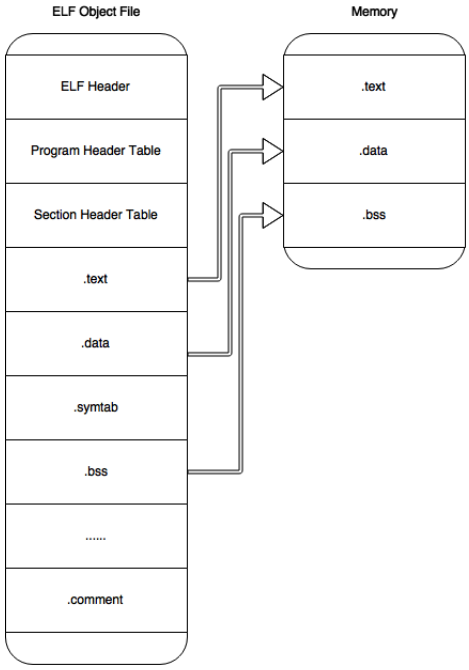

Figure 5.1: The loading of sections in RAM

As you can see in Figure 5.1 only the portions of the ELF file that are actually used by the program are loaded into memory. Examples of loadable sections are the .data, .bss, or .rodata sections for data, or code in the .text section. Sections containing debugging information or language-specific information are also included this way, but are typically not loaded into memory.

5.2 Relocation of Symbols

Inside each section there may be references to its own section or other sections. Examples of this are function calls or global data that needs to be accessed. Since these addresses are not known till run-time we need to relocate these references. The ELF format provides a special relocation section, as was already briefly described in Section 3.3. These special relocation sections contain entries for each relocation that needs to be applied. Each entry describes the location which needs to be adjusted and the operation which needs to be performed on it. For every section that needs relocation a separate relocation section is made. For example: If the .text section needs relocation, the data for these relocations can be found in .rela.text or .rel.text. For relocations in the .data section this would be .rel.data or .rela.data depending on the architecture the program is compiled for. The structure of the .rel and .rela section entries can be seen in Table 5.1 and Table 5.2.

Field	Description
r_offset	This is the offset from the start of the section that points to the location that needs to be modified. The section where the offset is calculated from is the section for which the .rel or .rela section provides relocation for. So a .rel.text section would calculate offsets from the start of the .text section.
r_info	This field consists of two parts. The first 32 bits represent the offset into the symbol table of the corresponding symbol that the location should be referring to. This can be the address of a function or an object for example. The last 32 bits describe the specific relocation to be applied. Different processors use different instructions. It might, for example, be more effective to use relative addressing instead of absolute addressing.

Table 5.1: The .Rel Entry structure

Field	Description
r_offset	This is the same as the .rel r_offset field.
r_info	This is the same as the .rel r_info field.
r_addend	This is a value that is used for relocation but is specific to the type of relocation used. Most of the time this value is just added at the location provided by r_offset.

Table 5.2: The .Rela Entry structure

The structure of both entries is relatively simple. Each of the tables provide an offset to the location and a specific type of relocation to be applied. The more difficult part lies in the fact that every processor uses different call, jump and reference instructions. A 32-bit processor will most likely use 32-bit addresses while a 64-bit processor will use either 32-bit or 64-bit. Some processors also have different types of addressing modes. Examples of this are relative jumps where instead of absolute addresses, offsets are used. The x86 architecture has the notion of short and long jumps, where short jumps use less bytes but are limited in the distance they can jump, while long jumps use absolute addresses. A dynamic loader that wishes to have full support for an architecture would have to implement all the different relocation operations that can possibly be performed on a location.

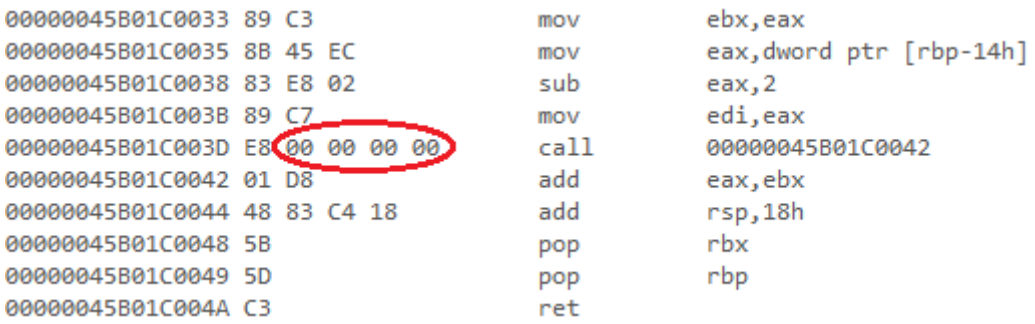

Figure 5.2: x64 Assembly before relocation is done

As you can see in Figure 5.2 the encircled 4 bytes are set to zero prior to relocation. Note that this type of call jumps to an address relative to the next instruction. The call in the example would jump to the next instruction if no relocation is done because the call's operand is zero. After relocation the call instruction will jump to the right location.

Chapter 6

Implementation

In this chapter the implementation of a framework for cross-platform native code plugins is described. First the selection of a programming language and initially supported platforms and object formats are discussed. After that the structure, general workings of the framework and the different calling conventions involved will be explained. At the end of this chapter the limitations of the framework will be discussed.

6.1 Programming Language

If we would like to achieve maximal performance of plugins we have to choose a language that compiles to machine code so that it can make maximal use of the hardware. Since C and C++ are one of the most popular languages that compile directly to machine code and the fact that most OS API's are written in C or C++, the decision was made to write the plugins in C and the framework in C++. Another reason why C is preferred is because its compilers are well-known for optimizing code. The reason why we decided to write the plugins in C instead of C++ is that object files written in C are generally more compatible then C++ object files, see also Section 6.8.1.

6.2 Support of Formats and Operating Systems

Since time was limited we could not implement support for all major object file formats. Since ELF is simpler than the PE-COFF format from Windows and Apple's Mach-O format, we chose to support ELF to explore the possibilities of loading ELF-based libraries on other platforms. The framework has been implemented on Linux, Windows and Mac OS X allowing ELF binaries to run on those operating systems. Within these boundaries we specifically chose the ELF format for 64-bit object files to explore the possibilities with the latest architectures. The framework however can be easily extended to more formats and platforms.

6.3 Support of Position Independent Code

Position Independent Code has the ability to share code with other processes using System Calls. Since plugins are almost never shared between multiple applications because of its application-specific nature, there is no reason to prefer PIC over PDC in the context of this work. PDC is also faster than PIC because it does not contain any indirection when accessing data or functions and it also does not occupy a valuable register to hold the GOT. Both the GOT and the PLT also give a larger memory footprint to the plugins because the PLT stubs need to be stored in memory as was shown in Chapter 4. Another reason to support only PDC is that PIC is more complex to load because we need to map the code into each process's address space that wants to use it. The Global Offset Tables and Procedure Linkage Tables also have to be created on the fly containing addresses of symbols used inside the library, which is another step that PDC does not need. Lazy Evaluation can be done at plugin-level so there is no need for Lazy Evaluation in used in PIC. Plugins can be loaded and unloaded by the application itself and thus loading it only when necessary (i.e

Lazy Evaluation). This is the reason why the choice was made not to support PIC, even though it can be implemented in future revisions.

6.4 Structure of the Framework

The framework consists of a few C++ source and header files which contain a number of classes. All the classes and their purpose are listed in Table 6.1.

Class	Description
Plugin	The Plugin class is the main interface point of the application and the plugin. For every plugin a new subclass of this class should be created which contains the methods and a small piece of stub code to ensure the method is called with the right Calling Convention.
DynamicLibrary	This is an abstract base class for all Dynamically Loaded Library formats that the framework supports. It also holds all the symbols that are exported by the plugin. The Plugin object holds a reference to this class to abstract away the specific format that is used for the plugin.
ELF::ELF	This is an abstract base class that serves as a base class for all ELF implementations. This can be used to distinguish between the 32 and 64-bit implementation of the ELF file format.
ELF::ELF64	This is a concrete class that contains methods to parse and load an 64-bit formatted ELF file.

Table 6.1: The classes that are exposed to users of the framework

The lifecycle of a plugin can be seen in Figure 6.1. Notice that the plugin is loaded when the Plugin.Load() method is called, to allow keeping track of multiple plugins without having to load them directly. When the Plugin.Load() method is called the contents on disk are first parsed and the necessary data and code is copied into memory. After that the necessary relocations will be done. If the plugin is (temporary) not needed it can be unloaded using the Plugin.Unload() method to free memory. After the ELF sections are parsed, the sections with the PROG_BITS flag are loaded into memory. Sections with PROG_BITS as type are sections that need to be loaded in memory and form the program together.

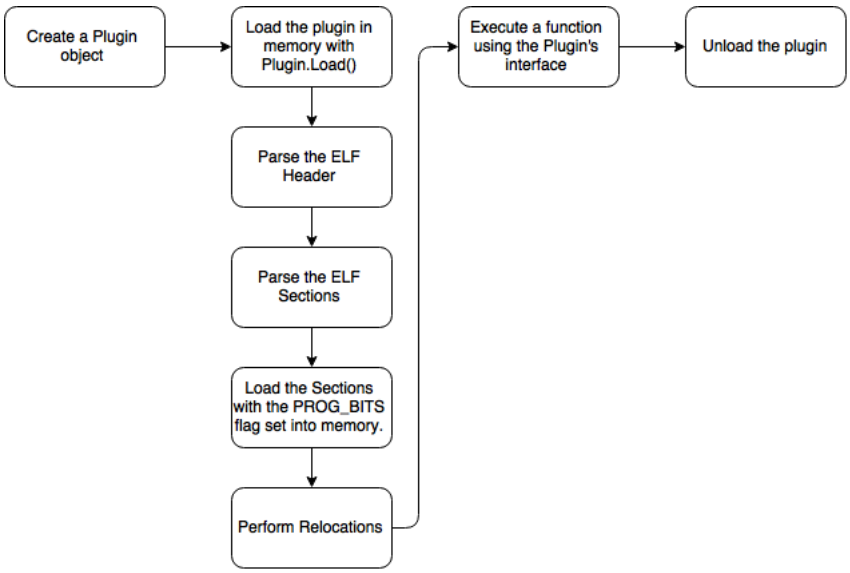

Figure 6.1: A Plugin's Lifecycle

6.4.1 Example of using the framework

Let us assume we want to load a plugin that exports the following code containing a function that calculates the n'th Fibonacci number.

```
int Fibonacci(int n)
{
   if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return (Fibonacci(n-1) + Fibonacci(n-2));
}
```

Then we would need to write a small piece of code that would serve as an interface between the plugin and the application. The interface would look like the following:

```
#include "Plugin/Plugin.h"

class ExamplePlugin : public Plugin
{
public:
        ExamplePlugin(char* filepath) : Plugin(filepath)
        {}

        int Fibonacci(int x);
};

int ExamplePlugin::Fibonacci(int x)
{
    Symbol* symbol = this->library->GetSymbol("Fibonacci");

#ifdef WIN32
// Fill the register arguments
long long registers[1];
registers[0] = x;
return System_V_Call(symbol->address, registers, 1);
#endif

    #if defined(LINUX) || defined(MAX_OSX)
    typedef int (*Fibonacci)(int);
    return ((Fibonacci) symbol->address)(x);
    #endif
}
```

It is possible to abstract away this boiler plate code in the future. The code can either be abstracted away by using C++ templates or macro's. Though when using macro's it would be necessary to create a macro for each number of parameters. The initialization of the plugin is done by the following piece of code.

```
// Create a ExamplePlugin object and load the plugin from disk.
ExamplePlugin plugin("Fibonacci.o");
plugin.Load();

// Execute the Fibonacci function with value 30.
std::cout << plugin->Fibonacci(30) << std::endl;
```

6.5 Parsing and Loading the ELF sections

Before we can execute the plugin we must first load its contents in memory correctly. The first step of parsing the ELF files contents is to parse the ELF Header. The fields that are of particular interest are the location of the Section Header Table, Program Header Table and the String Table. After determining their location we load the Section Header Table and parse each of the Section Headers. Note that we need the String Table to identify the different Sections by their name. The next step is to parse the sections with important data. We first load the .strtab inside memory because it contains the names of the Symbols inside the executable, afterwards the .symtab section is parsed which contains the actual Symbol data. It is important to store the

final addresses of these symbols so the application can access them when it needs to make a function call. After parsing the data we look up all the sections with type PROG_BITS and load them into memory. It is very important to load each of these sections on an address aligned by the alignment specified in the Section Header (sh_addralign field). The most common sections that are of the PROG_BITS type are the .text, .data, .rodata and the .bss section.

6.6 Performing the Relocation

After loading all necessary sections that make up the final program we might need to perform relocations. First a section starting with .rel or .rela is searched for. If none is found we are done with loading the plugin otherwise we need to parse those sections and do the relocation. Each relocation entry describes a relocation as is described in Chapter 5. We implemented the R_AMD64_PC32 relocation type which relocates references that are relative to the next instruction. Because Plugins tend to be small this is the most often used relocation on the x64 architecture. The framework however can be easily extended with relocations of different types to allow for broader support. The R_AMD64_PC32 relocation is computed using the formula $S + A - P$ where S is the value of the symbol which is referred to by the relocation entry, A is the addend field inside the .rela section and P is the address of the storage unit that is being relocated. When calculating $S - P$ we are calculating the offset from the address to the referred location. Because the relative jumps are calculated with an offset from the next instruction on x64 architectures, we need to add A to make the address relative to the next instruction instead of the data field that stores the address. This addend value is typically four because the address itself is contained in four bytes but might be more if padding using Nop instructions is done. This is why this value is stored inside the addend field instead of being a literal inside the formula. After all the relocations are performed the plugin is ready to be called by the application.

6.7 Calling Conventions

Another important aspect of making dynamic libraries run on multiple operating systems is that different calling conventions are in use. A calling convention is an agreement of the calling function and the called function on which registers to use to pass parameters. Often this also includes the information on which registers should be preserved across calls and which registers are not. An example: The callee might expect the first parameter to be passed in register RDI on x64 architectures to function correctly. If it is passed to another register the program will likely crash or produce unpredictable results.

For operating systems running on the x86 architecture multiple calling conventions exists. Some examples are: cdecl, fastcall, stdcall [10]. Because the 32-bit x86 architecture has been around since 1985, most popular compilers (for example GCC and MSVC) took the effort to add support for creating functions with different calling conventions over the years. Unfortunately this is not the case for the x64 architecture. At this time of writing only two well-known calling conventions are in use for the x64 architecture. One of them is the calling convention specified in the System V AMD64 ABI [7] which is widely adopted by most popular x64 platforms such as Linux and OS X. Microsoft on the other hand has implemented its own calling convention for Windows-based programs [11]. Unfortunately no compilers have built-in support to specify which calling convention to use when calling a function, but they might be included in further releases.

Table 6.2 shows the purpose of every register according to both the System V ABI and the Microsoft ABI on the x64 architecture.

Register	System V ABI	Preserved	Microsoft ABI	Preserved
Rax	Return Register	No	Return Register	No
Rbx	Preserved Register (Optionally a Base Pointer)	Yes	Preserved Register	Yes
Rcx	Fourth 64 bit argument	No	First 64 bit argument	No
Rdx	Third 64 bit argument	No	Second 64 bit argument	No
Rsp	Stack Pointer	Yes	Stack Pointer	Yes
Rbp	Preserved Register (Optionally a Frame Pointer)	Yes	Preserved Register (Optionally a Frame Pointer)	Yes
Rsi	Second Register Argument	No	Preserved Register	Yes
Rdi	First Register Argument	No	Preserved Register	Yes
R8	Fifth Register Argument	No	Third 64 bit argument	No
R9	Sixth Register Argument	No	Fourth 64 bit argument	No
R10	Temporary Register for Static Chain Pointer	No	Preserved Register	Yes
R11	Temporary Register	No	Preserved Register	Yes
R12 R15	Preserved Registers	Yes	Preserved Registers	Yes
Xmm0 Xmm1	Return Float Register	No	128 bit Registers	No
Xmm2 Xmm7	Floating Point Arguments	No	128 bit Registers	Partially
Xmm8 Xmm15	Temporary Registers	No	128 bit Registers	Partially
St0, St1	Return Long / Double Arguments	No	Floating Point Registers	No
St2, St7	Temporary Arguments	No	Floating Point Registers	No

Table 6.2: Register usage according the System V ABI and the Microsoft ABI.

Note that the only registers that both ABIs agree on are the return register, the stack pointer, the frame pointer and that R12 till R15 should be preserved across calls. This means that when calling a function that is compiled with a different ABI, we might have to push some registers on the stack because we cannot guarantee that they will be preserved across calls.

An example: We would like to to call a function that is compiled according to the System V ABI while the caller function is compiled according to the Microsoft x64 ABI. Because RSI, RDI, R10, R11 are not preserved in the System V ABI, we need to save those registers before we make the call. When the call returns we can pop the values off the stack again and continue with the program.

In our framework we needed to call a function with a System V calling convention from a Windows application that uses Microsoft's calling convention. Because at the time of writing compilers do not yet have support for calling functions with different calling conventions it is necessary to provide a small piece of assembly code that can safely call a function with a System V calling convention.

```
.code

;-----------------------------------------------------------------------
; int System_V_Call(void* address, int* registers, int registers_used)
;
; Rcx   Address         This is the address that we need to call.
;-----------------------------------------------------------------------
System_V_Call proc
```

```
        ;-------------------------------------------------------------------
        ; Save the Registers that need to be Preserved on the Stack
        ;-------------------------------------------------------------------
        push rbx
        push rsp
        push rbp
        push rsi
        push rdi
        push r12
        push r13
        push r14
        push r15

        ; Save the parameters in r10, r11, and r12
        mov r12, rcx        ; Address of the function
        mov r11, rdx        ; Pointer to Register arguments
        mov r10, r8         ; Amount of Registers

        ; Jump if 4 Register arguments
        cmp r10, 4
        je R4

        ; Jump if 3 Register arguments
        cmp r10, 3
        je R3

        ; Jump if 2 Register arguments
        cmp r10, 2
        je R2

        ; Jump if 1 Register arguments
        cmp r10, 1
        je R1

        ; Jump if no register arguments
        jmp L1

R4:
        mov rcx, [r11 + 24]
R3:
        mov rdx, [r11 + 16]
R2:
        mov rsi, [r11 + 8]
R1:
        mov rdi, [r11]

L1:

        ;-------------------------------------------------------------------
        ; Save the Registers that need to be Preserved on the Stack
        ;-------------------------------------------------------------------

        ; Call the function
        call r12

        ; Pop the Preserved Registers
        pop r15
        pop r14
        pop r13
        pop r12
        pop rdi
        pop rsi
        pop rbp
        pop rsp
        pop rbx

        ret
System_V_Call endp

end
```

This piece of assembly code has two goals, the first important task that it needs to fullfill is to make sure that the input for the functions is in the correct registers. This means that data inside array structure of which

the address is stored in r11, needs to be transferred to rdi, rsi, rdx, rcx. Note that if there are less then 4 parameters, only the first registers are filled with the necessary data. The second task that the code performs is saving the contents of the registers that need to be preserved according to the Microsoft calling convention on the stack. This way the Plugin can use all the registers that it needs without having to worry of overwriting a value that was stored inside a preserved register.

6.8 Limitations of the framework

The limitations of the framework can be roughly divided into two categories which are the limitations imposed by the plugin language and the difficulties of using external functions. The following sections will go in detail about each of these categories.

6.8.1 Limitations when using C++/C

The main issue when trying to communicate with plugins from the application is that many implementation details are undefined, meaning that developers of compilers are free to make the decision on how to implement the particular feature. C and C++ do not provide an Application Binary Interface (ABI) which define how certain features should be implemented such that output produced by compilers is compatible. The following sections describe important obstacles that limit us in using all of C++'s features when developing plugins.

Virtual Functions

As most object-oriented languages do, C++ also provides Virtual Functions which allow methods to be overridden within an inheriting class by a function with the same signature. Virtual Functions require every object in the underlying object hierarchy to contain a Virtual Function Table Pointer (VFTP). A VFTP is a pointer to a Virtual Function Table which contains the addresses of the functions that cannot be resolved at compile-time. The location of the pointer within the object layout is undefined and compilers are free to choose a location which is generally either at the start of the object followed by the objects contents, or appended at the end of the object. Because our framework allows us to compile plugins with different compilers we cannot know in advance where the VFTP is located in the object. Even if we did know, we might have to perform some very complex transformations of machine code to make it able to run on the application's environment. Aside the location of the VFTP, the actual format of the Virtual Function Table (VFT) itself might also be different and is not specified by the C++ specification. This can cause incompatibility between compilers that use a different implementations. Virtual Functions can be used internally by the plugin or application, but as soon as the plugin tries to pass an object to the application or the other way around, behavior is undefined. C++ makes heavy use of Virtual Functions and it is one of the core mechanisms to abstract away functionality, therefore this is a significant limitation. Further information about Virtual Functions can be found in the C++ [13].

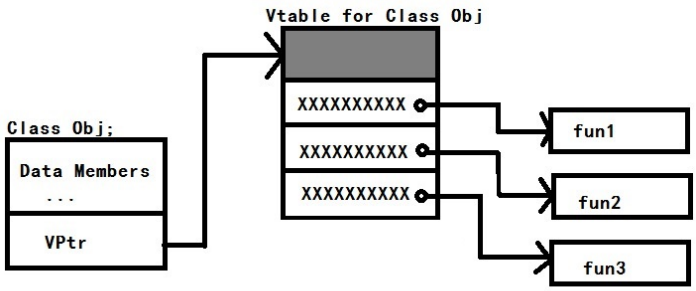

Figure 6.2: Virtual Function Table [12]

Alignment, Padding and Ordering

Aside Virtual Functions, the layout of an object is also undefined by the C++ specification. Some architectures require an object of x bytes to be aligned on a memory address that is a multiple of x but not all architectures

require this. Compilers are free to choose to align objects to a memory boundary and we cannot know how much alignment is applied to an object. Lets assume a structure looks like this:

```
struct Example
{
        char x;
        int y;
        long long z;
}
```

As you can see in Table 6.3 and Table 6.4 multiple layouts in memory could be used inside a program, making two object files with different memory layouts incompatible.

Offset from object's address	Size	Attribute
0	1	x
1	3	unused
4	4	y
9	8	z

Table 6.3: The output of a compiler that applies padding

Offset from object's address	Size	Attribute
0	1	x
1	4	y
5	8	z

Table 6.4: The output of a compiler that does not pad or align the attributes of the object

All these data layouts can be created by different compilers and we cannot know at run time which layout a compiler has produced. In practice most compilers produce the output that is shown in Table 6.3, but this is not guaranteed. As soon as a plugin uses a different layout for an object and tries to pass or receive an object from the application, the application will crash or produces unexpected results.

Exceptions and Debugging Information

The implementation model of Exceptions and Debugging Features is also undefined by the C++ specification. Some implementations store Exception information inside functions themselves while others try to move them out of the stack and place it on the heap. Debugging information depends heavily on the debugger used to debug the program.

Name Mangling

When C++ is compiled into object files, symbols are created to refer to functions and data so that they can be linked with other object files. The process of generating the names for each symbol is called Name Mangling. The names of symbols that are given to functions by a compiler might be different for each compiler implementation. Table 6.5 shows some of the most used Name Mangling Schemes.

Function: void h(int, char)	Symbol name
Microsoft Visual C++	?h@@YAXHD@Z
Borland C++	@h$qizc
Intel C++	_Z1hic
GCC v3 and v4	_Z1hic

Table 6.5: Name Mangling differences between compiler implementations

Because the generation of symbol names is different for most compilers, the framework would have to guess at run time which mangling scheme was used to compile the object file. It is possible to try and search for all popular name manglings of a function signature to find the function we need, but it is not optimal as there is a chance that a function gets associated with the wrong name. Another approach is to export symbols by the C naming conventions, which are more likely to be the same but also have a slight possibility to be different among implementations. (Examples of this are the Microsoft Visual C++ compiler which alters C symbol names when using _cdecl, _stdcall or _fastcall). Conversion of C++ names to C names can only be done with plain functions and not by member functions of classes because C has no support for classes.

Size of primitive types

The size of C++'s primitive types, such as char, int and float are also implementation defined. The C++ specification does mention a minimal size that each primitive type should at least have but no ABI exists that specifies the size of each primitive for each architecture. In practice virtually all compilers on the x86 and x64 architectures support the same sizes for each primitive, so even if its not defined in an ABI, we can assume that the sizes of the primitives in C and C++ are always the same.

Conclusion

Since so many implementation details of C++ are not defined by any specification the only things we can use to ensure portable code are the primitive types and plain C-style functions. This means that if we were programming in C++ we would basically be using a C-subset of the language. The lack of an Application Binary Interface, which specifies how features should be implemented in memory, gives compilers the ability to optimize their own code using an implementation they think is suitable, but it severely limits interchangeability between outputs produced by different compilers.

6.8.2 Calling external functions

Most code uses functions of other libraries to provide useful functionality. Since most operating systems use different calling conventions and use different system calls, it is impossible for plugins in our framework to call functions of any external library without having to perform very complex transformations and linkage in code. Every external function call would have to be linked to an equivalent version of the function on the operating system the plugin is executed on and every call would have to be transformed into the right calling convention. For instance Wine does exactly this by implementing the Windows API and providing equivalent functions on Unix-like systems. Because of these limitations it is better to limit the plugin to not call any external functions at all but rather let the application provide an interface of functions to the plugin. A good approach would be to initialize the plugin by calling a routine and pass it a set of function pointers that would provide it with memory allocation, file system access and other forms of output. This way an application can also give plugins access to different or limited functionality, where for example a plugin can only write to a certain directory or limits the amount of memory a plugin can use.

6.9 JVM vs Framework

A benefit of using our framework is that the code base and memory footprints are is very small (between 1000 and 3000 lines of code) which makes the chance of bugs lower compared to a JVM with a huge Trusted Code Base. Every new supported format increases the Trusted Code Base linearly, while each different supported operating system only requires a few lines extra.

Chapter 7

Experimental Evaluation

In this thesis, we have described a framework to load binary plugins in a truly cross-platform way. This allows us to run a binary plugin on an operating system without having native ELF support. To demonstrate the merits of this framework over cross-platform bytecode-based systems used on the same architecture, we have performed an experimental evaluation in which the performance of Java and native code plugins, implementing a number of computational kernels, is compared. These computational kernels were selected because of their intensive use of computational and memory resources, and practical use in real life software. The kernels that we used to perform our comparison are Matrix Multiplication, Image Blur, Image Scaling and Mandelbrot. Each of these kernels works on a N by N matrix, which either consists of floating-point values or RGB values. Because each kernel is working on a matrix they are well suited to be scaled to a level where they could be measured accurately.

7.1 Experimental Setup

The experimental configuration can be see in Table 7.1. The test program is started once and executes the kernels 50 times, timing each time when the kernel finishes. The framework runs on Windows and the plugins are compiled with GCC 5.1 on the Ubuntu system.

Linux Version	Ubuntu 14.04 TLS
Windows version	Windows 8.1
Processor	Intel Core i5-4690
RAM	8 Gigabytes
GCC Version	5.1
C++ Timer API	std::chrono
Java Version	Java SE 1.8.45
Java Timer API	System.nanoTime

Table 7.1: Configuration of the System

The GCC compiled kernels use the following optimization options:

```
gcc
-Ofast
-fomit-frame-pointer
-fexpensive-optimizations
-nostartfiles
-nodefaultlibs
```

```
-nostdlib
-s
-std=c99
-fno-exceptions
-fno-ident -Xlinker
--hash-style=sysv
-Xlinker
--build-id=none
-fno-asynchronous-unwind-tables
-fbranch-target-load-optimize2
-fmodulo-sched-allow-regmoves
-fgcse-sm
-fgcse-las
-fgcse-after-reload
-funsafe-loop-optimizations
-fsched-pressure
-fsched-critical-path-heuristic
-mavx
```

The use of -mavx flag which enables AVX optimizations is justified because the Java JIT compiler is allowed to apply SIMD optimizations as well. The C and Java code that implements the kernels can be found in the Appendix A.

Java code has been run inside a small test application which repeats each kernel 50 times just like the C test application. Since the JIT compiler at run-time will almost surely make better decisions on the spot, Java does not provide any compile time optimizations and rather delays this till run-time.

7.2 Results

7.2.1 Matrix Multiplication

The Matrix Multiplication computation takes as input three matrixes which are all of the size $n \times n$. The first two matrixes are then multiplied together and the result is stored inside the third matrix. The kernel is executed with 3 matrixes $n = 1250$. For each kernel the minimum, maximum and average time is recorded over 50 runs of the kernel. These metrics were chosen to provide insight of the predictability of execution time. As can be seen in Table 7.2 the C code is slightly faster then Java. When looking at the individual timings for the Java program we noticed that the first runs are lot slower because the run-time Hotspot Optimizer still needs to trigger to optimize the code to perform better. This is why the maximum time of the Java program is much higher then the average running time.

Matrix Multiplication	Minimum Duration	Average Duration	Maximum Duration
Java SE 1.8.45	22.196 seconds	22.278 seconds	24.315 seconds
GCC 5.1	18.074 seconds	18.105 seconds	18.539 seconds

Table 7.2: Matrix Multiplication Results

7.2.2 Image Blur

The Image Blur computation sets each pixel of the image to the average of its surrounding pixels. The inputs for this algorithm are two byte arrays that represent the RGB pixels of the image and the radius that represents the size of the area around the pixel that will be looked at. Each channel occupies one byte and each pixel therefore uses 3 bytes. For this test the image was 3000 by 3000 pixels and the radius was set to 10.

Figure 7.1: Polar Bear - Original

Figure 7.2: Polar Bear - Blurred

Image Blur	Minimum Duration	Average Duration	Maximum Duration
Java SE 1.8.45	12.011 seconds	12.173 seconds	15.200 seconds
GCC 5.1	3.073 seconds	3.165 seconds	3.462 seconds

Table 7.3: Image Blur Results

From the results of Table 7.3 it can be seen that the Image Blur algorithm is on average roughly 4 times faster then the Java equivalent.

7.2.3 Image Scaling

The Image Scaling computation simply scales the image by a factor n, which can make the image smaller ($n < 1$) or larger ($n > 1$). The Image Scaling algorithm that has been implemented finds the closest representing pixel in the original picture for every pixel in the new image and sets the pixel to that value. For the comparison between Java and C an image of size 2500 by 2500 was used and a scale factor of 2.0.

Image Scaling	Minimum Duration	Average Duration	Maximum Duration
Java SE 1.8.45	1.882 seconds	1.907 seconds	2.181 seconds
GCC 5.1	0.702 seconds	0.697 seconds	0.718 seconds

Table 7.4: Image Scaling Results

The results in Table 7.4 show that C code is roughly $2,5$ times as fast as the Java version.

7.2.4 Mandelbrot

The Mandelbrot computation [14] generates the famous Mandelbrot image which can be seen below. For this experiment we have obtained a C version of the code from [15]. In this code parameters can be configured. For the experiment the following parameters have been used: A depth of 256, iteration size of 256 and an image of size 10000x10000 to generate the image. The limits we used were -2.5, 1.5 for the x-axis and -1.5, 1.5 for the y-axis.

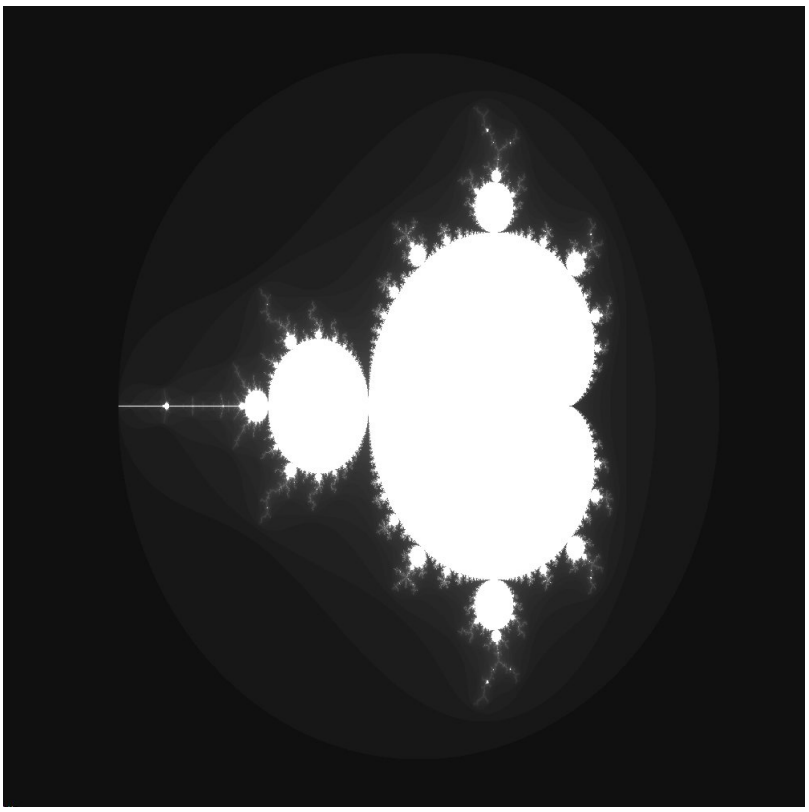

Figure 7.3: Mandelbrot output

Mandelbrot	Minimum Duration	Average Duration	Maximum Duration
Java SE 1.8.45	14.118 seconds	14.180 seconds	14.493 seconds
GCC C code	12.832 seconds	12.856 seconds	13.051 seconds
Optimized AVX	1.825 seconds	1.851 seconds	1.861 seconds

Table 7.5: Mandelbrot Results

Aside the normal C variant of Mandelbrot we also compared an hand-optimized version of the Mandelbrot algorithm using AVX instructions that operates on more data simultaneously, allowing for greater performance in plugins. Notice that this optimized version is roughly 8 times faster then the Java equivalent. This is one of the most compelling reasons to use native code rather than Java. In Java it is not possible to create hand-optimized code with AVX instructions. The AVX code can be found in Appendix A.

7.3 Conclusion

We can conclude from these experiments that performance and predictability of execution time can be good reasons to use native plugins instead of Java code that can be dynamically loaded. Native code is faster in general and when using optimized assembly code native code can even be multiple times faster. Aside the average performance of Java and Native code, we also see that the Hotspot Optimizer that Java uses tends to execute after a couple of runs making the first runs of an algorithm a lot slower. For systems that need to rely on a stable execution time this is another good reason to use native plugins, which have a more predictable execution time. The experiment also showed that it is possible to use another compiler to produce the plugins, while compiling the framework with a different compiler.

Chapter 8

Conclusions

In this thesis we have described a solution to the inability of using native plugins on different platforms. This solution consists of a framework that can run plugins of a specific object file format on an operating system that does not have built-in support for this object file format. As such, it becomes possible to compile a siongle native-code plugin binary for an architecture which can then be used on different operating systems. A performance comparison between native machine code and Java shows that native code can be up to roughly 6 times as fast as code optimized by a JIT compiler. So, for applications where speed is essential it is viable to distribute cross-platform plugins without having to use a JIT compiler.

There are however a number of limitations which disallow us to use all the features C and C++. That is why it is not so practical to develop cross-platform plugins in C or C++. Almost all of these limitations follow from the fact that C and C++ do not specify any implementation details of structures in memory. Newer languages that compile to native code such as D [16] have already learned from this mistake and have created an ABI for the x86 architecture which guarantees portability between object files between compilers. Only compiler support for different calling conventions would then be necessary to enable smooth interaction between plugins and the application without having to write assembly. In the future a new universal format for libraries could be created which can be linked dynamically by the applications on each platform according to a specification ABI. More exploration can also be done in the creating of libraries holding intermediate platform-independent code such as what Google's PNaCL tries to achieve.

Appendices

Appendix A

Plugin Code

The source code listed below is used to measure the performance difference between Java and C++.

A.1 C Code

A.1.1 Matrix Multiplication

```
void Matrix_Multiplication(int n, int A[][1250], float B[][1250], float R[][1250])
{
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
    {
      R[i][j] = A[i][0] * B[0][j];
      for(int k = 0; k < n; k++)
        R[i][j] += A[j][k] * B[k][j];
    }
}
```

A.1.2 Image Blur

```
struct BMPInfoHeader
{
  const unsigned int size;              // The Size of this Header.
  unsigned int width;                   // Width of the image.
  unsigned int height;                  // Height of the image.
  unsigned short colorplanes;           // Amount of colorplanes
  unsigned short bitsperpixel;          // Amount of bytes per pixel.
  unsigned int compressionMethod;       // Compression method. 0 - None
  unsigned int imagesize;               // Imagesize of non-compressed bitmap, 0 for uncompressed.
  unsigned int horizontalResolution;    // Horizontal Resolution in pixel per meter [signed int].
  unsigned int verticalResolution;      // Vertical Resolution in pixel per meter [signed int].
  unsigned int numberofcolors;          // Number of colors in the color palette. Default 0.
  unsigned int numberofimportantcolors; // Number of important colors used, Default 0;
};

typedef struct BMPInfoHeader BMPInfoHeader;

void Image_Blur(BMPInfoHeader* header, int radius, unsigned char* input, unsigned char* output)
{
  for (unsigned int x = 0; x < header->width; x++)
    for (unsigned int y = 0; y < header->height; y++)
    {
      int avgB = 0;
      int avgR = 0;
      int n = 0;
      for(int i = -radius; i <= radius; i++)
        for(int j = -radius; j <= radius; j++)
        {
          if(i + x < 0 || j + y < 0 || i + x >= header->width || j + y >= header->height)
```

```
            continue;

          avgB += input[(x + i)*3 + (y+j)*header->width*3 + 0];
          avgG += input[(x + i)*3 + (y+j)*header->width*3 + 1];
          avgR += input[(x + i)*3 + (y+j)*header->width*3 + 2];
          n += 1;
        }

      output[(y*header->width*3) + x*3 + 0] = avgB / n;
      output[(y*header->width*3) + x*3 + 1] = avgG / n;
      output[(y*header->width*3) + x*3 + 2] = avgR / n;
    }
}
```

A.1.3 Image Scaling

```
void Image_Scaling(BMPInfoHeader* header, float* scale, unsigned char* input, unsigned char* output)
{
  unsigned int newWidth = header->width * *scale;
  unsigned int newHeight = header->height * *scale;

  for (int x = 0; x < newWidth; x++)
    for (int y = 0; y < newHeight; y++)
    {
      int Oldx = (int)((float)x / *scale);
      int Oldy = (int)((float)y / *scale);
      int B = input[(Oldy*header->width * 24 / 8) + Oldx * 24 / 8 + 0];
      int G = input[(Oldy*header->width * 24 / 8) + Oldx * 24 / 8 + 1];
      int R = input[(Oldy*header->width * 24 / 8) + Oldx * 24 / 8 + 2];
      output[(y*newWidth * 24 / 8) + x * 24 / 8 + 0] = B;
      output[(y*newWidth * 24 / 8) + x * 24 / 8 + 1] = G;
      output[(y*newWidth * 24 / 8) + x * 24 / 8 + 2] = R;
    }
}
```

A.1.4 Mandelbrot

This code originates from [15].

```
struct spec
{
  /* Image Specification */
  int width;
  int height;
  int depth;

  /* Fractal Specification */
  float xlim[2];
  float ylim[2];
  int iterations;
};

void mandel_basic(unsigned char *image, const struct spec *s)
{
  float xdiff = s->xlim[1] - s->xlim[0];
  float ydiff = s->ylim[1] - s->ylim[0];
  float iter_scale = 1.0f / s->iterations;
  float depth_scale = s->depth - 1;

  for (int y = 0; y < s->height; y++)
  {
    for (int x = 0; x < s->width; x++)
    {
      float cr = x * xdiff / s->width + s->xlim[0];
      float ci = y * ydiff / s->height + s->ylim[0];
      float zr = cr;
      float zi = ci;
      int k = 0;
      float mk = 0.0f;
      while (++k < s->iterations) {
        float zr1 = zr * zr - zi * zi + cr;
        float zi1 = zr * zi + zr * zi + ci;
```

```
        zr = zr1;
        zi = zi1;
        mk += 1.0f;
        if (zr * zr + zi * zi >= 4.0f)
          break;
      }
      mk *= iter_scale;
      mk = sqrtf(mk);
      mk *= depth_scale;
      int pixel = mk;
      image[y * s->width * 3 + x * 3 + 0] = pixel;
      image[y * s->width * 3 + x * 3 + 1] = pixel;
      image[y * s->width * 3 + x * 3 + 2] = pixel;
    }
  }
}
```

A.1.5 Mandelbrot AVX

This code originates from [15].

```
void mandel_avx(unsigned char *image, const struct spec *s)
{
  __m256 xmin = _mm256_set1_ps(s->xlim[0]);
  __m256 ymin = _mm256_set1_ps(s->ylim[0]);
  __m256 xscale = _mm256_set1_ps((s->xlim[1] - s->xlim[0]) / s->width);
  __m256 yscale = _mm256_set1_ps((s->ylim[1] - s->ylim[0]) / s->height);
  __m256 threshold = _mm256_set1_ps(4);
  __m256 one = _mm256_set1_ps(1);
  __m256 iter_scale = _mm256_set1_ps(1.0f / s->iterations);
  __m256 depth_scale = _mm256_set1_ps(s->depth - 1);

  for (int y = 0; y < s->height; y++){
    for (int x = 0; x < s->width; x += 8) {
        __m256 mx = _mm256_set_ps(x + 7, x + 6, x + 5, x + 4,
                                  x + 3, x + 2, x + 1, x + 0);
        __m256 my = _mm256_set1_ps(y);
        __m256 cr = _mm256_add_ps(_mm256_mul_ps(mx, xscale), xmin);
        __m256 ci = _mm256_add_ps(_mm256_mul_ps(my, yscale), ymin);
        __m256 zr = cr;
        __m256 zi = ci;
        int k = 1;
        __m256 mk = _mm256_set1_ps(k);
        while (++k < s->iterations)
        {
          __m256 zr2 = _mm256_mul_ps(zr, zr);
          __m256 zi2 = _mm256_mul_ps(zi, zi);
          __m256 zrzi = _mm256_mul_ps(zr, zi);

          /* zr1 = zr0 * zr0 - zi0 * zi0 + cr */
          /* zi1 = zr0 * zi0 + zr0 * zi0 + ci */
          zr = _mm256_add_ps(_mm256_sub_ps(zr2, zi2), cr);
          zi = _mm256_add_ps(_mm256_add_ps(zrzi, zrzi), ci);

          /* Increment k */
          zr2 = _mm256_mul_ps(zr, zr);
          zi2 = _mm256_mul_ps(zi, zi);
          __m256 mag2 = _mm256_add_ps(zr2, zi2);
          __m256 mask = _mm256_cmp_ps(mag2, threshold, _CMP_LT_OS);
          mk = _mm256_add_ps(_mm256_and_ps(mask, one), mk);

          /* Early bailout? */
          if (_mm256_testz_ps(mask, _mm256_set1_ps(-1)))
              break;
        }
        mk = _mm256_mul_ps(mk, iter_scale);
        mk = _mm256_sqrt_ps(mk);
        mk = _mm256_mul_ps(mk, depth_scale);
        __m256i pixels = _mm256_cvtps_epi32(mk);
        unsigned char *dst = image + y * s->width * 3 + x * 3;
        unsigned char *src = (unsigned char *)&pixels;
        for (int i = 0; i < 8; i++) {
          dst[i * 3 + 0] = src[i * 4];
```

```
            dst[i * 3 + 1] = src[i * 4];
            dst[i * 3 + 2] = src[i * 4];
          }
    }
  }
}
```

A.2 Java

The code below is based of the C equivalent. Mandelbrot in Java is therefore indirectly based on [15].

A.2.1 Matrix Multiplication

```
public static void MatrixMultiplication(int n, float[][] A, float[][] B, float[][] R)
{
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
    {
      R[i][j] = A[i][0] * B[0][j];
      for(int k = 0; k < n; k++)
        R[i][j] += A[j][k] * B[k][j];
    }
}
```

A.2.2 Image Blur

```
public static void Image_Blur(int width, int height, int radius, char[] input, char[] output)
{
  for (int x = 0; x < width; x++)
    for (int y = 0; y < height; y++)
    {
      int avgB = 0;
      int avgG = 0;
      int avgR = 0;
      int n = 0;
      for(int i = -radius; i <= radius; i++)
        for(int j = -radius; j <= radius; j++)
        {
          if(i + x < 0 || j + y < 0 || i + x >= width || j + y >= height)
            continue;

              avgB += input[(x + i)*3 + (y+j) * width*3 + 0];
              avgG += input[(x + i)*3 + (y+j) * width*3 + 1];
              avgR += input[(x + i)*3 + (y+j) * width*3 + 2];
              n += 1;
        }

          output[(y * width*3) + x*3 + 0] = (char) (avgB / n);
          output[(y * width*3) + x*3 + 1] = (char) (avgG / n);
          output[(y * width*3) + x*3 + 2] = (char) (avgR / n);
    }
}
```

A.2.3 Image Scaling

```
public static void Image_Scaling(int width, int height, float scale, char[] input, char[] output)
{
  int newWidth = (int) (width * scale);
  int newHeight = (int) (height * scale);

  for (int x = 0; x < newWidth; x++)
    for (int y = 0; y < newHeight; y++)
    {
      int Oldx = (int)((float)x / scale);
      int Oldy = (int)((float)y / scale);
      int B = input[(Oldy*width * 24 / 8) + Oldx * 24 / 8 + 0];
      int G = input[(Oldy*width * 24 / 8) + Oldx * 24 / 8 + 1];
      int R = input[(Oldy*width * 24 / 8) + Oldx * 24 / 8 + 2];
```

```
      output[(y*newWidth * 24 / 8) + x * 24 / 8 + 0] = (char) B;
      output[(y*newWidth * 24 / 8) + x * 24 / 8 + 1] = (char) G;
      output[(y*newWidth * 24 / 8) + x * 24 / 8 + 2] = (char) R;
    }
}
```

A.2.4 Mandelbrot

```
public static void Mandelbrot(char[] image, Specification s)
{
  float xdiff = s.xlim1 - s.xlimo;
  float ydiff = s.ylim1 - s.ylimo;
  float iter_scale = 1.0f / s.iterations;
  float depth_scale = s.depth - 1;

  for (int y = 0; y < s.height; y++) {
    for (int x = 0; x < s.width; x++) {
      float cr = x * xdiff / s.width  + s.xlimo;
      float ci = y * ydiff / s.height + s.ylimo;
      float zr = cr;
      float zi = ci;
      int k = 0;
      float mk = 0.0f;
      while (++k < s.iterations) {
        float zr1 = zr * zr - zi * zi + cr;
        float zi1 = zr * zi + zr * zi + ci;
        zr = zr1;
        zi = zi1;
        mk += 1.0f;
        if (zr * zr + zi * zi >= 4.0f)
          break;
      }
      mk *= iter_scale;
      mk = (float) Math.sqrt(mk);
      mk *= depth_scale;
      int pixel = (int) mk;
      image[y * s.width * 3 + x * 3 + 0] = (char) pixel;
      image[y * s.width * 3 + x * 3 + 1] = (char) pixel;
      image[y * s.width * 3 + x * 3 + 2] = (char) pixel;
    }
  }
}
```

Glossary

ABI stands for Application Binary Interface and is a specification that specifies how structures produced by compilers are to be implemented in memory, allowing for more compatibility with other object files.. 22

ASCII stands for American Standard Code for Information Interchange and is a popular character-encoding scheme. Each character is represented by a number, which can be represented by 8 bits. 7

CLR stands for Common Language Run-time and is a Virtual Machine that can run MSIL bytecode which is produced by C#. 1

ELF stands for Executable and Linkage Format and is the object file format that is used on Linux/Unix operating systems.. 5, 19

GOT stands for Global Offset Table and is a table in memory which holds the addresses of symbols at run-time.. 12

JIT stands for Just In Time and describes a model of execution. Code that runs Just In Time is most commonly represented in an intermediate format and is translated to machine instructions at run-time.. i

JVM stands for Java Virtual Machine and is the Virtual Machine that runs Java bytecode.. 1

Mach-O is an object file format that is used exclusively on Apple operating systems.. 1, 19

PDC stands for Position Dependent Code and is a name for code that is dependent on the location that it is loaded and needs relocation unless it's loaded on a predefined address.. 11

PE-COFF stands for Portable Executable - Common Object File Format and is an object file format that is used exclusively on Windows operating systems.. 1, 19

PIC stands for Position Independent Code and is a name for code that can be executed no matter what address it is loaded on.. 11

PLT stands for Procedure Linkage Table and is a table in memory which holds code to call functions indirectly, allowing Position Independent Code to run without being bound to an address.. 12

VFT stands for Virtual Function Table and is a table in memory which is used to implement run-time polymorphism.. 25

VFTP stands for Virtual Function Table Pointer and is the pointer which is appended to a class object that points to the Virtual Function Table. The location of the pointer inside an object is implementation defined.. 25

Bibliography

[1] Kam Low *Pluga*,
[online] 29 May 2014,
http://sourcey.com/building-a-simple-cpp-cross-platform-plugin-system/,
(Accessed: 14 August 2015)

[2] Gigi Sayfan,
A Cross-platform Plugin Framework for C/C++,
[online] 25 November 2007,
http://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/204202899?cid=RSSfeed_DDJ_Cpp,
(Accessed: 14 August 2015)

[3] Giridhar Pemmasani *NDISWrapper*,
[online] 28 November 2013,
http://sourceforge.net/projects/ndiswrapper/ (Accessed: 14 August 2015)

[4] *The Wine Project*,
[online],
https://www.winehq.org/,
(Accessed: 14 August 2015)

[5] Google *Google PNaCL*,
[online],
https://developer.chrome.com/native-client/nacl-and-pnacl,
(Accessed: 14 August 2015)

[6] *Global structure of an ELF file*,
[online],
http://nairobi-embedded.org/004_elf_format.html,
(Accessed: 14 August 2015)

[7] Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell,
Executable and Linkable Format (ELF),
[online] October 7 2013,
Draft Version 0.99.6,
http://www.x86-64.org/documentation/abi.pdf,
(Accessed: 14 August 2015)

[8] John R. Levine,
Linkers And Loaders,
1st Edition, Morgan Kaufmann, 25 October 1999

[9] Eli Bendersky
The process of calling a function in a shared library,
[online] 3 November 2011,
http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/,
(Accessed: 14 August 2015)

[10] Microsoft Developer Network,
x86 Calling Conventions,
[online],
https://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx,
(Accessed: 14 August 2015)

[11] Microsoft Developer Network
Microsoft x64 Calling Convention,
[online],
https://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx,
(Accessed: 14 August 2015)

[12] *Structure of the Virtual Function Table*,
[online],
http://www.programering.com/a/MzMyQDNwATA.html (Accessed: 14 August 2015)

[13] The C++ Standards Committee,
ISO/IEC 14882:2014,
[online] 15 December 2014,
http://www.iso.org/iso/catalogue_detail.htm?csnumber=64029,
(Accessed: 14 August 2015)

[14] Benoit Mandelbrot,
Fractals and Chaos: The Mandelbrot Set and Beyond,
1st Edition, Springer Science & Business Media, 29 Juni 2013

[15] Claudio Leite,
Mandelbrot C and AVX code,
[online] 15 Juli 2015,
https://github.com/skeeto/mandel-simd/ (Accessed: 14 August 2015)

[16] Walter Bright, Andrei Alexandrescu,
The D Language,
[online],
http://dlang.org/,
(Accessed: 14 August 2015)