COMMON LANGUAGE INFRASTRUCTURE FOR RESEARCH (CLIR): EDITING AND OPTIMIZING .NET ASSEMBLIES

A Thesis
by
Shawn H. Windle

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

December 2012
Major Department: Computer Science

COMMON LANGUAGE INFRASTRUCTURE FOR RESEARCH (CLIR): EDITING AND OPTIMIZING .NET ASSEMBLIES

A Thesis
by
Shawn H. Windle
December 2012

APPROVED BY:

Dr. James B. Fenwick Jr.
Chairperson, Thesis Committee

Dr. Cindy Norris
Member, Thesis Committee

Dr. James T. Wilkes
Member, Thesis Committee

Dr. James T. Wilkes
Chairperson, Computer Science

Edelma D. Huntley
Dean, Research and Graduate Studies

ABSTRACT

COMMON LANGUAGE INFRASTRUCTURE FOR RESEARCH (CLIR): EDITING AND OPTIMIZING .NET ASSEMBLIES.

(December 2012)

Shawn H. Windle, Appalachian State University

Thesis Chairperson: Dr. James B. Fenwick Jr.

In 2002, Microsoft released the .NET Framework as it's implementation of the Common Language Infrastructure (CLI). The subsequent release of Mono, an open-source implementation of the CLI, allowed the .NET Framework audience to also include Max OS X, Linux and Unix users. These tools enabled high-level .NET development, but low-level researchers such as code optimizers have few tools available to manipulate .NET assembly files.

This thesis presents the Common Language Infrastructure for Research (CLIR) comprised of three components: the Common Language Engineering Library (CLEL), the Common Language Optimizing Framework (CLOT), and a suite of utility applications. CLIR enables researchers to experiment with and develop tools for the .NET Framework languages. CLEL provides the means to read, edit and write low-level .NET assemblies. CLOT uses CLEL to provide a framework for code optimization including algorithms and data structures for three traditional optimizations. Evaluations of program performance demonstrate meaningful decrease in program execution time due to the application of these optimizations, thus validating CLIR and enabling further .NET optimization research.

ACKNOWLEDGEMENTS

I would like to express my profound appreciation to my thesis chairperson, Dr. James B. Fenwick Jr., for all of the hard work and time he put into helping me, not just with this thesis, but also for all of the classes I took at Appalachian State University with him. I would also like to thank Dr. Cindy Norris, Professor Kenneth Jacker and Dr. James Wilkes for all of their advice and help. I would also like to acknowledge that without the support from my family this thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In conjunction with several other companies, Microsoft developed the Common Language Infrastructure (CLI) [37] in 2000. The CLI detailed a new programming and execution environment. A platform neutral virtual machine, type system and instruction set called the Common Intermediate Language (CIL) are core components of the CLI. The CLI was submitted to the European Computer Manufacturers Association (ECMA) and the International Organization for Standardization (ISO) and accepted in December 2001. In 2002, Microsoft released its implementation of the CLI called the .NET Framework. A new language was also released with .NET called C# [22]. Instead of creating a completely new low-level file format, Microsoft extended its existing *.exe* Portable Executable format and used it as the basis of the .NET assembly file format. A .NET compiler translates program source code files into an assembly containing CIL instructions. At runtime, the virtual machine translates the CIL to native machine-specific assembly instructions. Delaying the generation of the machine-specific assembly instructions to runtime allows the CIL code to be portable to any machine with a CLI-compliant virtual machine. Chapter 2 will give background information on the CLI, the .NET Framework, and the CIL.

Since the .NET Framework comes bundled with the Windows operating system, there is a large audience to use the .NET Framework. Mono, an open source implementation of the .NET Framework, runs on Linux, Mac OS X, and Unix, as well as Windows. With .NET and Mono, any developer can develop and execute .NET programs on any of the major platforms. With the ensuing surge in .NET programming, researchers, particularly code optimizers, need access to the low-level CIL code. Unfortunately, there are very few options available to easily manipulate .NET assemblies.

This thesis presents the Common Language Infrastructure for Research (CLIR) as a layered framework for conducting .NET assembly research. A reusable library called the Common Language Engineering Library (CLEL) is the CLIR component that allows the programmer to read, edit and write .NET assemblies. CLEL can be used to develop many other tools such as decompilers, optimizers, compilers, and virtual machines. To demonstrate the usefulness of CLEL, an optimization framework called the Common Language Optimizing Toolset (CLOT) is presented. Chapter 3 details the development of CLEL, CLOT, and their API's. CLOT uses CLEL to implement three different optimizations. These optimizations are Branch Instruction Replacement, Constant Propagation and Method Inlining. Chapter 4 details the background theory for the optimizations developed for this thesis. Chapter 5 evaluates the results of applying the aforementioned code optimizations to several programs. The optimizations are shown to have meaningful effects on program performance. Thus, CIL optimization is beneficial and the usefulness of the CLIR is validated. Chapter 6 concludes with a summary of the work and suggestions for future directions.

Chapter 2

Background

This chapter presents an overview and history of the Common Language Infrastructure (Section 2.1), an overview of the lower-level assembly file format (Section 2.2), the instruction format (Section 2.3), and summarizes some related work (Section 2.4).

2.1 The Common Language Infrastructure

In August 2000, Microsoft, Hewlett-Packard, Intel, and others began development of the Common Language Infrastructure (CLI). The CLI is a specification that details a broad set of features and rules for developing object-oriented programming languages and execution environments. The CLI was developed to be as programming language neutral as possible in order to support as wide a group of programming languages as possible. The CLI was ratified by the European Computer Manufacturers Association (ECMA) in December 2001 and the International Organization for Standardization (ISO) in April 2003. In 2002, Microsoft released an implementation of the CLI specification called the .NET Framework, which contained support for four programming languages: C#, Visual Basic .NET, JScript and Managed C++ .NET. On June 30, 2004, an open source implementation of the CLI

specification called Mono was released with support for the C# programming language [39]. Later versions of Mono added support for generics and newer versions of the .NET Framework.

The CLI is composed of three sections: the Common Type System (CTS), the Common Language System (CLS) and the Virtual Execution System (VES) [37]. The CTS defines a complete set of built-in types and methods for the user to combine and name new types. Two types are supported: reference types and value types. Reference types are always allocated on the heap and accessed by a reference to the object. Value types, on the other hand, are allocated on the stack and can be accessed directly. The CLI Specification includes all of these types: bool, char, object, string, float32, float64, int8, int16, int32, int64, native int, native unsigned int, typedref, unsigned int8, unsigned int16, unsigned int32, and unsigned int64 [37]. A person developing a higher level programming language would choose a subset of these built-in types to implement in their programming language. The CTS contains no primitives, only types. Therefore, when a programmer uses the "int" keyword like in C#, the compiler maps this to the System.Int32 type, which implements the int32 value type in the CTS. The CTS also allows the language designer to create new built-in types.

Switching to the programmer's perspective, the CTS also provides the standard object oriented features. Inheritance allows the programmer to create a new type, which is called a child type, by extending an existing type, which is called the parent type. Extending the parent type can be accomplished by adding new methods to the child type and overriding existing methods in the parent type. The CTS only supports single inheritance, thus a child type can only have one parent type. Further, all types, except System.Object, either inherit directly or indirectly from the System.Object type.

The process of taking a value type and converting it to a reference type is called boxing [37]. To box a value type, new memory is allocated on the heap, the value type's value is copied to this address and a reference to this new reference type is returned. Unboxing is the process of taking a reference type and converting it to a value type.

Each type in the CTS has one or more methods associated with it, with a default constructor method being required for all CTS types [37]. These methods describe the operations that are allowed on a certain type. Types also can contain zero or more fields. The CTS supports seven mechanisms for the accessibility of types and fields: Compiler-Controlled, Private, Family, Assembly, Family-and-Assembly, Family-or-Assembly, and Public [37]. These accessibility types range from Public, which is visible to all types, to Private, which is only visible to to the types that declare the field or method. Language designers are not required to implement all of the accessibility types and can choose a subset of these to implement.

One of the goals of the CLI was programming language interoperability and the Common Language System (CLS) was developed to achieve this goal. The CLS is a subset of the CTS types used for calls between different CLI-compliant programming languages. All of the built-in types are in the CLS except int8, native unsigned int, typedref, unsigned int16, unsigned int32, and unsigned int64. The CTS defines all possible types, but some programming language designers may decide not to implement certain types. To ensure programming language interoperability between a programming language that is being designed and other CLI-compliant languages, the types in the CLS must be implemented [37].

The Virtual Execution System (VES) defines the low-level assembly file format, metadata, the Common Intermediate Language (CIL) instruction format, and the rules

governing the Virtual Machine (VM) and the Just-In-Time (JIT) compiler [37]. A compiler takes code written in one language and outputs it in another form. Traditionally, this output form is machine code that is specific to a certain architecture. However, the output of a compiler for a CLI-compliant programming language is called an assembly and contains architecture independent code called the Common Intermediate Language (CIL). The CIL is an instruction set for an abstract stack machine [37]. A stack machine has no registers; instead, data are pushed on a stack, operations are performed using the data on the top of the stack, and operation results are pushed onto the stack. Method parameters and return values are also passed by pushing them onto the stack. When an assembly is run, a Virtual Machine (VM) reads in the assembly, parses it and passes it to the JIT compiler to generate machine code. Portability is obtained because architecture-dependent machine code is not generated until the assembly is run. Therefore an assembly can be run on any architecture with an implementation of a CLI-compliant VM and JIT compiler.

Beside the CIL instructions, the assembly also contains metadata. These metadata contain descriptions of the assembly called the manifest, infomation about the environment the assembly was compiled in, a description of the types in the assembly, signatures of all the methods in the assembly, attributes about the instructions and data in the assembly, and security information [37]. This information can be used by compilers, debuggers and virtual machines to see what types and methods an assembly exports for others to use. The assembly format is an extension of the Windows Portable Executable (PE) format [36][37] which is used by Windows executables. Just like Windows executables, assemblies either have an ".exe" or ".dll" extension. An assembly with an ".exe" extension is an application and an assembly with a ".dll" extension is a library.

The .NET Framework contains a group of libraries called the Base Class Library (BCL). The BCL contains types that can be used for string manipulation, networking, drawing graphical user interfaces, and a wide variety of other uses. Prior to the .NET Framework release, Microsoft released two compilers with its Integrated Development Environment Visual Studio (VS): Visual Basic and Visual C++. The C++ compiler read ANSI C++ and produced executables with native machine instructions. When Microsoft released the .NET Framework, Microsoft rewrote the Visual Basic and C++ compilers to produce assemblies. The .NET Framework introduced two new compilers: C# and JScript. The Visual Studio environment also contains a tool called *ildasm* that allows the user to view, but not edit, the contents of an assembly [50].

To be compatible with the .NET Framework, Mono also shipped with its own implementation of the BCL with most of the same types. Similar to *ildasm*, Mono contains a tool called *monodis*. Beta compilers for Visual Basic .NET and JScript and better generics support in the C# compiler were added in Mono 1.2.

2.2 Overview of the Assembly .text section

The Assembly file format is an extension of the Portable Executable (PE) file format [36][37] used by Windows executables. Since a main focus of this thesis is the optimization of the code in assemblies, this section will cover the decomposition and explanation of the fields and sections needed to perform the optimizations detailed in Chapter 4. The example assembly implements the bubble sort [12] algorithm in C# and was compiled into "BubbleSort.exe" using the Mono 1.2.6 C# compiler. See Appendix A for the C# code and Appendix B for the hexadecimal dump for this example. Appendix C covers this example in more detail.

Fields in an assembly are given as either offsets from the beginning of the file or as a relative virtual address (RVA), which is the virtual address of a section plus the offset to the field. For the example in Appendix C, the .text section begins at offset 0x200[1] and RVA 0x2000. Therefore when this assembly is run, the .text section will be loaded at virtual address 0x2000 and fields in the .text section will have an RVA based on this virtual address.

Offset	**RVA**	**Name**	**Value**
2ec	20ec	Type/Size Flags	1330
2ee	20ee	Max Stack	001f
2f0	20f0	Code Size	0000 0106
2f4	20f4	LocalVarSig Token	1100 0001
2f8	20f8	Code	02280a00000102730a0000027d04000001027b040000011f5b8c01 0000036f0a00000326027b040000011b8c010000036f0a00000326 027b040000011f658c010000036f0a00000326027b04000001198c 010000036f0a00000326027b040000011f3a8c010000036f0a0000 0326027b040000011ff28c010000036f0a00000326027b04000001 20000000c78c010000036f0a00000326027b040000011f2c8c0100 00036f0a00000326027b04000001178c010000036f0a0000032602 7b0400000120000002a68c010000036f0a00000326022806000002 160a3800000015027b04000001066f0a000004280a000005161758 0a06027b040000016f0a0000063fffffffda2a

Table 2.1: Method header - public BubbleSort()

The first part in the .text section to mention is the Method Headers, which starts at offset 0x2ec and RVA 0x20ec in the "BubbleSort.exe" assembly. Each method header represents one method in an assembly. Method headers are one of two types, either tiny or fat. If bits zero and one of the first byte of the method header are B10, then it is a tiny method. If the same two bits are instead B11, then it is a fat method. The first method header shown in Table 2.1 is fat because bit zero and one of 0x13 (B0001 00<u>11</u>) are B11. Since this is a fat method, it starts with a flags and size field that is two bytes in length. The "flags and size" field in Table 2.1 is 0x3013 (B0011 0000 0001 0011) because the assembly file is in little endian format. Bits zero to eleven (B0011 <u>0000 0001 0011</u>) are the flags for

[1]Numbers that begin with "0x" are in hexadecimal. Binary numbers start with "B". All numbers in tables are in hexadecimal. Numbers not in a table or beginning with "B" or "0x" are in decimal.

this method and bits twelve through fifteen (B0011 0000 0001 0011) are the size of this method header in 4-byte integers excluding the code field. Therefore this method header has three 4-byte integers for the Type/Size, Max Stack, Code Size, and LocalVarSig Token fields. All Fat methods have the same five fields shown in Table 2.1. The max stack field indicates the maximum size of the stack during the method's execution. The code size field tells how many bytes the upcoming code field is.

The LocalVarSig token is a four byte reference into a table contained in the metadata section that contains type information about the local variables in this method. The most significant byte of this token is the table number and the other three bytes are an index into that table. In Table 2.1, the LocalVarSigToken references row 1 (0x00 0001) in table 0x11, which is the LocalVarSig metadata table.

The final field contains the Common Intermediate Language (CIL) code of this fat method. The CIL instruction format will be covered in Section 2.3. The other method headers can be seen in Appendix C.

A tiny method header only contains the flags, code size and code fields. A tiny method has no local variables, no exceptions, no extra data sections, the stack usage does not exceed eight values and the code size is 64 bytes or less [37].

Offset	**RVA**	**Name**	**Value**
544	2344	Offset	0000 0298
548	2348	Size	0000 0010
54c	234c	Name	2347 5549 4400 0000

Table 2.2: Stream header - #GUID

Five similar headers that describe persistent streams in the assembly follow the method header section. These five streams are #~, #US, #Strings, #Blob, and #GUID. The #~ stream contains information about the metadata tables later in the assembly. The

#US stream contains string literals that were in the original source code. The #Strings stream contains method and field names and other compiler generated strings. The #Blob stream contains encoded types used by the metadata tables later in the assembly. The #GUID stream contains Global Unique Identifiers (GUID) used to uniquely identify assemblies. Table 2.2 shows the #GUID stream. The offset field gives the location of the stream in the metadata (see Table C.19 in Appendix C). The size field contains the size of the stream in bytes. The #GUID stream in Table 2.2 is 16 (0x0000 0010) bytes long. The final field, name, gives a string representation of the name of the stream. For example, the ASCII string "#GUID\0\0\0" is encoded in the bytes 0x2347 5549 4400 00 in Table 2.2.

Offset	**RVA**	**Name**	**Value**
558	2358	Reserved1	0000 0000
55c	235c	Major Version	01
55d	235d	Minor Version	00
55e	235e	Heap Sizes	00
55f	235f	Reserved2	01
560	2360	Valid	0000 0009 0002 0557
568	2368	Sorted	0000 0000 0000 0000
570	2370	Table Rows	0000 0001 0000 0004 0000 0002 0000 0001 0000 0004 0000 0002 0000 0007 0000 0003 0000 0001 0000 0001

Table 2.3: #~ stream

The five streams follow the stream headers. The #~ stream located at RVA 0x2358 is shown in Table 2.3. The Reserved1 and Reserved2 fields are reserved for future use. The #~ stream describes the other metadata tables that follow. The major and minor version fields indicate how the metadata is encoded in this assembly. The Heap Sizes field in the #~ stream is a bit field that contains a flag if a stream has a size that is greater than or equal to 65,536 bytes. If bit 0 is set, then all indexes from any metadata table into the #Strings stream are 4-bytes wide. If that bit is not set, then indexes into the #Strings stream are 2-bytes wide. The #GUID stream (bit 1) and #Blob stream (bit 2) also have a bit in Heap

Sizes reserved for this purpose. Some of the fields in metadata tables discussed later (like the Name field in the TypeDef table in Table 2.5) index into these streams. Since none of the bits are set in the Heap Sizes field, then all stream indices in the metadata tables are 2-bytes wide. The valid field is a bit vector that has a bit set for each metadata table that comes later in the assembly. Since 8 bytes, which is 64 bits, are used in the Valid field to represent what tables are in an assembly, there are 64 such tables possible. This assembly uses 10 of these tables because 10 bits are set in 0x0000 0009 0002 0557. According to the Valid field, this assembly contains the Module Table, TypeRef Table, TypeDef Table, Field Table, MethodDef Table, Param Table, MemberRef Table, StandAloneSig Table, Assembly Table, and the AssemblyRef Table. The Sorted field is a bit vector that contains a bit for each metadata table that is sorted. Since the sorted field is 0x0000 0000 0000 0000, none of the metadata are sorted according to any criteria.

The Table Rows field is a list of 4-byte integers, one for each of the metadata tables used in this assembly. Each 4-byte integer in the Table Rows field gives the number of rows in each metadata table. For example, since bit 6 is set in the valid bit vector, the MethodDef table exists in this assembly. Since the MethodDef table is the fifth table that exists in the Valid field, the fifth 4-byte integer in the table rows field (0x0000 0004) holds the number of rows in the MethodDef table. Note that the first row of a metadata table is indexed as 1 and not 0.

Offset	**RVA**	**ResolutionScope**	**Name**	**Namespace**
5a2	23a2	0006	000a	0011
5a8	23a8	0006	001e	0028
5ae	23ae	0006	003b	0011
5b4	23b4	0006	004e	0011

Table 2.4: TypeRef table

Not all of the metadata tables will be covered in this section. Only those directly related to the optimizations performed in this thesis. See Appendix C for a more detailed explanation of all of the metadata tables in this example.

The metadata table TypeRef (see Table 2.4) starts at RVA 0x23a2. Each of the four rows in the TypeRef Table represents an external type that is referenced in this assembly. The ResolutionScope is an encoded index into either the Module, ModuleRef, AssemblyRef or TypeRef Table that represents the module or assembly this type came from. Take row one for example. Since bits 0 and 1 are B10 in the ResolutionScope 0x0006, this is an index into the AssemblyRef Table [37]. Encoding B00, B01, and B11 represents the tables Module, ModuleRef, and TypeDef respectively. The other 14 bits represent the row number [37]. The ResolutionScope of the first row refers to row 0x0001 of the AssemblyRef Table. The Name field is a byte index into the #Strings stream which contains the name of this type. For example, row one's name references the ASCII string "Object" at offset 0x686 in the #Strings stream (0x67c + 0x000a = 0x686 in Table 2.11). The Namespace field is a byte index into the #Strings stream which contains the string representation of the namespace this type is in. For example, the ASCII string "System" is pointed at by row one's Namespace field. Therefore, row one represents the external type System.Object.

Offset	**RVA**	**Flags**	**Name**	**Namespace**	**Extends**	**FieldList**	**MethodList**
5ba	23ba	0000 0000	007e	0000	0000	0001	0001
5c8	23c8	0010 0001	0073	0000	0005	0001	0001

Table 2.5: TypeDef table

The TypeDef Table (see Table 2.5) at RVA 0x23ba can be seen in Table 2.5. Each row in the TypeDef Table represents a type that is defined in the assembly. The Flags field contains a bitset that represent the visibility of a type. For example, row two has the Public

flag (0x0000 0001), which means this type is globally visible, and the BeforeFieldInit flag (0x0010 0000), which indicates non-static fields must be initialized before static fields can be accessed. The Name and Namespace fields are indices into the #Strings stream, which contains the string representation of this type. The Extends field is an encoded index into either the TypeDef, TypeRef or TypeSpec Table. Bits 0 and 1 are used to identify the table: TypeDef (B00), TypeRef (B01), and TypeSpec (B10). The other 14 bits represent the row number [37]. Therefore, row two's Extends field 0x0005 is encoded as TypeRef Table (B0000 0000 0000 0101) and row 1 (B0000 0000 0000 0101). TypeRef row 1 represents the *System.Object* type. Therefore this type extends (or inherits) from the *System.Object* type. The FieldList field indexes into the Field Table to show which fields are a part of this type. The index into the Field Table is the start of a line of fields that belong to that type. The line ends when either the Field Table ends or the next types fields begin. The MethodList is an index into the MethodDef Table that represents the line of methods that belong to a certain type. In this example, row 2 represents the "BubbleSort" type whose methods start at row 1 of the MethodDef Table and whose fields start at row 1 of the Field Table.

Offset	**RVA**	**Flags**	**Name**	**Signature**
5d6	23d6	0001	0096	0027

Table 2.6: Field table

Each row in the Field Table (Table 2.6) represents a field that belongs to a type. The Flags field is a bitset that represents the access permissions of the field. For example, the Flag field in row one has the private flag (0x0001) set. The Name field is an index into the #Strings stream, which contains the string representation of the name of this field. In this case, the ASCII string "nums" is found by using the offset in row one's Name field to index into the #Strings stream. The Signature field is a index into the #Blob stream that

represents the type of this field. Thus, this encodes the private field called “nums” seen in the source code in Appendix A.

Offset	RVA	RVA	ImplFlags	Flags	Name	Signature	ParamList
5dc	23dc	0000 20ec	0000	1886	0018	0001	0001
5ea	23ea	0000 2200	0000	0081	009b	0001	0001
5f8	23f8	0000 2294	0000	0096	0086	00a8	0035
606	2406	0000 22e0	0000	0096	00ba	003b	0003

Table 2.7: MethodDef table

Each row in the MethodDef Table (see Table 2.7) represents a method that is defined in this assembly. The RVA field is the RVA to the Method Header that contains the information and CIL about the method to which this row refers. The ImplFlags field contains flags like method is implemented in CIL code (0x0000) and method is implemented in native code (0x0001). The Flags field contains flags for the visibility of this method. The Name field indexes into the #Strings stream with the ASCII representation of the name of this method. The Signature field is an index into the #Blob stream, which contains the encoded representation of the parameters and return type of this method. The ParamList field is an index into the Param Table, which starts a line of parameters that belong to this method [37].

In Table 2.7, the RVA field in row one references Table 2.1, which is the method header for the BubbleSort constructor. The ImplFlags for row one is 0x0000 because the method is implemented in CIL. The Flags field has the RTSpecialName (0x1000), SpecialName (0x0800), HideBySig (0x0080) and Public (0x0006) flags set. The first two flags are set because this method is treated special because it is a constructor. The third flag tells how inheritance should hide this method if it is overridden [37]. The last flag tells the visibility of this method. The Name value references the ASCII string “.ctor” by following

the offset 0x694 (0x67c + 0x0018 = 0x694) in the #Strings stream. This string is the internal name for constructors. The Signature value references offset 0x741 (0x0740 + 0x0001 = 0x0741) in the #Blob stream (see Table 2.13), which contains the bytes 0x3020 0001. The first byte is the size of the #Blob row minus the size byte. The next byte contains a calling convention flag and a method type flag. There are two mutually exclusive calling convention flags: DEFAULT (0x00) and VARARG (0x05). The DEFAULT flag is set when a method passes it's parameters by pushing them onto the stack before calling the method. The VARARG flag is used when a method has a variable number of arguments. Special CIL instructions are used to put and get the parameters on and off the stack when the VARARG flag is set. There is only one method type flag: HASTHIS (0x20). The presence of this flag means that this is an instance method. If the HASTHIS flag is not set, then this is a static method. This method has the DEFAULT and HASTHIS flags set. The next byte contains the number of parameters. The BubbleSort constructor has no parameters. The next byte encodes the return type. Since this is a constructor, the magic number ELEMENT_TYPE_VOID (0x01) is used [37]. If the method has parameters an encoded value or magic number for each parameter would follow. The ParamList is an index into the Param table that begins the list of parameters for this method. Note that the method represented by row 2 has parameters that start at index 1. Therefore, the BubbleSort constructor has parameters row one to one, which represents no parameters.

Offset	**RVA**	**Flags**	**Sequence**	**Name**
614	2414	0000	0001	00ad
61a	241a	0000	0002	00b3

Table 2.8: Param table

Offset	RVA	Class	Name	Signature
620	2420	0009	0018	0001
626	2426	0011	0018	0001
62c	242c	0011	0041	000e
632	2432	0011	0045	0013
638	2438	0021	0056	0018
63e	243e	0011	0060	001d
644	2444	0011	006a	0021

Table 2.9: MemberRef table

The Param Table can be seen in Table 2.8. Each row in the Param Table represents one parameter used by a method. The Flags field contains flags about the usage of a parameter. The Flags field contains flags about whether a parameter is called by reference or by value. The Sequence field contains the number of total parameters in the method that own this parameter. The Name field is an index into the #Strings stream which contains the ASCII representation of this parameter's name.

Offset	RVA	Signature
64a	244a	002b
64c	244c	002f
64e	244e	002b

Table 2.10: StandAloneSig table

The MemberRef Table follows the Param Table (see Table 2.9). Each row in the MemberRef Table represents either a reference to an external method or field in another assembly. The Class field is an encoded index into either the TypeDef (B000), TypeRef (B001), ModuleRef (B010), MethodDef (B011), or TypeSpec (B100) table. Bits 0 through 2 are used to determine which table and the other 13 bits are the row number [37]. For example, the Class field in row 1 indexes into the TypeRef Table (B0000 0000 0000 1001) at row 1 (B0000 0000 0000 1001). The Name field is a byte index into the #Strings stream, which represents the name of this external field or method. Therefore the name of row 1 is

".ctor" that is the internal name for a constructor. The Signature field is a byte index into the #Blob stream, which contains either the type of the external field or the parameters and return type for the external method.

Offset	RVA	Value	String
67c	247c	00	
67d	247d	6d73 636f 726c 6962 00	mscorlib
686	2486	4f62 6a65 6374 00	Object
68d	248d	5379 7374 656d 00	System
694	2494	2e63 746f 7200	.ctor
69a	249a	4172 7261 794c 6973 7400	ArrayList
6a4	24a4	5379 7374 656d 2e43 6f6c 6c65 6374 696f 6e73 00	System.Collections
6b7	24b7	496e 7433 3200	Int32
6bd	24bd	4164 6400	Add
6c1	24c1	6765 745f 4974 656d 00	get_Item
6ca	24ca	436f 6e73 6f6c 6500	Console
6d2	24d2	5772 6974 654c 696e 6500	WriteLine
6dc	24dc	6765 745f 436f 756e 7400	get_Count
6e6	24e6	7365 745f 4974 656d 00	set_Item
6ef	24ef	4275 6262 6c65 536f 7274 00	BubbleSort
6fa	24fa	3c4d 6f64 756c 653e 00	¡Module¿
703	2503	4275 6262 6c65 536f 7274 2e65 7865 00	BubbleSort.exe
712	2512	6e75 6d73 00	nums
717	2517	646f 4275 6262 6c65 536f 7274 00	doBubbleSort
724	2524	7377 6170 00	swap
729	2529	6669 7273 7400	first
72f	252f	7365 636f 6e64 00	second
736	2536	4d61 696e 00	Main
73b	253b	00	

Table 2.11: #Strings stream

The StandAloneSig Table can be seen in Table 2.10. Each row of the StandAloneSig Table is referenced by one Method Header. The Signature field is a byte index into the #Blob stream, which contains the encoded types of the local variables in a method.

Several data streams follow these tables beginning with the #Strings stream (see Table 2.11) starting at RVA 0x247c. The #Strings stream is a byte stream of ASCII strings separated by the null character. The #Strings stream is used by many parts of the assembly including the metadata sections.

Offset	RVA	Value	String
73c	253c	00	
73d	253d	00	
73e	253e	00	
73f	253f	00	

Table 2.12: #US stream

This is followed by the #US stream (see Table 2.12) starting at RVA 0x253c. The #US stream is a byte stream of ASCII strings that were string literals used by the programmer in the original source code. Since there were no string literals in the code, the #US stream is empty. The four null bytes were added because the size of each stream must be a multiple of four.

Offset	RVA	Byte(s)
740	2540	00
741	2541	0320 0001
745	2545	08b7 7a5c 5619 34e0 89
74e	254e	0420 0108 1c
753	2553	0420 011c 08
758	2558	0400 0101 1c
75d	255d	0320 0008
761	2561	0520 0201 081c
767	2567	0306 1209
76b	256b	0307 0108
76f	256f	0507 0308 0808
775	2575	0520 0201 0808
77b	257b	0300 0001
77f	257f	00

Table 2.13: #Blob stream

The #Blob stream starting at RVA 0x2540 can be seen in Table 2.13. The #Blob stream conatins encoded type information about method parameters, fields and local variables. Consider the swap method for example (see Appendix A). The swap method takes two ints as parameters and returns void. The swap method is represented by the third row in the MethodDef Table (see Table 2.7). Note the value of the Name field (0xa8). This offset points to the ASCII string “swap” at offset 0x724 (0x67c + 0xa8 = 0x724) in the

#Strings stream (see Table 2.11). The ParamList in the same row of the MethodDef table contains the value 0x35. This is an offset into the #Blob stream that contains the parameters and return types for the swap method. These encoded types are at offset 0x775 (0x740 + 0x35 = 0x775) in the #Blob stream. The first byte, which is 0x05, is the length of the #Blob entry minus the size byte itself. The next byte contains flags. The HASTHIS (0x20) flag and the DEFAULT (0x00) flag are set in this example. The HASTHIS flag specifies that this is an instance method. The DEFAULT flag specifies that this method passes it's parameters on the stack. The next byte contains the number of parameters of this method. In this case, swap has 0x02 parameters. Next is the encoded type for the method's return value: 0x01 (ELEMENT_TYPE_VOID) for a void method. Next comes an encoded value or magic number for each of the parameters. In this case, two 0x08 (ELEMENT_TYPE_I4) follow, which represents the int32 value type [37].

2.3 Common Intermediate Language Instruction Format

This section covers the encoding and use of a subset of the Common Intermediate Language (CIL) instructions used by Common Language Infrastructure (CLI) assemblies. The CIL instructions covered will be the ones necessary to understand the optimizations covered in Chapter 4 of this thesis. CIL instructions are composed of one or two operation code (or opcode) bytes followed by zero or more extra bytes.

The first group of instructions to cover are branch instructions. Branch instructions are instructions that can change the flow of instruction execution. Branch instructions come in two types: unconditional and conditional branches. Unconditional branches are always taken and conditional branches are taken depending on the result of a comparison operation.

Bytes	CIL instruction
2b 08	br.s 0x8
38 00000021	br 0x21

Table 2.14: Unconditional branch examples

Table 2.14 contains examples of the only two unconditional branch instructions in the CLI: br and br.s. The Bytes column is in hexadecimal and the intermediate values in the CIL instruction column is in decimal. Both instructions start with a 1-byte opcode. After the 1-byte opcode, the br.s example has a 1-byte intermediate value following it. All intermediate values in CIL instructions are signed unless noted. When the example br.s instruction is executed, execution will jump eight bytes from the end of the br.s instruction in the byte instruction stream. The br instruction is a long branch because it uses four bytes to encode the jump offset. The br example in Table 2.14 transfers control to 0x21 (or 33) bytes after the end of the br instruction.

Bytes	CIL instruction
3d fffffff2	bgt 0xfffffff2
37 ff	blt.un.s 0xff
39 00000010	br.false 0x10

Table 2.15: Conditional branch examples

All of the conditional and unconditional branch instructions also come in two forms: a long and a short version. The short version has a 1-byte intermediate value and the long version has a 4-byte intermediate. Table 2.15 gives a few examples of conditional branches. Whether these branches are taken or not depends on the result of a comparison to the top of the internal stack. The bgt example will branch to the instruction 14 bytes before (0xfffffff2 = -14) the end of the bgt instruction if and only if the top of the stack is greater then the second element on the stack. The second example is a short branch (".s") and the

intermediate is interpreted as an unsigned value (".un"). Therefore, if the top of the stack is less than the second element on the stack, execution will jump 0xff (or 255) bytes after the blt.un.s instruction. Note, long and short forms are available for every branch, but not every branch has an unsigned equivalent. Comparisons using booleans are also allowed as in the last example from Table 2.15. If the boolean value false is found on the top of the stack, execution jumps 0x10 (or 16) bytes after the end of the br.false instruction.

Bytes	**CIL instruction**
16	ldc.i4.0
1f 14	ldc.i4.s 0x14
04	ldarg.2
11 20	ldloc.s 0x20
14	ldnull
7b 04000001	ldfld 0x4000001

Table 2.16: Load examples

The next group of CIL instructions are used to push elements to the top on the stack. Table 2.16 contains some, but not all, of the instructions to load elements onto the stack. Also, note that some instructions have side effects that place elements to the stack (like the add instruction), but they are not covered here. The first two instructions in Table 2.16 load constants onto the stack. For example, ldc.i4.0 pushes zero as a four-byte integer. The instructions that push constants come in both the long and short versions. The next example, ldc.i4.s, pushes a 1-byte signed intermediate onto the stack. Similar instructions to load *floats* (ldc.r4) and *doubles* (ldc.r8) are also available.

Internally each method numbers its parameters and instructions that load parameters onto the stack and uses these numbers to reference the parameters instead of using the parameter's name. For non-instance methods (i.e., static), parameters are numbered left to right starting at 0. Therefore, the first parameter is numbered 0, the next is 1, then

is 2 and so on. For instance methods, the first parameter is numbered 1, the next is 2 and so on. Instance methods start numbering at 1 because parameter number 0 is reserved for the *this* pointer. The next example, ldarg.2, loads argument 2 onto the stack. There are special 1-byte ldarg.0 to ldarg.3 instructions that load arguments with numbers built into the opcode. There are also short and long version of the ldarg instructions to load arguments with numbers larger than 3.

Local variables are also numbered starting at 0 where they are declared. The ldloc.s example loads local variable 0x20 (or 32) onto the stack. The ldloc group of instruction also has a few instructions that have the local number built in (ldloc.0 to ldloc.3) and short and long versions of the ldloc instruction. The ldnull instruction loads the null value onto the stack.

The final example in Table 2.16, ldfld, loads a field onto the stack. The intermediate value is a 4-byte token. The first byte is a table constant and the other three bytes are the row number. Table 0x04 is the Field metadata table. Therefore, this example loads the field represented by row one in the Field metadata table. Similar instructions to store into local variables, arguments and fields are available also.

Bytes	**CIL instruction**
28 0a000001	call 0xa000001
6f 0a000002	callvirt 0xa000002
27 0a000003	jmp 0xa000003
29 11000006	calli 0x11000006

Table 2.17: Method call examples

Table 2.17 contains instructions used to invoke method calls. There are only four CIL instructions to invoke methods: call, callvirt, jmp and calli. For all four of these instructions the execution environment automatically pushes a reference to the variable

used to invoke the method if this is an instance or virtual method. This variable is not checked to see if it is null and thus an exception may be thrown at runtime [43]. A call may be used to invoke a static method in which no reference is pushed onto the stack. The intermediate token value associated with the call is an index into the MethodDef or MethodRef metadata tables. The first example calls the method defined by row one of the MethodRef table (0x0a). Arguments to the method must be pushed onto the stack before the method is called. If the method has a return value, it will be on the stack after the call instruction executes.

The callvirt instruction is used to call instance and virtual, but not static methods. If this is a call to an instance or virtual method, code is automatically generated for the callvirt instruction to check if the variable calling this method is null. If this is a virtual method call, the runtime inspects the variable calling this method to call this method polymorphically [43]. Parameters and return values are passed on the stack. The second example, callvirt, invokes the method defined by row two of the MethodRef table.

The jmp instruction pushes no parameters because when it is called because it inherits the parameters of the caller. Therefore the method being called must have the same number and types of parameters as the caller [26]. The next example uses the jmp instruction to call the method specified by row three of the MethodRef table.

For calli instructions, a function pointer to the method that is to be called must be pushed onto the stack using either the ldftn or ldftnvirt instruction after the method's parameters have been pushed. The calli instruction is used to call native code. The token in the calli instruction describes the signature of the method being called [37]. The final example uses the calli instruction to call the method that has the signature described in row 6 of the StandAlongSig table (0x11).

2.4 Related Work

After Microsoft released the .NET Framework and its specification in 2001, the company also released a reference implementation of the CLI called the Shared Source Common Intermediate Language [48]. The license for this implementation strictly forbids it from being used commercially, but the source code is available for educational purposes. Microsoft's Visual Studio, which is a .NET integrated development environment, ships with a tool called *ildasm* that allows the user to view, but not edit .NET assemblies.

Mono is an open source implementation of the .NET Framework which was first released in 2004 [39]. In 2011, Mono added support for .NET 4.0 and has announced upcoming support for the current version of .NET 4.5. There is another open source implementation of the .NET Framework that was started by the Free Software Foundation which is called DotGNU. However, DotGNU supports an older version of .NET, a new release has been made available since 2009. Both Mono and DotGNU have the compilers, the .NET Framework Common libraries and the virtual machines needed to run .NET code. Unfortunately, they are not structured as reusable libraries.

Microsoft Research released a technical overview of .NET in 2000 [27]. There has also been research in writing compilers for various languages that output .NET assemblies. For example, compilers have been written for Mercury [14], which is a functional programming language similar to Haskell and ML, C [20] and Ada [7]. Microsoft Research also released a paper on how generics were implemented in the .NET Framework [53]. Others have evaluated the .NET Framework to see if it can be used in real-time systems [54]. Papers have also been written on validating and sandboxing CIL code for various security reasons like limiting the impact of running untrusted code [5].

With all of this research in the .NET community, it is surprising that little deals with very little deals with optimizing .NET CIL code. Research has been published about optimizing Java's intermediate bytecode language [41]. This thesis develops a reusable library called the Common Language Infrastructure for Research (CLIR) to provide researchers with the tools to read, write and edit .NET assemblies and a toolset of optimizations, algorithms, and data structures.

Chapter 3

CLIR: Common Language Infrastructure for Research

This chapter presents the Common Language Infrastructure for Research (CLIR) developed as a part of this thesis, which enables low-level research on the Microsoft .NET intermediate language. CLIR itself consists of two research components, and a third component that demonstrates the use and practicality of CLIR. The first research component is the Common Language Engineering Library (CLEL). CLEL forms the foundation by providing functionality allowing the reading and writing of low-level .NET assemblies. The second research component is the Common Language Optimization Toolset (CLOT). CLOT uses the capabilities of CLEL to provide higher-level functionality targeting code optimization.

This chapter introduces the CLEL and details the application programming interface (API) available to researchers. Chapter 4 covers the CLOT component and Chapter 5 describes a use of these CLIR components. Figure 3.1 depicts the relationship of these components.

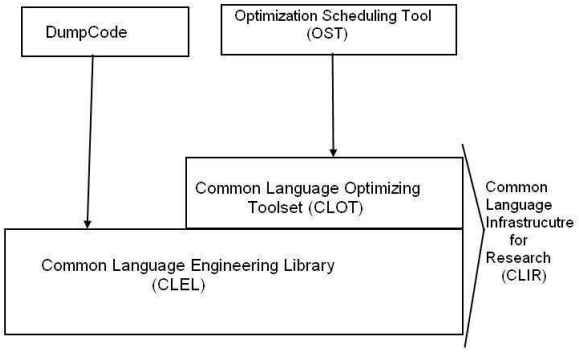

Figure 3.1: CLIR overview

3.1 CLEL: Common Language Engineering Library

CLEL
+ CLEL(ICLELReader reader) + CLEL(String local_path) + void WriteAssembly(ICLELWriter writer) + Token GetEntryPointToken() + List<ClassDescriptor> GetInternalClasses() + List<MethodDescriptor> GetMethodDescriptorForClass(Token token) + String GetMethodsName(Token token) + List<CLELInstruction> GetCodeForMethod(Token token) + void SetMethodsCode(MethodDescriptor md, List<CLELInstruction> code, MethodLocalsBlobInfo locals)

Figure 3.2: CLEL class

The Common Language Engineering Library (CLEL) is the foundation layer that supports .NET language research projects such as optimization. The CLEL provides functionality to read and write assemblies. The CLEL exposes its functionality through the CLEL class, which is shown in Figure 3.2. A "client," such as an optimization in CLOT or the DumpCode utility, instantiates a CLEL object and accesses internals of assembly files via methods in the CLEL object. The design of the CLEL component follows the facade design pattern [17] whereby the CLEL class acts as a simpler entry point to a more complex set of classes. This arrangement is depicted in Figure 3.3.

As is common with the facade pattern design, the CLEL class acts as a type of wrapper to simplify the interface. Most of the real work is done by the CLELAssembly

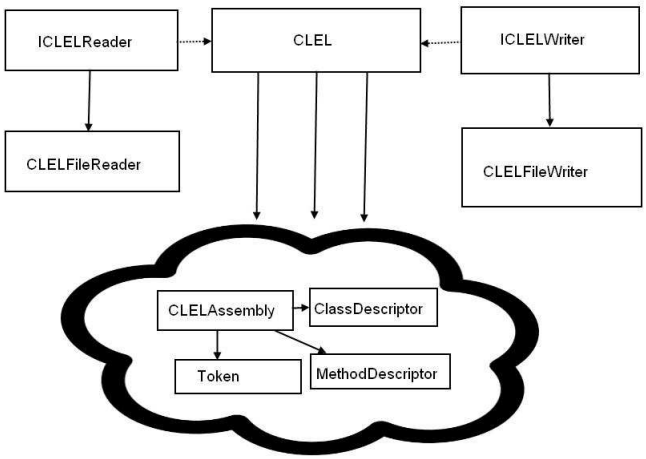

Figure 3.3: CLEL overview

class, which implements the methods that parse, edit and write .NET assemblies. The CLEL class validates a client request and then passes it on to a similarly named method in the CLELAssembly class. The CLEL class API is shown in Figure 3.2 and the CLELAssembly class is shown in Figure 3.6.

ICLELReader
+ bool IsLittleEndian
+ int ReadByte(ref byte val) + int ReadShort(ref short val) + int ReadInt(ref int val) + int ReadLong(ref long val) + int Read(byte [] buffer, int start, int length) + void Close()

Figure 3.4: ICLELReader interface

ICLELWriter
+ bool IsLittleEndian
+ void WriteByte(byte val) + void WriteShort(short val) + void WriteInt(int val) + void WriteLong(long val) + void Write(byte [] buffer, int start, int length) + void Close()

Figure 3.5: ICLELWriter interface

When creating a new CLEL object, the constructor requires an instance of the ICLELReader interface (see Figure 3.4). This interface defines the low-level methods necessary to read a stream of bytes that comprise a .NET assembly. Typically, this byte stream comes from a file stored on a local disk. CLEL provides an implementation of the reader interface, CLELFileReader, for this common usage. However, researchers can easily provide custom implementations of the interface to read .NET assemblies from other sources (e.g., via a network web service). This ability to use alternate sources is termed a "de-

pendency injection" [13][15] and provides future flexibility without needing to modify the base classes (i.e., CLEL and CLELAssembly). The ICLELWriter interface works similarly and CLEL provides a default implementation CLELFileWriter that writes to local files on disk. Because reading and writing to local disk files is so common, the CLEL class also provides an even simpler constructor that accepts a filename and internally manages the CLELFileReader.

Since the .Net assembly file format is an extension of the Windows executable Portable Executable (PE) file format [37], the fields that are the same in both are kept in a class called CLELExecutable. The CLELAssembly class inherits from the CLELExecutable class and implements its own text section, which is the significant difference between PE files and .NET assemblies, using the AssemblyTextSection class. The AssemblyTextSection contains all the methods needed to get and set all the fields in the various tables and sections in the .NET assemblies text section.

CLELAssembly
- AssemblyTextSection _text
+ CLELAssembly(ICLELReader reader) + void Write(ICLELWriter writer) + Token GetEntryPointToken() + List<ClassDescriptor> GetInternalClasses() + List<MethodDescriptor> GetMethodDescriptorForClass(Token token) + String GetMethodsName(Token token) + List<CLELInstruction> GetCodeForMethod(Token token) + void SetMethodsCode(MethodDescriptor md, List<CLELInstruction> code, MethodLocalsBlobInfo locals)

Figure 3.6: CLELAssembly class

The CLELAssembly class contains the method GetEntryPointToken that returns an internal reference token that is used to locate the Main method in the metadata tables [37]. If the current assembly is a dynamically linked library (DLL) rather than a .NET assembly, then this method returns null. The Common Language Infrastructure (CLI), which defines the specification of the .NET assembly file format, uses tokens to locate many objects in

its internal metadata tables. The Token class (see Figure 3.7) implements this CLI data structure and is devised of two pieces of data: a table constant and a row number. The table constant tells which metadata table the object is in and the row number is used to locate which row in the table represents the object.

Token
+ byte Table + int Row
+ Token(int encoded_token) + Token(byte table, int row) + Token Copy() + int GetEncodedToken()

Figure 3.7: Token class

ClassDescriptor
+ int Type
+ ClassDescriptor(int type) + byte [] ToBytes()

Figure 3.8: ClassDescriptor class

BuiltInClassDescriptor
+ byte BuiltInType
+ BuiltInClassDescriptor(byte builtin_type) + byte [] ToBytes()

Figure 3.9: BuiltInType class

MethodDescriptor
+ Token MethodToken + List<ParameterDescriptor> Parameters()
+ MethodLocalsBlobInfo Locals()
+ MethodDescriptor(CLELAssembly assembly, Token token) + List<CLELInstruction> GetMethodsCode() + void SetMethodsCode(List<CLELInstruction> code, MethodLocalsBlobInfo locals)

Figure 3.10: MethodDescriptor class

The GetInternalClasses method of the CLELAssembly class returns a list of all of the classes defined in the current .NET assembly. Each member of this list is a ClassDescriptor (see Figure 3.8). There are many different types of classes supported in .NET and the type field in the ClassDescriptor is used to distinguish between each type. Some of these types are built-in types (like int, float, and bool), arrays, structures, classes and generic classes. Each of these different .NET types are represented by CLEL classes that inherit from ClassDescriptor. For example, the BuiltInType class (see Figure 3.9) represents the built-in types mentioned earlier (int, float, etc.).

The GetMethodDescriptorsForClass method of the CLELAssembly class returns a list of all of the methods for the class indicated by the token parameter. Each method in the list is represented by a MethodDescriptor object (see Figure 3.10). Using a MethodDescriptor object, researchers can retrieve the code, set the code, get the parameters, and set the parameters for that associated method. The GetCodeForMethod method, in the CLELAssembly API, returns the code for the given MethodDescriptor as a list of CLELInstruction objects. The CLELInstruction class (see Figure 3.11) serves as a parent class for each of the instructions provided by CLEL. Thus, each instruction in CLI is implemented by a class that inherits from CLELInstruction, setting the appropriate opcode and implementing any extra fields the instruction needs. The list of all of the opcodes in CLI is implemented as static constants in the CLELOpcode class. The GetBytes method of the abstract CLELInstruction class is overridden by the concrete child class and returns the bytes that represent that specific instruction in the .NET assembly.

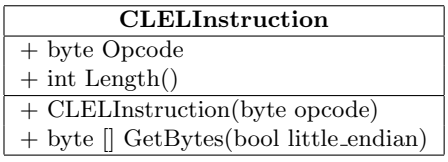

Figure 3.11: CLELInstruction class

To change the code of a method in the .NET assembly, CLELAssembly provides the SetMethodsCode method. Parameters to the SetMethodsCode method indicate which .NET assembly method to update, the new CLI code, and a MethodLocalsBlobInfo object that represents the new local variable information.

3.2 CLOT: Common Language Optimization Toolset

The Common Language Engineering Library (CLEL) described in the preceding section provides the foundational capabilities of reading and writing .NET assemblies. Researchers generally have higher-level ambitions including, for example, performing a code transformation that will improve program performance. Such code transformations are typically called optimizations. The CLIR provides the Common Language Optimization Toolset (CLOT) as a collection of classes that implement various optimizations and optimization utilities. Researchers can easily add new optimizations and new utilities to the CLOT.

opt	**opt.lib**	
IOptimization	ClassIFE	
OptimizationConfiguration	ConvergePassBIR	
	OnePassBIR	
	PeepholeBIR	
	LocalCP	
opt.Analysis.Graph	**opt.Analysis.Set**	**opt.Analysis.Branch**
Graph	BitSet	BranchTable
GraphNode	GenKill	
GraphArc	ReachingDefinitions	
ControlFlowGraph		
CallGraph		

Figure 3.12: CLOT namespaces overview

CLOT consists of a namespace called *opt* with two nested namespaces called *lib* and *Analysis* that organize the constituent classes. Figure 3.12 gives a visual overview of the namespaces and the classes contained in each namespace in CLOT. The *opt* namespace itself contains general purpose optimization classes and interfaces, such as IOptimization described below. The *opt.lib* namespace contains concrete optimizations, such as the various versions of Branch Instruction Replacement (BIR), whose implementation is described in Chapter 4. The *opt.Analysis* namespace provides utility data structures and algorithms commonly used by many optimizations. The *opt.Analysis* namespace is further subdivided into three namespaces: Branch, Set, and Graph. The *opt.Analysis.Graph* namespace con-

tains graph-related data structures specifically. For example, there is an abstract Graph class and a concrete ControlFlowGraph class. The *opt.Analysis.Set* namespace contains classes used to implement set theoretic algorithms such as GenKill and ReachingDefinitions [3]. The *opt.Analysis.Branch* namespace contains utility classes for maintaining low-level control flow and branch information.

IOptimization
+ String Name + String Key + String Description
+ void DoOptimization(CLEL clel,CLELLogger log)

Figure 3.13: IOptimization interface

Every CLOT optimization must implement the IOptimization interface, which is shown in Figure 3.13. Three of the required methods extract identification information about the optimization, and invoking the DoOptimization method initiates execution of the optimization. Notice that this method requires a CLEL object that encapsulates the .NET assembly that is the target of the optimization.

OptimizationConfiguration
- String _path - Dictionary<String,String> _config
+ OptimizationConfiguration(String path) + String GetConfigValue(String name)

Figure 3.14: OptimizationConfiguration class

Each optimization can optionally have a configuration file represented by the OptimizationConfiguration class (Figure 3.14). The OptimizationConfiguration constructor takes a path to an xml file containing configuration name and value pairs. The configuration file allows the researcher to tweak an optimization without changing code. It is up to the person writing the optimization to load their own configuration file and to manage it. For example, the method inlining optimization can result in an increase in code size so a threshold can be established to control the amount of inlining performed. This

threshold can be more easily reconfigured using the configuration file. Therefore the OptimizationConfiguration class is used by the optimizations that implement the IOptimization interface.

When the optimization is called using the DoOptimization method, a shared CLELLogger object is passed in along with the CLEL object, which represents the current assembly. The CLELLogger object is opened by the OptimizationController and can be used to write any data that might be useful later. This log could be used when debugging the optimization or it could be used to keep track of statistics like how often the optimization was applied to the code.

The *opt.Graph* namespace contains very useful high-level abstractions that deserve a more detailed description. The CallGraph and ControlFlowGraph classes are concrete classes based on the generic Graph class. The CallGraph represents the entire program, with the methods as nodes and method calling instructions as the arcs in the graph. The ControlFlowGraph represents a single method with basic blocks of single-entry-single-exit code as nodes. An arc is added between two nodes if a branch ends at one block and goes to the second block [3] [47].

The graph data structures employ the Visitor Design Pattern [42], which allows a researcher to easily inject node-specific processing as the graph is traversed. In order to traverse the CallGraph and ControlFlowGraph, the CallGraphVisitor and ControlFlowGraphVisitor classes are used. The CallGraphVisitor class, which takes the CallGraph to traverse as a constructor parameter, has an *accept* method that takes one parameter of IGraphVisit type. The ControlFlowGraphVisitor, which takes a ControlFlowGraph as a constructor parameter, has a similar *accept* method. The IGraphVisit interface defines a single method to implement, which is the *visit* method that only takes a GraphNode as a

parameter. This GraphNode is the current node that is being visited as the visitor classes traverse the graph. This way the optimization researcher does not need to write these data structures, rather they need only to implement the IGraphVisit interface with a class that contains the specific logic they need for their optimization.

Three optimizations are provided by default with CLOT: branch instruction replacement, constant propagation, and method inlining. These optimizations are described in detail in Chapter 4.

3.3 User Applications

CLEL forms the foundation of reading and writing .NET assemblies. A user application can directly make use of CLEL functionality. The CLOT sits on top of the CLEL and provides higher-level functionality related to optimizations. This section describes two user applications: DumpCode and the Optimization Scheduling Tool (OST).

A tool called DumpCode was written to help examine and debug assemblies as CLEL and CLOT were written. DumpCode is a user application that directly accesses CLEL functionality. Similar to the Linux objdump tool, which prints an executables assembly code, DumpCode prints the CIL code in an assembly in a more readable form. DumpCode was built using CLEL as a foundation to parse the assembly and return the code to print. DumpCode then prints the namespace, class name and string representation of each CIL instruction. As CLEL was written, DumpCode was used to verify that CLEL was parsing and writing the CIL in the assembly correctly. While writing the optimizations covered in this thesis, DumpCode was also used to verify that the transformations applied to the CIL code were correct. Other than the uses for this thesis, DumpCode could be helpful to anyone needing to view the CIL code in an assembly.

The Optimization Scheduling Tool is a much larger effort providing a user the ability to run optimizations in a variety of different orderings. Aho et al. [1] demonstrate that optimization orderings do in fact alter the resulting code. Thus, being able to easily reconfigure tests is crucial for optimization researchers. The OST is a graphical utility that achieves this reconfiguration.

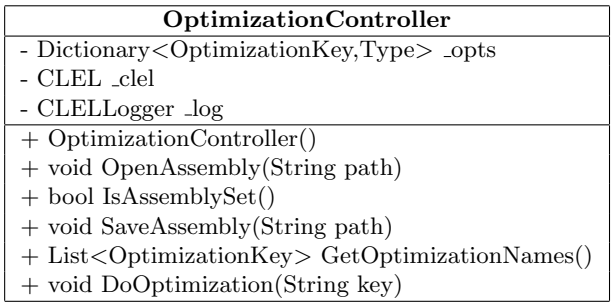

Figure 3.15: OptimizationController class

The OptimizationController class is shown in Figure 3.15. This class scans the assemblies in the *opt.lib* directory and uses reflection to examine every class contained there. If a class has implemented the IOptimization interface then the OST recognizes this as an available optimization. Thus, the determination of available optimizations occurs dynamically. After writing a new optimization, a researcher only needs to put the optimization class file into the correct folder and OST will automatically find it and allow it to be scheduled. In particular, OST does not need rebuilding for each optimization written. Reflection also allows OST to access the optimization name and description and to initiate the optimization execution (by calling the DoOptimization method).

When OST starts, a list of all available optimizations are loaded into the leftmost text box shown in Figure 3.16. The text box on the right lists what optimizations will be run and in what order. The user can add optimizations from the left text box by selecting one and clicking the button with the plus on it. To delete optimizations, select the optimization

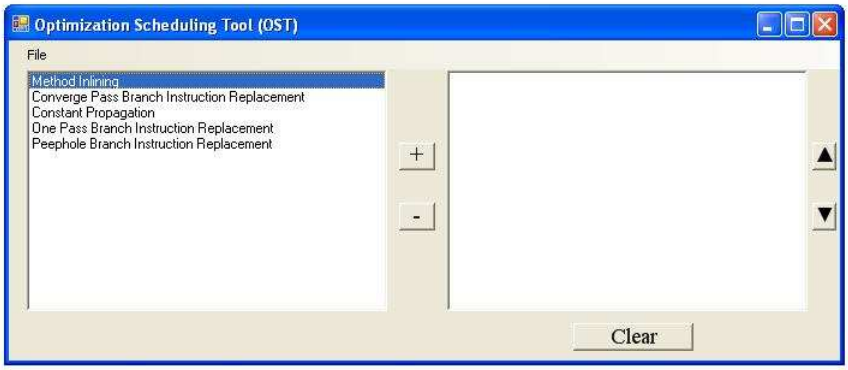

Figure 3.16: Optimization Scheduling Tool

from the right text box and click the minus button. The optimizations are performed in order top to bottom. The up and down arrows on the far right can be used to reorder them.

The file menu has an open item that allows the user to select the .NET assembly to be optimized. All of the optimizations in the right text box will be applied when the run item is chosen from the file menu. The save item in the file menu allows the optimized .NET file to be saved to disk.

OptimizationKey
+ String Name + String Key + String Description
+ OptimizationKey(String name,String key,String description)

Figure 3.17: OptimizationKey class

In order to populate the text box of available optimizations, the GetOptimization-Names method is called and it returns a List of OptimizationKeys. Each OptimizationKey represents an optimization that is available to run. The OptimizationKey class (Figure 3.17) contains information like the name of the optimization, a lookup key and a description. The OpenAssembly method is called within OST to load an assembly. The IsAssemblySet returns true or false depending on whether there is currently an assembly loaded. The DoOptimization method is used to run each optimization selected by the user.

Chapter 4

Optimizations Background and Implementation

4.1 Optimization Overview and Goals

Three different optimizations were chosen to demonstrate the usefulness of the Common Language Infrastructure for Research (CLIR) and by extension the Common Language Engineering Library (CLEL). The goal is to show that a wide range of optimizations can be developed using the Common Language Optimizing Toolset (CLOT) components. Optimizations share the algorithms and data structure contained in CLOT and also the ability to read and write .NET assemblies by using the CLEL component.

The three optimizations selected were Branch Instruction Replacement (BIR), Constant Propagation (CP) and Method Inlining. Branch Instruction Replacement is a type of peephole optimization that replaces one or more branch instructions with equivalent instructions that are more efficient. Three variations of BIR, which are called Peephole, Converge and One Pass BIR, are presented in Section 4.2. Constant Propagation replaces

a value in an expression with a constant value. Section 4.3 discusses the CP optimization. Method Inlining attempts to improve performance by replacing a method call with the body of the called method, thus removing the overhead of calling a method. Method Inlining uses interprocedural analysis to determine which methods to inline and is presented in Section 4.4.

4.2 Branch Instruction Replacement

4.2.1 Peephole Optimization Background

A peephole optimization is performed by examining a sliding window, which is called a peephole, of instructions to see if any of them can be replaced with a more efficient sequence of instructions [1]. This efficiency could be because the new instructions use fewer bytes in the code stream, take fewer cycles to execute on the CPU, or because the new instructions use less power when executed by the CPU. This optimization can be included as part of the compilation process or can be done independently after the code generation stage.

1) x = x - 0
2) x = x * 1
3) x = x * 2

Figure 4.1: Peephole examples

There are many types of peephole optimizations available. A common one is called Algebraic Simplification and Reduction of Strength [1]. Figure 4.1 has a few examples of these types of peephole optimization opportunities. If instructions like number 1 or 2 in Figure 4.1 are seen as the peephole slides through the code, they can be removed because they have no effect on the value of the variable x since subtracting zero or multiplying by one does not change the value of x. This algebraic simplification reduces the number of instructions executed as well as memory consumption because the code size decreased. Line

3 from Figure 4.1 is an example of Reduction of Strength in which an expensive instruction is replaced with a less expensive, but equivalent instruction. Since multiplying a value by 2 is the same as shifting it's binary value left by one bit, the instruction at line 3 can be rewritten as left shifting x by one. Typically, CPU architectures can shift a value faster then multiplying a value. Therefore the result will be the same, but execute faster.

Peephole optimizations examine the code in the peephole; no other code is considered. The peephole size can vary to examine one or several instructions at a time, but only those instructions are analyzed at that time. There have been several strategies to find opportunities for optimization in the peephole. One strategy converts the target assembly code in the peephole into a string, represents the peephole optimization as a regular expression and applies the regular expression to the code in the peephole [46]. By using a regular expression library, new peephole optimizations can be added dynamically with only a modest overhead.

Another approach is to have an input specification that describes both the code to find and the replacement code [19]. The input is turned into a finite state machine (FSM) that models patterns as a graph with states and arcs between the states representing transitions from one state to another. Each FSM is applied to the peephole as it slides through the code. A specific instruction could cause the transition from one state to another. One of the states represents successfully finding the optimization pattern, in which case the FSM triggers a replacement in the code.

4.2.2 Branch Instruction Replacement Implementation

Recall from Chapter 2 that the Common Intermediate Language (CIL) is the intermediate code resulting from program compilation. CIL has two groups of instructions for branches:

short and long branches. The short version has a 1-byte opcode and one byte for a numeric offset. The offset indicates how far in the code to branch. The long version also has one byte for the opcode, but uses four bytes for the offset. Sometimes a .NET compiler will emit a long branch when a short branch will suffice. For example, the br instruction, which is an unconditional long branch, is unnecessary if the offset can fit within one byte, a range of -128 to 127. See Figure 4.2 for an example of BIR.

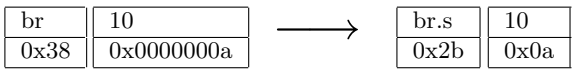

Figure 4.2: BIR example

The Branch Instruction Replacement (BIR) optimization was devised to replace long branches that don't need four bytes to represent the offset. Since the 10 (0x0a) in Figure 4.2 can be represented in one byte, the br instruction was converted to a br.s instruction, which is the short version of the unconditional br branch. By doing this instruction replacement, the code size decreased by three bytes because a 5-byte instruction was replaced by a 2-byte instruction.

```
for each class c
    for each method m in c
        for each instruction i in m
            if i is a long branch with offset between -128 and 127
                replace i with equivalent short branch with same offset
```

Figure 4.3: BIR algorithm (version 1)

The implementation of BIR must first find the branch replacement possibilities. A first attempt might look something like the pseudocode in Figure 4.3. This algorithm is a peephole optimization with the peephole window size being one instruction.

Unfortunately the simple algorithm in Figure 4.3 ignores the fact that changing the size of a branch impacts the offset with other branches. Consider the code fragment in Figure 4.4. If BIR is applied to the *br 15* instruction then this instruction uses three bytes

```
blt 160
.........
br 15
.........
//destination of blt instruction
```

Figure 4.4: Branch body

less than before. This causes the subsequent code to effectively "move up" from where it was. Thus the destination of the blt instruction is now only 157 bytes away instead of 160. An attempt to fix this problem could be to increase the size of the peephole window perhaps, but the *blt 160* instruction could always be outside this window.

```
                for each class c
                    for each method m in c
                        for each instruction i in m
O(i)                        if branch instruction
                                add to branch info table

                        for each branch b1 in branch info table
O(b^2)                      for each branch b2 in branch info table
                                if b2 is between b1 start and end offset
                                    add b1 to b2 over dictionary

                        for each branch b3 in branch info table
                            if b3 is a long branch with offset between -128 and 127
O(b^2)                          get list of branches that branch over b3 over_list from over dictionary
                                    for each branch b4 in over_list
                                        update offset of b4

O(cm(i+b^2+b^2)) = O(cmi+cmb^2+cmb^2) = O(cmi+cmb^2)
```

Figure 4.5: Peephole BIR algorithm (version 1.1)

A correct BIR algorithm can be seen in Figure 4.5. Before the instructions of a method are processed, a branch information table is created to keep track of the method's branches. Because of this persistent data, BIR is no longer technically a peephole optimization. By making the changes in version 1.1 of the BIR algorithm, the algorithm is no longer a pure peephole optimization because now instructions outside of the peephole window need to be examined. This table records the location in the code of each branch, its offset into the code, and what type of branch it is. This branch table is then processed

to create a table called *over* which maps each branch to a list of all the other branches that branch over it. In the Figure 4.4 example, the br instruction maps to the blt instruction. Therefore, when the br instruction is changed to a br.s instruction, we know to update blt's offset.

A theoretical performance analysis of version 1.1 of the BIR algorithm in Table 4.5 results in $O(cmi + cmb^2)$ where c is the number of classes, m is the number of methods and b is the number of branches in the input code. The cmi factor is fixed because every BIR implementation needs to examine each instruction in the input at least once before it can decide if it is replaceable or not. The second factor, cmb^2, is the true cost of this approach.

```
br 128
.........
blt 18
.........
//destination of br instruction
```

Figure 4.6: Branch body, with missed replacement

```
for each class c
    for each method m in c
        for each instruction i in m                                          O(i)
            if branch instruction
                add to branch info table

        for each branch b1 in branch info table                              O(b^2)
            for each branch b2 in branch info table
                if b2 is between b1 start and end offset
                    add b1 to b2 over dictionary

        do                                                                   O(b^3)
            changed = false
            for each branch b3 in branch info table
                if b3 is a long branch with offset between -128 and 127
                    changed = true
                    replace with equivalent short branch with same offset
                    get list of branches that branch over b3 over_list from over dictionary
                    for each branch b4 in over_list
                        update offset of b4
        while(changed)

O(cm(i+b^2+b^3)) = O(cmi+cmb^2+cmb^3) = O(cmb^3)
```

Figure 4.7: Converge BIR algorithm (version 2)

It should be noted that it is possible for version 1.1 of the BIR algorithm to miss some replacement opportunities. Consider the code fragment in Figure 4.6. Since version 1.1 of the algorithm only makes one pass through the code, a branch replacement opportunity will be missed. The br instruction will be considered to be replaced and will not be because 128 is greater than 127, which is the maximum values that can be stored in a 1-byte offset. Then the blt instruction will be considered and it will be replaced causing the offset of the br instruction to be updated to 125. Now the br instruction is replaceable, but the algorithm only makes one pass through the code and the br was already rejected for replacement.

BIR version 2 (see Figure 4.7) was developed to capture these missed replacement opportunities. Basically, the version 1.1 algorithm is repeated multiple times until no branch changes occur. This change ensures that all of the BIR replacement opportunities possible were found. Unfortunately, the worst case theoretical analysis is now $O(cmb^3)$. The first factor, cmi, is the same fixed factor like in version 1.1 of the algorithm. The second factor, cmb^2, is also the same as version 1.1. The final factor, cmb^3, is caused by adding the *do while* loop that continues to make passes through the code until it makes a pass through the code where no branches were replaced. In the case where every branch is replaceable eventually and only one branch is replaced with each pass, the *do while* loop will make b passes through the code doing cmb^2 with each pass, giving a running time of cmb^3. This version of the BIR algorithm is called Converge BIR.

Another version of the algorithm was developed to try and find a better way to process the branches in a method so that only one pass through the code is needed and no missed replacement opportunities occur. This version of the algorithm is called One Pass BIR. Consider Figure 4.6 again, the missed opportunity could have been avoided if the blt instruction was processed before the br instruction. Then the br instruction's offset would

```
                for each class c
                    for each method m in c
                        for each instruction i in m
O(i)                        if branch instruction
                                add to branch info table

                        for each branch b1 in branch info table
                            for each branch b2 in branch info table
O(b^2)                          if b2 is between b1 start and end offset
                                    add b1 to b2 over dictionary
                                    add b2 to b1 inner list in priority queue

                        while(priority queue not empty)
                            get and remove top of priority queue, which has branch b3 and inner_list
                            if b3 is a long branch with offset between -128 and 127
                                replace with equivalent short branch with same offset
                                get list of branches that branch over b3 over_list from over dictionary
O(b^2log b)                     for each branch b4 in over_list
                                    update offset of b4
                            for each branch list over_list2 in priority queue
                                for each branch b5 in over_list2
                                    if b5 equals b3
                                        remove b5 from over_list2

O(cm(i+b^2+b^2log b)) = O(cmi+cmb^2+cmb^2log b) = O(cmb^2log b)
```

Figure 4.8: One Pass BIR algorithm (version 3)

be updated to 125 before considered for replacement. In general, if all the "inner" branches of an "outer" branch were processed before that "outer" branch was processed, then each branch could be processed a single time. Pseudocode for version 3 of the algorithm is shown in Figure 4.8. A new list called *inner* maps the "inner" branches to its "outer" branches.

```
    br 25                   br 25    blt 10
    ........                blt 10
    blt 10
    ........
    //destination of blt    beq 8
    ........
    //destination of br
    ........
    beq 8
    ........
    //destination of beq
    ........
```

Figure 4.9: Branch lists example

Figure 4.8 contains pseudocode for this third version. First, an information table is created that contains a row for each branch in the code. This row contains the type of

branch it is, the index into the code list the branch is at, the byte offset into the code's instruction stream where the branch is and the byte offset into the code's instruction stream of where the branch's destination is. Next a list is built for each branch in the method using this information table; for example, see Figure 4.9. This list contains all other branch instructions that this current branch jumps over. Since the br instruction jumps over the blt instruction, the blt instruction is added to br's list. Once these lists are built, they are stored in a priority queue sorted by the length of the list. Thus the top of the priority queue contains the list with the smallest number of items in the list. The *while* loop checks if the priority queue is empty. If not, the next branch is processed and removed from the top of the priority queue.

Since there is always an inner most branch at any point (assuming the code contains at least one branch), there will be a list that is empty because this inner most branch does not branch over anything. Since a priority queue is used, this empty list will be on the top of the priority queue. We also know that the offset it has will not change because there are no branches in its body to examine. Therefore, we can now examine this branch's offset to see if it can be replaced or not without fear that whether it can be replaced or not will change in the future.

The next block in the pseudocode checks if the branch removed from the top of the priority queue is replaceable. If so, the branch is replaced and all other branches affected are updated. The next *for* loop removes the found branch from the other lists in the dictionary, so that the next inner most branch can be found and processed. After all of the branches have been examined, the *while* loop ends. By using a priority queue to process each branch only once, the theoretical performance analysis was reduced to O($cmb^2 log\ b$) because retrieving and removing the top of the priority queue is done in $log\ b$.

4.3 Constant Propagation

4.3.1 Constant Propagation Background

Another class of optimizations can be solved using set theory, which is a branch of mathematics that studies groups of objects called sets and the operations that can be done on sets. One optimization that can be solved with the help of set theory is constant propagation (CP). CP replaces an expression with a constant value if it can be proved that the expression is always the same value at this specific point in the program [1].

```
int a = 5;                    int a = 5;
int b = 3;                    int b = 3;
final int c = 2;              final int c = 2;
while(b < 100)                while(b < 100)
{                             {
    b = a + b + c;                b = 5 + b + 2;
}                             }
1) Before                     2) After
```

Figure 4.10: Constant propagation example

Some programming languages, like Java for example, have a keyword that can be used to mark variables or fields as constant. It is up to the compiler or interpreter to make sure that variables or fields marked with this keyword are not changed once they are initialized. In Java, this keyword is *final* [4]. Therefore, any variable or field in Java marked with *final* can be be propagated without any analysis [8]. However, consider the use of the variable a in Figure 4.10. The variable is assigned before the *while* loop and is never reassigned in the loop, so its value does not change during the execution of the loop. Therefore, anywhere the value of the variable a is used, it can be replaced by the constant 5.

CP not only improves code efficiency, but can also produce opportunities for other optimizations. For example, $b = 5 + b + 2$ uses two addition operations, but this can

```
int c = 10;              int c = 10;
int d = 4;               int d = 4;
int e = c + d - 11;      int e = 10 + 4 - 11;      int e = 3;
       (1)                      (2)                    (3)
```

Figure 4.11: Constant propagation can uncover other optimization opportunities

be replaced with $b = b + 7$ which only executes one addition operation. More generally, constant folding replaces expressions consisting only of literal constants with the expression's calculated value.

Dead code elimination is an optimization that removes unnecessary code [1]. In Figure 4.11, CP is applied to the code fragment in column 1 resulting in the code in 2. Then constant folding replaces "10 + 4 - 11" with the value 3. Since the variables c and e no longer used in any statements, the initialization statements can be removed by dead code elimination resulting in the code in column 3.

One way CP can be implemented is by calculating *reaching definitions* using data-flow analysis, which is a technique for gathering information about the values in a program. Reaching definitions describe which definitions of each variable reach each point in a program [1]. A definition is created by an instruction n (*gen[n]*) when a variable is assigned and that same definition is killed (*kill[m]*) at an instruction m that reassigns a variable. This analysis shows the lifetime of a value in a variable and lets us know if an expression can be replaced or not.

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$
$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Figure 4.12: Reaching definitions algorithm

Figure 4.12 shows the algorithm for calculating reaching definitions [3]. In the algorithm, *in*, *out*, *gen* and *kill* are sets. The variables n and p are code instructions. The expression *pred[n]* is a set of statements that precede n during execution. The *in*

and *out* sets are initially set to empty. The *in* and *out* sets in Figure 4.12 are computed for each statement n in the code repeatedly until there are no changes to any of the sets. The definitions that flow into a statement are the union of the reaching definitions of all statements that precede it. These preceding statements also include any definitions coming from branches that can be taken. The reaching definitions coming out of a statement are the reaching definitions coming in to it with the definitions that are killed by the current statement removed and then unioned with the definitions that are generated by the current statement. By doing this, the *out* reaching definition set only contains definitions that reach past the statement. If only one definition of a variable reaches a statement, then that variable can be replaced by that definition because it is impossible for it to have any other value, no matter what path is taken in the code during execution.

In order to calculate reaching definitions, a control flow graph is needed. A control flow graph [10] is a graph where the nodes represent the basic blocks of code. A basic block is a single-entry, single-exit block of continuous code. A method can be broken into basic blocks by separating the code where either a branch is located or the target site of a branch. These basic blocks are joined with arcs that represent the branches that separate them. The reaching definitions algorithm is applied to each basic block as the control flow graph is traversed.

The *gen* and *kill* sets [3] are also needed for the Reaching Algorithm. For each statement in the code *gen* and *kill* sets are calculated. A statement is added to the *gen* set for each assignment statement because it is said to generate the value that is being assigned to the variable at that point in the code. Items would be added to the *kill* set for all other statements in the code that assign to the same variable that was assigned to in the current statement. If a statement does not assign a value, it's *gen* and *kill* sets are empty.

	Gen	**Kill**
1) a = 1;	1	3,5
2) b = 1;	2	4
3) a = a + b;	3	1,5
4) b = a + a;	4	2
5) a = b + b;	5	1,3

Figure 4.13: A *gen* and *kill* set example

Figure 4.13 shows five example code statements and their respective *gen* and *kill* sets. Statement 1 generates a value assigned to variable *a*, which is added to the *gen* set of the statement. Any other definition of *a* is killed, so the definitions at statement 3 and 5 are added to the *kill* set of statement 1. These same rules are used to calculate the *gen* and *kill* sets for the rest of the statements in the example.

Once the reaching definitions information is calculated, constant propagation can be applied. If a statement uses a variable that has a single definition reaching the statement then the variable can be replaced with that definition.

4.3.2 Constant Propagation Implementation

This section covers the implementation of constant propagation (CP) using the CLOT optimization framework and CLEL. In order to implement the reaching definitions described in the previous subsection, a class called GenKill was written to calculate the *gen* and *kill* sets. The GenKill class is part of the opt.Analysis.Set namespace in CLOT. Since the reaching definitions algorithm contains set theoretic operations like union and intersection, a BitSet class was created. This class represents a set as a series of bits. If an item is in the set, the bit is set. If not, the bit is not set. This enables the implementation of union efficiently by applying the binary "or" operation on the bits in the first set with the bits in the second set. Similarly, intersection was implemented by using the binary "and" operation.

```
                    ldloc.0
                    ldloc.1
                    add
c = a + b;          stloc.2
```

Figure 4.14: Example assignment statement decomposed

For calculating the *gen* and *kill* sets, a BitSet is assigned to each local variable in the code being analyzed. This BitSet holds the indexes of the instructions that assign the local variable. The Common Intermediate Language (CIL), which is the instruction set of the .NET Framework, has several instructions that can store a value in a local variable. The types and formats of store CIL instructions were covered in Section 2.3. For example, an assignment statement and the potential CIL is shown in Figure 4.14. Variables *a* and *b* are loaded onto the stack, the top two items on the stack are added and the result is stored in *c*. The code is scanned for store CIL instructions. For each store instruction, if it stores a value into a local variable, the BitSet for this local variable has the index of this CIL store instruction added to its BitSet. For the example in Figure 4.14, the BitSet for the variable *c* will set the bit for the index of that stloc instruction.

The other data structure needed for CP is a control flow graph. A class called ControlFlowGraph is provided in the opt.Analysis.Graph namespace. The nodes in the control flow graph represent the basic blocks from the input code. The arcs in the control flow graph represent branches that connect the basic blocks. A ControlFlowGraphVisitor class is provided to iterate throught the control flow graph and visit each node.

A class called ReachingDefinitions, which is in the opt.Analysis.Set namespace, implements the reaching definitions algorithm described in 4.3.1. The *gen* and *kill* sets are calculated and then the control flow graph is constructed before both are passed to the ReachingDefinitions class. Each instruction in the input code is assigned both an *in* and an *out* set that are initially empty. The reaching definitions algorithm is applied to each

node in the control flow graph until the in and out sets converge to a solution; that is, until there are no changes to the sets.

```
1)  ldc.i4 5        // load constant 5
2)  stloc.0         // assign a
3)  ldc.i4.3        // load constant 3
4)  stloc.1         // assign b
5)  ldc.i4 2        // load constant 2
6)  stloc.2         // assign c
7)  br 6            // branch to conditional
8)  ldloc.0         // load a
9)  ldloc.1         // load b
10) add             // add top two items on stack
11) ldloc.2         // load c
12) add             // add top two items on stack
13) stloc.1         // store result in b
14) ldloc.1         // load b
15) ldc.i4.s 100    // load constant 100
16) blt -14         // if less than branch to top of while loop
```

Figure 4.15: Constant propagation example

Once the reaching definitions information has been calculated, it can now be determined if any constants can be propagated. At each instruction that loads a local variable, it is examined to see if a constant can be propagated to it for this variable. The *in* set for that instruction is examined to identify the definitions of the local variable that reach that instruction. If all the definitions reaching this load instruction store the same constant value then this value can be propagated to the current load instruction.

The CIL code in Figure 4.15 comes from the code fragment in Figure 4.10 shown earlier. It was determined that the value of 5 assigned to the variable a can be propagated to the statement inside the *while* loop because the variable a was never redefined anywhere else after it was assigned. When the ldloc.0 instruction at line 6 is examined, the reaching definitions information indicates that only one definition of a from line 2 reaches line 6 and thus this value at line 2 may be propagated. Recall that CIL is run on a stack-based virtual machine, so the store instruction on line 2 is storing a constant value from the top of the stack into the variable a. Thus, in this example the ldloc.0 instruction at line 6, which

loads the value of a onto the stack, is replaced with an ldc.i4.5 instruction, which loads the constant 5 onto the stack and ultimately into the variable *a* created by line 2.

The value assigned to the variable *b* in line 4 cannot be propagated to the statement at line 10 in the *while* loop because two definitions of the variable *b* reach this statement, one from line 4 and one from the stloc.1 at line 9 in the body of the *while* loop. Since two definitions reach the instruction at line 10 and one is not a constant, no propagation is possible here.

4.4 Method Inlining

4.4.1 Method Inlining Background

The last optimization implemented was Method Inlining. Method inlining replaces a method call instruction with the body of the called method [1]. In assembly code, a method call consists of many "setup" instructions including saving registers and loading parameters into registers. The "setup" code is followed by a jump or branch instruction that transfers control to the called method [10]. This is followed by "teardown" code that handles any return value and restores registers to their state before the method call. This optimization removes the overhead of method "setup" and "teardown," which results in fewer instructions to execute. Notice that while the "setup" and "teardown" code are removed, the instructions of the called method's body are inserted. Thus, generally the code size of the method containing the original call increases. For this reason, smaller methods are typically selected as candidates to inline. Method inlining can also expose opportunities for other optimizations, like CP.

The big question is how to choose which methods to inline. Interprocedual analysis is the process of gathering information about the program to help guide the optimization [47]. This analysis can come in two flavors: static and dynamic. Static program analysis is the process of analyzing the program and how variables and data are used without running the program. A call graph models the calling structure of a program by having methods as nodes and call instructions as arcs. Once the call graph is created, it can be traversed and the smallest methods can be inlined [49].

Some disadvantages of method inlining are code bloat, instruction cache misses and increase pressure on register allocation [10]. The code bloat causes increased memory usage. In order to improve performance a CPU can have an I-cache to store commonly executed instructions. As the code size of a method increases, it may cause I-cache thrashing as commonly executed instructions are evicted from the cache. This causes the CPU to have to refetch the instruction from main memory, which is slower than the I-cache.

During register allocation, it is decided which local variables will be kept in which registers. Keeping a local variable in a register is desired because if they are not kept in a register, they are stored on the stack. Retrieving a variable from the stack is slower than if it had been kept in a register. Therefore, as more methods are inlined, potentially the number of local variables in a method could increase making more and more of the local variables spill onto the stack. Using the number of local variables a method has to decide if it should be inlined or not is another statistic that can guide static analysis and used [11]. Thus the number of local variables within a method can also be used to help determine whether a method should be inlined.

Dynamic program analysis collects statistics about the program as it executes to better inform the optimization process. Several studies have been done using such statistics

when inling methods in Java [21][6]. During several sample executions of the program, counts can be collected for how many times each method was executed. These counts can be used to help inform what methods to inline.

Another problem arises with Object-Oriented Programming (OOP) languages like Java and .Net languages like C# and Visual Basic .NET. OOP languages use permission keywords (e.g., private) to hide methods and fields from the outside. If a method in one class tries to inline a method in another class, there could be access problems. The callee could access private methods or fields in its class that will not be accessible in the caller's class. To get around this, the entire object can be inlined. All of the callee's methods are inlined and a copy of any fields in the callee are copied into the caller. Wimmer and Mossenbock investigated object inlining within the Java virtual machine [52]. Since the implementation inlined the methods at runtime, it could select the methods to inline based on I-cache performance. The rest of this section will cover how method inlining was implemented using the CLOT optimization framework and CLEL.

4.4.2 Method Inlining Implementation

The Common Intermediate Language (CIL) has four instructions to invoke a method: call, calli, callvirt and jmp (see Section 2.3). If the method being invoked in CIL is an instance method, the *this* pointer is pushed onto the stack first. The parameters are then pushed onto the stack. The next instruction is the method call. If there is a return value, it will be on the stack after the method invocation.

1) Not a constructor
2) Caller and callee must be part of the same class
3) Not a method in an interface
4) Not a polymorphic method
5) Fits under the code growth restriction

Figure 4.16: Rules for method inlining

The first step in implementating method inlining is the choice of an algorithm that selects the methods to inline. For this thesis, a greedy algorithm was chosen. A call graph is created from the input code where each method is a node in the call graph. Directed arcs are added to the call graph between two nodes to represent when a method is called. The call graph nodes are traversed and if a node passes the five rules shown in Figure 4.16, it is inlined. First, the method cannot be a constructor. The reason for not inlining constructors is that CLI virtual machines do not handle constructors the same way as other methods. If a constructor is inlined, the object might not be created properly.

Second, the caller and the callee must be part of the same class. This restriction avoids the access problems mentioned earlier in Section 4.4.1. This restriction also implies that if the caller and callee are in different assemblies, they will also not be a candidate for inlining. Third, the method to inline cannot be part of an interface because interface methods have no implementation. To fulfill this restriction, only methods whose code size is greater than zero are considered for inlining.

The fourth rule is that polymorphic method calls are not inlined. Polymorphic method calls are dynamically dispatched based on the type of the variable calling the method. Since inlining is done prior to execution, inlining a polymorphic method could create the situation where the wrong method was inlined.

The method inlining optimization uses a configuration file. This file lets the user set the percentage of code growth to allow before the method inlining algorithm stops. The fifth and final rule is the method to inline must be the smallest method found. If the method to inline passes all of these restrictions, it is inlined. If a method is not found that fits all of these restrictions, the method inlining optimization is completed. Additional code analysis could be performed to relax these restrictions, but this is our starting point.

To promote reuse, CLOT separates the decision of whether to inline or not from the class that does the actual inlining. The inlining is done by the MethodInline class in the opt.Tools.Methods namespace. The MethodInline class contains a method called Inline. It is passed the caller and callee's MethodDescriptors, which represent the two methods, the caller and callee code and the index of the call instruction in the caller to inline. By abstracting out the method inlining to a separate class, it can be reused by any optimization.

A few details regarding the new inlined code are in order. Normally, a called method's formal parameters are an alias to the callee's arguments. The Inline method removes the aliasing of the callee's parameters. For instance, a local variable in the caller may be passed into the callee's method argument. Instructions in the callee using this parameter must be updated to use the variable passed in by the caller. New local variables are also created in the caller to hold any local variables formally created in the callee. New local variables are created via CLEL by adding a new ClassDescriptor to the List in the MethodLocalsBlobInfo returned by the get_Locals method on the MethodDescriptor class (see Section 3.1). Return instructions in the callee are also converted to unconditional branches to the end of the newly inlined section of code.

Also, the instructions that push the parameters and call the method are removed from the caller. The body of the callee is inserted in place of these instructions. If there is a return value, a store instruction is inserted after the newly inlined code. After modifying the caller's code and local variables, the SetMethodsCode method on the CLELAssembly class (see Section 3.1) is used to save the changes. This repairs any offsets or fields necessary to make the method header in the assembly valid. It will also update the blob stream, which holds the encoded count and types for the local variables of all methods in the assembly.

Chapter 5

Optimization Tests

The Common Language Infrastructure for Research (CLIR) is a layered toolset, thus a stepwise evaluation was performed. The bottom layer of CLIR, the CLEL, was evaluated by the successful and useful implementation of tools that use CLEL. Since the CLOT layer of CLIR uses the CLEL, if CLOT evaluates as successful and useful then CLEL, by extension, must be successful and useful.

CLOT provides the capability to analyze and optimize CIL code. Indeed, three optimizations were implemented using the framework provided by CLOT. The ability to build these three optimizations partially affirms the success and usefulness of CLOT. However, this affirmation gains strength if the optimized code is measurably improved. Indeed, if the optimizations improve CIL program performance then it shows that research into CIL code optimizations is worthwhile and since the CLIR enables such research then CLIR itself is validated. The remainder of this chapter evaluates the effectiveness of the optimizations implemented in CLOT and begins with a description of the testing environment.

5.1 Introduction and Testing Environment

In order to determine the effectiveness of the optimizations described in the previous chapter, the results of applying the optimizations to three programs are discussed. These three programs are The Game of Life [18][16] based on John Conway's rules for how cellular automata replicates, Huffman Coding [12][38], which implements David A. Huffman's algorithm for lossless data compression, and ZipFolder [23], which compresses all the files in a folder using the zip algorithm. The Game of Life and Huffman Coding are both written in Visual Basic .NET and ZipFolder is written in C#. See Table 5.1 for the relative sizes of each program.

Program	Number of Classes	Number of Methods	Lines of CIL Code
Game of Life	1	7	158
Huffman Coding	3	15	401
ZipFolder	6	50	1768

Table 5.1: Relative program sizes

The input for each of the test programs was fixed to make sure that each run of the program was exactly the same. The input was also chosen to be of a sufficient size to cause the execution time to be long enough so that changes in the execution time caused by the optimizations studied here would be more noticeable. The Game of Life was run with the same starting generation on a 50 by 50 grid and ran through 100 generations. The Huffman Coding program was run with the first volume of Edgar Allan Poe's complete works as input. Every time the ZipFolder program was run, it was run on a folder of source code and executables, containing among other things a complete F# compiler.

The tests in this chapter were executed on a computer running Slackware Linux 12.0 with a 566 megahertz Celeron CPU that has 128 kilobytes of cache and 256 megabytes

of main memory. A Perl script was written to run each program 100 times and find the average execution time. The Linux *time* command was used in the Perl script to obtain execution times. Version 1.2.6 of Mono, which is an open source implementation of the .NET Framework, was used in these tests. Since machine code, not the Common Intermediate Language (CIL), is what is actually run on the CPU, Mono's Ahead Of Time (AOT) compilation feature was used to translate the assembly into a native image. Then *objdump* is used to translate the native image into human-readable assembly. This process will provide a better idea of how the optimizations actually affected the runtime of each program. All Mono optimizations were turned off at every stage of the tests in this chapter. The following three sections discuss how each of the three optimizations implemented for CLOT affected performance.

5.2 One Pass Branch Instruction Replacement

Recall from Section 4.2 that three versions of BIR were written for this thesis. However, the tests were only run on the best version that makes a single pass through the code. Also recall the CIL has two types of branches: long and short. The long branch type has a one byte operation code and a four byte offset. In contrast, the short branch type has a one byte operation code, but only one byte for the branch offset. Replacing a five byte instruction with a two byte instruction reduces the code size by three bytes for each replacement. This optimization was described as an example of how to compact CIL code in [26], but no tests or results were given. This section sheds light on the effects of this optimization.

Branches are generated in CIL by many common higher level language constructs including loops, if statements, and try-catch exception blocks. The Mono C# and VB .NET compilers generally generate long branches in CIL unless optimizations are turned on

in the compiler. Optimizations in Mono can be enabled in two places, at compile time and at runtime. When the -O flag in C# or the /optimize flag in VB .NET is not explicitly specified, a simple set of default optimizations implicitly are performed that provide a balance between compile and runtime and the benefits gained by the optimizations [40]. However, BIR is not one of these simple optimizations. Therefore in order to make sure that short branches are generated whenever possible, the optimization flag must be used explicitly when compiling the code.

Branch	**Number of Long Branches**	**Number Replaced**	**Percent replacement**
br	29	27	93%
brtrue	9	6	67%
brfalse	21	21	100%
leave	2	2	100%
TOTALS	**61**	**56**	**92%**

Table 5.2: Branches replaced for Game of Life

Branch	**Number of Long Branches**	**Number Replaced**	**Percent Replaced**
br	47	45	96%
brtrue	19	19	100%
brfalse	27	24	89%
TOTALS	**93**	**88**	**95%**

Table 5.3: Branches replaced for Huffman

Tables 5.2 through 5.4 show the number of each type of long branches in the three test programs and how many of each was replaced with a smaller, equivalent branch during BIR. A vast majority of all of the long branches were replaced in all three programs. Table 5.5 shows the changes in code size after BIR was run on each of these test programs. The largest difference was ZipFolder with 561 bytes less code. Since ZipFolder has more lines of higher level code, more branches were available for possible replacement. This implies that the larger the program, the more the code size can be decreased by performing BIR.

Branch	Number of Long Branches	Number Replaced	Percent Replaced
br	48	43	90%
brtrue	24	22	92%
brfalse	54	53	98%
beq	14	13	93%
bge	5	5	100%
bgt	3	3	100%
ble	4	4	100%
blt	14	13	93%
bneun	11	10	91%
bgtun	1	1	100%
bltun	1	1	100%
leave	19	19	100%
TOTALS	**198**	**187**	**94%**

Table 5.4: Branches replaced for ZipFolder

Program	Code Size Before	Code Size After	Difference
Game of Life	1474	1306	-168 (-11.40%)
Huffman	3022	2758	-264 (-8.74%)
ZipFolder	8688	8127	-561 (-6.46%)

Table 5.5: Code size changes (in bytes)

Program	Without BIR	With BIR	Difference
Game of Life	6.3684	6.3901	+0.0217 (+0.34%)
Huffman	10.3626	10.3201	-0.0425 (-0.41%)
ZipFolder	39.2829	39.1041	-0.1788 (-0.46%)

Table 5.6: Average execution time (in seconds) for BIR

Execution time remained roughly the same, within a small margin of error, after BIR was applied as seen in Table 5.6. However, the advantage of BIR is that the intermediate code size is reduced. This small savings relative to total program size might not make much difference on a desktop or laptop computer, but on an embedded computer, where resources are limited, the savings could help keep more of the code in the CPU's cache and improve performance [25].

5.3 Constant Propagation

Recall, constant propagation replaces instructions that load values from memory with instructions that load constant values. This replacement is allowed if the value being loaded from memory can be proven to always be the constant value. This can decrease execution time because it removes costly memory accesses [28][51][8].

```
public CRC32() {
        UInt32 dwPolynomial = 0xEDB88320;
        for (i=0; i < 256; i++) {
                for (j=8; j > 0; j− −) {
                        dwCrc = (dwCrc >> 1) ^ dwPolynomial;
                }
        }
}
```

Table 5.7: ZipFile C# code fragment

Original CIL	**Optimized CIL**	
ldc.i4 -306674912	ldc.i4 -306674912	//initialize dwPolynomial
stloc.0	stloc.0	
ldloc.3	ldloc.3	// load dwCrc
ldc.i4.1	ldc.i4.1	// load 1
shr.un	shr.un	// shift right
ldloc.0	ldc.i4 -306674912	// OPTIMIZED LINE!
xor	xor	
stloc.3	stloc.3	// store into dwCrc

Table 5.8: ZipFile CIL

Assembly of Original CIL	**Assembly of Optimized CIL**
mov -0x1c(%ebp),%ecx	xor $0xebd88320,%eax
xor %ecx,%eax	

Table 5.9: ZipFile Assembly

Program	Number of Constants Propagated	Lines of CIL Code
Game of Life	24	158
Huffman	41	401
ZipFolder	14	1768

Table 5.10: Number of Constants Propagated

Program	Without CP	With CP	Difference	
Game of Life	6.3686	6.2835	-0.0851	(-1.34%)
Huffman	10.3626	9.8489	-0.5137	(-4.96%)
ZipFolder	39.2829	37.4725	-1.8104	(-4.61%)

Table 5.11: Average execution time (in seconds)

Tables 5.7 through 5.9 illustrate the code transformations of applying CP. Table 5.7 shows a code fragment of C# .NET from the ZipFile program. Notice that the *dwPolynomial* local variable is assigned a constant value that is not changed. The variable is then used in the body of the loop statement. Because the programmer uses a variable, the value is stored into memory and repeatedly loaded from memory. This can be seen in the low-level CIL code on the left side of Table 5.8. In particular, the *dwPolynomial* variable is local variable 0 and this variable's value is stored by the *stloc.0* instruction (the second CIL instruction). Several instructions below is the *ldloc.0* instruction that loads the value for this variable. Notice that the CP optimization replaces this *ldloc.0* instruction with a *ldc.i4* instruction to load the constant value. Table 5.9 follows this change even further into the native machine assembly code. Here, the *mov* instruction that access memory is eliminated and the constant value has been propagated successfully into the computation (the *xor* instruction).

Table 5.10 shows the number of constants propagated by CP, and Table 5.11 shows the average execution times for the test programs measured both before and after CP was applied. The benefits of optimizing with CP are impressive, resulting in execution decreases of nearly 5% in some cases. But several other observations catch our interest. First, there

were far fewer opportunities for CP in the ZipFile program, which was the largest program. Second, in spite of this, optimizing ZipFile still resulted in an impressive decrease of 4.61% in execution time. Lastly, there was a significant difference between the gains showed by the Huffman and ZipFile programs compared to the Game of Life program.

```
Private Function FindProbabilitiesForSymbols(ByVal Data() AsByte) As Long()
        Dim I As Integer

        For I = 0 To 255
                B(I) = 0
        Next
End Function
```

Table 5.12: Huffman Coding VB.NET code fragment

Original CIL	**Optimized CIL**	
ldc.i4.0	ldc.i4.0	
stloc.s 5	stloc.s 5	// initialize zero
ldloc.s 5	ldc.i4.0	// changed: load constant 0
stloc.2	stloc.2	// I = 0
ldc.i4 255	ldc.i4 255	// initialize stop
stloc.3	stloc.3	
ldc.i4.1	ldc.i4.1	// initialize step
stloc.4	stloc.4	
ldloc.2	ldloc.2	
ldloc.s 4	ldc.i4.1	// changed: load constant 1
add	add	// add step to I
stloc.2	stloc.2	// store into I
ldloc.2	ldloc.2	// load I
ldloc.3	ld.i4 255	// changed: load constant 255
cgt	cgt	// test for branch

Table 5.13: FindProbabilitiesForSymbols CIL

Investigating the actual code produced, the last two observations stem from the same underlying issue. If constants are propagated into code that is executed many times, for example into loops, then a single CP application can yield noticeable execution decreases. Indeed, the ZipFile code in Table 5.7 optimized a statement inside two loops. This statement

is executed 2,048 times each time a CRC32 object is created. Conversely, fewer of the CP opportunities in the Game of Life were in loops.

As to the first observation, ZipFolder was a C# program compiled by the Mono C# .NET compiler whereas the other two programs were Visual Basic programs compiled by the Mono VB.NET compiler. Investigating the CIL generated by these two compilers revealed that the VB.NET compiler creates more inefficient code that contains more opportunities for CP. Table 5.12 shows a Visual Basic code fragment from the Huffman program and Table 5.13 shows the resulting CIL generated. The *for* loop shown in Table 5.12 uses the local variable "I" for the loop index. The loop upper bound is a constant, but the Mono compiler generates a temporary local variable for this upper bound and uses this local variable each time through the loop. In addition, the loop index is increased by 1 each time through the loop; this is called the step value. The Mono VB.NET compiler also uses a temporary local variable for this step value. Neither of these two temporary variables are changed after their initial assignment and thus are constants that can be propagated. In the CIL in Table 5.13, local variable 2 is used for the programmer variable "I", local variable 3 is used for the upper bound value 255, and local variable 4 is used for the step value 1. In particular, notice the following instruction sequence: *ldc.i4 255* and *stloc.3*. These two instructions store the value 255 into the temporary local variable. Then, the second to last statement *ldloc.3* loads this value from memory to be used in the loop comparison (the *cgt* instruction).

5.4 Method Inlining

Recall that method inlining substitutes the method setup code (e.g, pushing parameters onto stack) and method invocation code (including saving registers) at a call site with the

body of the called method. The key to method inlining is the selection of those methods to inline. Many algorithms have been proposed [2][6][44], but, in order to demonstrate method inlining, this thesis uses a simple, greedy algorithm.

As methods are inlined at the call sites of a method (the "caller"), the caller's code grows. Our algorithm continues inlining methods at call sites until this code growth exceeds a certain threshold value that is a percentage of the original file size. This percentage is specified in an external configuration file, thus allowing researchers to quickly rerun a test with a different threshold value. As each call site is encountered, the method called will be inlined if the five conditions for inlining detailed in Section 4.4.2 are satisfied.

Program	**Orig**	**10%**	**Diff at 10%**	**20%**	**Diff at 20%**
Game of Life	6.3684	6.3687	+0.0003 (+0.00%)	6.3119	-0.0565 (-0.89%)
Huffman	10.3626	10.0829	-0.2797 (-2.70%)	10.152	-0.2106 (-2.03%)
ZipFolder	39.2829	39.2099	-0.0730 (-0.19%)	39.0962	-0.1867 (-0.48%)

Table 5.14: Average execution time (in seconds) for Method Inlining at 10% and 20%

Program	**30%**	**Diff at 30%**	**40%**	**Diff at 40%**	**50%**	**Diff at 50%**
Game of Life	6.3353	-0.0331 (-0.52%)	6.3016	-0.0668 (-1.05%)	6.2947	-0.0737 (-1.16%)
Huffman	9.8874	-0.4752 (-4.59%)	8.8128	-1.5498 (-14.96%)	9.7068	-0.6558 (-6.33%)
ZipFolder	38.3509	-0.932 (-2.37%)	37.5546	-1.7283 (-4.40%)	39.3491	+0.0662 (+0.17)

Table 5.15: Average execution time (in seconds) for Method Inlining at 30%, 40% and 50%

The performance increase gained by method inlining is offset partially by an increase in code size, an increase in cache misses, and increased register pressure [9]. Thus, some inlining can decrease execution time but too much inlining can result in increased execution time. Tables 5.14 and 5.15 show the results of performing method inlining on the three test programs at varying threshold levels. It can be seen that a threshold of about 40% achieved impressive decreases in execution time. The Game of Life program did not appear

to benefit much from method inlining. Further investigation revealed that there are only seven methods in the program which decreased the chances to apply the optimization. In addition, the methods were already experiencing some register pressure issues that became more exacerbated by inlining. Indeed, a better thresholding mechanism may be based on register pressure rather than code size.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The Common Language Infrastructure for Research (CLIR) provides a three layer toolset for programming language research of the .NET environment. The CLIR layered architecture is *open* in that users can access functionality in all layers. The bottom layer of CLIR is the Common Language Engineering Library (CLEL). CLEL supplies the foundational capabilities of reading and writing .NET assembly files. The second layer, the Common Language Optimizing Toolset (CLOT), sits on top of CLEL and supplies a suite of optimization tools. CLOT provides three optimizations as well as the data structures and analysis algorithms needed by the optimizations. The success of the CLOT layer validates the usefulness and correctness of the CLEL layer. The top layer of CLIR contains standalone applications useful for researchers that employ lower layers of CLIR. The *DumpCode* utility directly accesses the CLEL layer to create a human readable form of a .NET assembly. The *Optimization Scheduling Tool (OST)* utility provides a graphical interface that allows easy selection and ordering of optimizations to perform on a .NET assembly file.

As mentioned in Section 2.4, little research was found regarding optimizing .NET assembly code. Whereas, a wide variety of research for optimizing higher-level languages is readily available. This thesis sheds some light on the potential gains of optimizing low-level .NET assembly code. CLOT was used to implement three optimizations, each of which was run on a test suite of three programs of various size. The one pass Branch Instruction Replacement (BIR) optimization uses a peephole-based approach to replace an expensive operation with a cheaper, equivalent one. BIR had little effect on execution time for all three test programs. However, BIR did achieve a modest memory savings between 6.46% and 11.40% for the test programs because the replacement instructions were smaller. This could positively impact programs that are executed across a network by decreasing the amount of code transferred.

The second optimization was constant propagation (CP). CP is a classic intraprocedural optimization requiring commonly-used optimizing data structures, such as a control flow graph, and useful algorithms including reaching definitions. Many opportunities were found for CP and significant decreases in program execution time resulted. In addition, it was discovered that the Mono VB .NET compiler generates extra local variables in the CIL that are not generated by the C# compiler. The final optimization was method inlining, which is an interprocedural technique. This thesis implements a greedy algorithm that replaces the smallest methods until the code size exceeds a threshold value. Varying threshold values between 10% to 50% were explored. Generally, 40% achieved the best decreases in program execution time.

The successful implementation of the three optimizations serves as a validation of the CLOT component of CLIR. The CLOT framework allows other researchers to use the tools currently implemented or extend these tools in new ways or build new tools entirely.

The user applications of the top CLIR layer were deployed to perform the optimization experiments. The optimizations were found to have beneficial impacts and thus validates the CLIR concept generally.

6.2 Future Work

There are several distinct directions of future work possible. One direction involves continued optimization experimentation. For example, other method selection algorithms could be investigated for method inlining. In addition, other thresholding calculations could be attempted including an estimate of potential register pressure. Also, CLOT could be expanded with other well-known traditional optimizations such as common subexpression elimination and dead code elimination. Another direction is the expansion of the CLEL layer to improve and/or provide access to all the metadata tables and assembly fields. For example, CLEL does not currently allow for adding a class to an assembly.

Another direction of future work consists of a more rigorous experimentation of optimizing .NET CIL code. This thesis has shown that optimizing CIL code can be beneficial. However, a determination of which optimizations and which orderings of optimization application was not studied. In addition, deficiencies in high-level .NET compilers (e.g., VB and C#) were uncovered. There may be further improvements to those tools that may be discovered. Lastly, this thesis concentrated on CIL code generated by the open-source Mono .NET compiler. Future efforts could experiment with the Microsoft Visual Studio .NET compiler.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, & Tools*. Pearson Education, Boston, MA, second edition, 2007.

[2] Aigner, G. and Holzle, U. Eliminating virtual function calls in c++ programs. *ECOOP '96 Conference Proceedings*, 1996.

[3] Appel, A. W. *Modern Compiler Implementation in Java*. Press Syndicate of the University of Cambridge, Cambridge, England, first edition, 1999.

[4] Arnold, K., Gosling, J., and Holmes,D. *The Java Programming Language*. Addison Wesley, Boston, MA, fourth edition, 2006.

[5] Besson, F., de Grenier de Latour, T., and Jensen, T. Secure calling contexts for stack inspection. *PPDP '02 Proceedings of the 4th ACM SIGPLAN international conference on principles and practice of declarative programming*, Pages 26 - 87, 2002.

[6] Bradel, B. J. and Abdelrahman, T. S. The use of traces for inlining in java programs. *LCPC '04 Proceedings of the 17th international conderence on Languages and Compilers for High Performace Computing*, 2004.

[7] Carlisle, M. C., Sward, R. E., and Humphries, J. W. Weaving ada 95 into the .net environment. *SIGAda '02 Proceedings of the 2002 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*, Pages 22 - 26, 2002.

[8] Caudill, S. and Machkasova, E. Empirical studies of java optimizations. In *Midwest Instruction and Computing Symposium*, 2005.

[9] Chakrabarti, D. R. and Liu, S. Inline analysis: Beyond selection heuristics. In *CGO '06 Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[10] Cooper, K. and Torczon, L. *Engineering a compiler*. Morgan Kaufmann Publishers, San Francisco, CA, first edition, 2003.

[11] Cooper, K. D., Wall, M. W., and Torczon, L. Unexpected side effects of inline substitution: a case study. *ACM Letters on Programming Language and Systems*, Volume 1 Number 1, 1992.

[12] Cormen, T. H., Leiserson, C. E., Rivest, R. R., and Stein, C. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, third edition, 2009.

[13] Dearle, A. Software deployment, past, present and future. *FOSE '07 2007 Future of Software Engineering*, 2007.

[14] Dowd, T., Henderson, F., and Ross, P. Compiling mercury to the .net common language runtime. *Electronic Notes in Theoretical Computer Science*, 59 No. 1, 2001.

[15] Fowler, M. Inversion of control and the dependency injection pattern. World Wide Web electronic publication, http://www.martinfowler.com/articles/injection.html, 2004.

[16] Foxall, J. D. *Sams Teach Yourself Visual Basic 2010 in 24 Hours*. Sams Publishing, Carmel, IN, first edition, 2010.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, first edition, 1994.

[18] Gardner, M. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American 223: 120 - 123*, October 1970.

[19] Ghosh, M., Kumar, R., and Chakrabarti, P. P. Fsm matchers: A post compilation optimization technique for extensible architectures. *International Conference on High Performance Computing (HiPC 2004)*, Workshop on New Horizons in Compiler Analysis and Optimizations, 2004.

[20] Hanson, D. R. Lcc.net: Targeting the .net common intermediate language from standard c. World Wide Web electronic publication, http://research.microsoft.com/pubs/69973/tr-2002-112.pdf, 2003.

[21] Hauble, C., Wimmer, C., and Mossenbock, H. Evaluation of trace inlining heuristics for java. *SAC '12 Proceeding of the 27th Annual ACM Symposium on Applied Computing*, 2012.

[22] Hejlsberg, A., Torgersen, M., Wiltamuth, S., and Golde, P. *The C# Programming Language*. Addison Wesley, Boston, MA, fourth edition, 2010.

[23] Huggins, B. Zipfolder project homepage. World Wide Web electronic publication, http://zipfolder.codeplex.com, 2007.

[24] Intel. Intel architecture software developer's manual volume 2: Instruction set reference. World Wide Web electronic publication, http://www.intel.com/design/intarch/manuals/243191.htm, 1999.

[25] Johnson, N. E. Code size optimization for embedded processors. Master's thesis, University of Cambridge, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-607.pdf, November 2004.

[26] Lidin, S. *Expert .NET 2.0 IL Assembler*. Apress, Berkeley, CA, first edition, 2006.

[27] Meijer, E. and Gough, J. Technical overview of the common language runtime. World Wide Web electronic publication, http://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf, 2000.

[28] Metzger, R. and Stroud, S. Interprocedural constant propagation: an empirical study. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2 Issue 1-4, March - Dec. 1993, 1993.

[29] Microsoft Corporation. String structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms646987.aspx.

[30] Microsoft Corporation. Stringfileinfo structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms646989(VS.85).aspx.

[31] Microsoft Corporation. Stringtable structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms646992.aspx.

[32] Microsoft Corporation. Var structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms646994.aspx.

[33] Microsoft Corporation. Varfileinfo structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms646995.aspx.

[34] Microsoft Corporation. Vs_fixedfileinfo structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms646997.aspx.

[35] Microsoft Corporation. Vs_versioninfo structure. World Wide Web electronic publication, http://msdn.microsoft.com/en-us/library/ms647001.aspx.

[36] Microsoft Corporation. Microsoft portable executable and common object file format specification. World Wide Web electronic publication, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, 2008.

[37] Miller, J. S. and Ragsdale, S. *The Common Language Infrastructure Annotated Standard*. Addison Wesley, Boston, MA, first edition, 2004.

[38] MKA Software. Adding huffman coding to your vb .net application. World Wide Web electronic publication, http://mka-soft.com/index.php/easy-coding/47-add-huffman, 2011.

[39] Mono Project. Mono project's history. World Wide Web electronic publication, http://www.mono-project.com/History, 2008.

[40] Mono Project. Aot. World Wide Web electronic publication, http://www.mono-project.com/AOT, 2011.

[41] Nystrom, N. J. Bytecode-level analysis and optimization of java classes. Master's thesis, Purdue University, ftp://ftp.cs.purdue/pub/hosking/papers/nystrom.pdf, August 1998.

[42] Pressman, R. *Software Engineering - A Practitioner's Approach (6th ed.)*. McGraw Hill, New York, NY, sixth edition, 2004.

[43] Richter, J. *CLR via C#*. Microsoft Press, Redmond, WA, second edition, 2006.

[44] Sewe, A., Jochem, J., and Mezini, M. Next in line, please!: Exploiting the indrect benefits of inlining by accurately predicting further inlining. *SPLASH '11 Workshops Proceedings of the compilation of the co-located workshops on DSM '11, TMC '11, AGERE! '11 AOOPES '11, NEAT '11, and VMIL '11*, 2011.

[45] Software Developer's Journal. Browsing through headers - an introduction to reverse engineering. World Wide Web electronic publication, http://en.sdjournal.org/products/articleInfo/28, 2006.

[46] Spinellis, D. Declarative peephole optimization using string pattern matching. *ACM SIGPLAN LAN Notes*, 34(2):47-51, February 1999.

[47] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Fransico, CA, first edition, 1997.

[48] Stutz, D., Neward, T., and Shilling, G. *Shared Source CLI Essentials*. O'Reilly, Sebastopol, CA, first edition, 2003.

[49] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 27 Issue 4, July 2005, 2005.

[50] Thai,T. and Lam, H. Q. *.NET Framework Essentials*. O'Reilly, Sebastopol, CA, third edition, 2003.

[51] Verbrugge, C., Co, P., and Hendren, L. Generalized constant propagation a study in c. In *In Proceedings of the 1996 International Conference on Compiler Construction*, 1996.

[52] Wimmer, C. and Mossenbock, H. Automatic feedback-directed object inlining in the java hotspot virtual machine. *VEE '07 Proceedings of the 3rd international conference on Virtual execution environments*, 2007.

[53] Yu, D., Kennedy, A., and Syme, D. Formalization of generics for the .net common language runtime. *POPL '04 Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles on programming languages*, Pages 39 - 51, 2004.

[54] Zerzelidis, A. and Wellings, A. J. Requirements for a real-time .net framework. *ACM SIGPLAN Notices*, 40 Issue 2, 2005.

Appendix A

Bubble Sort in C#

For this thesis Appendix A through C will take an example program and decompose the assembly file format. What follows is a program written in C# using the bubble sorting [12] algorithm to sort a list of integers. Appendix B contains a listing of the contents of the assembly in hexadecimal and Appendix C covers the various sections, tables and fields that are represented by the assembly in Appendix B.

```
using System;
using System.Collections;

public class BubbleSort
{
   private ArrayList nums;

   public BubbleSort()
   {
      nums = new ArrayList();
      nums.Add(91);
      nums.Add(5);
      nums.Add(101);
      nums.Add(3);
      nums.Add(58);
      nums.Add(-14);
      nums.Add(199);
      nums.Add(44);
      nums.Add(1);
      nums.Add(678);
      doBubbleSort();
      int i;
```

```
      for(i = 0;i < nums.Count;i++)
         Console.WriteLine(nums[i]);
   }

   private void doBubbleSort()
   {
      int i,j,k;
      for(k = 0;k < nums.Count;k++)
         for(i = 0;i < nums.Count;i++)
            for(j = 0;j < nums.Count;j++)
               if((int)nums[i] < (int)nums[j])
                  swap(i,j);
   }

   public void swap(int first,int second)
   {
      int temp = (int)nums[first];
      nums[first] = nums[second];
      nums[second] = temp;
   }

   public static void Main()
   {
      new BubbleSort();
   }
}
```

Appendix B

BubbleSort.exe hexadecimal dump

What follows is the output after running "BubbleSort.exe" through the Linux *xxd* program. The first column is the offset from the beginning of the file. The middle columns contain the bytes of the file in hexadecimal. The last column displays the file bytes in ASCII.

```
0000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
0000010: b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 0000 0000 0000 0000 0000 8000 0000  ................
0000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ........!..L.!Th
0000050: 6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
0000060: 7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS
0000070: 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$.......
0000080: 5045 0000 4c01 0300 8121 f648 0000 0000  PE..L....!.H....
0000090: 0000 0000 e000 0e01 0b01 0600 0006 0000  ................
00000a0: 0004 0000 0000 0000 0020 0000 0020 0000  ......... ... ..
00000b0: 0040 0000 0000 4000 0020 0000 0002 0000  .@....@.. ......
00000c0: 0400 0000 0000 0000 0400 0000 0000 0000  ................
00000d0: 0080 0000 0002 0000 0000 0000 0300 0000  ................
00000e0: 0000 1000 0010 0000 0000 1000 0010 0000  ................
00000f0: 0000 0000 1000 0000 0000 0000 0000 0000  ................
0000100: 1820 0000 4f00 0000 0040 0000 e402 0000  . ..O....@......
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0060 0000 0c00 0000 0000 0000 0000 0000  .`..............
0000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000150: 0000 0000 0000 0000 1020 0000 0800 0000  ......... ......
0000160: 0000 0000 0000 0000 6420 0000 4800 0000  ........d ..H...
0000170: 0000 0000 0000 0000 2e74 6578 7400 0000  .........text...
0000180: 9005 0000 0020 0000 0006 0000 0002 0000  ..... ..........
0000190: 0000 0000 0000 0000 0000 0000 2000 0060  ............ ..`
```

```
00001a0: 2e72 7372 6300 0000 e402 0000 0040 0000  .rsrc........@..
00001b0: 0004 0000 0008 0000 0000 0000 0000 0000  ................
00001c0: 0000 0000 4000 0040 2e72 656c 6f63 0000  ....@..@.reloc..
00001d0: 0c00 0000 0060 0000 0002 0000 000c 0000  .....`..........
00001e0: 0000 0000 0000 0000 0000 0000 4000 0042  ............@..B
00001f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000200: ff25 1020 4000 0000 0000 0000 0000 0000  .%. @...........
0000210: 4020 0000 0000 0000 5a20 0000 0000 0000  @ ......Z ......
0000220: 0000 0000 4e20 0000 1020 0000 0000 0000  ....N ... ......
0000230: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000240: 0000 5f43 6f72 4578 654d 6169 6e00 6d73  .._CorExeMain.ms
0000250: 636f 7265 652e 646c 6c00 4020 0000 0000  coree.dll.@ ....
0000260: 0000 0000 4800 0000 0200 0000 e822 0000  ....H........"..
0000270: a802 0000 0100 0000 0400 0006 e822 0000  ............."..
0000280: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000290: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00002e0: 0000 0000 0000 0000 0000 0000 1330 1f00  .............0..
00002f0: 0601 0000 0100 0011 0228 0100 000a 0273  .........(.....s
0000300: 0200 000a 7d01 0000 0402 7b01 0000 041f  ....}.....{.....
0000310: 5b8c 0300 0001 6f03 0000 0a26 027b 0100  [.....o....&.{..
0000320: 0004 1b8c 0300 0001 6f03 0000 0a26 027b  ........o....&.{
0000330: 0100 0004 1f65 8c03 0000 016f 0300 000a  .....e.....o....
0000340: 2602 7b01 0000 0419 8c03 0000 016f 0300  &.{..........o..
0000350: 000a 2602 7b01 0000 041f 3a8c 0300 0001  ..&.{.....:.....
0000360: 6f03 0000 0a26 027b 0100 0004 1ff2 8c03  o....&.{........
0000370: 0000 016f 0300 000a 2602 7b01 0000 0420  ...o....&.{.... 
0000380: c700 0000 8c03 0000 016f 0300 000a 2602  .........o....&.
0000390: 7b01 0000 041f 2c8c 0300 0001 6f03 0000  {.....,.....o...
00003a0: 0a26 027b 0100 0004 178c 0300 0001 6f03  .&.{..........o.
00003b0: 0000 0a26 027b 0100 0004 20a6 0200 008c  ...&.{.... .....
00003c0: 0300 0001 6f03 0000 0a26 0228 0200 0006  ....o....&.(....
00003d0: 160a 3815 0000 0002 7b01 0000 0406 6f04  ..8.....{.....o.
00003e0: 0000 0a28 0500 000a 0617 580a 0602 7b01  ...(......X...{.
00003f0: 0000 046f 0600 000a 3fda ffff ff2a 0000  ...o....?....*..
0000400: 1330 0e00 8600 0000 0200 0011 160c 386d  .0............8m
0000410: 0000 0016 0a38 5100 0000 160b 3835 0000  .....8Q.....85..
0000420: 0002 7b01 0000 0406 6f04 0000 0a79 0300  ..{.....o....y..
0000430: 0001 4a02 7b01 0000 0407 6f04 0000 0a79  ..J.{.....o....y
0000440: 0300 0001 4a3c 0800 0000 0206 0728 0300  ....J<.......(..
0000450: 0006 0717 580b 0702 7b01 0000 046f 0600  ....X...{....o..
0000460: 000a 3fba ffff ff06 1758 0a06 027b 0100  ..?......X...{..
0000470: 0004 6f06 0000 0a3f 9eff ffff 0817 580c  ..o....?......X.
0000480: 0802 7b01 0000 046f 0600 000a 3f82 ffff  ..{....o....?...
0000490: ff2a 0000 1330 0a00 3e00 0000 0300 0011  .*...0..>.......
```

```
00004a0: 027b 0100 0004 036f 0400 000a 7903 0000  .{.....o....y...
00004b0: 014a 0a02 7b01 0000 0403 027b 0100 0004  .J..{......{....
00004c0: 046f 0400 000a 6f07 0000 0a02 7b01 0000  .o....o.....{...
00004d0: 0404 068c 0300 0001 6f07 0000 0a2a 0000  ........o....*..
00004e0: 1e73 0100 0006 262a 4253 4a42 0100 0100  .s....&*BSJB....
00004f0: 0000 0000 0c00 0000 7631 2e31 2e34 3332  ........v1.1.432
0000500: 3200 0000 0000 0500 7000 0000 2401 0000  2.......p...$...
0000510: 237e 0000 9401 0000 c000 0000 2353 7472  #~..........#Str
0000520: 696e 6773 0000 0000 5402 0000 0400 0000  ings....T.......
0000530: 2355 5300 5802 0000 4000 0000 2342 6c6f  #US.X...@...#Blo
0000540: 6200 0000 9802 0000 1000 0000 2347 5549  b...........#GUI
0000550: 4400 0000 0000 0000 0000 0000 0100 0001  D...............
0000560: 5705 0200 0900 0000 0000 0000 0000 0000  W...............
0000570: 0100 0000 0400 0000 0200 0000 0100 0000  ................
0000580: 0400 0000 0200 0000 0700 0000 0300 0000  ................
0000590: 0100 0000 0100 0000 0000 8700 0100 0000  ................
00005a0: 0000 0600 0a00 1100 0600 1e00 2800 0600  ............(...
00005b0: 3b00 1100 0600 4e00 1100 0000 0000 7e00  ;.....N.......~.
00005c0: 0000 0000 0100 0100 0100 1000 7300 0000  ............s...
00005d0: 0500 0100 0100 0100 9600 2700 ec20 0000  ..........'.. ..
00005e0: 0000 8618 1800 0100 0100 0022 0000 0000  ..........".....
00005f0: 8100 9b00 0100 0100 9422 0000 0000 8600  .........".......
0000600: a800 3500 0100 e022 0000 0000 9600 ba00  ..5....".........
0000610: 3b00 0300 0000 0100 ad00 0000 0200 b300  ;...............
0000620: 0900 1800 0100 1100 1800 0100 1100 4100  ..............A.
0000630: 0e00 1100 4500 1300 2100 5600 1800 1100  ....E...!.V.....
0000640: 6000 1d00 1100 6a00 2100 2b00 2f00 2b00  `.....j.!.+./.+.
0000650: 0480 0000 0000 0000 0000 0000 0000 0000  ................
0000660: 0000 7300 0000 0100 0000 8813 0000 0000  ..s.............
0000670: 0000 0500 0100 0000 0000 0000 006d 7363  .............msc
0000680: 6f72 6c69 6200 4f62 6a65 6374 0053 7973  orlib.Object.Sys
0000690: 7465 6d00 2e63 746f 7200 4172 7261 794c  tem..ctor.ArrayL
00006a0: 6973 7400 5379 7374 656d 2e43 6f6c 6c65  ist.System.Colle
00006b0: 6374 696f 6e73 0049 6e74 3332 0041 6464  ctions.Int32.Add
00006c0: 0067 6574 5f49 7465 6d00 436f 6e73 6f6c  .get_Item.Consol
00006d0: 6500 5772 6974 654c 696e 6500 6765 745f  e.WriteLine.get_
00006e0: 436f 756e 7400 7365 745f 4974 656d 0042  Count.set_Item.B
00006f0: 7562 626c 6553 6f72 7400 3c4d 6f64 756c  ubbleSort.<Modul
0000700: 653e 0042 7562 626c 6553 6f72 742e 6578  e>.BubbleSort.ex
0000710: 6500 6e75 6d73 0064 6f42 7562 626c 6553  e.nums.doBubbleS
0000720: 6f72 7400 7377 6170 0066 6972 7374 0073  ort.swap.first.s
0000730: 6563 6f6e 6400 4d61 696e 0000 0000 0000  econd.Main......
0000740: 0003 2000 0108 b77a 5c56 1934 e089 0420  .. ....z\V.4... 
0000750: 0108 1c04 2001 1c08 0400 0101 1c03 2000  .... ......... .
0000760: 0805 2002 0108 1c03 0612 0903 0701 0805  .. ............
0000770: 0703 0808 0805 2002 0108 0803 0000 0100  ...... .........
0000780: 5a64 03bf 392f 0846 95a9 37ad 5959 531b  Zd..9/.F..7.YYS.
0000790: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
00007a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000800: 0000 0000 0000 0000 0000 0000 0000 0100  ................
0000810: 1000 0000 1800 0080 0000 0000 0000 0000  ................
0000820: 0000 0000 0000 0100 0100 0000 3000 0080  ............0...
0000830: 0000 0000 0000 0000 0000 0000 0000 0100  ................
0000840: 0000 0000 4800 0000 5840 0000 8c02 0000  ....H...X@......
0000850: 0000 0000 0000 0000 8c02 3400 0000 5600  ..........4...V.
0000860: 5300 5f00 5600 4500 5200 5300 4900 4f00  S._.V.E.R.S.I.O.
0000870: 4e00 5f00 4900 4e00 4600 4f00 0000 0000  N._.I.N.F.O.....
0000880: bd04 effe 0000 0100 0000 0000 0000 0000  ................
0000890: 0000 0000 0000 0000 3f00 0000 0000 0000  ........?.......
00008a0: 0400 0000 0200 0000 0000 0000 0000 0000  ................
00008b0: 0000 0000 4400 0000 0100 5600 6100 7200  ....D.....V.a.r.
00008c0: 4600 6900 6c00 6500 4900 6e00 6600 6f00  F.i.l.e.I.n.f.o.
00008d0: 0000 0000 2400 0400 0000 5400 7200 6100  ....$.....T.r.a.
00008e0: 6e00 7300 6c00 6100 7400 6900 6f00 6e00  n.s.l.a.t.i.o.n.
00008f0: 0000 0000 7f00 b004 ec01 0000 0100 5300  ..............S.
0000900: 7400 7200 6900 6e00 6700 4600 6900 6c00  t.r.i.n.g.F.i.l.
0000910: 6500 4900 6e00 6600 6f00 0000 c801 0000  e.I.n.f.o.......
0000920: 0100 3000 3000 3700 6600 3000 3400 6200  ..0.0.7.f.0.4.b.
0000930: 3000 0000 2800 0200 0100 5000 7200 6f00  0...(.....P.r.o.
0000940: 6400 7500 6300 7400 5600 6500 7200 7300  d.u.c.t.V.e.r.s.
0000950: 6900 6f00 6e00 0000 2000 0000 2400 0200  i.o.n... ...$...
0000960: 0100 4300 6f00 6d00 7000 6100 6e00 7900  ..C.o.m.p.a.n.y.
0000970: 4e00 6100 6d00 6500 0000 0000 2000 0000  N.a.m.e..... ...
0000980: 2400 0200 0100 5000 7200 6f00 6400 7500  $.....P.r.o.d.u.
0000990: 6300 7400 4e00 6100 6d00 6500 0000 0000  c.t.N.a.m.e.....
00009a0: 2000 0000 2800 0200 0100 4c00 6500 6700   ...(.....L.e.g.
00009b0: 6100 6c00 4300 6f00 7000 7900 7200 6900  a.l.C.o.p.y.r.i.
00009c0: 6700 6800 7400 0000 2000 0000 3800 0b00  g.h.t... ...8...
00009d0: 0100 4900 6e00 7400 6500 7200 6e00 6100  ..I.n.t.e.r.n.a.
00009e0: 6c00 4e00 6100 6d00 6500 0000 4200 7500  l.N.a.m.e...B.u.
00009f0: 6200 6200 6c00 6500 5300 6f00 7200 7400  b.b.l.e.S.o.r.t.
0000a00: 0000 0000 2c00 0200 0100 4600 6900 6c00  ....,.....F.i.l.
0000a10: 6500 4400 6500 7300 6300 7200 6900 7000  e.D.e.s.c.r.i.p.
0000a20: 7400 6900 6f00 6e00 0000 0000 2000 0000  t.i.o.n..... ...
0000a30: 1c00 0200 0100 4300 6f00 6d00 6d00 6500  ......C.o.m.m.e.
0000a40: 6e00 7400 7300 0000 2000 0000 2400 0200  n.t.s... ...$...
0000a50: 0100 4600 6900 6c00 6500 5600 6500 7200  ..F.i.l.e.V.e.r.
0000a60: 7300 6900 6f00 6e00 0000 0000 2000 0000  s.i.o.n..... ...
0000a70: 4800 0f00 0100 4f00 7200 6900 6700 6900  H.....O.r.i.g.i.
0000a80: 6e00 6100 6c00 4600 6900 6c00 6500 6e00  n.a.l.F.i.l.e.n.
0000a90: 6100 6d00 6500 0000 4200 7500 6200 6200  a.m.e...B.u.b.b.
```

```
0000aa0: 6c00 6500 5300 6f00 7200 7400 2e00 6500  l.e.S.o.r.t...e.
0000ab0: 7800 6500 0000 0000 2c00 0200 0100 4c00  x.e.....,.....L.
0000ac0: 6500 6700 6100 6c00 5400 7200 6100 6400  e.g.a.l.T.r.a.d.
0000ad0: 6500 6d00 6100 7200 6b00 7300 0000 0000  e.m.a.r.k.s.....
0000ae0: 2000 0000 0000 0000 0000 0000 0000 0000   ...............
0000af0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b00: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b10: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b20: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b30: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b40: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b50: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b60: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b80: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000b90: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000ba0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000bb0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000bc0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000bd0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000be0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000bf0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c00: 0020 0000 0c00 0000 0230 0000 0000 0000  . .......0......
0000c10: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c20: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c30: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c40: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c50: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c60: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c80: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000c90: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000ca0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000cb0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000cc0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000cd0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000ce0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000cf0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d00: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d10: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d20: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d30: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d40: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d50: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d60: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d70: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d80: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000d90: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
0000da0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000db0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000dc0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000dd0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000de0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000df0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Appendix C

Assembly File Format

The assembly file format is an extension of the Portable Executable (PE) file format [36][37] used by Windows executables. An overview of the PE file format is shown in Figure C.1. The assembly file format is the same as the PE file format except the .text section is structured differently. See [36] or [37] for the layout of the .text section of the PE file format. The layout of the assembly .text section can be seen in Figure C.2.

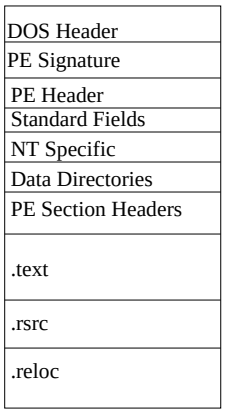

Figure C.1: PE and assembly file format

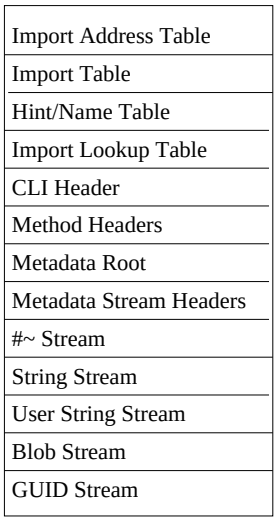

Figure C.2: Assembly .text section

The assembly file format begins with the DOS Header, which is a DOS executable stub used to print an error message if the assembly file is run in DOS. Next is the PE Signature, which is a magic number to mark the beginning of the PE file. The next two

Offset	RVA	Name	Value
0	0	Magic Number	4d5a
2	2	Last Page of File	0090
4	4	Pages in File	0003
6	6	Relocations	0000
8	8	Size of Header	0004
a	a	Min Paragraphs	0000
c	c	Max Paragraphs	ffff
e	e	Initial SS Value	0000
10	10	Initial SP Value	00b8
12	12	Checksum	0000
14	14	Initial IP Value	0000
16	16	Initial CS Value	0000
18	18	Relocation Table	0040
1a	1a	Overlay Number	0000
1c	1c	Reserved	0000 0000 0000 0000
24	24	OEM Identifier	0000
26	26	OEM Information	0000
28	28	Reserved2	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
3c	3c	PE Signature Offset	0000 0080
40	40	DOS Stub	0e1f ba0e 00b4 09cd 21b8 014c cd21 5468 6973 2070 726f 6772 616d 2063 616e 6e6f 7420 6265 2072 756e 2069 6e20 444f 5320 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000
80	80	PE Signature	5045 0000

Table C.1: DOS header

sections, PE Header and Standard Fields, contain information about where and how this assembly was created and the offset and size of the code and data sections. Next is the NT Specific section and it contains information about the environment used when an assembly file was created. The Data Directories section follows next and contains the size and offsets to additional tables used in the .text section. The Section Headers contain the name, size, and offsets of the sections contained in an assembly file. Three common sections follow one after another beginning with the .text section. This section contains the class and method information along with the code. The .rsrc section follows the .text section and contains unmanaged resources used by the assembly file which are stored as a tree. Icons, fonts and configuration files are examples of resources that can be stored in the .rsrc section. These resources are called unmanaged because this section was inherited from the PE file format and are not managed by the Common Language Infrastructure runtime. Managed resources are kept in the .text section. Following the .rsrc section is the .reloc section. This section

keeps a table of sections to be moved once the assembly is run. If interested, see Section 5 and 6 of [36] for the other optional sections that may appear.

The rest of this section will cover the decomposition and explanation of an example assembly file in more detail. A simple program using the bubble sort algorithm [12] was written in C# and compiled into "BubbleSort.exe" using the Mono 1.2.6's C# compiler. See Appendix A for the C# code and Appendix B for a print out of the assemblies byte values. If one follows along using Appendix B, note that multibyte values are in little endian format.

The assembly begins with the DOS Header (see Table C.1). Offsets in the assembly file format are given in either offsets from the beginning of the file or as a Relative Virtual Address (RVA). An RVA is the offset to a field from the beginning of that section plus the virtual address that the section was loaded at. The DOS Header begins with the DOS executable magic number 0x4d5a, which is "MZ" in ASCII [45]. The next two fields give a minimum length for the DOS file. The Pages in File field give the number of 512 byte pages and the Last Page of File field is the number of remaining bytes in the last page. Therefore, the entire file is at least 1168 bytes (((0x3-1)*512)+0x90 = 1160). The Relocations field tells us there are no sections in the DOS file that are to be relocated when the file is loaded. The Size of Header field is the size of this header in paragraphs, which is 16 bytes in length. Therefore, this header is 4*16 or 64 (0x40) bytes, which is the size of the DOS Header up to the DOS Stub field. The Min Paragraphs and Max Paragraphs field has the minimum and maximum number of extra paragraphs of memory this DOS executable needs for a .bss section. The .bss section is a block of memory allocated when the program is run and is used for uninitialized data. The Min and Max Paragraphs are set to default values since the DOS executable stub that follows does not need any extra memory [36].

The next two fields give the initial values for the Stack Segment (SS) register and Stack Pointer (SP) register used when the DOS executable is loaded. The SS register contains the address of the bottom of the system stack and the SP register contains the address of the current frame on the system stack. The Checksum field is used to check for modifications in the file since the Checksum field was calculated. Since the Checksum field is 0x0, it is not used. The next two fields give the initial values for the Instruction Pointer (IP) register and Code Segment (CS) register. The IP register contains the address of the next instruction to execute. The CS register contains the address of the base of the code section. The Relocation Table gives the offset from the beginning of this section to the beginning of the code segment. Therefore, the code segment for this DOS file is 0x40 (0x0+0x40 = 0x40). At offset 0x40 is the stub of a DOS executable. Overlay number is used if this file is part of a multi-file DOS executable. Since Overlay Number is 0x0, this is the main part of the DOS executable. The Reserved field is not used and reserved for future use. The OEM Identifier and OEM Information can be used to track who created a DOS executable, but is not used here. The Reserved2 field is not used and reserved for future use. The PE Signature Offset gives the offset to the PE Signatures [45].

Byte(s)	**Assembly Instruction**	**Note**
0e	push cs	Push code segment address
1f	pop ds	Pop code segment address into data segment register
ba000e	mov dx, offset message	Load address of message
b409	mov ah, 9h	9h is OS's print service
cd21	int 21h	Call OS
b84c01	mov ax, 4c01h	4c01h is OS's exit service
cd21	int 21h	Call OS
5468 6973 2070 726f 6772 616d 2063 616e 6e6f 7420 6265 2072 756e 2069 6e20 444f 5230 6d6f 6465 2e0d 0d0a 24	message db "This program cannot be run in DOS mode.",0dh,0ah,'$'	message variable in data section

Table C.2: DOS stub instructions [24][45]

Offset	RVA	Name	Value
84	84	Machine	014c
86	86	Number of Sections	0003
88	88	Time/Date Stamp	48f6 2181
8c	8c	Pointer to Symbol Table	0000 0000
90	90	Number of Symbols	0000 0000
94	94	Optional Header Size	00e0
96	96	Characteristics	010e

Table C.3: PE header

Offset	RVA	Name	Value
98	98	Magic	010b
9a	9a	LMajor	06
9b	9b	LMinor	00
9c	9c	Code Size	0000 0600
a0	a0	Initialized Data Size	0000 0400
a4	a4	Uninitialized Data Size	0000 0000
a8	a8	Entry Point RVA	0000 2000
ac	ac	Base of Code	0000 2000
b0	b0	Base of Data	0000 4000

Table C.4: Standard fields

Next is the DOS Stub. If this assembly is run in DOS, the DOS Stub is loaded and executed. The DOS Stub is a mix of Intel x86 assembly instructions and a data section (see Table C.2 for how the DOS Stub from Table C.1 is decomposed). Note that in Table C.2, numbers ending in the character 'h' are in hex format.

The code begins by pushing the code segment address. Then it pops the code segment address into the data segment address. Therefore, the offset of the variable *message* is calculated from the beginning of the code segment. This is done because the message variable is placed right after the code. Next the offset of the message variable is loaded and the Operating System (OS) is called to print the message. Then the exit service call is executed by the OS. Finally, the message variable is declared. The 0x0d byte in the message variable is a line-feed character and the 0x0a byte is a newline character. The '$' character is used as a null terminator for the string. The DOS Stub is padded with seven bytes of zeros after this code.

Finally the PE Signature follows in Table C.1. The PE Signature contains the magic number "PE\0\0" in ASCII. The PE Signature is used to mark the beginning of the PE file after the DOS Header.

The PE Header table follow the PE Signature. The format of the PE Header can be seen in Table C.3. At offset 0x84 is the Machine field. Since the Machine value is 0x14c, this assembly was compiled on an Intel 386 or later machine. The Number of Sections value indicates that there are three sections in this assembly. The Section Headers section later in the assembly will list and give more information about each of these sections (see Tables C.7, C.8, and C.9). If one looks ahead, one will see that these three sections are the .text, .rsrc and .reloc sections. The Time/Date Stamp indicates the number of seconds between January 1, 1970 12:00:00 AM and the date and time this assembly was created. Therefore, this assembly was created on October 15, 2008 12:59:45 PM (January 1, 1970 12:00:00 + 0x48f62181 seconds = October 15, 2008 12:59:45 PM).

The Pointer to Symbol Table and Number of Symbols fields contain an offset to the symbol table and number of symbols in the symbol table. Since the Pointer to Symbol Table and Number of Symbols is zero, there is not an embedded symbol table in this assembly. The Optional Header Size indicates the total size of three optional tables if they exist. The first of these optional tables is the Standard Fields and it's size is 28 bytes. The second optional table is the NT Specific table and it's size is 68 bytes. The last optional table is the Data Directories table and it's size is 128 bytes. Since all three of these optional tables exist, the Optional Header Size is 0xe0 (28 + 68 + 128 = 224 or 0xe0). The Characteristics field indicates the attribute flags of this assembly. The four flags that are set are 0x0100, which means that this is a 32 bit-word based assembly, 0x0008, which means that the symbol table has been removed, 0x0004, which means that line numbers have been removed, and 0x0002, which means that this is an executable file [37].

The Standard Fields table, as shown in Table C.4, follows at offset 0x98. The 0x010b value in the Magic field means this is an executable file. The LMajor and LMinor fields contain the major and minor numbers of the linker used to create this assembly. The Code

Size field indicates that the size of the .text section is 0x600 bytes. The Initialized Data Size field means that the .rsrc section is 0x400 bytes long and a zero in the Uninitialized Data Size field means there is no .bss section. The .bss section is a block of memory created at runtime to hold uninitialized data.

Recall that a Relative Virtual Address (RVA) is calculated by adding the offset from the beginning of a section to a field and the virtual address that that section begins at. The Base of Code field indicates that the .text section will be loaded at the virtual address 0x0000 2000. The Entry Point RVA value of 0x2000 means that the Entry Point table begins at the first byte of the .text section. The Base of Data field indicates that the .rsrc section begins at virtual address 0x0000 4000. These virtual addresses are used to calculate any RVAs used in these sections.

Offset	**RVA**	**Name**	**Value**
b4	b4	Image Base	0040 0000
b8	b8	Section Alignment	0000 2000
bc	bc	File Alignment	0000 0200
c0	c0	OS Major	0004
c2	c2	OS Minor	0000
c4	c4	User Major	0000
c6	c6	User Minor	0000
c8	c8	SubSys Major	0004
ca	ca	SubSys Minor	0000
cc	cc	Reserved	0000 0000
d0	d0	Image Size	0000 8000
d4	d4	Header Size	0000 0200
d8	d8	File Checksum	0000 0000
dc	dc	SubSystem	0003
de	de	DLL Flags	0000
e0	e0	Stack Reserved Size	0010 0000
e4	e4	Stack Commit Size	0000 1000
e8	e8	Heap Reserved Size	0010 0000
ec	ec	Heap Commit Size	0000 1000
f0	f0	Loader Flags	0000 0000
f4	f4	Number of Data Directories	0000 0010

Table C.5: NT specific

The NT Specific section directly follows the Standard Fields section (see Table C.5). The Image Base field gives the preferred address the beginning of the assembly should be

loaded at. The Section Alignment value means that every section in the assembly, like the .text, .rsrc, and .reloc sections, must have a size that is a multiple of 0x0000 2000 bytes. The File Alignment value means that the total assembly, from the first byte of the DOS Header to the last byte of the .reloc section, must have a size that is a multiple of 0x0000 0200 bytes. Sections are padded with zero bytes to make sure that they are aligned. The OS, User, and SubSys major and minor numbers give version information about the OS and system used to create this assembly. The Mono C# compiler set these to default values to mimic the Microsoft .NET compilers. The Reserved field is not used and reserved for future use.

The Image Size field contains the value of the total number of bytes used by the entire assembly in memory. The Section Headers later in Tables C.7 to C.9 give the virtual addresses used by each section. The Header Size field is the combined sizes of all the tables from the DOS Header to the beginning of the .text section.

The File Checksum field is used to detect if the assembly has been modified. Since the File Checksum field is set to zero, it is not used in this example. The SubSystem field tells what version of the Windows kernel subsystem is expected to run this assembly. A value of 0x0003 for the SubSystem field means that this assembly is expected to run in the Windows character subsystem (i.e., Window's command-line). The DLL Flags field contains flags for various miscellaneous flags for how DLL's should run, but none are set in this example. The Stack Reserved Size field tells how many bytes this assembly expects to use for it's stack. The Stack Commit Size field tells how many bytes to add to the stack each time if the assembly goes over what it has reserved with the Stack Reserved Size. The Heap Reserved Size and Heap Commit Size serve the same purpose except they are used to record how much memory to allocate for the heap. The Loader Flags are reserved and at this time there are no flags defined, but in the future they may be added. The Number

of Data Directories field tells the number of entries in the next table. Therefore, there are 0x0000 0010 (or 16) Data Directories in the next table.

Offset	RVA	Name	Value
f8	f8	Export Table	0000 0000 0000 0000
100	100	Import Table	0000 004f 0000 2018
108	108	Resource Table	0000 02e4 0000 4000
110	110	Exception Table	0000 0000 0000 0000
118	118	Certificate Table	0000 0000 0000 0000
120	120	Base Relocation Table	0000 000c 0000 6000
128	128	Debug	0000 0000 0000 0000
130	130	Copyright	0000 0000 0000 0000
138	138	Global Ptr	0000 0000 0000 0000
140	140	TLS Table	0000 0000 0000 0000
148	148	Load Config Table	0000 0000 0000 0000
150	150	Bound Import	0000 0000 0000 0000
158	158	Import Address Table	0000 0008 0000 2010
160	160	Delay Import Descriptor	0000 0000 0000 0000
168	168	CLI Header	0000 0048 0000 2064
170	170	Reserved	0000 0000 0000 0000

Table C.6: Data directories

Next is the Data Directories table (see Table C.6). Each entry in the Data Directories contains an encoded size and RVA for a table that follows later in the assembly. If the encoded value is 0x0000 0000 0000 0000, then the table does not exist. The first four bytes is the size of the section and the last four bytes is the RVA of the section. For example, since the Import Table has a value of 0x0000 004f 0000 2018, the size of the Import Table is 0x0000 004f and it's RVA is 0x0000 2018. Also, since the Base of Code field in the Standard Fields (see Table C.4) is 0x0000 2000, the Import Table is 0x18 bytes after the beginning of the .text section. The Import Table contains RVA's of other tables like the Import Lookup Table, the Import Address Table, and the Name Table, which are explained when they are encountered later in the assembly.

The .rsrc section is represented by the Resource Table at offset 0x108. Since the Resource Table has the value 0x0000 02e4 0000 4000, the .rsrc section begins at RVA 0x0000 2000 and has a size of 0x0000 02e4. Note that this information about the .rsrc section is

repeated in the .rsrc Section Header covered later in Table C.8. The Base Relocation Table represents the .reloc section and is also repeated later in Table C.9. The Import Address Table gives the RVA and size of the Import Address Table seen later in Table C.10. The CLI Header field give the RVA and size of the table with additional information about the CLI system used by this assembly. Other tables not used here are the Debug table that contains debug information generated and inserted into the assembly at compile time, the Exception Table which give the RVA and size of the table that contains the exception handler information in the assembly, the Copyright field which contains the RVA and size of a table that contains the copyright information about this assembly, and the Certificate Table which contains the RVA and size of the table that contains the public and private key cryptography information used in digitally signing an assembly [37].

Offset	RVA	Name	Value
178	178	Name	2e74 6578 7400 0000
180	180	Virtual Size	0000 0590
184	184	Virtual Address	0000 2000
188	188	Size of Raw Data	0000 0600
18c	18c	Pointer To Raw Data	0000 0200
190	190	Pointer To Relocations	0000 0000
194	194	Pointer To Line Numbers	0000 0000
198	198	Number of Relocations	0000
19a	19a	Number of Line Numbers	0000
19c	19c	Characteristics	6000 0020

Table C.7: Section header: .text

Offset	RVA	Name	Value
1a0	1a0	Name	2e72 7372 6300 0000
1a8	1a8	Virtual Size	0000 02e4
1ac	1ac	Virtual Address	0000 4000
1b0	1b0	Size of Raw Data	0000 0400
1b4	1b4	Pointer To Raw Data	0000 0800
1b8	1b8	Pointer To Relocations	0000 0000
1bc	1bc	Pointer To Line Numbers	0000 0000
1c0	1c0	Number of Relocations	0000
1c2	1c2	Number of Line Numbers	0000
1c4	1c4	Characteristics	4000 0040

Table C.8: Section header: .rsrc

Next are the Section Headers beginning at offset 0x178. The Number of Sections field at offset 0x86 in the PE Header (see Table C.3) indicates that there are 0x0003 sections. Each of the three section has a Section Header with the same fields but with different values

Offset	RVA	Name	Value
1c8	1c8	Name	2e72 656c 6f63 0000
1d0	1d0	Virtual Size	0000 000c
1d4	1d4	Virtual Address	0000 6000
1d8	1d8	Size of Raw Data	0000 0200
1dc	1dc	Pointer To Raw Data	0000 0c00
1e0	1e0	Pointer To Relocations	0000 0000
1e4	1e4	Pointer To Line Numbers	0000 0000
1e8	1e8	Number of Relocations	0000
1ea	1ea	Number of Line Numbers	0000
1ec	1ec	Characteristics	4200 0040

Table C.9: Section header: .reloc

in each field. The first Section Header is for the .text section (see Table C.7). The ASCII value ".text\0\0\0" is contained in the Name field. The Virtual Size contains the value 0x0000 0590, which is the size of the .text section without the padding. The Virtual Address field indicates that the .text section begins at RVA 0x0000 2000. The Size of Raw Data indicates that the .text section is 0x0000 0x0600 bytes in size with padding. The Pointer To Raw Data field tells us that the .text section begins at offset 0x0000 0x0200. Since there are no relocations or line numbers, the next four fields are zero. The Characteristics field contains flags for the .text section. Three flags are set for the .text section: contains executable code (0x0000 0002), section can be executed as code (0x2000 0000), and section can be read (0x4000 0000).

Offset	RVA	Name	Value
210	2010	Hint/Name Table RVA	0000 2040

Table C.10: Import address table

The fields in the Section Header for the .rsrc (see Table C.8) and .reloc (see Table C.9) sections can be understood similar to the ones for the .text section. The Characteristic flags for the .rsrc section are section can be read (0x4000 0000) and section contains initialized data (0x0000 0040). The Characteristic flags for the .reloc section are section

Offset	RVA	Name	Value
218	2018	ImportLookupTable RVA	0000 205a
21c	201c	DateTime Stamp	0000 0000
220	2020	Forward Chain	0000 0000
224	2024	Name RVA	0000 204e
228	2028	ImportAddressTable RVA	0000 2010

Table C.11: Import table

can be read (0x4000 0000), section can be discarded (0x0200 0000) and section contains initialized data (0x0000 0040). Since each section must begin on an offset that is a multiple of 0x0200 and the .text section is next, the assembly is padded with zero bytes from the end of the .reloc Section Header at offset 0x1f0 to offset 0x200. Therefore the .text section must begin at 0x0200 because it is the closest multiple of the File Alignment field from the NT Specific table at Table C.5.

The .text section follows next at offset 0x200 and RVA 0x2000. The Import Address Table (see Table C.10) begins at RVA 0x2010. The Import Address Table contains the RVA of the Hint/Name Table. Four bytes of padding follow the Import Address Table. The purpose of the Import Address Table is to find the Hint/Name Table RVA.

Next is the Import Table (see Table C.11) at RVA 0x2018. The Import Table contains the Import Lookup Table's RVA and the Import Address Table's RVA. The rest of the Import Table concerns itself with information about the _CorExeMain method in mscoree.dll. The DateTime Stamp field is set to 0x0000 0000 in this assembly, but is later loaded with the Date/Time Stamp of an external DLL called mscoree.dll, which is loaded to run the _CorExeMain method. The Name RVA field contains the RVA to the ASCII string of name of the DLL to load. Since the Forward Chain is set to zero, the DLL that is referenced in the Name RVA field is the DLL that the ImportTable is looking for and not a forward reference to another one. The Import Table is followed by 20 bytes of undocumented zeros.

Offset	RVA	Name	Value
240	2040	Hint	0000
242	2042	Name	5f43 6f72 4578 654d 6169 6e00
24e	204e	Name	6d73 636f 7265 652e 646c 6000

Table C.12: Hint/Name table

Offset	RVA	Name	Value
25a	205a	Hint/Name Table RVA	0000 2040

Table C.13: Import lookup table

The Hint/Name Table (see Table C.12) follows at RVA 0x2040. The Hint/Name Table contains the names of the main library of the Virtual Machine called "mscoree.dll\0" and the method that begins the execution of this assembly called "_CorExeMain\0". The Import Lookup Table (see Table C.13) that follows at RVA 0x205a contains the Hint/Name Table's RVA. The Import Lookup Table is used to find the Hint/Name Table. Six bytes of padding follow the Import Lookup Table to being it to a four byte alignment.

Offset	RVA	Name	Value
264	2064	Cb	0000 0048
268	2068	Major Runtime Version	0002
26a	206a	Minor Runtime Version	0000
26c	206c	Metadata	0000 0000 0000 22e8
274	2074	Size of Metadata	0000 02a8
278	2078	Flags	0000 0001
27c	207c	Entry Point Token	0600 0004
280	2080	Resources	0000 0000 0000 22e8
288	2088	Strong Name Signature	0000 0000 0000 0000
290	2090	Code Manager Table	0000 0000 0000 0000
298	2098	VTable Fixups	0000 0000 0000 0000
2a0	20a0	Exported Address Table Jumps	0000 0000 0000 0000
2a8	20a8	Managed Native Header	0000 0000 0000 0000

Table C.14: CLI header table

The CLI Header (see Table C.14) follows next at RVA 0x2064. The CLI Header contains information about the CLI framework that the assembly was built against. The Cb field contains the size of the CLI Header in bytes. The Major and Minor Runtime Version fields contain the the minimum version of the framework this assembly needs to

run. In this case, this assembly needs at least a 2.0 .NET-compatible framework. The Metadata and Size of Metadata fields gives the RVA and size of the Metadata table, which is a large table containing class, method and metadata information about this assembly. The Flags field has one flags set: assembly contains CIL only code (0x0000 0001).

The Entry Point Token field contains an encoded token that references the Main() method in this assembly. The encoded token is comprised of two pieces: a table and row number. The highest byte of the token contain the table number and the lower three bytes contain the row number. In this case, the Entry Point Token 0x0600 0004 point to row 0x00 0004 in table 0x06. Table 0x06 is the MethodDef table and row 0x00 0004 contains the necessary information to look the Main() method up. More information about how to lookup methods, classes and fields will follow later in the Metadata section (see Table C.19 to Table C.39).

Offset	**RVA**	**Name**	**Value**
2ec	20ec	Type/Size	1330
2ee	20ee	Max Stack	001f
2f0	20f0	Code Size	0000 0106
2f4	20f4	LocalVarSig Token	1100 0001
2f8	20f8	Code	02280a00000102730a0000027d04000001027b040000011f5b8c01 0000036f0a00000326027b040000011b8c010000036f0a00000326 027b040000011f658c010000036f0a00000326027b04000001198c 010000036f0a00000326027b040000011f3a8c010000036f0a0000 0326027b040000011ff28c010000036f0a00000326027b04000001 20000000c78c010000036f0a00000326027b040000011f2c8c0100 00036f0a00000326027b04000001178c010000036f0a0000032602 7b0400000120000002a68c010000036f0a00000326022806000002 160a3800000015027b04000001066f0a000004280a000005161758 0a06027b040000016f0a0000063fffffffda2a

Table C.15: Method header - public BubbleSort()

The Resources field hold the RVA of CLI resources. An assembly can be given what is called a Strong Signature. A Strong Signature is a way to uniquely identify an assembly. Four fields are used in a Strong Signature: Assembly Name, Version, Culture, and a Public Key Token [50]. If an assembly had a Strong Signature, the Strong Name Signature field

would contain an RVA and size to reference it. Since the VTable Fixups field is set to zero, there are no sections to relocate. If an assembly contains native code, the Exported Address Table Jumps and Managed Native Header fields would be used to load, jump to and execute it. After the CLI Header is 60 bytes of padding until the first Method Header at 0x2ec.

Offset	**RVA**	**Name**	**Value**
400	2200	Type/Flags	1330
402	2202	Max Stack	000e
404	2204	Code Size	0000 0086
408	2208	LocalVarSig Token	1100 0002
40c	220c	Code	160c380000006d160a3800000051160b3800000035027b04000001066 f0a00000479010000034a027b04000001076f0a00000479010000034a 3c000000802060728060000030717580b07027b040000016f0a00000 63ffffffba0617580a06027b040000016f0a0000063ffffff9e0817 580c08027b040000016f0a0000063ffffff822a

Table C.16: Method header - private void doBubbleSort()

Offset	**RVA**	**Name**	**Value**
494	2294	Type/Flags	1330
496	2296	Max Stack	000a
498	2298	Code Size	0000 003e
49c	229c	LocalVarSig Token	1100 0003
4a0	22a0	Code	027b04000001036f0a00000479010000034a0a027b0400000103027 b04000001046f0a0000046f0a000007027b0400000104068c010000 036f0a0000072a

Table C.17: Method header - public void swap(int first,int second)

Method Headers are next and each Method Header represents one method in the assembly. Method Headers are one of two types, either Tiny or Fat. If bits zero and one of the first byte of the Method Header are B10, then it is a Tiny method. If the same two bits are instead B11, then it is a Fat method. For example, the first Method Header's (see Table C.15) type is Fat because bit zero and one of 0x13 are B11. If a Fat method, in the first two bytes, bits 12 through 15 give the size of this Method Header except the Code. In this case, bits 12 to 15 are B0011. Therefore the total size used for the Type/Size, Max

Stack, Code Size, and LocalVarSig Token fields is three 4-byte integers.

All Fat methods have the same five fields shown in Table C.15. The Max Stack field indicates the maximum depth the stack every gets during execution. The Code Size field tells how many bytes the upcoming Code field is. When fields need to reference a Metadata table, a token is used. The LocalVarSigToken is a token that references table 0x11, which is the LocalVarSig metadata table, at row 0x1. This row of the LocalVarSig table contains the type information about the local variables in this method. Then the Common Intermediate Language (CIL) code follows in this Fat method. The CIL instruction format was covered in Section 2.3. Table C.16 and Table C.17 also contain Method Headers that are Fat.

Offset	**RVA**	**Name**	**Value**
4e0	22e0	Type/Flags	1e
4e1	22e1	Code	7306000001262a

Table C.18: Method header - public static void Main()

Offset	**RVA**	**Name**	**Value**
4e8	22e8	Signature	42534a42
4ec	22ec	Major Version	0001
4ee	22ee	Minor Version	0001
4f0	22f0	Reserved	0000 0000
4f4	22f4	Length	0000 000c
4f8	22f8	Version	7631 2e31 2e34 3332 3200 0000
504	2304	Flags	0000
506	2306	Streams	0005

Table C.19: Metadata root

Table C.18 contains the first Tiny Method Header. The Tiny Method Header type only contains the Type/Flags and the Code fields. After masking out the bits not used by the Tiny/Fat flag (bits 2 to 7) from the Type/Flags field, B000111 is returned. Therefore, Code Size is 7 bytes. The Code field contains the Common Intermediate Language (CIL) instruction bytes. Whether a Method Header is encoded as Tiny or Fat depends on certain

features of the method. If code size can fit in 6 or less bits, there are no exceptions, no local variables and the maximum depth of the stack is equal to or less than eight, then the method is Tiny. If any of these conditions fail, then it is a Fat method [37].

Next is the Metadata Root (see Table C.19) at RVA 0x22e8. The Metadata Root begins the metadata section of the assembly. The ASCII string "BSJB" is contained in the Signature field. The Signature field is a magic number to mark the beginning of the Metadata Root. The Major and Minor Version fields indicate that the metadata section in this assembly is encoded using the 1.1 version of the CLI specification. The Reserved field is reserved for future use. The Length field contains the length of the version string that follows. Therefore the 0x0000 000c bytes that follow is the Version field. The ASCII string "v1.1.4322" is in the Version field. This is the major, minor and build number of the .NET Framework this assembly was built with. The Mono C# compiler that was used to build this assembly mimicked that version of the .NET Framework in order to insure portability. Since the Flags field is 0x0000, there are no flags for the metadata in this assembly. Finally, the Streams field indicates that there are 0x0005 streams in this assembly. The content of these five streams (#~, #Strings, #US, #Blob, and #GUID) are covered later (see Tables C.25 to C.39).

Offset	RVA	Name	Value
508	2308	Offset	0000 0070
50c	230c	Size	0000 0124
510	2310	Name	237e 0000

Table C.20: Stream header - #~

Offset	RVA	Name	Value
514	2314	Offset	0000 0194
518	2318	Size	0000 00c0
51c	231c	Name	2353 7472 696e 6773 0000 0000

Table C.21: Stream header - #Strings

After the Method Headers, comes five similar headers that describe persistent streams in the assembly. These five streams are #~, #US, #Strings, #Blob, and #GUID. The #~ stream contains information about the metadata tables later in the assembly.

Offset	RVA	Name	Value
528	2328	Offset	0000 0254
52c	232c	Size	0000 0004
530	2330	Name	2355 5300

Table C.22: Stream header - #US

Offset	RVA	Name	Value
534	2334	Offset	0000 0258
538	2338	Size	0000 0040
53c	233c	Name	2342 6c6f 6200 0000

Table C.23: Stream header - #Blob

Offset	RVA	Name	Value
544	2344	Offset	0000 0298
548	2348	Size	0000 0010
54c	234c	Name	2347 5549 4400 0000

Table C.24: Stream header - #GUID

Offset	RVA	Name	Value
558	2358	Reserved1	0000 0000
55c	235c	Major Version	01
55d	235d	Minor Version	00
55e	235e	Heap Sizes	00
55f	235f	Reserved2	01
560	2360	Valid	0000 0009 0002 0557
568	2368	Sorted	0000 0000 0000 0000
570	2370	Table Rows	0000 0001 0000 0004 0000 0002 0000 0001 0000 0004 0000 0002 0000 0007 0000 0003 0000 0001 0000 0001

Table C.25: #˜ stream

The #US stream contains string literals that were in the original source code. The #Strings stream contains method and field names and other compiler generated strings. The #Blob stream contains encoded types used by the metadata tables later in the assembly. The #GUID stream contains Global Unique Identifiers (GUID) used to uniquely identify assemblies. Each stream header has an Offset field that tells the offset from the beginning of the Metadata Root (see Table C.19) to that stream. For example, 0x22e8 plus 0x0000 0070 equals 0x2358, which is the RVA of the #˜ stream. The Size field contains the size of the stream in bytes. For example, the #GUID stream is 0x0000 0010 bytes long. The final field, Name, gives a string representation of the name of the stream. For example, the ASCII string "#GUID" is stored in the bytes 0x2347554944 in Table C.24.

The #~ stream (see Table C.25) is next at RVA 0x2358. The Reserved1 and Reserved2 fields are reserved for future use and not used now. The Major and Minor Version fields represent the version of how the metadata is encoded in this assembly. The Heap Sizes field is a bit field that contains a flag if a stream (or sometimes called a heap the CLI specification) has a size that is greater than or equal to 65,536 bytes. If the #Strings bit is set (0x01), then all indexes into the #Strings stream encoded in this assembly are 4-bytes wide. If that bit is not set, then indexes into the #Strings stream are 2-bytes wide. Since none of the bits are set in the Heap Sizes field, then all stream (or heap) indexes are 2-bytes wide. The Valid field is a bit vector that has a bit set for each Metadata table that come later in the assembly. According to the Valid field, this assembly contains the Module Table, TypeRef Table, TypeDef Table, Field Table, MethodDef Table, Param Table, MemberRef Table, StandAloneSig Table, Assembly Table, and the AssemblyRef Table (see Table C.26 to Table C.35). Next is the Sorted field, which is a bit vector that contains a bit for each metadata table that is sorted. Since the Sorted field is 0x0000 0000 0000 0000, none of the metadata tables are sorted according to any criteria. Next is the Table Rows field, which is a list of 4-byte integers, one for each metadata table in this assembly. Each 4-byte integer in the Table Rows field gives the number of rows in each metadata table. For example, the Module Table has one row and the TypeRef Table has four rows. Note that the first row of a metadata table is indexed as 1 and not 0.

Offset	**RVA**	**Generation**	**Name**	**Mvid**	**Encld**	**EncBaseId**
598	2398	0000	0087	0001	0000	0000

Table C.26: Module table

The first metadata table is the Module Table (see Table C.26) at RVA 0x2398. The Module Table contains a row for each module that is in this in this assembly. CLI-compliant code can be compiled into separate modules and combined into one assembly, but

this assembly only contains one module. The Generation value is reserved and not used. The Name field is an byte index into the #Strings stream. The ASCII string "BubbleSort.exe" is at byte index 0x87. The Mvid field is an index into the #GUID stream for a GUID used to distinguish two modules if they have the same name. Therefore, row 1 in the #GUID stream (see Table C.39) contains this assembly's unique GUID. The Encld and EncBaseId fields are reserved and not used.

Offset	RVA	ResolutionScope	Name	Namespace
5a2	23a2	0006	000a	0011
5a8	23a8	0006	001e	0028
5ae	23ae	0006	003b	0011
5b4	23b4	0006	004e	0011

Table C.27: TypeRef table

Offset	RVA	Flags	Name	Namespace	Extends	FieldList	MethodList
5ba	23ba	0000 0000	007e	0000	0000	0001	0001
5c8	23c8	0010 0001	0073	0000	0005	0001	0001

Table C.28: TypeDef table

The next metadata table is the TypeRef Table (see Table C.27) at RVA 0x23a2. Each of the four rows in the TypeRef Table represents an external type that is referenced in this assembly. The ResolutionScope is an encoded index into either the Module, ModuleRef, AssemblyRef or TypeRef Table that represents the module or assembly this type came from. Take row one for example. Since bits 0 and 1 are B10 in the ResolutionScope 0x0006, this is an index into the AssemblyRef Table [37]. Module uses B00, ModuleRef uses B01 and TypeRef uses B11 in bits 0 and 1 to represent these tables. The other 14 bits represent the row number [37]. Therefore the ResolutionScope of the first row refers to row 0x0001 of the AssemblyRef Table. The Name field is an byte index into the #Strings stream that contains the name of this type. For example, row one's name references the ASCII string "Object"

using it's offset 0x000a into the #Strings stream. The Namespace field is a byte index into the #Strings stream with the string representation of the namespace this type is in. For example, the string "System" is pointed at by row one's Namespace field. Therefore, row one represents the external type System.Object.

The next metadata table to cover is the TypeDef Table (see Table C.28) at RVA 0x23ba. Each row in the TypeDef Table represents the types that are defined in this assembly. The Flags field is a bitset of flags that represent the visibility of a type. For example, row two has the Public flag (0x0000 0001), which means this type is globally visible, and the BeforeFieldInit flag (0x0010 0000), which indicates non-static fields must be initialized before static fields can be accessed.

The Name and Namespace fields are indexes into the #Strings stream, which contains the string representation of this type. The Extends field is an encoded index into either the TypeDef, TypeRef, or TypeSpec Table. Bits 0 and 1 are used to encode which table: TypeDef (B00), TypeRef (B01), and TypeSpec (B10) [37]. The other 14 bits represent the row number [37]. Therefore, row two's Extends field (0x0005) is encoded as TypeRef Table and row 1. TypeRef row 1 represents the "System.Object" type. Therefore, this type extends (or inherits) from the System.Object type.

Offset	**RVA**	**Flags**	**Name**	**Signature**
5d6	23d6	0001	0096	0027

Table C.29: Field table

The FieldList field indexes into the Field Table to show which fields go with this type. The index into the Field Table is the start of a line of fields that belong to that type. The line ends when either the Field Table ends or the next types fields begin. The MethodList is an index into the MethodDef Table that represents the line of methods that belong to a certain type. Therefore, in this example, row 2 represents the "BubbleSort"

type whose methods start at row 1 of the MethodDef Table and whose fields start at row 1 of the Field Table.

Next is the Field Table (see Table C.29). Each row in the Field Table represents a field that belongs to a type. The Flags field is a bitset that represents the access permissions of the field. For example, the Flag field in row one has the private flag (0x0001) set. The Name field is an index into the #Strings stream, which contains the string representation of the name of this field. In this case, the ASCII string "nums" is referenced by the Name field in row one. The Signature field is a index into the #Blob stream that represents the type of this field. Thus, this encodes the private "nums" field seen in the Bubble Sort source code in Appendix A.

Offset	**RVA**	**RVA**	**ImplFlags**	**Flags**	**Name**	**Signature**	**ParamList**
5dc	23dc	0000 20ec	0000	1886	0018	0001	0001
5ea	23ea	0000 2200	0000	0081	009b	0001	0001
5f8	23f8	0000 2294	0000	0096	0086	00a8	0035
606	2406	0000 22e0	0000	0096	00ba	003b	0003

Table C.30: MethodDef table

Next is the MethodDef Table (see Table C.30). Each row in the MethodDef Table represents a method that is defined in this assembly. The RVA field (column three in Table C.30) is the RVA to the Method Header that contains the information and CIL about the method this row refers to. The ImplFlags field contains flags like method is implemented in CIL (0x0000) and method is implemented in native (0x0001). The Flags field contains flags for the visibility of this method. The Name field indexes into the #Strings stream with the ASCII representation of the name of this method. The Signature field is an index into the #Blob stream, which contains the encoded representation of the parameters and return type of this method. The ParamList field is an index into the Param Table, which starts a line of parameters that belong to this method.

For example, the RVA field in row one references Table C.15, which is the method header for the BubbleSort constructor. The ImplFlags for row one is 0x0000 because the method is implemented in CIL. The Flags field has the RTSpecialName (0x1000), SpecialName (0x0800), HideBySig (0x0080), and Public (0x0006) flags set. The first two flags are set because this method is treated special because it is a constructor. The third flag tells how inheritance should hide this method if it is overridden [37]. The last flag tells the visibility of this method. The Name value references offset 0x694 (0x67c + 0x0018 = 0x694) in the #Strings stream. The ASCII string ".ctor" is at this offset into the #Strings stream. This string is the internal name for constructors.

The Signature value references offset 0x741 (0x0740 + 0x0001 = 0x0741) in the #Blob stream (see Table C.38), which contains the bytes 0x3020 0001. The first byte is the size of the #Blob row minus the size byte. The next byte is the calling conventions for the method. The next byte contains the number of parameters. Therefore, the BubbleSort constructor has no parameters. The next byte encodes the return type. Since this is a constructor, the magic number ELEMENT_TYPE_VOID (0x01) is used. What would follow, if this method had parameters, would be an encoded value or magic number for each parameter.

The ParamList field is an index into the Param table that begins the list of parameters for this method. This list ends when the next begins or the table ends. Note that the method represented by row two has parameters that start at index 1. Therefore, the BubbleSort constructor has parameters row one to one, which represents no parameters.

Offset	**RVA**	**Flags**	**Sequence**	**Name**
614	2414	0000	0001	00ad
61a	241a	0000	0002	00b3

Table C.31: Param table

Next is the Param Table (see Table C.31). Each row in the Param Table represents one parameter used by a method. The Flags field contains flags about the usage of a parameter. The Flags field contains flags about whether a parameter is called by reference or by value. The Sequence field contains the number of total parameters in the method that own this parameter. The Name field is a byte index into the #Strings stream which contains the ASCII representation of the parameters name.

Offset	**RVA**	**Class**	**Name**	**Signature**
620	2420	0009	0018	0001
626	2426	0011	0018	0001
62c	242c	0011	0041	000e
632	2432	0011	0045	0013
638	2438	0021	0056	0018
63e	243e	0011	0060	001d
644	2444	0011	006a	0021

Table C.32: MemberRef table

Next is the MemberRef Table (see Table C.32). Each row in the MemberRef Table represents either a reference to an external method or field in another assembly. The Class field is an encoded index into either the TypeRef, ModuleRef, MethodDef, TypeSpec, or TypeDef Table. Bits 0 and 1 are used to determine which table and the other 14 bits is the row number [37]. For example, row one indexes into the TypeRef Table at row two. The Name field is a byte index into the #Strings stream, which represents the name of this external field or method. The Signature field is a byte index into the #Blob stream, which contains either the type of the external field or the parameters and return type for the external method.

Offset	**RVA**	**Signature**
64a	244a	002b
64c	244c	002f
64e	244e	002b

Table C.33: StandAloneSig table

Next is the StandAloneSig Table (see Table C.33). Each row of the StandAlongSig Table is referenced by a Method Header. The Signature field is a byte index into the #Blob stream, which contains the encoded types of the local variables in a method.

Offset	RVA	Hash	Major	Minor	Build	Rev	Flags	Public	Name	Culture
650	2450	0000 8004	0000	0000	0000	0000	0000 0000	0000	0073	0000

Table C.34: Assembly table

The Assembly Table (see Table C.34) is next. The Assembly Table always has one row and describes the assembly it is contained in. Therefore, row one of the Assembly Table describes the BubbleSort assembly. The Hash field contains a constant that describes what hashing algorithm was used to build the Strong Name for this assembly. Since the BubbleSort assembly does not have a Strong Name, this value is not set. The absence for a Strong Name is also the reason there is no Major, Minor, Build and Revision numbers for this assembly.

The Flags field is a bitset containing flags about the assembly. Some of these flags are DisableJITCompilerOptimizer (0x4000) and PublicKey (0x0001), which means this assembly has a Public/Private key pair [37]. In this case none of the flags in the Flags field are set for the BubbleSort assembly.

Offset	RVA	Major	Minor	Build	Rev	Flags	Public	Name	Culture	Hash
666	2466	0001	0000	1388	0000	0000 0000	0005	0001	0000	0000

Table C.35: AssemblyRef table

The Public field is a byte index into the #Blob heap, which contains the public key for this assembly. The Name field is a byte index into the #Strings stream, which contains the ASCII name of this assembly. The Culture field is a byte index into the #Strings stream, which contains the ASCII representation of language used in this assembly.

Offset	RVA	Value	String
67c	247c	00	
67d	247d	6d73 636f 726c 6962 00	mscorlib
686	2486	4f62 6a65 6374 00	Object
68d	248d	5379 7374 656d 00	System
694	2494	2e63 746f 7200	.ctor
69a	249a	4172 7261 794c 6973 7400	ArrayList
6a4	24a4	5379 7374 656d 2e43 6f6c 6c65 6374 696f 6e73 00	System.Collections
6b7	24b7	496e 7433 3200	Int32
6bd	24bd	4164 6400	Add
6c1	24c1	6765 745f 4974 656d 00	get_Item
6ca	24ca	436f 6e73 6f6c 6500	Console
6d2	24d2	5772 6974 654c 696e 6500	WriteLine
6dc	24dc	6765 745f 436f 756e 7400	get_Count
6e6	24e6	7365 745f 4974 656d 00	set_Item
6ef	24ef	4275 6262 6c65 536f 7274 00	BubbleSort
6fa	24fa	3c4d 6f64 756c 653e 00	¡Module¿
703	2503	4275 6262 6c65 536f 7274 2e65 7865 00	BubbleSort.exe
712	2512	6e75 6d73 00	nums
717	2517	646f 4275 6262 6c65 536f 7274 00	doBubbleSort
724	2524	7377 6170 00	swap
729	2529	6669 7273 7400	first
72f	252f	7365 636f 6e64 00	second
736	2536	4d61 696e 00	Main
73b	253b	00	

Table C.36: #Strings stream

Offset	RVA	Value	String
73c	253c	00	
73d	253d	00	
73e	253e	00	
73f	253f	00	

Table C.37: #US stream

The AssemblyRef Table (see Table C.35) is next. Each row in the AssemblyRef Table defines an external assembly that is referenced by this assembly. The Major, Minor, Build and Revision fields give the versioning information about the external assembly. The Flags field contains the same type of flags that the Assembly Table (see Table C.34) has. The Public field is a byte index into the #Blob stream, which contains the assemblies public key. The Culture field is a byte index into the #Strings stream, which contains the ASCII representation of the local language or geographic area used in the assembly. The Hash

field field is a byte index into the #Blob stream, which contains the hash for the assembly that is referenced. At the end of the AssemblyRef Table is two bytes of zero padding.

Next is the #Strings stream (see Table C.36) starting at RVA 0x247c. The #Strings stream is a byte stream of ASCII strings separated by the null character. The #Strings stream is used by many parts of the assembly including the metadata sections.

Next is the #US stream (see Table C.37) starting at RVA 0x253c. The #US (or User String) stream is a byte stream of ASCII strings that were string constants used by the programmer in the original source code. Since there were no string constants in the C# code in Appendix A, the #US stream is empty. The four null bytes were added because the size of each stream must be a multiple of four.

Offset	**RVA**	**Byte(s)**
740	2540	00
741	2541	0320 0001
745	2545	08b7 7a5c 5619 34e0 89
74e	254e	0420 0108 1c
753	2553	0420 011c 08
758	2558	0400 0101 1c
75d	255d	0320 0008
761	2561	0520 0201 081c
767	2567	0306 1209
76b	256b	0307 0108
76f	256f	0507 0308 0808
775	2575	0520 0201 0808
77b	257b	0300 0001
77f	257f	00

Table C.38: #Blob stream

The #Blob stream is next (see Table C.38) starting at RVA 0x2540. The #Blob stream contains encoded type information about method parameters, fields and local variables. Take the swap method for example (see Appendix A). It takes two int's as parameters and returns void. The swap method is represented by the third row in the MethodDef table (see Table C.30). Note the value of the name column, 0xa8. This offset points to offset 0x724 (0x67c + 0xa8 = 0x724) in the #Strings stream (see Table C.36). The ASCII string

"swap" is at this offset into the #Strings stream. The ParamList in the same row of the MethodDef table contains the value 0x35. This is an offset into the #Blob stream that contains the parameters and return types for the swap method. At offset 0x775 (0x740 + 0x35 = 0x775) in the #Blob stream are these encoded types. The first byte, which is 0x05, is the length of the #Blob entry minus the size byte itself.

The next byte contains calling convention flags. There are two mutually exclusive calling convention flags: DEFAULT (0x00) and VARARG (0x05). The DEFAULT flag is set when a method passes it's parameters by pushing them onto the stack before calling the method. The VARARG flag is used when a method has a variable number of arguments. Special CIL instructions are used to put and get the parameters on and off the stack when the VARARG flag is set. There is only one method type flag: HASTHIS (0x20). The presence of this flag means that this is an instance method. If the HASTHIS flag is not set, then this is a static method. This method has the DEFAULT and HASTHIS flags set. This method is called in the default way.

The next byte contains the number of parameters this method has. In this case, swap has 0x02 parameters. Next is the encoded type for the methods return value. The value 0x01 (ELEMENT_TYPE_VOID) is the magic number for a void method. Next comes an encoded type for each of the number of parameters read earlier. In this case the next two encoded types are both ELEMENT_TYPE_I4 (0x08), which is the int32 value type [37].

Offset	**RVA**	**GUID**
780	2580	5a64 03bf 392f 0846 95a9 37ad 5959 531b

Table C.39: #GUID stream

The #GUID stream (see Table C.39) is next starting at RVA 0x2580. There is always at least one Global Unique Identifier (GUID) in the #GUID stream because it must store one GUID for the current assembly. The GUID is used to uniquely identify an

assembly. The rest of the .text section (offset 0x790 to 0x800) is padded with zeros so that the .text section is aligned according to the File Alignment field in the NT Specific Table (see Table C.5).

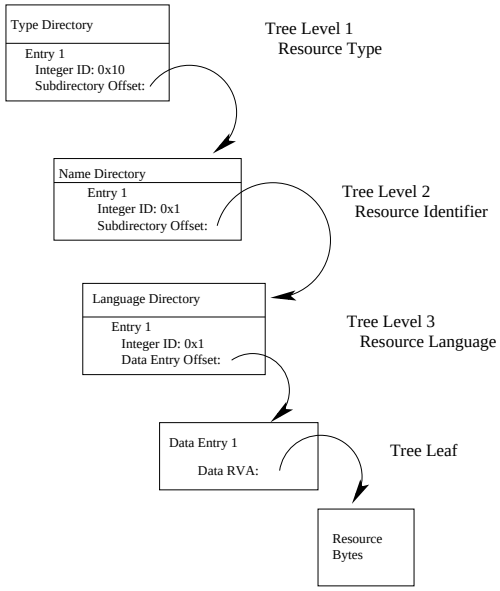

Figure C.3: Overview of .rsrc section

The .rsrc section is next starting at offset 0x800 and RVA 0x4000. The .rsrc section contains unmanaged resources. If the Bubble Sort example had any managed resources, which are managed by the virtual machine instead of the operating system, it would be stored in the .text section. Resources in the .rsrc are characterized by three values: type, name and language. These resources are stored in a tree that always has a depth of three where each level of the tree represents one of the three characteristics mentioned earlier (see Figure C.3). Each layer of the tree points to the next layer of the tree. Layer one contains information about the resource type. Layer two contains a resource identifier in case there is more than one resource of the same type. Layer three contains language information about the resource [37][26]. Resources in the tree are represented by following the path in the tree until you get to the leaf that points to the resource. For example, Figure C.3 has one resource represented by Type 0x10, Identifier 0x1, and Language 0x1.

Offset	RVA	Name	Value
800	4000	Characteristics	0000 0000
804	4004	Time/Date Stamp	0000 0000
808	4008	Major Version	0000
80a	400a	Minor Version	0000
80c	400c	Number of Name Entries	0000
80e	400e	Number of ID Entries	0001

Table C.40: Type directory

Offset	RVA	Name	Value
810	4010	Integer ID	0000 0010
814	4014	Subdirectory Offset	8000 0018

Table C.41: Entry 1

Offset	RVA	Name	Value
818	4018	Characteristics	0000 0000
81c	401c	Time/Date Stamp	0000 0000
820	4020	Major Version	0000
822	4022	Minor Version	0000
824	4024	Number of Name Entries	0000
826	4026	Number of ID Entries	0001

Table C.42: Name directory

Offset	RVA	Name	Value
828	4028	Integer ID	0000 0001
82c	402c	Subdirectory Offset	8000 0030

Table C.43: Entry 1

Table C.40 contains the type information for the one resource in the .rsrc section. The Characteristics field contains no flags and is reserved for future use. The Time/Date Stamp field contains the number of seconds since 00:00:00 UTC on January 1, 1970. In this case this field was not used. The Major and Minor Version field are used to version resources, but are not used here. Each Resource Directory Table contains one or more Entries. These Entries can be either identified by names or ids. The Number of Name Entries field contains the number of Entries identified by names. The Number of ID Entries field contains the number of Entries identified by ids. Entries contain either a name or id to distinguish them from other Entries.

Table C.41 contains the id 0x10. 0x10 is the magic number for the RT_VERSION type of resource that contains version information about the file that was embedded by the Mono C# compiler that was used to compile the example code in Appendix A. If bit 31 of the Subdirectory Offset field is a one, it contains an offset from the beginning of the .rsrc

Offset	RVA	Name	Value
830	4030	Characteristics	0000 0000
834	4034	Time/Date Stamp	0000 0000
838	4038	Major Version	0000
83a	403a	Minor Version	0000
83c	403c	Number of Name Entries	0000
83e	403e	Number of ID Entries	0001

Table C.44: Language directory

Offset	RVA	Name	Value
840	4040	Integer ID	0000 0001
844	4044	Data Entry Offset	0000 0048

Table C.45: Entry 1

section to the next layer of the tree. If the same bit is a zero, it points to a Data Entry leaf [37]. Therefore this Subdirectory Offset field points to the second layer of the tree since offset 0x800 plus 0x18 equals offset 0x818.

Table C.42 contains identifier information about a resource. The fields in this table are treated the same as the one in Table C.40. The one Entry (see Table C.43) contains the ID 0x1. The Subdirectory Offset points to the third layer of the tree.

Offset	RVA	Name	Value
848	4048	Data RVA	0000 4058
84c	404c	Size	0000 028c
850	4050	Code Page	0000 0000
854	4054	Reserved	0000 0000

Table C.46: Data entry 1

Offset	RVA	Name	Value
858	4058	wLength	028c
85a	405a	wValueLength	0034
85c	405c	wType	0000
85e	405e	szKey	0056 0053 005f 0056 0045 0052 0053 0049 004f 004e 005f 0049 004e 0046 004f 0000
87e	407e	Padding1	0000

Table C.47: VS_VERSIONINFO structure

Table C.44 contains language information about the resource. The fields in Table C.44 are treated the same as Table C.40. Layer three only has one Entry (see Table C.45)

and contains the language ID 0x1. Since bit 31 is not set in the Data Entry Offset field, it points to a Data Entry leaf in the tree.

Table C.46 is the leaf in the tree that points to the raw bytes of the one resource in the .rsrc section. The Data RVA field contains the RVA of where the resource bytes begin. The Size field contains the number of bytes in the resource. The Code Page field is used to decode resources and not used in this example. The Reserved field is not used and reserved for future use.

Offset	**RVA**	**Name**	**Value**
880	4080	dwSignature	feef 04bd
884	4084	dwStrucVersion	0000 0001
888	4088	dwFileVersionMS	0000 0000
88c	408c	dwFileVersionLS	0000 0000
890	4090	dwProductVersionMS	0000 0000
894	4094	dwProductVersionLS	0000 0000
898	4098	dwFileFlagsMask	0000 003f
89c	409c	dwFileFlags	0000 0000
8a0	40a0	dwFileOS	0000 0004
8a4	40a4	dwFileType	0000 0002
8a8	40a8	dwFileSubtype	0000 0000
8ac	40ac	dwFileDateMS	0000 0000
8b0	40b0	dwFileDateLS	0000 0000

Table C.48: VS_FIXEDFILEINFO structure

Offset	**RVA**	**Name**	**Value**
8b4	40b4	wLength	0044
8b6	40b6	wValueLength	0000
8b8	40b8	wType	0001
8ba	40ba	szKey	0056 0061 0072 0046 0069 006c 0065 0049 006e 0066 006f 0000
8d2	40d2	Padding	0000

Table C.49: VarFileInfo structure

Offset	**RVA**	**Name**	**Value**
8d4	40d4	wLength	0024
8d6	40d6	wValueLength	0004
8d8	40d8	wType	0000
8da	40da	szKey	0054 0072 0061 006e 0073 006c 0061 0074 0069 006f 006e 0000
8f2	40f2	Padding	0000
8f4	40f4	Language ID	04b0 007f

Table C.50: Var structure

Offset	RVA	Name	Value
8f8	40f8	wLength	01ec
8fa	40fa	wValueLength	0000
8fc	40fc	wType	0001
8fe	40fe	szKey	0053 0074 0072 0069 006e 0067 0046 0069 006c 0065 0049 006e 0066 006f 0000

Table C.51: StringFileInfo structure

Offset	RVA	Name	Value
91c	411c	wLength	01c8
91e	411e	wValueLength	0000
920	4120	wType	0001
922	4122	szKey	0003 0003 0037 0066 0003 0034 0062 0003
932	4132	Padding	0000

Table C.52: StringTable structure

Offset	RVA	Name	Value
934	4134	wLength	0028
936	4136	wValueLength	0002
938	4138	wType	0001
93a	413a	szKey	0050 0072 006f 0064 0075 0063 0074 0056 0065 0072 0073 0069 006f 006e 0000
958	4158	Value	0020 0000

Table C.53: String structure 1

Offset	RVA	Name	Value
95c	415c	wLength	0024
95e	415e	wValueLength	0002
960	4160	wType	0001
962	4162	szKey	0043 006f 006d 0070 0061 006e 0079 004e 0061 006d 0065 0000
97a	417a	Padding	0000
97c	417c	Value	0020 0000

Table C.54: String structure 2

Offset	RVA	Name	Value
980	4180	wLength	0024
982	4182	wValueLength	0002
984	4184	wType	0001
986	4186	szKey	0050 0072 006f 0064 0075 0063 0074 004e 0061 006d 0065 0000
99e	419e	Padding	0000
9a0	41a0	Value	0020 0000

Table C.55: String structure 3

Offset	RVA	Name	Value
9a4	41a4	wLength	0028
9a6	41a6	wValueLength	0002
9a8	41a8	wType	0001
9aa	41aa	szKey	004c 0065 0067 0061 006c 0043 006f 0070 0079 0072 0069 0067 0068 0074 0000
9c8	41c8	Value	0020 0000

Table C.56: String structure 4

Offset	RVA	Name	Value
9cc	41cc	wLength	0038
9ce	41ce	wValueLength	000b
9d0	41d0	wType	0001
9d2	41d2	szKey	0049 006e 0074 0065 0072 006e 0061 006c 004e 0061 006d 0065 0000
9ec	41ec	Value	0042 0075 0062 0062 006c 0065 0053 006f 0072 0074 0000 0000

Table C.57: String structure 5

Offset	RVA	Name	Value
a04	4204	wLength	002c
a06	4206	wValueLength	0002
a08	4208	wType	0001
a0a	420a	szKey	0046 0069 006c 0065 0044 0065 0073 0063 0072 0069 0070 0074 0069 006f 006e 0000
a2a	422a	Padding	0000
a2c	422c	Value	0020 0000

Table C.58: String structure 6

Offset	RVA	Name	Value
a30	4230	wLength	001c
a32	4232	wValueLength	0002
a34	4234	wType	0001
a36	4236	szKey	0043 006f 006d 006d 0065 006e 0074 0073 0000
a48	4248	Value	0020 0000

Table C.59: String structure 7

Offset	RVA	Name	Value
a4c	424c	wLength	0024
a4e	424e	wValueLength	0002
a50	4250	wType	0001
a52	4252	szKey	0046 0069 006c 0065 0056 0065 0072 0073 0069 006f 006e 0000
a6a	426a	Padding	0000
a6c	426c	Value	0020 0000

Table C.60: String structure 8

The one resource in the .rsrc section is represented as a VS_VERSIONINFO [35] structure. The VS_VERSIONINFO structure gives information like company name, file version, and product name about a file in unicode. The VS_VERSIONINFO structure is composed of VS_FIXEDFILEINFO [34], VarFileInfo [33], Var [32], StringFileInfo [30], StringTable [31], and String [29] structures that were parsed into Tables C.47 to C.62 for this example. See Microsoft's Developer Network for more information [35][34][33][32][30][31][29]. The rest of the .rsrc section, from offset 0xae4 to 0xbff, is padded with zero bytes.

Next is the last section in the assembly, which is the .reloc section. The .reloc section starts at offset 0xc00 and RVA 0x6000. The .reloc section is a collection of Fix Up

Offset	RVA	Name	Value
a70	4270	wLength	0048
a72	4272	wValueLength	000f
a74	4274	wType	0001
a76	4276	szKey	004f 0072 0069 0067 0069 006e 0061 006c 0046 0069 006c 0065 006e 0061 006d 0065 0000
a98	4298	Value	0042 0075 0062 0062 006c 0065 0053 006f 0072 0074 002e 0065 0078 0065 0000 0000

Table C.61: String structure 9

Offset	RVA	Name	Value
ab8	42b8	wLength	002c
aba	42ba	wValueLength	0002
abc	42bc	wType	0001
abe	42be	szKey	004c 0065 0067 0061 006c 0054 0072 0061 0064 0065 006d 0061 0072 006b 0073 0000
ade	42de	Padding	0000
ae0	42e0	Value	0020 0000

Table C.62: String structure 10

Offset	RVA	Name	Value
c00	6000	Page RVA	0000 2000
c04	6004	Block Size	0000 000c
c08	6008	Type/Offset Entries	3002
c0a	600a	Type/Offset Entries	0000

Table C.63: Fix Up 1

Tables, one after another. The .reloc section in this example only has one Fix Up Table (see Table C.63). The Page RVA gives the RVA of the section to relocate. In this example, RVA 0x0000 2000 is the .text section. The Block Size field gives the size of this Fix Up Table in bytes. Following the Block Size field is one or more Type/Offset Entries, which are two bytes each. To find the number of Type/Offset Entries, subtract eight from the Block Size and divide by two. In this case, there is two Type/Offset Entry ((0xc - 0x8)/0x2 = 2).

Each Type/Offset Entry contains a Fix Up type and an offset to apply to the section. Bits 1 to 4 of the Type/Offset Entry contains the Fix Up type. The first Type/Offset Entry has type B0010, which is the IMAGE_REL_BASED_LOW type. The

IMAGE_REL_BASED_LOW type calculates a delta by taking the difference of the Image Base field in Table C.5 and the offset of the section to relocate. If this delta is zero, this section fix up is skipped. If it is not zero, the low 16 bits of the delta are added to the offset in the Type/Offset Entry after the type is masked out. Then the lower 16 bits of the result are added to the RVA of the section to be relocated. Therefore, the delta for this Type/Offset is 0x003f e000 (0x0040 0000 - 0x2000). Since the delta is not zero, the lower 16 bits of this delta (0xe000) is added to the Type/Offset Entry after the type is masked out (0x3000) and the result is 0x1 0000. The lower 16 bits of this value is added to the current offset and then the section is relocated there. Therefore the .text section is relocated from where it is now at RVA 0x2000 to 0x2000 plus 0x0000. In the end the .text section did not move because the lower 16 bits of the previous result was 0x0000. Type/Offset Entry two at offset 0xc08 has type B0000, which means this section fix up is skipped. The rest of the .reloc section, from offset 0xc0c to 0xdff, is padded with zero bytes. That ends the explanation of the bytes in Appendix B generated by compiling the code in Appendix A.

Vita

Shawn H. Windle was born in New Orleans, Louisiana on August 23, 1976. He graduated from Warner Robins High School in Warner Robins, Georgia in 1994. He graduated from Lees-McRae College in 2003 with a Bachelor of Science degree in Computer Information Systems with a concentration in Mathematics. He entered Appalachian State University in 2005 to get a Master of Science degree in Computer Science. During his Computer Science studies, he received the NSF-funded CSEMS scholarship. He was also a Graduate Teaching Assistant for Professor Kenneth Jacker in his Spring 2005 Introduction To Computer Systems class. Currently he works as a programmer and system administrator at KSPCS in Boone, North Carolina. He received his Master of Science degree in Computer Science in December of 2012.