JaCIL: a CLI to JVM Compiler

Almann Goo

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
JaCIL: A CLI to JVM Compiler  
Master’s Project Proposal  

Almann T. Goo  
atg2335@cs.rit.edu  
Rochester Institute of Technology  
Department of Computer Science  

March 30, 2006  

Abstract  
Both the .NET Framework via the Common Language Infrastructure (CLI) [16] and the Java Virtual Machine (JVM) [15] provide for a managed, virtual execution environment for running high-level virtual machine code. As these two platforms become ubiquitous, it is of pragmatic and academic interest to have the capability of running CLI code on the JVM and vice-versa. The author proposes a project, named JaCIL (pronounced “jackal”), to create a byte-code compiler to translate from CLI binaries to binaries suitable for consumption by the JVM.

1 Introduction  
The concept of a Virtual Machine (VM) is far from a new concept, but undoubtedly with the advent of Sun’s Java Platform it has become far more popular as application code can be compiled to an platform independent image and executed on any machine that has a JVM implementation [12]. Microsoft has since introduced its own competing .NET Framework platform that has similar aims.

With the influence that a major company like Microsoft has, it would seem destined that its platform shall too rise in popularity creating two sets of applications and libraries that exist in one platform or another. Even though both platforms have strikingly similar features [12], they are not binary compatible with one another and thus “platform independent” code from one execution environment is unavailable to another.

Since each platform itself can be viewed as another machine, albeit much more high-level than a physical machine, it would seem reasonable then
that an implementation of either platform could be created on the other. This has in fact been done in the IKVM.NET [11] project with the JVM on the CLI—I discuss this in section 2.2.2. With that said, the author knows of no such implementation for other direction—a CLI implementation on the JVM.

1.1 The Java Virtual Machine

As many have stated, the JVM was developed as an execution environment for the Java Programming Language [12, 18, 13]. The JVM is an abstract stack-based machine with some of its major characteristics noted below: [15]

- Direct operations for primitive integers.
- A Notion of a reference which is the address, but no way for primitives to manipulate the value of such an address (i.e. taking the address of a local or incrementing such an address).
- A run-time heap shared by all program threads that are where all objects are allocated, complex types are never allocated in a stack frame.
- A private per-thread stack to store frames of method executions; containing an activation’s operand stack and local variables.

JVM class files, the binary image of a loadable component (class), contain various meta-data that define the class, its fields, and its methods. This structure also defines symbolic information accessed by byte-code instructions in the methods and the byte-code instructions themselves [15].

1.2 The Common Language Infrastructure

The CLI, from a high level, shares much in common with the description given of the JVM in section 1.1. It too has a stack-based architecture but has features that make it more “friendly” for various programming language paradigms to target [14]. Given, its high-level similarities, some of the compelling differences are highlighted below: [9, 16, 12]

- The CLI instruction set tends to be more generalized for operations such as arithmetic—most equivalent JVM instructions have the type information embedded in them.
The CLI supports user defined value types that can be allocated directly on the stack or directly within another object.

The concept of a managed pointer which is like a JVM reference address, but the Instruction Set Architecture (ISA) provides primitives to manipulate them to some degree. More specifically, the CLI allows for instructions that take an address of a local variable, argument, field, or method and perform indirection on such a pointer.

Support for tail calls. The CLI provides for invoking a method and discarding the stack frame of the caller. This is essential for languages that depend on tail recursion as a looping mechanism (i.e. Scheme).

CLI assemblies are the binary image of a loadable component in the CLI. Unlike their Java equivalent, they can consist of numerous types (classes) and contain global meta-data as well as meta-data for each class [9]. In fact it is not difficult to draw the parallel between Java Archive (JAR) files and CLI assemblies [16]. CLI assemblies, much like their Java analog contain similar types of meta-data.

2 Project Definition

The author proposes to develop a byte-code compiler to translate CLI compiled assemblies into equivalent JVM class files. The proposed system would translate types (e.g. classes), fields, methods, and appropriate meta-data defined in a CLI assembly into the equivalent JVM analog. The proposed system would also translate the Common Intermediate Language (CIL) code—the byte-code language of the CLI—that comprises the code within methods of a given assembly into semantically equivalent JVM byte-code instructions.

Since there are invariably constructs within the CLR that have no direct semantic equivalent in the JVM, a Java “hosting” library will also be designed and implemented to accommodate those semantics.

2.1 Project Scope

The proposed project consists of a designing and implementing a system comprising of the following components.

- A compiler facility to translate CLI assemblies to JVM class files.
- Provide an application to provide access to the compiler as a stand-alone, Ahead-Of-Time (AOT) compilation tool. This would be the most common way to access the compiler.
- Provide an API to the compiler to allow embedding of the compiler into other applications. This would potentially allow for Just-In-Time (JIT) compiler implementations to leverage this system in the future.

- A thin run-time “hosting” library for the JVM to provide compatibility to CLI semantics. Including, but not necessarily limited to the following:
  - Method compatibility for CLI primitive types such as integers and strings.
  - Base object class compatibility.
  - Compatibility for implementing CLI semantics that have no direct JVM equivalent. Some examples of this include the following.
    * Method pointers and indirect method calls.
    * Field, array element, and local variable pointers and de-referencing operations.
    * User defined value types that are allocated directly on the stack or in heap objects.
    * Non-virtual methods.

Please see section 3.1 for functional details of CLI features to be supported, section 3.2 for functional details on the compiler, and section 3.3 for functional details on the run-time component.

## 2.2 Related Work

The following subsections describe related work and their applicability to this project.

### 2.2.1 Previous Translation Efforts and Platform Comparisons

Perhaps the most prominent work in translating from the CLI to the JVM comes from Shiel and Bayley who have actually implemented a rudimentary translator from the CLI to the JVM [17]. The intent of their work was to discuss how the semantics of the CLI compare with the JVM and to use this study as a basis for formalizing CLI semantics [17].

As this paper gives a good portion of the CLI semantics that need to be translated, Shiel and Bayley’s work shall be useful in defining this project’s translational semantics between the CLI and the JVM.
2.2.2 IKVM.NET Java Virtual Machine

IKVM.NET is an open source implementation of the JVM on the CLI. It provides an VM implementation with a JVM to CLI JIT compiler, Java Standard API (via GNU Classpath), and support tools like an AOT compiler and a tool analog of the `java` command [11].

The IKVM.NET JVM can be leveraged by accessing well established Java APIs in the CLI. In practical terms, this will allow the author to utilize existing byte-code tooling libraries such as the ObjectWeb ASM Framework (ASM) [6], the Jakarta Byte Code Engineering Library [2], or the Serp Framework [7] to generate the JVM byte-code for the compiler.

2.2.3 Visual MainWin for J2EE

Visual MainWin for J2EE is a commercial product that consists of an AOT byte-code compiler from CLI assemblies to JVM class files, a run-time layer, and an implementation of the .NET Framework Standard API (via the Mono Project) [8].

Unfortunately, not much can be leveraged from this product because it is a closed, proprietary product. Also, the target audience of this product seems to be Java 2 Enterprise Edition (J2EE) integration from Visual Studio [8], so it is not clear how general purpose the product aims to be.

The fact that there is a commercial product that implements a system that may be close to a full CLI implementation for the JVM indicates promise for this project.

2.2.4 Retroweaver

This project provides a byte-code converter for Java 1.5 byte-code to Java 1.4 compatible byte-code. This project also provides a thin compatibility run-time to provide emulation of semantics not natively present on legacy JVM implementations [1]. Retroweaver is similar to the proposed project in that its goals are parallel—the key difference, of course, is that it is a JVM-to-JVM translation system.

Despite the Java only nature of this project, the work the project has done to provide compatibility on legacy VM–namely the compatibility layer–can be gleaned for insight.
2.2.5 The Mono and DotGNU Projects

Both the Mono [5] and the DotGNU [3] projects are open source projects providing an implementation of the .NET Framework including the CLI and the .NET Framework Standard API.

These projects point out other implementations of the CLI other than that of Microsoft and could be leveraged in future directions of this project. Specifically, having a working CLI to JVM byte-code compiler would enable the CLI byte-code portions of the libraries (which is likely most of it) to be converted to JVM byte-code and thus eliminating the need to directly implement a considerable portion of the CLI Standard API in Java. This would leave only the “native” portions of the CLI Standard API to port to Java—the end result being nearly a complete CLI implementation in Java.

This is in fact the strategy the IKVM.NET JVM implementation employs by using a statically JVM to CLI byte-code compiled version of the GNU Classpath Java Standard API [11, 4].

3 Functional Specification and Design

The following subsections describe a preliminary functional specification and design for this project.

3.1 Supported CLI Features

This section describes the explicit features of the CLI that shall be implemented by JaCIL. The features listed with subsections of this section are a minimal set of functionality that shall be supported; the implementation may in fact support more functionality than what is listed here. Furthermore, due to architectural differences between the CLI and the JVM, the project’s implementation may have compatibility issues to a feature that is supported. These compatibility issues with the implementation shall be specified in user documentation.

The following subsections follow closely in structural order to topics listed in Partitions II and III of ECMA-335, Common Language Infrastructure (CLI) Specification [9].
3.1.1 Assemblies

As stated earlier, a deployable unit in the CLI is an assembly. This project shall support assemblies in so far as their usage as a collection of types is concerned. The JaCIL Compiler shall not be required to translate assembly meta-data (i.e. attributes).

The CLI provides for top-level global fields and methods; that is, methods and fields that do not have an enclosing class [9]. JaCIL shall not be required to support this feature.

Any support for assembly meta-data that is provided by the implementation shall be noted in the final specification and/or user documentation as appropriate.

3.1.2 Types

The project shall support at a minimum the following subset built-in CLI types [9].

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Boolean</td>
<td>The boolean type.</td>
</tr>
<tr>
<td>System.Char</td>
<td>A 16-bit Unicode character.</td>
</tr>
<tr>
<td>System.Single</td>
<td>A 32-bit floating point number.</td>
</tr>
<tr>
<td>System.Double</td>
<td>A 64-bit floating point number.</td>
</tr>
<tr>
<td>System.Byte</td>
<td>An unsigned 8-bit integer.</td>
</tr>
<tr>
<td>System.SByte</td>
<td>An signed 8-bit integer.</td>
</tr>
<tr>
<td>System.Int16</td>
<td>A signed 16-bit integer.</td>
</tr>
<tr>
<td>System.Int32</td>
<td>A signed 32-bit integer.</td>
</tr>
<tr>
<td>System.Int64</td>
<td>A signed 64-bit integer.</td>
</tr>
<tr>
<td>System.Object</td>
<td>An object type.</td>
</tr>
<tr>
<td>System.String</td>
<td>A string type.</td>
</tr>
<tr>
<td>System.ValueType</td>
<td>A valuetype type.</td>
</tr>
<tr>
<td>type[]</td>
<td>A vector type.</td>
</tr>
<tr>
<td>type &amp;</td>
<td>A managed pointer type.</td>
</tr>
</tbody>
</table>

JaCIL shall only support zero-based index, single dimensional arrays—called vectors in the CLI. Implicitly, this means that jagged arrays (vectors of vectors) are supported. This does however mean that the project shall not be required to support CLI multi-dimensional arrays or arrays with non-zero lower bound.
3.1.3 User Defined Types

JaCIL shall support assembly definitions of user defined classes, interfaces, and value types. The implementation shall not be required to support inner classes or generics.

Type Definitions The only CLI attributes of these definitions that the system shall be required to support are the following CLI class attributes [9].

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>The defined type is abstract.</td>
</tr>
<tr>
<td>interface</td>
<td>The defined type is an interface.</td>
</tr>
<tr>
<td>sealed</td>
<td>The defined type cannot be derived.</td>
</tr>
<tr>
<td>public</td>
<td>The defined type is accessible outside of the assembly. See below.</td>
</tr>
<tr>
<td>private</td>
<td>The defined type is only accessible within the assembly. See below.</td>
</tr>
</tbody>
</table>

The system shall translate the access modifiers, public and private, as public in the translated JVM type. The rationale for this is that closest semantic for assembly private is package protected which is too granular assembly private is package which is too granular name spaces which are the CLI equivalent of JVM packages).

Within a user defined type, JaCIL shall support the declaration of methods and fields. The extent of support of these features is described below.

Method Definitions JaCIL shall support methods that take as parameters any supported type including user defined types. The system shall also support the return of any supported type including user defined types and void (no return value).

The system shall support the following method attributes for method declarations [9].
The system shall translate any method accessibility attributes that rely on assembly or compilation unit accessibility to public in the JVM. The rationale for this is the same as was stated for Type Declarations.

Furthermore, JaCIL shall support non-virtual methods, which are non-static methods that are defined without the virtual attribute.

The CLI defines calling conventions that are defined for method declarations [9]. This system shall support the instance and instance explicit modifiers that indicates that the method is an instance method and the semantics for the this argument. The only CLI calling kind that the system is required to support is the default one.

Methods in the CLI also specify implementation attributes; the following are supported by the translator [9].

<table>
<thead>
<tr>
<th>Accessibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>The defined method is abstract.</td>
</tr>
<tr>
<td>final</td>
<td>The defined virtual method cannot be overridden in a derived type.</td>
</tr>
<tr>
<td>virtual</td>
<td>The defined method is virtual.</td>
</tr>
<tr>
<td>static</td>
<td>The defined method is static.</td>
</tr>
<tr>
<td>public</td>
<td>The defined method is publicly accessible.</td>
</tr>
<tr>
<td>private</td>
<td>The defined method is accessible only to the type.</td>
</tr>
<tr>
<td>family</td>
<td>The defined method is accessible only to the type and its derived types. See below.</td>
</tr>
<tr>
<td>assembly</td>
<td>The defined method is accessible by the assembly. See below.</td>
</tr>
<tr>
<td>famorassem</td>
<td>The defined method is accessible by the assembly or a derived type. See below.</td>
</tr>
<tr>
<td>famandalassem</td>
<td>The defined method is accessible only to a derived type in the assembly. See below.</td>
</tr>
<tr>
<td>compilercontrolled</td>
<td>The defined method is accessible only to a compilation unit (e.g. a module). See below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cil</td>
<td>The method is implemented using CIL byte-code.</td>
</tr>
<tr>
<td>managed</td>
<td>The method should be managed by the CLI.</td>
</tr>
<tr>
<td>synchronized</td>
<td>The method should be synchronized on the instance.</td>
</tr>
</tbody>
</table>
JaCIL can only translate methods that have both the cil and managed implementation attributes. The JaCIL compiler shall emit an error if either one of these is not specified. This also implies that the system shall not be required to support native and/or unmanaged methods.

**Field Definitions**  JaCIL shall support fields of any supported type including user-defined types.

The following field attributes shall be supported by the system [9].

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initonly</td>
<td>The defined field cannot be modified after initialization.</td>
</tr>
<tr>
<td>literal</td>
<td>The defined field is a constant literal.</td>
</tr>
<tr>
<td>static</td>
<td>The defined field is static.</td>
</tr>
<tr>
<td>public</td>
<td>The defined field is publicly accessible.</td>
</tr>
<tr>
<td>private</td>
<td>The defined field is accessible only to the type.</td>
</tr>
<tr>
<td>family</td>
<td>The defined field is accessible only to the type and its derived types.</td>
</tr>
<tr>
<td></td>
<td>See below.</td>
</tr>
<tr>
<td>assembly</td>
<td>The defined field is accessible by the assembly.</td>
</tr>
<tr>
<td></td>
<td>See below.</td>
</tr>
<tr>
<td>famorassem</td>
<td>The defined is accessible by the assembly or a derived type.</td>
</tr>
<tr>
<td></td>
<td>See below.</td>
</tr>
<tr>
<td>famandassem</td>
<td>The defined field is accessible only to a derived type in the assembly.</td>
</tr>
<tr>
<td></td>
<td>See below.</td>
</tr>
<tr>
<td>compilercontrolled</td>
<td>The defined field is accessible only to a compilation unit (e.g. a module).</td>
</tr>
<tr>
<td></td>
<td>See below.</td>
</tr>
</tbody>
</table>

As with method access modifiers, support for field access modifiers that involve assembly or module access are to be translated as public for the reasons stated earlier in this section.

### 3.1.4 Exception Handling

JaCIL shall support CLI protected blocks (which are called try blocks in Java). Regarding CLI handlers for such blocks, the catch handler and finally handler shall be implementated by the system.
3.1.5 General CIL Support

The CLI defines managed program code to be written in Common Intermediate Language (CIL) byte-code instructions. Much like the JVM, it is possible to write code that is type-safe and not verifiable [9]. Even though it is legal to write unsafe code in the CLI, a semantic that does not really have a Java equivalent, this project shall not support operations that are not verifiable, as defined by the CLI specification.

The following paragraphs specify to what extent the CIL is supported by the project.

CIL Prefixes  The CLI supports special instruction prefix opcodes that can be used before certain instructions [9]. JaCIL shall not be required to support any of these.

CIL Base Instructions  The system shall be required to support the translation of the following base instructions defined by the specification [9].
<table>
<thead>
<tr>
<th>0x58</th>
<th>add</th>
<th>0x5F</th>
<th>and</th>
<th>0x3B</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2E</td>
<td>beq.s</td>
<td>0x3C</td>
<td>bge</td>
<td>0x2F</td>
<td>bge.s</td>
</tr>
<tr>
<td>0x3D</td>
<td>bgt</td>
<td>0x30</td>
<td>bgt.s</td>
<td>0x3E</td>
<td>ble</td>
</tr>
<tr>
<td>0x31</td>
<td>bge.s</td>
<td>0x3F</td>
<td>blt</td>
<td>0x32</td>
<td>blt.s</td>
</tr>
<tr>
<td>0x38</td>
<td>br</td>
<td>0x2B</td>
<td>br.s</td>
<td>0x39</td>
<td>brfalse</td>
</tr>
<tr>
<td>0x2C</td>
<td>brfalse.s</td>
<td>0x3A</td>
<td>brtrue</td>
<td>0x2D</td>
<td>brtrue.s</td>
</tr>
<tr>
<td>0x28</td>
<td>call</td>
<td>0x29</td>
<td>call1</td>
<td>0xFE01</td>
<td>ceq</td>
</tr>
<tr>
<td>0xFE02</td>
<td>cgt</td>
<td>0xC3</td>
<td>ckeq</td>
<td>0xFE04</td>
<td>clt</td>
</tr>
<tr>
<td>0x67</td>
<td>conv.i1</td>
<td>0x68</td>
<td>conv.i2</td>
<td>0x69</td>
<td>conv.i4</td>
</tr>
<tr>
<td>0x6A</td>
<td>conv.i8</td>
<td>0x6B</td>
<td>conv.r4</td>
<td>0x6C</td>
<td>conv.r8</td>
</tr>
<tr>
<td>0x2D</td>
<td>conv.u1</td>
<td>0xD1</td>
<td>conv.u2</td>
<td>0x5B</td>
<td>div</td>
</tr>
<tr>
<td>0x25</td>
<td>dup</td>
<td>0xDC</td>
<td>endfinally</td>
<td>0xFE09</td>
<td>ldarg</td>
</tr>
<tr>
<td>0x0E</td>
<td>ldarg.s</td>
<td>0x02</td>
<td>ldarg.0</td>
<td>0x03</td>
<td>ldarg.1</td>
</tr>
<tr>
<td>0x04</td>
<td>ldarg.2</td>
<td>0x05</td>
<td>ldarg.3</td>
<td>0xFE0A</td>
<td>ldarga</td>
</tr>
<tr>
<td>0x0F</td>
<td>ldarga.s</td>
<td>0x20</td>
<td>ldc.i4</td>
<td>0x21</td>
<td>ldc.i8</td>
</tr>
<tr>
<td>0x22</td>
<td>ldc.r4</td>
<td>0x23</td>
<td>ldc.r8</td>
<td>0x16</td>
<td>ldc.i4.0</td>
</tr>
<tr>
<td>0x17</td>
<td>ldc.i4.1</td>
<td>0x18</td>
<td>ldc.i4.2</td>
<td>0x19</td>
<td>ldc.i4.3</td>
</tr>
<tr>
<td>0x1A</td>
<td>ldc.i4.4</td>
<td>0x1B</td>
<td>ldc.i4.5</td>
<td>0x1C</td>
<td>ldc.i4.6</td>
</tr>
<tr>
<td>0x1D</td>
<td>ldc.i4.7</td>
<td>0x1E</td>
<td>ldc.i4.8</td>
<td>0x15</td>
<td>ldc.i4.m1</td>
</tr>
<tr>
<td>0x1F</td>
<td>ldc.i4.s</td>
<td>0xFE06</td>
<td>ldftn</td>
<td>0x46</td>
<td>ldind.i1</td>
</tr>
<tr>
<td>0x48</td>
<td>ldind.i2</td>
<td>0x4A</td>
<td>ldind.i4</td>
<td>0x4C</td>
<td>ldind.i8</td>
</tr>
<tr>
<td>0x47</td>
<td>ldind.u1</td>
<td>0x49</td>
<td>ldind.u2</td>
<td>0x4E</td>
<td>ldind.r4</td>
</tr>
<tr>
<td>0x4F</td>
<td>ldind.r8</td>
<td>0x50</td>
<td>ldind.ref</td>
<td>0xFE0C</td>
<td>ldloc</td>
</tr>
<tr>
<td>0x11</td>
<td>ldloc.s</td>
<td>0x06</td>
<td>ldloc.0</td>
<td>0x07</td>
<td>ldloc.1</td>
</tr>
<tr>
<td>0x08</td>
<td>ldloc.2</td>
<td>0x09</td>
<td>ldloc.3</td>
<td>0xFE0D</td>
<td>ldloca</td>
</tr>
<tr>
<td>0x12</td>
<td>ldloca.s</td>
<td>0x14</td>
<td>ldnull</td>
<td>0xFD</td>
<td>leave</td>
</tr>
<tr>
<td>0xDE</td>
<td>leave.s</td>
<td>0x5A</td>
<td>mul</td>
<td>0x65</td>
<td>neg</td>
</tr>
<tr>
<td>0x00</td>
<td>nop</td>
<td>0x66</td>
<td>not</td>
<td>0x60</td>
<td>or</td>
</tr>
<tr>
<td>0x26</td>
<td>pop</td>
<td>0x5D</td>
<td>rem</td>
<td>0x2A</td>
<td>ret</td>
</tr>
<tr>
<td>0x62</td>
<td>sh1</td>
<td>0x63</td>
<td>shr</td>
<td>0x64</td>
<td>shr.un</td>
</tr>
<tr>
<td>0xFE0B</td>
<td>starg</td>
<td>0x10</td>
<td>starg.s</td>
<td>0x52</td>
<td>stind.i1</td>
</tr>
<tr>
<td>0x53</td>
<td>stind.i2</td>
<td>0x54</td>
<td>stind.i4</td>
<td>0x55</td>
<td>stind.i8</td>
</tr>
<tr>
<td>0x56</td>
<td>stind.r4</td>
<td>0x57</td>
<td>stind.r8</td>
<td>0x51</td>
<td>stinf.ref</td>
</tr>
<tr>
<td>0xFE0E</td>
<td>stloc</td>
<td>0x13</td>
<td>stloc.s</td>
<td>0x0A</td>
<td>stloc.0</td>
</tr>
<tr>
<td>0x0B</td>
<td>stloc.1</td>
<td>0x0C</td>
<td>stloc.2</td>
<td>0x0D</td>
<td>stloc.3</td>
</tr>
<tr>
<td>0x59</td>
<td>sub</td>
<td>0x45</td>
<td>switch</td>
<td>0x61</td>
<td>xor</td>
</tr>
</tbody>
</table>

**CIL Object Model Instructions**  The system shall be required to support the translation of the following *object model instructions* defined by the specification [9].
3.2 The JaCIL Compiler

The JaCIL Compiler shall consist of a back-end consisting of a compiler framework that defines a specification for programmatic access to it as well as a front-end tool that interfaces with the API.

3.2.1 The JaCIL Compiler Framework

The JaCIL Compiler Framework shall specify an API that provides public access to the compiler from external and internal components. This framework should be flexible such that different “compiler profiles” can be implemented for the framework that can potentially specify different translational semantics for compilation.

The specified public API should be flexible that in-memory as well as file sources can be consumed by the compiler and in-memory as well as file targets can be generated from the compiler. This API should also be able to decide for the consumer which “compiler profile” implementation to use.

Below is an example of a potential public interface call in C# to the API. Note that this is for illustrative purposes only—the actual API specification will likely differ.
JaCILProfile profile =
    new JaCILStandardProfile();
JaCILOptions options = ...;
JaCILCompiler compiler =
    new JaCILCompiler( profile, options );

// compile from file
JaCILAssembly assembly =
    compiler.Compile( "Some.Assembly.dll" );

// out put to JAR file
assembly.WriteJavaArchive( "Some.Assembly.jar" );
// get the bytes of a JVM class file for a class
Byte[] bytes =
    compiler.GetClassBytes( "com.someplace.SomeClass" );

The compiler shall also provide a tool front-end to access the API described
above. The following examples demonstrate how such a tool may be invoked
from the command-line. Note that the actual command-line tool may have
different parameters than what is listed here, this is for demonstrative purposes
only.

$ jacilc --help
Usage: jacilc [options] assembly1 [assembly2...]
Compiles the given Common Language Runtime
Assemblies to JVM JAR Files.

$ jacilc MyAssembly.dll

$ jacilc --entry-point=MainProgramClass \
    MyAssembly.dll

$ jacilc --profile=standard \
    -o MyAssemblyNewName.jar \
    MyAssembly.dll

3.2.2 Supported Source Binaries

The compiler shall support CLI assembly binary images that limit their
functionality to the subset of CLI 2.0 features described explicitly in 3.1.

This specification indicates which features of the CLI are supported—it is
up to the implementation how to deal with features that are not supported.
The compiler may raise an error or ignore any such feature. Regardless of the behavior chosen by the implementation, the compiler should not raise an unexpected exception when such a feature is encountered.

### 3.2.3 Compiler Output Target

The compiler shall emit binaries that are compatible to the Java 1.5 Virtual Machine. This is to allow the compiler to make use of any of modern JVM features available.

### 3.2.4 Hosted Platform

![JaCIL Compiler Framework Component Diagram](image)

Although the compiler framework could target either the CLI or the JVM for its implementation, this tool will more than likely be hosted on the CLI. The rationale for this is pragmatic. Targeting the CLI, we have access to the .NET Standard API and the Mono Cecil Library [10]. Both of these libraries provide capabilities to introspect CLI assemblies. Furthermore, with the IKVM.NET Project [11] at our disposal we can leverage Java libraries that abstract JVM class files and can perform byte-code generation [6, 2, 7]. This
will allow the author to focus on the translational semantics instead of the concrete format of the source and target binaries.

Furthermore, the hosted platform shall be the .NET Framework 2.0. This is to allow the compiler access to all additions to the standard API from version 1.1 which include enhancements to the Reflection API.

Figure 1 illustrates how the compiler shall operate within the CLI environment. The purple colored components are JaCIL components.

### 3.3 The JaCIL Hosting Layer

![Component diagram of JaCIL Compiled Assembly in JVM.](image)

The JaCIL Hosting Layer will be a Java API that will provide run-time support for binaries generated from the compiler infrastructure defined above. This API shall be composed of an Emulation and Utility API and a Type Compatibility API that is described in further detail in the subsections below.

The JaCIL Hosting Layer shall be dependent on Java 1.5 for its implementation. This follows the same requirements of the binaries generated from the
compiler.

Figure 2 is a component diagram illustrating the interaction between a JaCIL translated assembly, the hosting layer components and the Java standard API in the JVM. The green component is the translated assembly and the purple components are part of the JaCIL implementation.

3.3.1 The Emulation and Utility API

The Emulation and Utility API is provided to emulate CLI semantic concepts that are not present on the JVM. The functionality that is provided by this library are for CLI features like managed pointers—pointers that can be manipulated safely in in CLI byte-code—that have no equivalent primitive in the JVM.

For example, the CLI instruction, ldflda, loads the address of an instance’s field as a managed pointer onto the operand stack. This managed pointer can be passed around like any other data type and can be de-referenced by the ldind and stind instructions which load the value pointed to by a managed pointer and store a value to the location referenced by the pointer respectively. The API could then implement a pointer class that would use reflection to actually implement the de-referencing operations (Using a “boxing” technique similar to the one described is suggested by [17] and [13]). The compiler would then emit code to use this pointer class when translating CLI instructions dealing with managed pointers.

Below is an example of potential CLI code that would be translated in (in a simplified CLI assembler syntax):

```cli
// y = &(x.aField)
ldloc.0
ldflda String SomeClass::aField
stloc.1

// *y = "New Value"
ldloc.1
ldstr "New Value"
stind
```

A possible translation by the compiler might look like this (in a simplified Java 1.5 assembler syntax):

```java
// y = &(x.aField)
new jacil/impl/FieldPointer
```

17
dup
aload_0
ldc [class] SomeClass
ldc [string] "aField"
invokevirtual jacil/emulation/impl/FieldPointer:
  <init>({Ljava/lang/Object;
      Ljava/lang/Class;
      Ljava/lang/String;})V
astore_1

// *y = "New Value"
aload_1
ldc [string] "New Value"
invokevirtual jacil/emulation/Pointer:
  setValue({Ljava/lang/Object;})V

Of course as the semantics of the CLI are investigated, this API would expand to support other non-native semantics in the JVM. This API is also essentially “private”—only generated code from the compiler should ever interact with it.

This API shall also contain utility classes and methods that shall be used to provide helpers for compiler generated code to call that would otherwise be redundant to in-line directly.

### 3.3.2 The Type Compatibility API

The *Type Compatibility API* provides the functionality of parts of the CLI Standard API that must be present for any assembly to run. Namely support for primitive types such as integers, floating point numbers, string, and the base object shall be provided. Also, the base exception hierarchy shall be implemented by this API.

To provide a consistent interface to all classes defined in assemblies, all public parts of this API shall follow the same name space as the CLI equivalent. Thus, for example, the base object class shall be defined as the Java class `System.Object`.

### 3.3.3 Hosting Layer Implications

The hosting layer proposed can be thought of as a “proto-CLI” implementation. This layer shall only provide the capability to *run* translated components that are limited to using this small subset of the CLI.
At first, future work not considered, this may seem to indicate that the immediate benefits of this project would be minimal. But by using the IKVM.NET Java Standard APIs from any language supported in the .NET Framework (C#, VB.NET, etc.) instead of the CLI equivalents would render assemblies that would have Java API references. This assembly could then be translated and run in any JVM as the referred standard API calls would be present on any compliant JVM. This would have the effect of bringing practically any .NET Framework language to the JVM without having to write a compiler for that language.

This is, of course, the beginning of having a full CLI implementation on the JVM and although it is not within scope of the proposed project, future work could translate the CLI byte-code portions of the Mono [5] or DotGNU [3] standard libraries as a starting point. It is also interesting to note that if such an endeavor were to be realized, the JaCIL Compiler Framework itself could be boot-strapped into working as a JIT compiler for this hypothetical CLI implementation.

### 3.4 Test and Performance Measurements

The project shall implement a suite of small applications to serve the following purposes: compatibility testing and performance testing.

The applications that shall serve the role of compatibility testing shall serve to test the features of the CLI that are supported by the system. All supported features that the implementation supports shall have at a minimum unit tests to demonstrate correctness. Some examples of these applications may include the following.

- Abstract class translation.
- Method invocation.
- Numeric instruction basic functionality.
- Boundary numeric instruction (check overflow cases).

The applications comprising the performance testing shall not aim to test implemented functionality (though they may be used for such purposes). These applications shall be small but more computationally expressive to help compare the performance of translated code. Some examples of such applications may include the following.

- Repeated object instantiation.
• Repeated indirection for managed pointers to fields, array elements, and local variables.

• Repeated indirect calls of managed pointers to methods.

• Fibonacci Number generation.

These programs shall be written in any appropriate CLI language. For more acute unit tests, such programs may be written using the \texttt{ilasm} CLI assembler than a high-level CLI language.

4 Deliverables

The deliverables will consist of the following items:

• Documentation describing the design/architecture of the system.

• API documentation

• User manual

• Technical report providing background and overview of the project and its results.

• Project source code, commented.

5 Final Report

The following is a preliminary draft of the overall major sections to be put into the final report.

1. Introduction, Overview, and Motivation

2. Related Work

3. Software Documentation
   (a) Specification
   (b) API Specification
   (c) Design

4. CLI Feature Implementation
6 Schedule

The following outlines a high-level schedule for the project—subject to updates.
<table>
<thead>
<tr>
<th>Week</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Literature survey, project proposal and preliminary proof of concept work.</td>
</tr>
<tr>
<td>1</td>
<td>Set up build infrastructure (makefiles, nant, ant, etc.) and proof of concept work.</td>
</tr>
<tr>
<td>2-3</td>
<td>Development of basic compiler architecture and implementation of generating stub JVM class files from CLI assemblies.</td>
</tr>
<tr>
<td>4-6</td>
<td>Implementation of compilation of base method code, including support for base instructions that do not require hosting layer assistance.</td>
</tr>
<tr>
<td>7-8</td>
<td>Implementation of compilation of other base instructions and the supporting hosting layer API.</td>
</tr>
<tr>
<td>9-11</td>
<td>Implementation of the object model instructions and supporting hosting layer API.</td>
</tr>
<tr>
<td>11-12</td>
<td>Final testing and performance review.</td>
</tr>
<tr>
<td>13-15</td>
<td>Development of final report and user documentation.</td>
</tr>
<tr>
<td>16-17</td>
<td>Final preparation for defense.</td>
</tr>
<tr>
<td>18</td>
<td>Project defense.</td>
</tr>
</tbody>
</table>

References


