Execution Engine Architecture

Version 1.9 Final

Copyright © 1999 Microsoft Corporation. All rights reserved.

Last updated: 8 June 2000

This is preliminary documentation and subject to change

Table of Contents

1 Audience and Related Specifications

This specification is intended for people interested in generating or analyzing programs that will be executed by the NGWS runtime. This includes people who write compilers that target the NGWS runtime (either with native code or IL), development tools or environments, or program analysis tools.

For more information about the EE, IL, and metadata, see the following specifications:

- The Virtual Object System (VOS) specification.
- The IL Instruction Set specification.
- The Assembler Programmers Reference specification.
- The Metadata Interfaces specification.
- The File Format specification.

2 Execution Engine Overview

NGWS provides an *Execution Engine* (EE) that manages the execution of source code after being compiled into Intermediate Language (IL), OptIL, or native machine code. All code based on NGWS IL or OptIL executes as *managed code;* that is code that runs under a "contract of cooperation" with NGWS. NGWS provides services such as memory management, cross language integration, exception handling, code access security, and automatic lifetime control of objects. In return, managed code must supply enough information in metadata to enable NGWS to locate and unwind stack frames. For a high level description of the features that NGWS provides to managed code, see the "NGWS Overview" specification.

A key feature of NGWS runtime is its ability to provide *software isolation* of programs running within a single address space. It does this by enforcing typesafe access to all areas of memory when running typesafe managed code. Some compilers generate IL that is not only typesafe but whose typesafety can be proven by simply examining the IL. This process, *verification*, allows servers to quickly examine user programs written in IL and only run those that it can demonstrate will not make unsafe memory references. This independent verification is critical to truly scalable servers that execute user-defined programs (scripts).

The EE provides the following services:

- Code management
- Software memory isolation
- Verification of the typesafety of IL
- Conversion of IL to native code
- Loading and execution of managed code (IL or native)
- Accessing metadata (enhanced type information)
- Managing memory for managed objects
- Insertion and execution of security checks
- Handling exceptions, including cross-language exceptions
- Interoperation between NGWS objects and COM objects
- Automation of object layout for late binding
- Supporting developer services (profiling, debugging, etc.)

The EE supplies the common infrastructure that allows tools and programming languages to benefit from cross-language integration. Any technical improvements to the EE will benefit all languages and tools that target NGWS.

One of the most important functions of the EE is on-the-fly conversion of IL (or OptIL) to native code. Source code compilers generate IL (or OptIL), and *JIT compilers* convert that IL to native code for specific machine architectures. As long as a simple set of rules is followed by the IL generator, the same IL code will run on any architecture that supports NGWS. Because the conversion from IL to native code occurs on the target machine, the generated native code can take advantage of hardware-specific optimizations. Other significant EE functions include class loading, verification, and support for security checks.

2.1 IL and OptIL

IL is a stack-based set of instructions designed to be easily generated from source code by compilers and other tools. Several kinds of instructions are provided, including instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and method invocation. There is also a set of IL instructions for implementing object-oriented programming constructs such as virtual method calls, field access, array access, and object allocation and initialization.

The IL instruction set can be directly interpreted by simply tracking the data types on the stack and emulating the IL instructions. It can also be converted efficiently into native code. The design of IL allows this process to produce optimized native code at reasonable cost. The design of IL allows programs that are not typesafe to be expressed, since this is essential for support of some common programming languages. At the same time, by following a simple set of rules, it is possible to generate IL programs that are not only typesafe but can easily be proven to be so (see the Verification section for more information about type safety and verification).

OptIL is a subset of IL that can be generated by optimizing compiler front ends. OptIL contains embedded annotations, which are IL instructions that supply control flow and register allocation information. Since OptIL is a subset of IL, any component that can execute or analyze IL can also analyze or execute OptIL (ignoring the embedded annotations if necessary). The OptJIT compiler (to be shipped in a future release), however, uses the embedded information to rapidly produce optimized native code. The correctness of this native code depends on the annotations, so they are subject to verification. OptIL is useful in situations where limited time and memory resources are available during the conversion to native code (ie at JIT time), yet the native code produced must meet high performance standards.

2.2 JIT Compilation

The EE provides three JIT compilers for converting IL to native code: EconoJIT, JIT, and OptJIT. Each JIT compiler has been designed to meet specific goals with respect to performance and resource usage. The performance characteristics are summarized in Figure 1. Because of the low overhead of the EconoJIT compiler, as well as the ease with which it can be ported to new architectures, NGWS does not include an interpreter for IL. (EconoJIT is so named because it performs the same task as the full JIT compiler, but using less computer resources. As a trade-off, the quality of the generated code is not so high).

JIT Compiler	Input Language	JIT Compiler Overhead	Compilation Speed	Quality of Output
EconoJIT	IL (incl. OptIL)	Very Small	Very Fast	Low
JIT	IL (incl. OptIL)	Medium to Large	Moderate	High
OptJIT (not in V1)	**OptIL only**	**Small**	**Fast**	**High**

Figure 1: Performance Characteristics of NGWS JIT Compilers

In some cases, tools vendors or researchers might want to design their own JIT compilers for use with NGWS. Using the standard interface between the EE and a JIT compiler, a third-party JIT compiler can be "plugged in" to the EE and interact appropriately. This interface (to be published in a future release) will consist of two parts: one used when IL is compiled to native code (the *JIT/EE interface*) and the other when the compiled code is executed (the *code manager*). The code manager performs the stack walks required for memory management, exception handling, and security checks. It also performs other functions, such as converting NGWS exceptions into the form expected by the source language processor's exception handlers. Vendors who design custom JIT compilers can use the Runtime's code manager or they can design a custom code manager to describe the layout of the method state for code they have compiled.

2.3 Class Loading

The EE's *class loader* loads the implementation of a class, expressed in IL, OptIL or native code, into memory, checks that it is consistent with assumption made about it by other previously loaded classes, and prepares it for execution. To accomplish this task, the class loader ensures that certain information is known, including the amount and the shape of the space that instances of the type require. In addition, the class loader determines whether references made by the loaded type are available at runtime and whether references to the loaded type are consistent.

The class loader checks for certain consistency requirements that are vital to the NGWS security enforcement mechanism. These checks constitute a minimal, mandatory, verification process that precedes the IL verification, which is more rigorous (and optional). In addition, the class loader supports security enforcement by providing some of the credentials required for validating code identity. For more details, see the NGWS Virtual Object System specification.

NGWS runtime allows only one class loader, its own. NGWS does not support user-written class loaders.

2.4 Verification

Typesafe programs reference only memory that has been allocated for their use, and they access objects only through their public interfaces. These two restrictions allow objects to safely share a single address space, and they guarantee that security checks provided by the objects' interfaces are not circumvented. Code access security, the NGWS runtime's security mechanism, can effectively protect code from unauthorized access only if there is a way to verify that the code is typesafe.

To meet this need, the NGWS runtime uses the information in type signatures to help determine whether IL code is typesafe. It checks to see that metadata is well-formed, and it performs control flow analyses to ensure that certain structural and behavioral conditions are met. The runtime declares that a program is successfully verified only if it is typesafe.

Used in conjunction with the strong typing of metadata and IL, such checking can ensure the typesafety of programs written in IL. NGWS requires code to be so checked before it is run, unless a specific (administratively controlled) security check determines that the code can be fully trusted.

2.5 Security Checks

The EE is involved in many aspects of NGWS''s security mechanism. In addition to the verification process required by code access security, the EE provides support that enables both declarative and imperative security checks to occur.

Declarative security checks take place automatically whenever a method is called. The permissions that are required in order to access the method are stored in the component's metadata. At run time, calls to methods that are marked as requiring specific permissions are intercepted to determine whether callers have the required permissions. A stack walk is sometimes necessary to determine whether each caller in the call chain also has the required permissions.

Imperative security checks occur when security functions, such as checking a code access permission, or asserting the right to use a specified permission, are invoked from within the code being protected. The EE supports this type of security check by providing trusted methods that enable code identity to be determined and allow permissions to be located and stored in the stack. In addition, the EE gives the security engine access to administrative information about security requirements.

2.6 Profiling and Debugging

The EE provides the ability to both debug (observe and modify the behavior) and profile (measure resource utilization) of running programs. It does this by providing three underlying services, described in detail in the Debugging Specifications and the Profiling Specification. Both profiling and debugging depend on information produced by the source language compiler and updated by the JIT compiler.

The EE provides an API for debugging that handles registration for and notification of events in the running program. This allows a debugger to control execution of a program, including setting and handling breakpoints, intercepting exceptions, modifying control flow, and examining or modifying program state (both code and data).

The EE also provides an API for use by tools that do program profiling. The API supports profiling of managed native code (e.g. the output of a JITter) both with and without inserting specific profiling probes into the code.

2.7 Interoperation with Unmanaged Code

The EE also provides for two-way transitions between managed and unmanaged code. This includes interoperation with existing COM clients and services (known as "COM interop") as well as previously compiled native DLLs (known as "PInvoke"). Where necessary because of data format or other differences, the EE supplies marshaling procedures that copy and/or reformat information across the boundary.

2.8 This Specification

The remainder of this specification provides information about aspects of the architecture of the NGWS Execution Engine that are relevant to the development of tools that generate or manipulate IL. The following topics are discussed:

- Virtual Execution System
- Supported data types

- Executable image information
- Machine state definitions
- Method calling information
- Exception handling
- OptIL

3 Virtual Execution System

By providing services such as class loading, verification, JIT compilation, and code management, the Execution Engine creates an environment for code execution called the Virtual Execution System. Figure 2 shows the major elements of the EE highlighted in gray, and it indicates with arrows the various paths that can be taken through this execution environment.

In most cases, source code is compiled into IL, the IL is loaded, compiled to native code on-the-fly using one of the JIT compilers, and executed. Note that for trusted code, verification can be omitted.

The EE's metadata engine enables the source code compiler to place metadata in the PE file along with the generated IL or OptIL. ("PE" stands for Portable Executable, the format used for executable (EXE) and dynamically linked library (DLL) files). During loading and execution, this metadata provides information needed for registration, debugging, memory management, and security. Also indicated in the diagram is the fact that classes from the NGWS Base Class Library can be loaded by the class loader along with IL, OptIL, or native code.

Another execution path that can be chosen involves pre-compilation to native code using a backend compiler. This option might be chosen if compiling code at run-time (that's to say, JIT compiling) is unacceptable due to performance requirements. As indicated in the diagram, precompiled native code bypasses verification and JIT compilation. Because precompiled native code is not verified, it must be considered fully trusted code in order to execute.

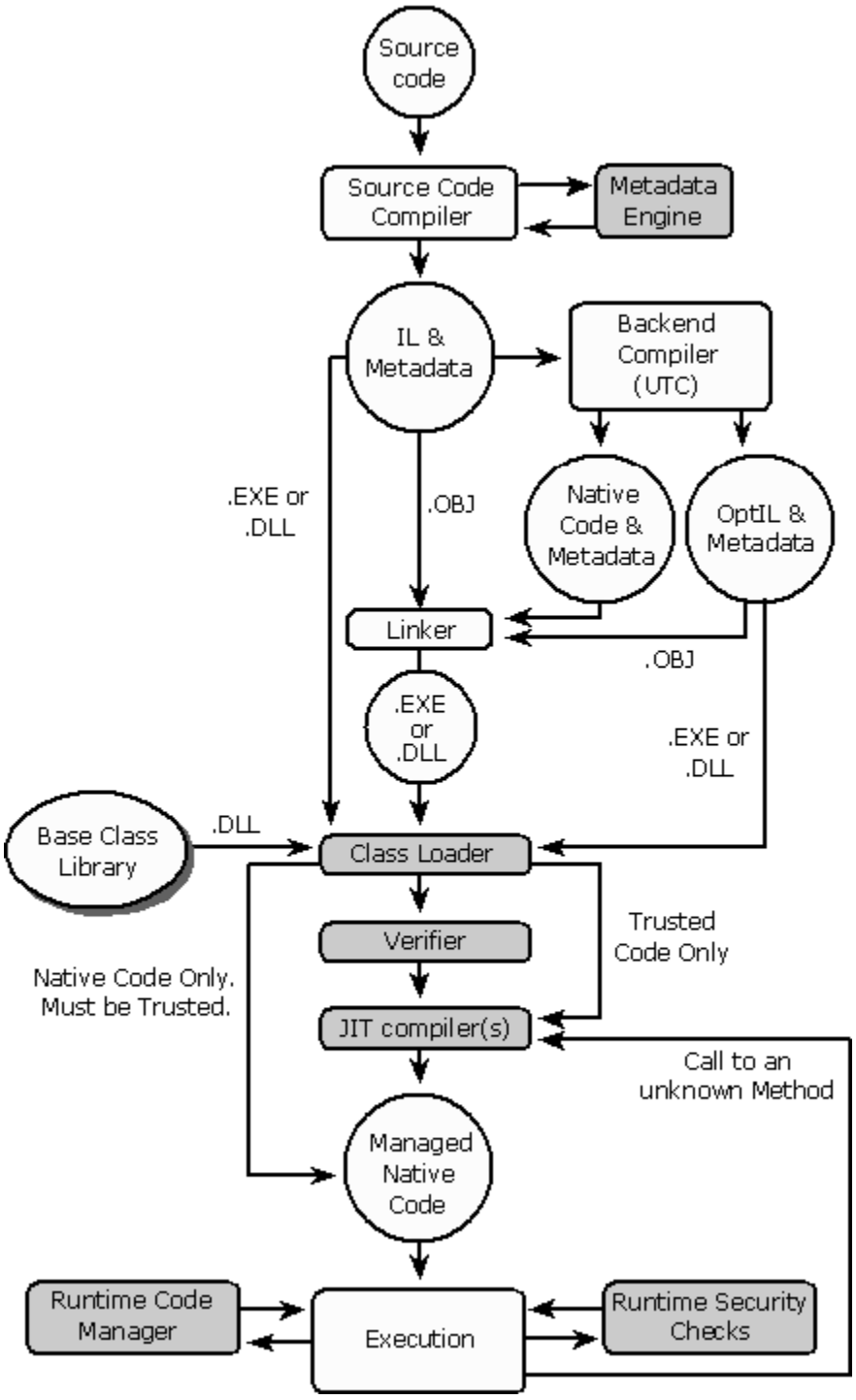

Figure 2: Overview of EE Architecture

4 Supported Data Types

The Execution Engine directly supports the data types shown in Table 1. That is, these data types can be manipulated using the IL instruction set.

Data Type	Description
I1	8-bit 2's complement signed value
U1	8-bit unsigned binary value
I2	16-bit 2's complement signed value
U2	16-bit unsigned binary value
I4	32-bit 2's complement signed value
U4	32-bit unsigned binary value
I8	64-bit 2's complement signed value
U8	64-bit unsigned binary value
R4	32-bit IEEE 754 floating point value
R8	64-bit IEEE 754 floating point value
I	natural size 2's complement signed value
U	natural size unsigned binary value, also unmanaged pointer
R4Result	Natural size for result of a 32-bit floating point computation
R8Result	Natural size for result of a 64-bit floating point computation
RPrecise	Maximum-precision floating point value
O	natural size object reference to managed memory
&	natural size managed pointer (may point into managed memory)

Table 1: Data Types Directly Supported by the EE

The EE model uses an evaluation stack. Instructions that copy values from memory to the evaluation stack we call "loads"; instructions that copy values from the stack back to memory we call "stores". The full set of data types in the table above can be represented in memory. However, the EE supports only a subset of these types in its operations upon values stored on its evaluation stack – I4, I8, I. In addition the EE supports an internal data type, F, to represent floating point values on the internal evaluation stack. The F type can be thought of as starting at the size of values loaded from memory and then expanded when combined with higher-precision values. Shorter values (I1, I2, U1, U2) are widened when loaded (memory-to-stack) and narrowed when stored (stack-to-memory). This reflects a computer model that assumes memory cells are 1, 2, 4, or 8 bytes wide but registers and stack locations are either 4 or 8 bytes wide. The support for short values consists of:

- Load and store instructions to/from memory: **ldelem, ldind, stind, stelem**
- Arithmetic with overflow detection: **add.ovf**, **mul.ovf, sub.ovf**
- Data conversion: **conv**, **conv.ovf**
- Loading constants: **ldc**

- Array creation: **newarr**

The signed integer (I1, I2, I4, I8, and I) and unsigned integer (U1, U2, U4, U8, and U) types differ only in how the bits of the integer are interpreted. For those operations where an unsigned integer is treated differently from a signed integer (e.g. comparisons or arithmetic with overflow) there are separate instructions for treating an integer as unsigned (e.g. **cgt.un** and **add.ovf.u**).

This instruction set design simplifies JIT compilers and interpreters of IL by allowing them to internally track a smaller number of data types. See the Evaluation Stack section.

As described below, IL instructions do not specify their operand types. Instead, the EE keeps track of operand types and the JIT generates the appropriate native code. For example, the single **add** instruction will add two integers or two floats from the stack.

4.1 Natural Size: I, R4Result, R8Result, RPrecise, U, O and &

The natural-size, or generic, types (I, R4Result, R8Result, RPrecise, U, O, and &) are a mechanism in the EE for deferring the choice of a value's size. These data types exist as IL types. But when compiled to native code, the JIT maps each to the natural size for that specific processor. (For example, data type I would map to I4 on a Pentium processor, but to I8 on an IA64 processor). So, the choice of size is deferred until JIT compilation, when the EE has been initialized and the architecture is known. This implies that field and stack frame offsets are also not known at compile time. For languages like Visual Basic, where field offsets are not computed early anyway, this is not a hardship. In languages like C or C++, a conservative assumption that they occupy 8 bytes is sometimes acceptable (for example, when laying out compile-time storage). The EE's generic types were designed to circumvent parts of this problem.

4.1.1 Unmanaged Pointers as Type U

For languages like C, when compiling all the way to native code, where the size of a pointer is known at compile time and there are no managed objects, the fixed-size unsigned integer types (U4 or U8) can serve as pointers. However choosing pointer size at compile time has its disadvantages. If pointers were chosen to be 32 bit quantities at compile time, the code would be restricted to 4gig of address space, even if it were run on a 64 bit machine. Moreover, a 64 bit EE would need to take special care so those pointers passed back to 32-bit code could always fit in 32 bits. If pointers were chosen at compile time to be 64 bits, the code could be run on a 32 bit machine, but pointers in **every** data structure would be twice as large as necessary on that EE.

It is desirable, especially when building library routines that are platform-agnostic, to defer the choice of pointer size from compile time to EE initialization time. In that way, the ***same IL code*** can handle large address spaces for those applications that need them, while also being able to reap the size benefit of 32 bit pointers for those applications that do not need a large address space.

For these reasons, the **U** type should be used to represent unmanaged pointers.

4.1.2 Managed Pointer Types: O and &

The **O** datatype represents an object reference that is managed by NGWS. As such, the number of specified operations is severely limited. In particular, references can only be used on operations that indicate that they operate on reference types (e.g. **ceq** and **ldind.ref**), or on operations whose metadata indicates that references are allowed (e.g. **call**, **ldsfld**, and **stfld**).

The **&** datatype (managed pointer) is similar to the **O** type, but points to the interior of an object. That is, a managed pointer is allowed to point to a field within an object or an element within an array, rather than to point to the 'start' of object or array.

Object references (**O**) and managed pointers (**&**) must be reported to the NGWS memory manager so that it can update their values as the items they point to are moved during garbage collection.

In summary, object references, or **O** types, refer to the 'outside' of an object, or to an object as-a-whole. But managed pointers, or **&** types, refer to the interior of an object.

In order to allow managed pointers to be used more flexibly, they are also permitted to point to areas that aren't under the control of the NGWS garbage collector, such as the evaluation stack, static variables, and unmanaged memory. This allows them to be used in many of the same ways that unmanaged pointers (**U**) are used. As a result, however, managed pointers are allowed to appear only as parameters or local variables; this guarantees that a managed pointer to a value on the evaluation stack doesn't outlast the life of location to which it points.

4.1.3 Portability: Storing Pointers in Memory

Several instructions, including **calli**, **cpblk**, **initblk**, **ldind.***, and **stind.***, expect an address on the top of the stack. If this address is derived from a pointer stored in memory, there is an important portability consideration.

1. Code that stores pointers in a natural sized integer or pointer location (types **I**, **O**, **U**, or **&**) is always fully portable.
2. Code that stores pointers in an 8 byte integer (type **I8** or **U8**) *can* be portable. But this requires that a **conv.ovf.u** instruction be used to convert the pointer from its memory format before its use as a pointer. This may cause a runtime exception if run on a 32-bit machine.
3. Code that uses any smaller integer type to store a pointer in memory (**I1**, **U1**, **I2**, **U2**, **I4**, **U4**) is *never* portable, even though the use of a U4 or I4 will work correctly on a 32-bit machine.

4.1.4 Natural Size Floating-Point: R, R4Result, R8Result, and RPrecise

To support a wide range of hardware architectures, the EE follows the recommendations of the ANSI C9x committee by providing not only the two IEEE storage formats (32-bit and 64-bit) but three additional types that are not portable

across architectures. The type **R4Result** is a type large enough to hold the results of calculations that use **R4** (i.e. IEEE 32-bit) arguments. Similarly, the type **R8Result** is a type large enough to hold the results of calculations that use **R8** (i.e. IEEE 64-bit) arguments. Finally, the type **RPrecise** can hold a floating-point value of the maximum precision supported conveniently on the target architecture, but containing at least 64 bits (for example, in current-generation Pentium processors, this would be 80 bits). In terms of precision, the following are always true:

R4 <= R4Result <= RPrecise

R8 <= R8Result <= RPrecise

R4 < R8 <= RPrecise

R4Result <= R8Result <= RPrecise

4.2 Handling of Short Integer Data Types

The Execution Engine defines an evaluation stack that contains either 4-byte or 8-byte integers, but a memory model that encompasses in addition 1-byte and 2-byte integers. To be more precise, the following rules are part of the Execution Engine model:

- Loading from 1-byte or 2-byte locations (arguments, locals, fields, statics, pointers) expands to 4-byte values. For locations with a known type (e.g. local variables) the type being accessed determines whether the load sign-extends (signed locations) or zero-extends (unsigned locations). For pointer dereference (**ldind.***), the instruction itself identifies the type of the location (e.g. **ldind.u1** indicates an unsigned location, while **ldind.i1** indicates a signed location).
- Storing into a 1-byte or 2-byte location truncates to fit and will not generate an overflow error. Specific instructions (**conv.ovf.***) can be used to test for overflow before storing.
- Calling a method in essence assigns values from the evaluation stack to the arguments for the method, hence it truncates just as any other store would.
- Returning from a method in essence assigns a value to an invisible return variable, so it also truncates as a store would. Since the value of this return variable is then placed on the evaluation stack, it is then sign-extended or zero-extended as would any other load. Notice that this truncation followed by extending is *not* identical to simply leaving the computed value unchanged.

It is the responsibility of any translator from IL to native machine instructions to make sure that these rules are faithfully modeled through the native conventions of the target machine. The Execution Engine does specify, for example, whether truncation of short integer arguments occurs at the call site or in the target method.

4.3 Handling of Floating Point Datatypes

The Execution Engine assumes floating-point calculations are handled as described in the IEEE 754 standard, "IEEE Standard for Binary Floating-point Arithmetic". This standard describes encoding of floating point numbers, definitions of the basic operations and conversion, rounding control, and exception handling.

The standard defines three special values, **NaN**, (not a number), **+infinity**, and **–infinity**. These values are returned on overflow conditions. A general principle is

that operations that have a value in the limit return an appropriate infinity while those that have no limiting value return **NaN**, but see the standard for details. The following examples show the most commonly encountered cases:

```
X rem 0 = NaN
0 *  +infinity = 0 * -infinity = NaN
(X / 0) = +infinity, if X>0
                NaN, if X=0
                -infinity, if X < 0
NaN op X = X op NaN = NaN for all operations
 (+infinity) + (+infinity) =  (+infinity)
X / (+infinity) = 0
X mod (-infinity) = -X
(+infinity) - (+infinity) =  NaN
```

For purposes of comparison, infinite values act like a number of the correct sign but with a very large magnitude when compared with finite values. **NaN** is 'unordered' for comparisons (see **clt**, **clt.un**).

While the IEEE 754 spec also allows for exceptions to be thrown under unusual conditions (overflow, invalid operand, …), the EE does not generate these exceptions. Instead, the EE uses the **NaN** return values and provides the instruction **ckfinite** to allow users to generate an exception if a result is **NaN**, **+infinity**, or **–infinity**.

The rounding mode defined in IEEE 754 is set by the EE to round to the nearest number, and neither the IL nor the base class library provide a mechanism for modifying this setting. The EE does not specify what happens if fully trusted code modifies the rounding mode.

For conversion to integers, the default operation supplied by the IL is "truncate towards zero". There are base class libraries supplied to allow floating-point numbers to be converted to integers using any of the other three traditional operations (**round** to nearest integer, **floor** (truncate towards **–infinity**), **ceiling** (truncate towards **+infinity**)).

Storage locations for floating point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are **R4**, **R8**, and **RPrecise**. Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables) floating point numbers are represented using the internal **F** type. This type can be thought of as starting at the size of value loaded from storage and then expanding as needed. This design allows the EE to choose a platform-specific high-performance representation for floating point numbers until they are placed in storage locations. For example, it may be able to leave floating point variables in hardware registers that provide more precision than a user has requested. At the same time, IL generators can force operations to respect language-specific rules for representations through the use of conversion instructions.

When a value of type **F** is put in a storage location it is automatically coerced to the required size, which may involve a loss of precision or the creation of an out-of-range marker (a **NaN**). To detect values that cannot be converted to a particular storage type, use a conversion instruction (**conv.r4**, **conv.r8**, **conv.r4result**, **conv.r8result**, or **conv.rprecise**) and then check for a non-finite value using

ckfinite. To detect underflow when converting to a particular storage type, a comparison to zero is required before and after the conversion.

4.4 IL Instructions and Numeric Types

Most IL instructions that deal with numbers take their operands from the evaluation stack (see the Evaluation Stack section), and these inputs have an associated type that is known to the JIT compiler. As a result, a single operation like **add** can have inputs of any numeric data type, although not all instructions can deal with all combinations of operand types. Binary operations other than addition and subtraction require that both operands must be of the same type. Addition and subtraction allow an integer to be added to or subtracted from a managed pointer (types **&** and **O**).

Instructions fall into the following categories:

Numeric: These instructions deal with both integers and floating point numbers, and consider integers to be signed. Simple arithmetic, conditional branch, and comparison instructions fit in this category.

Integer: These instructions deal only with integers. Bit operations and unsigned integer division/remainder fit in this category.

Floating point: These instructions deal only with floating point numbers.

Specific: These instructions deal with integer and/or floating point numbers, but have variants that deal specially with different sizes and unsigned integers. Integer operations with overflow detection, data conversion instructions, and operations that transfer data between the evaluation stack and other parts of memory (see the Method State section) fit into this category.

Unsigned/unordered: There are special comparison and branch instructions that treat integers as unsigned and consider unordered floating point numbers specially (as in "branch if greater than or unordered"):

Load constant: The load constant (**ldc.***) instructions can be used to load constants of type I4, I8, R4 or R8. Natural size constants (type I) must be created by conversion from I4 (conversion from I8 would not be portable) using **conv.i** or **conv.u.** Similarly, constants of type R4Result, R8Result, or RPrecise must be created by conversion from one of the fixed-size floating point types (R4 or R8).

shows the IL instructions that deal with numeric values, along with the category to which they belong. Instructions that end in ".*" indicate all variants of the instruction (based on size of data and whether the data is treated as signed or unsigned).

add	Numeric
add.ovf.*	Specific
and	Integer
beq[.s]	Numeric
bge[.s]	Numeric
bge.un[.s]	Unsigned/unordered

div	Numeric
div.un	Integer
ldc.*	Load constant
ldelem.*	Specific
ldind.*	Specific
mul	Numeric

bgt[.s]	Numeric
bgt.un[.s]	Unsigned/unordered
ble[.s]	Numeric
ble.un[.s]	Unsigned/unordered
blt[.s]	Numeric
blt.un[.s]	Unsigned/unordered
bne.un[.s]	Unsigned/unordered
ceq	Numeric
cgt	Numeric
cgt.un	Unsigned/unordered
ckfinite	Floating point
clt	Numeric
clt.un	Unsigned/unordered
conv.*	Specific
conv.ovf.*	Specific

mul.ovf.*	Specific
neg	Integer
newarr.*	Specific
not	Integer
or	Integer
rem	Numeric
rem.un	Integer
shl	Integer
shr	Integer
shr.un	Specific
stelem.*	Specific
stind.*	Specific
sub	Numeric
sub.ovf.*	Specific
xor	Integer

Table 2: IL Instructions by Numeric Category

4.5 IL Instructions and Pointer Types

The Execution Engine has the ability to track pointers to objects and to collect objects that are not longer reachable (memory management by "garbage collection"). This process moves objects in order to reduce the working set and thus must modify all pointers to those objects as they move. For this to work correctly, pointers to objects must only be used in certain ways. The **O** (object reference) and **&** (managed pointer) datatypes are the formalization of these restrictions.

The use of object references is tightly restricted in the IL. They are used almost exclusively with the "virtual object system" instructions, which are specifically designed to deal with objects. In addition, a few of the base instructions of the IL can handle object references. In particular, object references can be:

1. Loaded onto the evaluation stack to be passed as arguments to methods (**ldloc**, **ldarg**), and stored from the stack to their home locations (**stloc, starg**)
2. Duplicated or popped off the evaluation stack (**dup**, **pop**)
3. Tested for equality with one another, but not other data types (**beq**, **beq.s**, **bne**, **bne.s**, **ceq**)
4. Loaded-from / stored-into unmanaged memory, in type unsafe code only (**ldind.ref**, **stind.ref**)
5. Create a null reference (**ldnull**)
6. Returned as a value (**ret**)

Managed pointers have several additional base operations.

1. Addition and subtraction of integers, in units of *bytes*, returning a managed pointer (**add**, **add.ovf.u, sub, sub.ovf.u**)
2. Subtraction of two managed pointers to elements of the same array, returning the number of *bytes* between them (**sub**, **sub.ovf.u**)
3. Unsigned comparison and conditional branches based on two managed pointers (**bge.un, bge.un.s, bgt.un, bgt.un.s, ble.un, ble.un.s, blt.un, blt.un.s, cgt.un**, **clt.un**)

Since the memory manager runs asynchronously with respect to programs and updates managed pointers, both the distance between distinct objects and their relative position can change. To allow for the possibility that data layout might be changed asynchronously, the ordering and distance between fields *within* an object must be assumed to change. Thus, arithmetic operations upon managed pointers are intended *only* for use on pointers to elements of the same array. Other uses of arithmetic on managed pointers is unspecified.

4.6 Aggregate Data

The EE supports *aggregate data*, that is, data items that have sub-components (arrays, structures, or object instances) but are passed by copying the value. The sub-components can include references to managed memory. Aggregate data is represented using a *value type*, which can be instantiated in two different ways:

- **Boxed**: as an Object, carrying full type information at runtime, and typically allocated on the heap by the NGWS memory manager.
- **Unboxed**: as a "value type instance" which does *not* carry type information at runtime and which is never allocated directly on the heap. It can be part of a larger structure on the heap – a field of a class, a field of a boxed value type, or an element of an array. Or it can be in the local variables or incoming arguments array (see the Method State section). Or it can be allocated as a static variable or static member of a class or a static member of another value type.

Because value type instances, specified as method arguments, are copied on method call, they do not have "identity" in the sense that Objects (boxed instances of classes) have; see the VOS specification.

Support for value types at the IL level is driven by these requirements:

1. Memory management must remain efficient.
2. Value type instances have no space overhead (hence, no runtime type information is stored in the instance itself).
3. Value type instances are first-class (i.e., they can be passed as arguments, returned as values, stored in variables, and stored in data structures).

4.6.1 Homes for Values

The **home** of a data value is where it is stored for possible reuse. The EE directly supports the following home locations:

1. An incoming **argument**
2. A **local variable** of a method
3. An instance **field** of an object or value type

4. A **static** field of a class, interface, or module
5. An **array element**

For each home location, there is a means to compute (at runtime) the address of the home location and a means to determine (at JIT compile time) the type of a home location. These are summarized in Table 3.

Type of Home	Runtime Address Computation	JITtime Type Determination
Argument	**ldarga** for by-value arguments or **ldarg** for by-reference arguments	Method signature
Local Variable	**ldloca** for by-value locals or **ldloc** for by-reference locals	Locals signature in method header
Field	**ldflda**	Type of field in the class, interface, or module
Static	**ldslfda**	Type of field in the class, interface, or module
Array Element	**ldelema** for single-dimensional zero-based arrays or call the instance method **Address**	Element type of array

Table 3: Address and Type of Home Locations

In addition to homes, built-in values can exist in two additional ways (i.e. without homes):

1. as constant values (typically embedded in the IL instruction stream using **ldc.*** instructions)
2. as an intermediate value on the evaluation stack, when returned by a method or IL instruction.

4.6.2 Operations on Value Type Instances

Value type instances can be created, passed as arguments, returned as values, and stored into and extracted from locals, fields, and elements of arrays (i.e., copied). Like classes, value types can have both static and non-static members (methods and fields). But, because they carry no type information at runtime, value type instances are not substitutable for items of type `Object`; in this respect, they act like the primitive types `int`, `long`, and so forth. There are two new operations, box and unbox, that convert between value type instances and Objects.

4.6.2.1 Initializing Instances of Value Types

There are three options for initializing the home of a value type instance. You can zero it by loading the address of the home (see Table 3) and using the **initobj** instruction (for local variables this can also be accomplished by setting the **zero initialize** bit in the method's header). You can call a user-defined constructor by loading the address of the home (see Table 3) and then calling the constructor directly. Or you can copy an existing instance into the home, as described in 4.6.2.2.

4.6.2.2 Loading and Storing Instances of Value Types

There are two ways to load a value type onto the evaluation stack:

- Directly load the value from a home that has the appropriate type, using an **ldarg**, **ldloc**, **ldfld**, or **ldsfld** instruction
- Compute the address of the value type, then use an **ldobj** instruction

Similarly, there are two ways to store a value type from the evaluation stack:

- Directly store the value into a home of the appropriate type, using a **starg**, **stloc**, **stfld**, or **stsfld** instruction
- Compute the address of the value type, then use a **stobj** instruction

4.6.2.3 Passing and Returning Value Types

Value types are treated just as any other value would be treated:

- **To pass a value type by value**, simply load it onto the stack as you would any other argument: use **ldloc**, **ldarg**, etc., or call a method which returns a value type. To access a value type parameter that has been passed by value use the **ldarga** instruction to compute its address or the **ldarg** instruction to load the value onto the evaluation stack.
- **To pass a value type by reference**, load the address of the value type as you normally would (see Table 3). To access a value type parameter that has been passed by reference use the **ldarg** instruction to compute the address of the argument and then the **ldobj** instruction to load it onto the evaluation stack.
- **To return a value type**, just load the value onto an otherwise empty evaluation stack and then issue a **ret** instruction.

4.6.2.4 Calling Methods

Static methods on value types are handled no differently from static methods on an ordinary class: use a **call** instruction with a metadata token specifying the value type as the class of the method. Non-static methods (i.e. instance and virtual methods) are supported on value types, but they must be given special treatment. A non-static method on a class (rather than a value type) expects a **this** pointer which is an instance of that class. This makes sense for classes, since they have identity and the **this** pointer represents that identity. Value types, however, have identity only when boxed. To address this issue, the **this** pointer on a non-static method of a value type is a by-ref parameter of the value type rather than an ordinary by-value parameter.

A non-static method on a value type my be called in the following ways:

- Given an unboxed instance of a value type, the **call** instruction can be used to invoke the function, passing as the first parameter (the **this** pointer) the address of the instance. The metadata token used with the **call** instruction must specify the value type itself as the class of the method.
- Given a boxed instance of a value type, the **call** instruction can be used to invoke the function, passing the boxed instance as the first parameter (the **this** pointer).

The metadata token used must specify **System.Object** as the class of the method.

To call a non-static method of an interface that is implemented by a value type or a virtual method inherited from **System.Object** you must box the value type and use a **callvirt** instruction. For a method on an interface, the metadata token must specify the interface as the type of the method, and for an inherited method it must specify **System.Object** as the class of the method.

4.6.2.5 Boxing and Unboxing

Box and **unbox** are conceptually equivalent to (and may be seen in higher-level languages as) casting between a value type instance and **System.Object**. Because they change data representations, however, boxing and unboxing are like the widening and narrowing of various sizes of integers (the **conv** and **conv.ovf** instructions) rather than the casting of reference types (the **isinst** and **castclass** instructions). The **box** instruction is a widening (always typesafe) operation that converts a value type instance to **System.Object** by making a copy of the instance and embedding it in a newly allocated object. **Unbox** is a narrowing (runtime exception may be generated) operation that converts a **System.Object** (whose runtime type must be a value type) to a value type instance. This is done by computing the address of the embedded value type instance without making a copy of the instance.

4.6.2.6 Castclass and IsInst on Value Types

Casting to and from value type instances isn't possible (the equivalent operations are **box** and **unbox**). When boxed, however, it is possible to use the **isinst** instruction to see whether a value of type `Object` is the boxed representation of a particular class.

4.6.3 Opaque Classes

Some languages provide multi-byte data structures whose contents are manipulated directly by address arithmetic and indirection operations. To support this feature, NGWS allows value types to be created with a specified size but no information about their data members. Instances of these "opaque classes" are handled in precisely the same way as instances of any other class, but the **ldfld**, **stfld**, **ldflda**, **ldsfld**, and **stsfld** instructions cannot be used to access their contents.

5 Executable Image Information

The File Format Specification provides details of the NGWS PE file format, and the Metadata Specification describes an API that can be used to access the metadata in a PE file. The EE relies on the following information about each method defined in a PE file:

- The *instructions* composing the method body, including all exception handlers.
- The *signature* of the method, which specifies the return type and the number, order, parameter passing convention, and primitive data type of each of the arguments. It also specifies the native calling convention (this does *not* affect the IL virtual calling convention, just the native code).
- The *exception handling array*. This array holds information delineating the ranges over which exceptions are filtered and caught. See the Exception Handling section.
- The size of evaluation stack that the method will require. (This is one of the factors checked by the Verifier)
- The size of the locals array that the method will require.
- A "zero init flag" that indicates whether the local variables and memory pool should be initialized by the EE (see also **localloc**).
- Type of each local variable in the form of a signature of the local variable array (called the "locals signature").

In addition, the file format is capable of indicating the degree of portability of the file. There are two kinds of restrictions that can be described:

- Restriction to a specific (32-bit or 64-bit) natural size for integers.
- Restriction to a specific "endian-ness" (i.e. whether bytes are stored left-to-right or right-to-left within a machine word).

By stating what restrictions are placed on executing the code, the EE class loader can prevent non-portable code from running on an architecture that it cannot support.

6 Machine State

One of the design goals of the Execution Engine is to hide the details of a method call frame from the IL code generator. This allows the EE (and not the IL code generator) to choose the most efficient calling convention and stack layout. To achieve this abstraction, the call frame is integrated into the EE. The machine state definitions below reflect these design choices, where machine state consists primarily of global state and method state.

6.1 The Global State

The Execution Engine manages multiple concurrent threads of control (not necessarily the same as the threads provided by a host operating system), multiple managed heaps, and a shared memory address space. A thread of control can be thought of, somewhat simplistically, as a singly linked list of *method states*, where a new state is created and linked back to the current state by a method call instruction – the traditional model of a stack-based calling sequence. Notice that this model of the thread of control doesn't correctly explain the operation of **tail.**, **jmp**, **jmpi**, or **throw** instructions.

Figure 3 illustrates the machine state model , which includes threads of control, method states, and multiple heaps in a shared address space. Method state, shown separately in Figure 4, is an abstraction of the stack frame. Arguments and local variables are part of the method state, but they can contain Object References that refer to data stored in any of the managed heaps.

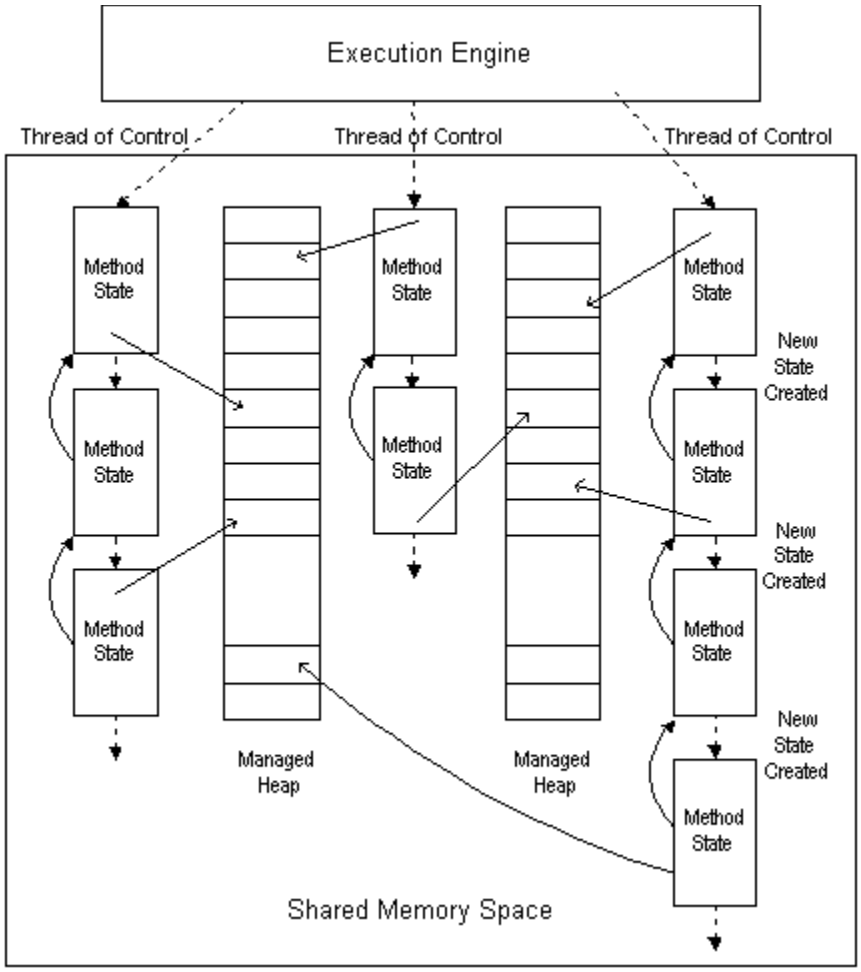

Figure 3. Machine State Model

6.2 The Memory Store

By "memory store" we mean the regular process memory that the Execution Engine operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a data object. The Execution Engine accesses data objects in the memory store via the **ldind.*** and **stind.*** instructions.

6.2.1 Alignment

Alignment of datatypes larger than 1 byte is dependent on the target CPU. It is strongly recommended that primitive datatypes be aligned to the size of that datatype. That is I2 and U2 start on even address; I4, U4, and R4 start on an address divisible by 4; and I8, U8, and R8 start on an address divisible by 8. The natural size types (I, U, and &) are always generated by the EE aligned to their natural size (4 bytes or 8 bytes, depending on architecture). When generated externally, these should also be aligned to their natural size, but portable code may choose to enforce the stronger restriction of 8 byte alignment which is guaranteed to be architecture independent.

There is a special prefix instruction, **unaligned.**, that can immediately precede a **ldind**, **stind, in itblk**, or **cpblk** instruction. It indicates that the data may not be fully aligned and requires that the JIT generate code that will not cause unaligned memory faults.

6.2.2 Byte Ordering

For datatypes larger than 1 byte, the byte ordering is dependent on the target CPU. Code that depends on byte ordering may not run on all platforms. The PE file format (see the Executable Image Information section) allows the file to be marked to indicate that it depends on a particular type ordering.

Figure 4. Method State

6.3 Method State

Method state describes the environment within which a method executes. (In conventional compiler terminology, it corresponds to a superset of the information captured in the "invocation stack frame"). NGWS method state consists of the following items:

- An instruction pointer (**IP**). This points to the next IL instruction to be executed by the EE in the present method.
- An *evaluation stack*. The stack is empty upon method entry. Its contents are entirely local to the method and are preserved across call instructions (that's to say, if this method calls another, once that other method returns, our evaluation stack contents are "still there"). The evaluation stack is not addressable. At all times it is possible to deduce which one of a reduced set of types is stored in any stack location (see the Evaluation Stack section).
- A *local variable array* (starting at index 0). Values of local variables are preserved across calls (in the same sense as for the evaluation stack). A local variable can hold any data type. However, a particular slot must be used in a type consistent way (where the type system is the one described in the Evaluation Stack section). Local variables are initialized to 0 before entry if the initialize flag for the method is set (see the Opaque Classes section). The address of an individual local variable can be taken using the **ldloca** instruction.
- An *argument array*. The values of the current method's incoming arguments (starting at index 0). These can be read and written by logical index. The address of an argument can be taken using the **ldarga** instruction. The address

of an argument is also implicitly taken by the **arglist** instruction for use in conjunction with typesafe iteration through variable-length argument lists.

- A *methodInfo* handle. This contains read-only information about the method. In particular it holds the signature of the method, the types of its local variables, and data about its exception handlers.
- A *local memory pool*. The EE includes instructions for dynamic allocation of objects from the local memory pool (**localloc**). Memory allocated in the local memory pool is *addressable*. The memory allocated in the local memory pool is reclaimed upon method context termination.
- A *return state* handle. This handle is used to restore the method state on return from the current method. Typically, this would be the state of the method's caller; however, both the return state and the return instruction pointer can be adjusted through the code manager (see the Code Manager Specification). (This item corresponds approximately to what in conventional compiler terminology would be the *dynamic link*)
- A *security descriptor*. This descriptor is not directly accessible to managed code but is used by the NGWS security system to record security overrides (**assert**, **permit-only**, and **deny**). From verified code the only way to access this information is through the code manager (see the Code Manager Specification).

Note that we describe the four areas of the method state – incoming arguments array, local variables array, local memory pool and evaluation stack – as if logically distinct areas. This is important, since this is a specification of the EE architecture. However, in practice, the EE may actually map these areas into one contiguous array of memory, held as a conventional stack frame on the underlying, target architecture.

6.3.1 The Evaluation Stack

Associated with each method state is an evaluation stack. Most EE instructions retrieve their arguments from the evaluation stack and place their return values on the stack. Arguments to other methods and their return values are also placed on the evaluation stack. When a procedure call is made the arguments to the called methods become the incoming arguments array (see Section 6.3.2) to the method. This may require a memory copy, or simply a sharing of these two areas by the two methods.

The evaluation stack is made up of slots that can hold any data type, including an unboxed instance of a value type. The type state of the stack (the stack depth and types of each element on the stack) at any given point in a program must be identical for all possible control flow paths. For example, a program that loops an unknown number of times and pushes a new element on the stack at each iteration would be prohibited.

While the EE, in general, supports the full set of types described in Table 1, the EE treats the evaluation stack in a special way. While some JIT compilers may track the types on the stack in more detail, the EE only requires that values be one of:

- I8, an 8-byte signed integer
- I4, a 4-byte signed integer
- I, a signed integer of either 4 or 8 bytes, whichever is more convenient for the target architecture

- F, a floating point value (R4, R8, R4Result, R8Result or RPrecise)
- &, a managed pointer
- O, an object reference
- *, a "transient pointer," which can be used only within the body of a single method, that points to a value known to be in unmanaged memory (see the IL Instruction Set specification for more details. * types are generated internally within the EE; they are not created by the user).

The other types are synthesized through a combination of techniques:

- Shorter integer types in other memory locations are zero-extended or sign-extended when loaded onto the evaluation stack; these values are truncated when stored back to their home location.
- Special instructions perform numeric conversions, with or without overflow detection, between different sizes and between signed and unsigned integers.
- Special instructions treat an integer on the stack as though it were unsigned.
- Instructions that create pointers which are guaranteed not to point into the memory manager's heaps (e.g. **ldloca**, **ldarga**, and **ldsflda**) produce transient pointers (type ***) which can be used wherever a managed pointer (type **&**) or unmanaged pointer (type **U**) is expected.
- When a method is called, an unmanaged pointer (type **U** or ***) is permitted to match a parameter that requires a managed pointer (type **&**). The reverse, however, is *not* permitted since it would allow a managed pointer to be "lost" by the memory manager.
- A managed pointer (type **&**) can be explicitly converted to an unmanaged pointer (type **U**), although this is not verifiable and may produce a runtime exception.

6.3.2 Local Variables and Arguments

Part of each method state is an array that holds local variables and an array that holds arguments. Like the evaluation stack, each element of these arrays can hold any single data type or an instance of a value type. Both arrays start at 0 (that is, the first argument or local variable is numbered 0). The address of a local variable can be computed using the **ldloca** instruction, and the address of an argument using the **ldarga** instruction.

Associated with each method is metadata that specifies:

- whether the local variables and memory pool memory must be initialized when the method is entered
- the type of each argument and the length of the argument array (but see below for variable argument lists)
- the type of each local variable and the length of the local variable array.

The EE inserts padding as appropriate for the target architecture. That is, on some 64-bit architectures all local variables may be 64-bit aligned, while on others they may be 8-, 16-, or 32-bit aligned. The IL generator must make no assumptions about the offsets of local variables within the array. In fact, the EE is free to reorder the elements in the local variable array, and different JITters may choose to order them in different ways.

6.3.3 Variable Argument Lists

The Execution Engine works in conjunction with the NGWS Class library to implement methods that accept argument lists of unknown length and type ("varargs methods"). Access to these arguments is through a typesafe iterator in the Class Library, called **System.ArgIterator**.

The IL includes one instruction provided specifically to support the argument iterator, **arglist**. This instruction can be used only within a method that is declared to take a variable number of arguments. It returns a value that is needed by the constructor for a **System.ArgIterator** object. Basically, the value created by **arglist** provides access both to the address of the argument list that was passed to the method and a runtime data structure that specifies the number and type of the arguments that were provided. This is sufficient for the class library to implement the user visible iteration mechanism.

From the EE point of view, varargs methods have an array of arguments like other methods. But only the initial portion of the array has a fixed set of types and only these can be accessed directly using the **ldarg**, **starg**, and **ldarga** instructions. The argument iterator allows access to both this initial segment and the remaining entries in the array.

6.3.4 Local Memory Pool

Part of each method state is a local memory pool. Memory can be explicitly allocated from the local memory pool using the **localloc** instruction. All memory in the local memory pool is reclaimed on method exit, and that is the only way local memory pool memory is reclaimed (there is no instruction provided to *free* local memory that was allocated during this method invocation). The local memory pool is used to allocate objects whose type or size is not known at compile time and which the programmer does not wish to allocate in the managed heap.

Because the local memory pool cannot be shrunk during the lifetime of the method, a language implementation cannot use the local memory pool for general-purpose memory allocation .

7 Control Flow

The IL instruction set provides a rich set of instructions to alter the normal flow of control from one IL instruction to the next.

- **Conditional and Unconditional Branch** instructions for use within a method, provided the transfer doesn't cross a protected region boundary (see the Exception Handling section).
- **Method call** instructions to compute new arguments, transfer them and control to a known or computed destination method (see the Method Calls section).
- **Tail call** prefix to indicate that a method should relinquish its stack frame before executing a method call (see the Method Calls section).
- **Return** from a method, returning a value if necessary.
- **Method jump** instructions to transfer the current method's arguments to a known or computed destination method (see the Method Calls section).
- **Exception-related** instructions (see the Exception Handling section). These include instructions to initiate an exception, transfer control out of a protected region, and end a filter, catch clause, or finally clause.

While the EE supports arbitrary control transfers within a method, there are several restrictions that must be observed, and which are tested by the verifier:

1. Control transfer is never permitted to enter a catch handler or finally clause (see the Exception Handling section) except through the exception handling mechanism.
2. Control transfer out of a protected region (see the Exception Handling section) is only permitted through an exception instruction (**leave**, **end.filter**, **end.catch**, or **end.finally**).
3. The evaluation stack must be empty after the return value is popped by a **ret** instruction.
4. All slots on the stack must have the same data type at every point within the method body, regardless of the control flow that allows execution to arrive there.
5. In order for the JIT compilers to efficiently track the data types stored on the stack, the stack must normally be empty at the instruction following an unconditional control transfer instruction (**br**, **br.s**, **ret**, **jmp**, **jmpi**, **throw**, **end.filter**, **end.catch**, or **end.finally**). The stack is allowed to be non-empty at this point only if at some earlier location within the method there has been a forward branch to that location.
6. Control is not permitted to simply "fall through" the end of a method. All paths must terminate with one of these instructions: **ret**, **throw**, **jmp**, **jmpi**, or (**tail.** followed by **call**, **calli**, or **callvirt**).

8 Method Calls

An important design goal of the EE is to abstract the layout of native method frames, including calling convention. That is, instructions emitted by the IL code generator contain sufficient information for different implementations of the EE to use different native calling convention. All method calls initialize the method state areas (see the Method State section) as follows:

- The incoming arguments array is set by the caller to the desired values.
- The local variables array always has **null** for Object types and for fields within value types that hold objects. In addition, if the "zero init flag" is set in the method header, then it is initialized to 0 for all integer types and 0.0 for all floating point types. Value Types are not initialized by the EE, but verified code will supply a call to an initializer as part of the method's entry point code.
- If the "zero init flag" is set in the method header the local memory pool is initialized to all zeros.
- The evaluation stack is empty.

8.1 Call Site Descriptors

To support this flexibility, call sites need additional information that enables an interpreter or JIT compiler to synthesize any native calling convention. All IL calling instructions (**call, calli,** and **callvirt**) include as part of the instruction a description of the call site. This description can take one of two forms. The simpler form, used with the **calli** instruction, is a "call site description" (represented as a metadata token for a stand-alone call signature, see the Metadata Specification) that provides:

- The number of arguments being passed.
- The data type of each argument.
- The order in which they have been placed on the call stack.
- The native calling convention to be used

The more complicated form, used for the **call** and **callvirt** instructions, is a "method reference" (a metadata **methodref** token, see the Metadata Specification) that augments the call site description with an identifier for the target of the call instruction.

8.2 Calling Instructions

The IL has three call instructions that are used to transfer new argument values to a destination method. Under normal circumstances, the called method will terminate and return control to the calling method.

- **call** is designed to be used when the destination address is fixed at the time the IL is linked. In this case, a method reference is placed directly in the instruction. This is comparable to a direct call to a static function in C. It can be used to call static or instance methods or the (statically known) superclass method within an instance method body.

- **calli** is designed for use when the destination address is calculated at run time. A method pointer is passed on the stack and the instruction contains only the call site description.
- **callvirt**, part of the IL VOS instruction set, uses the class of an object (known only at runtime) to determine the method to be called. The instruction includes a method reference, but the particular method isn't computed until the call actually occurs. This allows an instance of a subclass to be supplied and the method appropriate for that subclass to be invoked. The **callvirt** instruction is used both for instance methods and methods on interfaces. For further details, see the VOS specification and the IL Instruction Set specification.

In addition, each of these instructions can be immediately preceded by a **tail.** instruction prefix. This specifies that the calling method terminates with this method call (and returns whatever value is returned by the called method). The **tail.** prefix instructs the JIT compiler to discard the caller's method state prior to making the call (if the call is from untrusted code to trusted code the frame cannot be fully discarded for security reasons). When the called method executes a **ret** instruction, control returns not to the calling method but rather to wherever that method would itself have returned (typically, return to caller's caller). Notice that the **tail.** instruction shortens the lifetime of the caller's frame so it is unsafe to pass managed pointers (type **&**) as arguments.

Finally, there are two instructions that indicate an optimization of the **tail.** case:

- **jmp** is followed by a **methodref** or **methoddef** token and indicates that the current method's state should be discarded, its arguments should be transferred intact to the destination method, and control should be transferred to the destination. The signature of the calling method must exactly match the signature of the destination method.
- **jmpi** takes a computed destination address on the stack, pops it off the stack, discards the current method state, transfers the current arguments to the destination method, and transfers control to the destination method. The signature of the calling method must exactly match the signature of the destination method.

8.3 Computed Destinations

The destination of a method call can be either encoded directly in the IL instruction stream (the **call** and **jmp** instructions) or computed (the **callvirt**, **calli**, and **jmpi** instructions). The destination address for a **callvirt** instruction is automatically computed by the Execution Engine based on the method token and the value of the first argument (the **this** pointer). The method token must refer to a virtual method on a class that is a direct ancestor of the class of the first argument. The EE computes the correct destination by, effectively, locating the nearest ancestor of the first argument's class that supplies an implementation of the desired method (the implementation can be assumed to be more efficient than the linear search implied here).

For the **calli** and **jmpi** instructions the IL code is responsible for computing a destination address and pushing it on the stack. This is typically done through the use of a **ldftn** or **ldvirtfn** instruction at some earlier time. The **ldftn** instruction includes a metadata token in the IL stream that specifies a method, and the instruction pushes the address of that method. The **ldvirtfn** instruction takes a metadata token for a virtual method in the IL stream and an object on the stack. It

performs the same computation described above for the **callvirt** instruction but pushes the resulting destination on the stack rather than calling the method.

The **calli** instruction includes a call site description that includes information about the native calling convention that should be used to invoke the method. The EE does *not* check that this correctly matches the calling convention for the method that is being called; any mismatch will result in unpredictable behavior. The **jmpi** instruction requires that the destination method have the same calling convention and the method that contains the **jmpi** instruction; any mistmatch will result in unpredictable behavior.

8.4 Virtual Calling Convention

The IL provides a "virtual calling convention" that is converted by the JIT into a native calling convention. The JIT determines the optimal native calling convention for the target architecture. This allows the native calling convention to differ from machine to machine, including details of register usage, local variable homes, copying conventions for large call-by-value objects (as well as deciding, based on the target machine, what is considered "large"). This also allows the JIT to reorder the values placed on the IL virtual stack to match the location and order of arguments passed in the native calling convention.

The EE uses a single uniform calling convention for all method calls. It is the responsibility of the JITters to convert this into the appropriate native calling convention. The virtual calling convention is:

1. If the method being called is an instance method (class or interface) or a virtual method, first push the **this** pointer. For methods on Objects (including boxed value types), the **this** pointer is of type **O** (object reference). For methods on value types, the **this** pointer is provided as a by-ref parameter; that is, the value is a pointer (managed, **&**, or unmanaged, ***** or **I**) to the instance.
2. Push the remaining arguments in left-to-right order (that is, push the lexically, leftmost argument first). The Parameter Passing section describes how each of the three parameter passing conventions (by-value, by-reference, and typed reference) should be implemented.
3. Execute the appropriate call instruction (**call**, **calli**, or **callvirt** any of which may be preceded by **tail.**).

8.5 Parameter Passing

The EE supports three kinds of parameter passing, all indicated in metadata as part of the signature of the method. Each parameter to a method has its own passing convention (e.g., the first parameter may be passed by-value while all others are passed by-ref). Parameter may be passed as follows:

- **By-value** parameters, where the **value** of an object is passed from the caller to the callee.
- **By-ref** parameters, where the **address** of the data is passed from the caller to the callee, and the type of the parameter is therefore a managed or unmanaged pointer.
- **Typed reference** parameters, where a runtime representation of the data type is passed along with the address of the data, and the type of the parameter is therefore one specially supplied for this purpose.

It is the responsibility of the IL generator to follow these conventions. The verifier checks that the types of parameters match the types of values passed, but is otherwise unaware of the details of the calling convention.

8.5.1 By-Value Parameters

For primitive types (integers, floats, etc.) the caller copies the value onto the stack before the call. For `Object`s the object reference (type **O**) is pushed on the stack. For managed pointers (type **&**) or unmanaged pointers (type **U**), the address is passed from the caller to the callee. For value types, the protocol described in the Operations on Value Type Instances section is used.

8.5.2 By-Ref Parameters

By-Ref Parameters are the equivalent of C++ reference parameters or PASCAL **`var`** parameters: instead of passing as an argument the value of a variable, field, or array element, its address is passed instead; and any assignment to the corresponding parameter actually modifies the corresponding caller's variable, field, or array element. Much of this work is done by the higher-level language, which hides from the user the need to compute addresses to pass a value and the use of indirection to reference or update values.

Passing a value by reference requires that the value have a home (see the Homes for Values section) and it is the address of this home that is passed. Constants, and intermediate values on the evaluation stack, cannot be passed as by-ref parameters because they have no home.

The EE provides instructions to support by-ref parameters:

- calculate addresses of home locations (see Table 3)
- load and store primitive data types through these address pointers (**ldind.***, **stind.*, ldfld,** etc.)
- copy value types (**ldobj** and **cpobj**).

Some addresses (e.g., local variables and arguments) have lifetimes tied to that method invocation. These cannot be referenced outside their lifetimes, and so they should not be stored in locations that last beyond their lifetime. The IL does not (and cannot) enforce this restriction, so the IL generator must enforce this restriction or the resulting IL will not work correctly. For code to be verifiable (see the Verification section) by-ref parameters may **only** be passed to other methods or referenced via the appropriate **stind** or **ldind** instructions.

8.5.3 Typed Reference Parameters

By-ref parameters and value types are sufficient to support statically typed languages (C++, Pascal, etc.). They also support dynamically typed languages that pay a performance penalty to box value types before passing them to polymorphic methods (Lisp, Scheme, SmallTalk, etc.). Unfortunately, they are not sufficient to support languages like Visual Basic that require by-reference passing of unboxed data to methods that are not statically restricted as to the type of data they accept. These languages require a way of passing *both* the address of the home of the data *and* the static type of the home. This is exactly the information that would be

provided if the data were boxed, but without the heap allocation required of a box operation.

Typed reference parameters address this requirement. A typed reference parameter is very similar to a standard by-ref parameter but the static data type is passed as well as the address of the data. Like by-ref parameters, the argument corresponding to a typed reference parameter must have a home. If it were not for the fact that the verifier and the memory manager must be aware of the data type and the corresponding address, a by-ref parameter could be implemented as a standard value type with two fields: the address of the data and the type of the data. Like a regular by-ref parameter, a typed reference parameter can refer to a home that is on the stack, and that home will have a lifetime limited by the call stack. Thus, the IL generator must apply appropriate checks on the lifetime of by-ref parameters; and the verifier imposes the same restrictions on the use of typed reference parameters as it does on by-ref parameters (see the By-Ref Parameters section).

A typed reference is passed by either creating a new typed reference (using the **mkrefany** instruction) or by copying an existing typed reference. Given a typed reference argument, the address to which it refers can be extracted using the **refanyval** instruction; the type to which it refers can be extracted using the **refanytype** instruction.

8.5.4 A Note on Interactions

A given parameter can be passed using any one of the parameter passing conventions: by-value, by-ref, or typed reference. No combination of these is allowed for a single parameter, although a method may have different parameters with different calling mechanisms.

There are a pair of non-obvious facts about the parameter passing convention:

1. A parameter that has been passed in as typed reference cannot be passed on as by-ref or by-value without a runtime type check and (in the case of by-value) a copy.
2. A by-ref parameter can be passed on as a typed reference by attaching the static type.

Table 4 illustrates the parameter passing convention used for each data type.

Type of data	Pass By	How data is sent
Built-in value type (int, float, etc.)	Value	Copied to called method, type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method
User-defined value type	Value	Called method receives a copy; type statically known at both sides
	Reference	Address sent to called method, type statically known at both sides
	Typed reference	Address sent along with type information to called method

Object	Value	Reference to data sent to called method, type statically known and class available from reference
	Referenc e	Address of reference sent to called method, type statically known and class available from reference
	Typed reference	Address of reference sent to called method along with static type information, class (i.e. dynamic type) available from reference

Table 4: Parameter Passing Conventions

9 Exception Handling

The EE supports an exception handling model based on the idea of exception objects and protected blocks of code. When an exception occurs, an object is created to represent the exception. All exceptions objects are boxed instances of some subclass of **System.Exception**. Users can create their own exception classes by subclassing **System.Exception**.

There are four kinds of handlers for protected blocks. A single protected block can have exactly one handler associated with it:

1. A **finally handler** which must be executed whenever the block exits, regardless of whether that occurs by normal control flow or by an unhandled exception.
2. A **fault handler** which must be executed if an exception occurs, but not on completion of normal control flow.
3. A **type-filtered handler** that handles any exception of a specified class or any of its sub-classes.
4. A **user-filtered handler** that runs a user-specified set of IL instructions to determine whether the exception should be ignored (i.e. execution should resume), handled by the associated handler, or passed on to the next protected block.

Protected regions, the type of the associated handler, and the location of the associated handler and (if needed) user-supplied filter code are described through an Exception Handler Table associated with each method. The exact format of the Exception Handler Table is specified in detail in the File Format Specifiation. Details of the exception handling mechanism are specified in the Exception Specification.

9.1 Exceptions Thrown by the EE Itself

EE instructions can throw the following exceptions as part of executing individual instructions. The documentation on a particular instruction will list all the exceptions the instruction can throw (except for the general purpose ExecutionEngineException described below that can be generated by all instructions).

Base Instructions

ArithmeticException

DivideByZeroException

ExecutionEngineException

InvalidAddressException

OverflowException

SecurityException

StackOverflowException

Object Model Instructions

TypeLoadException

IndexOutOfRangeException

InvalidAddressException

InvalidCastException

MissingFieldException

MissingMethodException

NullReferenceException

OutOfMemoryException

SecurityException

StackOverflowException

The ExecutionEngineException is special. It can be thrown by any instruction and indicates an unexpected inconsistency in the EE. Code that has been passed through the code verifier should never throw this exception (it is a defect in either the verifier or the EE if it does). However, unverified code can cause this error if the code is corrupt or inconsistent in some way.

Note that, because of the verifier, there are no exceptions for things like 'MetaDataTokenNotFound.' The verifier can detect this inconsistency before the instruction is ever executed (the code is then considered unverified). If the code has not been verified, this type of inconsistency would raise the generic ExecutionEngineException.

Exceptions can also be thrown by the NGWS runtime, as well as by user code, using the **throw** instruction. The handing of an exception is identical, regardless of the source.

9.2 Overview of Exception Handling

See the Exception Handling specification for details.

Each method in an executable has associated with it a (possibly empty) array of exception handling information. Each entry in the array describes a protected block, its filter, and its handler (which may be a **catch** handler, a **finally** handler, or a **fault** handler). When an exception occurs, the EE searches the array for the first protected block that

- Protects a region including the current instruction pointer *and*
- Is a catch handler block *and*
- Whose filter wishes to handle the exception

If a match is not found in the current method, the calling method is searched, and so on. If no match is found the EE will dump a stack trace and abort the program. If a match is found, the EE walks the stack back to the point just located, but this time calling the **finally** and **fault** handlers. It then starts the corresponding exception handler. Stack frames are discarded either as this second walk occurs or after the handler completes, depending on information in the exception handler array entry associated with the handling block.

Some things to notice are:

- The ordering of the exception clauses in the Exception Handler Table is important. If handlers are nested, the most deeply nested try blocks must come before the try blocks that enclose them.
- Exception handlers can access the local variables and the local memory pool of the routine that catches the exception, but any intermediate results on the evaluation stack at the time the exception was thrown are lost.
- An exception object describing the exception is automatically created by the execution engine and pushed onto the evaluation stack as the first item upon entry of a filter or catch clause.
- Execution cannot be resumed at the location of the exception. This restriction may be relaxed in the future.

9.3 IL Support for Exceptions

The IL has special instructions to:

- **Throw** and **rethrow** a user-defined exception.
- **Leave** a protected block and execute the appropriate **finally** clauses within a method, without throwing an exception. This is also used to exit a **catch** clause. Notice that leaving a protected block does **not** cause the fault clauses to be called.
- End a user-supplied filter clause (**endfilter**) and return a value indicating whether to handle the exception.
- End a finally clause (**endfinally**) and continue unwinding the stack.

9.4 Lexical Nesting of Protected Blocks

This section summarizes restrictions that are described in detail in the Exception Handling Specification.

The following restrictions below refer to the *lexical* nesting of **try** blocks and their associated handlers:

1. A **try** block may have associated with it any *one* of the following:
   - a **catch** block (with an implied filter based on the type of the exception)
   - a **filter** block and a **catch** block
   - a **finally** block
   - a **fault** block
2. A single **try** block, **filter** block, **catch** block, **fault** block, or **finally** block must constitute a contiguous block of IL instructions.
3. The exception table fully specifies the range of the **try**, **catch**, **fault** and **finally** blocks, but only specifies the entry point for the **filter** block. The **filter** block lexically ends with the (required and unique) **endfilter** instruction for that block.
4. Multiple **try** blocks that specify precisely the same range of instructions are considered to be a single try block with multiple associated handler blocks. The associated handler blocks must be either **catch** blocks or **filter** and **catch** blocks,

in any combination. There cannot be any **finally** or **fault** blocks associate with these **try** blocks. (To model a source-level construct that has, for example, a **try** with two associated **catch** handlers and a **finally**, you must have three entries: two **try**/**catch** entries specifying the same region of instructions, and an enclosing **try**/**finally** that covers both the other **try** block *and* their handlers.)

5. The addresses included in **catch** blocks, **filter** blocks, **fault** blocks, and **finally** blocks must not overlap one another, nor can any handler block be shared between multiple **try** blocks.
6. The region of IL instructions associated with a **try** block cannot include its own **filter**, **fault**, **finally**, or **catch** block (i.e. it is just the protected code, not the handlers that are associated with it).
7. A block of any kind (except the **try** block associated with a **fault** block) that encloses a **try** block must include all of the code associated with the inner **try** as well as the handlers associated with the inner **try**.
8. A **try** block that has an associated **fault** block may overlap another **try** block that has a **fault** block (but the **fault** blocks themselves may not overlap one another).
9. A **try** block cannot appear within a **filter** block.

9.5 Control Flow Restrictions on Protected Blocks

The following restrictions are about the control flow into, out of, and between **try** blocks and their associated handlers.

1. Correct IL code must not enter a **filter**, **catch**, **fault** or **finally** block except through the NGWS exception handling mechanism.
2. There are only two ways to enter a **try** block from outside its lexical body:

   a) **Branching to or falling into the try block's first instruction**. The branch can be made using a conditional branch, an unconditional branch, or a **leave** instruction.

   b) **Using a leave instruction within the catch block associated with the try**. In this case correct IL code can branch to any address within the **try** block, not just its first instruction.
3. Upon entry to a **try** block the evaluation stack must be empty.
4. The only ways correct IL code can leave a **try**, **filter**, **catch** or **finally** block are as follows:

   a) **throw** from any of them.

   b) **leave** from the body of a **try** or **catch** (in this case the destination of the **leave** must have an empty evaluation stack and the **leave** instruction has the side-effect of emptying the evaluation stack).

   c) **endfilter** may appear only as the lexically last instruction of a **filter** block, and it must always be present (even if it is immediately preceded by a **throw** or other unconditional control flow). If reached, the evaluation stack must contain an **I4** when the **endfilter** is executed, and the value is used to determine how exception handling should proceed.

d) **endfinally** from anywhere within a **finally**, with the side-effect of emptying the evaluation stack.

e) **rethrow** from within a **catch** block, with the side-effect of emptying the evaluation stack.

f) fall through the end of a **try** block (falling through the end of any other kind of block is not permitted).

5. When the try block is exited with a leave instruction, the evaluation stack must be empty for correct IL.
6. When a catch or filter clause is exited with a leave instruction, the evaluation stack must be empty for correct IL. This involves popping off the exception object from the evaluation stack which was automatically pushed onto the stack.
7. Correct IL code must not exit any block using a **ret** instruction.

10Atomicity of Memory Accesses

The EE makes several assumptions about atomicity of memory references, and these translate directly into rules required of either programmers or translators from high-level languages into IL.

- Read and write access to word-length memory locations (types **I** and **U**) that are properly aligned is atomic. Correct translation from IL to native code requires generation of native code sequences that supply this atomicity guarantee. Note that there is no guarantee about atomic update (read-modify-write) of memory.
- Read and write access to 4-byte data (**I4** and **U4**) that is aligned on a 4-byte boundary is atomic, even on a 64-bit machine. Again, there is no guarantee about atomic read-modify-write.
- One- and Two-byte data that does not cross a word boundary will be read atomically, but writing will write the entire word back to memory.
- No other memory references are performed atomically.

When the EE controls the layout of managed data, it pads the data so that if an object starts at a word boundary all of the fields that require 4 or fewer bytes will be aligned so that reads will be atomic. The managed heap always aligns data that it allocates to maintain this rule, so heap references (type **O**) to data that does not have explicit layout will occur atomically where possible. Similarly, static variables of managed classes are allocated so that they, too, are aligned when possible. The EE aligns stack frames to word boundaries, but does not attempt to align to an 8-byte boundary on 32-bit machines even if the frame contains 8-byte values.

11OptIL: An Instruction Set Within IL

A fundamental issue associated with generating intermediate IL is how much of the work is done by the IL generator and how much of the work is done by the Execution Engine (via a JIT compiler). The IL instruction set was designed to be easy for compilers to generate so that IL can be generated quickly in rapid application development (RAD) environments, where compile speed and ease of debugging are at a premium.

On the other hand, in situations where load time is important, it is useful to do as much work as possible in the code generator, before the executable is loaded. In particular it is useful to do expensive optimizations like common sub-expression elimination, constant folding, loop restructuring, and even register allocation in the code generator (as would be done in a traditional compiler). The instruction set should be able to represent such optimized code as well.

Finally, in some environments it is important that the JITter be small and run in a nearly constant amount of memory, even for large methods. The instruction set should allow a compiler to compute information and pass it on to the JITter that will reduce the memory required by the JITter (e.g., register allocation and branch targets).

In the NGWS runtime environment, an optimizing compiler can best express many optimizations by generating OptIL. OptIL is optimized code represented using the same IL instruction set; however, OptIL differs from non-OptIL code in the following ways

- Many transformations will have been done (e.g., loop restructuring, constant folding, CSE).
- The code will obey certain conventions (e.g., method calls are not nested).
- There will be additional annotations (e.g., exactly when each variable is used for the last time).

The exact restrictions an executable must satisfy to be of this form are described in the "Opt-IL Specification". The "IL Instruction Set" specification contains a detailed description of each of the IL instructions.

Note that an OptIL program is still a valid IL program (it can be run by the normal EE), but because it has been optimized by the code generator it can be compiled to native code very quickly and using little memory.