Introducing reference counting into the Common Language Runtime

Memory management strategies in a managed world

Arild Fines

Introducing reference counting into the Common Language Runtime: Memory management strategies in a managed world

by Arild Fines

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

1.1. The complexity of software

Computer software is getting more and more complex. Software projects of today, while equal in actual scope, are far more ambitious in their goals than those of the IBM OS/360 project to which Frederick Brooks relates in his seminal work “The Mythical Man-Month”[1]. While software engineering as a field is better understood now than it was in the mid-sixties, you will find few researchers in the field claiming that this understanding has kept up with the growth in the complexity of software projects.

In fact, in a world where Visual Basic remains one of the most popular programming languages, one can easily argue that the developer pool has a larger portion of relatively unskilled programmers now than ever. Yet, some of the software even from this segment has complexity requirements that would leave the Fred Brooks anno 1965 dumbfounded. And while not all of this software will stand as shiny examples of software engineering, it is hard to deny the fact that software written by unskilled developers plays an important part in our society.

While software engineering hasn't quite been able to meet the expectations once held for it, it is obvious that something has enabled programmers to build increasingly more complex software. And if this “something” isn't a vastly increased understanding of the art and science of software engineering, what is it?

The answer is complex and probably worth of a thesis or two of its own, but one thing is certain: the tools and languages we use today are orders of magnitude better than what the developers under Brooks had available. OS/360 was written wholly in assembly code, which was (and still is) hard to write and debug, mainly because of its lack of abstraction and its conceptual distance from the problem domain. Fred Brooks himself mentioned the advent of higher level programming languages as one of the things most likely to advance the field of software engineering [1].

In contrast to the situation in Brooks' day, we currently have at our disposal a vast array of very high level languages. Languages like Visual Basic, Python,

[1] Likewise, in his retrospective essay “No Silver Bullet”[2], he mentions object-oriented programming as one of the things that indeed came closest to being a “silver bullet”.

Java and C# maintain a high level of abstraction from the actual hardware, and often come with massive standard libraries that enables the developer to focus on the core functionality of his application, rather than constantly having to reinvent the wheel.

While the availability of high quality standard libraries go a long way in explaining the increased productivity of todays' developers, it is hardly the only reason. One other feature the aforementioned languages all have in common is that they free the developer from the responsibility of manually managing memory. Memory related bugs are all too common in traditional lower-level languages (who has never accessed a pointer whose backing memory has been freed[2]?), and tracking down these bugs can take up an inordinate amount of developer time, especially in more complex software. The newer languages basically make this a non-issue by handling the allocation and de-allocation of memory in the language runtime, thus relieving the developer of this rather tedious burden.

And while hard core C/C++ developers still maintain, in some kind of misunderstood machismo, that memory management "is too important to be left to the computer"[3] it is becoming more and more clear that automatic memory management, in various implementations, is here to stay. The past couple of years have seen a steady shift towards the use of higher level programming languages and tools for all but the most low-level of development tasks (operating systems, drivers, 3D graphics engines etc.) And the higher level languages all invariably support automatic memory management.

1.2. The Common Language Runtime

Enter the Common Language Runtime (CLR): A rich execution engine that provides services and support for a wide variety of computer languages, ranging from low level ones, such as C and C++, to very high level languages such as IronPython, Nemerle and Boo. However, any language that targets the CLR is pretty much confined to the memory model provided by the execution engine.

[2] This is commonly referred to as a "dangling" pointer.

[3] It is a long-standing joke: C programmers have long understood that memory management is so critical it can't be left up to the system, and Lisp programmers have long understood that memory management is so critical, it can't be left up to the programmers.

This model, based on a tracing garbage collector, is in its very nature undeterministic: it provides no guarantees as to when an object will be marked as garbage and disposed of. If the object holds onto resources not related to memory, such as file handles and other items in scarce supply, disposing of an object at the earliest possible moment might be crucial in order to attain the maximum possible performance.

To illustrate, an example might be in order:

Example 1.1. Memory pressure

```
for (int i = 0; i < 42000; i++)
{
   LargeObject obj = new LargeObject();
   obj.DoStuff();
}
```

In the above code, 42000 instances of the `LargeObject` is created inside a tight loop. Now, what happens at the end brace? Is the object created inside the loop disposed of, releasing the memory allocated and any unmanaged resources associated with it? In the CLR, the answer is no. New instances of `LargeObject` will continue to be allocated, but the old ones will linger up until the time when the garbage collector runs out of space to allocate the new ones.

If “Large” in `LargeObject` does not refer to the actual amount of heap memory allocated for the object, but another resource held by it, this might take a long time. Consider the case where you have a class like this:

Example 1.2. A type holding onto an expensive unmanaged resource

```
class LargeObject
{
  Foo()
  {
    this.expensiveUnmanagedResource = AcquireResource();
  }

  ~LargeObject()
  {
     FreeResource(this.expensiveUnmanagedResource);
  }

  // ...

  private IntPtr expensiveUnmanagedResource;

}
```

In this case, objects of the `LargeObject` type only occupy 4 bytes of heap memory, plus whatever memory is required for internal bookkeeping by the execution engine. We will assume the latter number is 8 bytes, leaving us with 12 bytes per object. Now, the garbage collector should be able to allocate 42000 * 12 = 504 000 bytes on the heap without even breaking a sweat, and a garbage collection might not occur at all in the timeframe of the loop. But this does not take into account the cost of the unmanaged resource. If *`expensiveUnmanagedResource`* represents a file handle, the underlying operating system will run out of these a long time before the garbage collector runs out of memory on one of its heaps.

I will in this thesis investigate the feasibility of introducing a hybrid model of memory management in the CLR, one in which certain objects can be tracked by a reference counting mechanism instead of the tracing garbage collector. If `LargeObject` was managed by reference counting instead of the CLR's garbage collector, each instance would be destroyed and its unmanaged resource released at the end of each iteration of the loop.

The reason for choosing a hybrid model over a pure reference counted mechanism lies in the fact that reference counting implementations have

drawbacks of their own: they can be inefficient in terms of execution time and memory, and they do not handle cycles (in which an object directly or indirectly holds a reference to itself) well. There are existing languages that use a hybrid mechanism, of which the most notable is the Python programming language[4], but the implementation aimed for in this paper differs from these in that the tracing garbage collector will remain the primary memory management mechanism, leaving reference counting for specific scenarios in which they are generally useful.

Another reason for going with a hybrid is the number of implicit assumptions made in the CLR regarding the memory model. For example, there is a class in the *Base Class Libraries* named `GC`, which allows the programmer to have some control over the tracing garbage collector directly. To completely replace the memory management model of the CLR would require either removing this class entirely (which would technically violate the ECMA standard describing the CLR) or making its methods into no-ops.

The primary vehicle for the implementation of such a mechanism in the CLR will be the *Shared Source CLI* (SSCLI), a source implementation of the CLR released by Microsoft. My thesis will focus on determining which changes need to be made to the SSCLI in order to support an hybrid memory management model as described above.

The SSCLI consists of a large amount of code, and considerable time will have to be spent in getting acquainted with it in order to complete this implementation. This paper will therefore describe the workings of the SSCLI in some detail, as well as going into the workings of the current tracing garbage collector in some detail.

Chapter 2. Automatic memory management

2.1. Garbage collection

Automatic memory management is commonly referred to as *garbage collection*, referring to how the mechanism automatically picks up the garbage after the programmer. This term is not altogether precise in some cases, since some of the algorithms focus more on detecting and picking up all *non*-garbage objects, followed by a sweep of whatever remains. The word “garbage” in this context refers to objects that are no longer in use by the program.

2.1.1. The history of garbage collection

While the previous chapter may have appeared to portray garbage collection as a new trend in programming languages, it actually has a history almost as long as that of high level programming itself. In fact, the second high level programming language ever created, Lisp, sported a garbage collector already in its initial specification in 1960 [3]. Subsequent versions of Lisp, including Scheme and Common Lisp, have all included various forms of garbage collectors. It is hard to conceive of a language in the extended Lisp family (functional languages) featuring manual memory management, due to the dynamic nature of their runtime environments.

The Symbolics Lisp Machine was unique in that it had direct hardware support for garbage collection.

The SIMULA programming language was designed and implemented by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo between 1962 and 1967. Recognized as the first object-oriented language, it was also one of the first imperative languages in wide use to utilize a garbage collector. The garbage collector used in Simula-67 required four passes over the object graph in order to determine what was garbage and dispose of it.

The language Smalltalk, inspired by Simula, was developed by Alan Kay and others in the 1970s at Xero'x Palo Alto Research Center (PARC). It also included garbage collection as a central element. Again, this was made necessary by a highly dynamic runtime environment.

In modern times, pretty much every new language introduced support some form of automatic memory management. The concept of garbage collection is usually an inherent feature of the language in question. As mentioned earlier, functional languages like Lisp would be hard to conceive without some kind of automatic memory management. Very high level languages, those often referred to as “scripting”languages (Python, Perl etc...), would also be hard to imagine having manual memory management. For this reason, language specifications usually either explicitly or implicitly indicate that a language should use some form of garbage collection.

2.1.2. Garbage collection and performance

Garbage collectors have long had a bad reputation among developers focussing on raw performance as the primary goal, and they were mostly considered an unneeded luxury by programmers developing in languages like C and C++. Garbage collection by the system would often be a time-consuming operation, and would cause noticeable pauses during program execution. Many programmers felt that this was a waste of system resources and that they would be better off managing memory manually.

This has changed in later years for several reasons: Computer hardware is now faster than it ever was (and is still getting faster, despite dire predictions of the opposite), making the aforementioned pauses far less noticeable, and there has been a lot of research effort into creating ever better algorithms for effective garbage collection. Computer memory (RAM) is cheaper per megabyte now than it has ever been, and virtual memory mechanisms ensure that a large memory space is available to applications.

Of course, progress in this area has been offset somewhat by the need for more and more (and bigger!) objects in even small programs, but Jacob Marner makes a good case[5] for why it is perfectly conceivable to use a garbage collected language like Java for a soft real-time application such as a 3D computer game.

2.1.3. Types of garbage collection mechanisms

One can categorize garbage collection mechanisms in various ways, but one possible rough taxonomy would be to classify them into “tracing” garbage collectors and “reference counting” garbage collectors.

2.1.3.1. Tracing garbage collectors

A tracing garbage collector operates on the principle that an object that cannot be reached from any other reachable object in the system is garbage. In other words, an object that cannot be part of any future computation can be safely disposed of. It determines which objects fulfil this criterion by building a graph of object references starting from a set of well-known "*roots*". These roots usually include local method/function variables in stack frames that are currently live. They also include static fields and global variables.

These references are traced recursively, eventually obtaining the transitive closure of all objects reachable from the roots. Any object that is *not* found in this graph is considered garbage and is cleaned up by the system.

Cleaning up means, in simple implementations, merely to mark the memory previously occupied by the object as vacant and available for future memory allocations. Such a simple scheme has the same problem a traditional memory allocation function like `malloc()` has: the memory heap used for the allocations will get fragmented, and it requires quite a lot of bookkeeping merely to keep track of the occupied and vacant areas of the heap. New allocations will have to search through a set of lookup tables in order to find a section of memory large enough to accommodate the request. In some cases, there may not be a single segment in the heap large enough to fit the new object even if the total available amount of memory far exceeds the demands. This is called *fragmentation*.

2.1.3.1.1. Compacting garbage collectors

For this reason, garbage collectors in actual use also *compact* the heap when performing a garbage collection. Heap compaction in this context means that all *live* objects are copied to and placed together in a single continuous segment of memory. Dead objects are not copied. This makes subsequent memory allocations a very simple operation. The allocator can just maintain a pointer to the end of this segment, and whenever a request for memory comes in, it can satisfy this request by just returning the value of this pointer and incrementing it. A `new` statement then merely becomes a pointer increment. This is orders of magnitude faster than a traditional `malloc()` implementation.

Such a scheme also has the added benefit of increased locality (although this is dependent on the exact details of the compaction algorithm). Related objects

will tend to be placed close to each other in memory, something that makes it far more likely that the processor can maintain them in one of its caches.

2.1.3.1.2. Generational garbage collectors

Of course, this scheme requires that a lot of memory is copied around. Additionally, pointers to the objects in the old locations need to be backpatched to point to the new locations. These can be time-consuming operations.

A generational garbage collector exploits the fact that most objects have a relatively short life. To do this, it divides the heap into two or more *generations*. Objects are initially allocated in the heap belonging to the first generation. If an object survives a garbage collection, it is copied or *promoted* into the next higher generation. When working in this manner, the collector can restrict garbage collection runs to a single generation, which can be collected very quickly. Full garbage collections, spanning all generations, only need to be performed every once in a while.

Conservative versus exact garbage collectors

One other categorization of garbage collectors goes to the “exactness” of its ability to detect garbage. A *conservative garbage collector* assumes that any bit pattern inside an object that, if interpreted as a pointer, points to a valid object, is in fact such a pointer. A conservative collector usually has no knowledge of the exact binary layout of an object, and is therefore the only choice in garbage collectors for languages designed without consideration for one, such as C and C++. Conservative collectors might in some cases yield false positives, causing legitimate garbage to remain uncollected.

In contrast, an *exact garbage collector* has intimate knowledge of the binary layout of an object, and can accurately determine which fields are pointers. An exact collector yields no false positives and will collect all garbage. Such a collector usually requires explicit support for it in the language compiler or the runtime engine.

2.1.3.2. Reference counting

Another form of garbage collection, although not commonly referred to as such, is *reference counting*. In this mechanism, each object maintains a count of how many other objects are referring to it. For every time another object obtains a reference to this object, the other object is responsible for

incrementing this count. Likewise, each time a reference to a reference counted object goes out of scope or the object is no longer needed, this count needs to be decremented by one.

Obviously, this mechanism breaks down in the case where one object references itself, either directly or indirectly. This is called a *cycle*. Traditional reference counting mechanisms are unable to deal with this case. Languages and environments depending on reference counting either supplement the mechanism with a tracing garbage collector or mandate that cycles are not permitted.

Reference counting can be either explicit or implicit. In an environment that uses implicit reference counting, reference counts are incremented and decremented automatically as a side effect of variable assignments. Explicit reference counting requires the programmer to explicitly call functions or methods that adjust the reference count.

The choice of implicit or reference counting is not necessarily an either/or, even within a single language/technology. Some languages accessing COM objects (which are reference counted) use implicit reference counting, while others again use explicit. Languages like Visual Basic 6 are examples of the former, while C/C++ are the most common examples of the latter.

2.2. Advantages and disadvantages of the two mo

Obviously, the two schemes of automatic memory management both have their benefits and disadvantages (otherwise, there wouldn't really be a need for two disparate models in the first place!).

Advantages of tracing garbage collectors include:

Programming simplicity
: Programming in an environment that provides a tracing garbage collector is very simple; usually you can just allocate objects and forget about them. The garbage collector will deal with unreachable objects.

Locality of reference
: The allocation strategies of most implementations of tracing garbage collectors (allocating a new object is often just a matter of incrementing a pointer) will usually lead to related objects being located close to one another in memory. This leads to a greater chance of objects being in the CPU's cache when accessing them from methods of other objects, as well

as reducing the program's working set and reducing the number of page faults associated with the program.

These benefits are highly dependent on factors such as the exact allocation and compaction mechanisms chosen, though.

Among the disadvantages, we find:

No deterministic destruction
: It is usually not possible to determine *when* an object under the control of a garbage collector is destroyed. In quite a few cases, such objects hold onto external resources such as files, operating system synchronization primitives, window handles etc... Some of these resources are in short supply, and should be disposed of as soon as possible.

  Programmers cannot then rely on the garbage collector to release these resources in a timely. They are thus forced to provide an explicit `Close()` or `Dispose()` method on types with non-trivial finalization requirements.

Unpredictable runtime behavior
: A tracing garbage collector is not entirely predictable; it might decide to stop and perform a collection at any time. Furthermore, a full collection might take a relatively long time to perform, causing the program to pause while it is running. Although faster hardware and better garbage collection algorithms have gone a long way in mitigating this factor in later years, there is still a widespread perception (justified or not) of this being a major issue.

The benefits of reference counting are the same as the drawbacks of tracing collectors:

Deterministic destruction
: Given the source code of a program using a reference counted memory management solution, one can determine the exact sequence point in execution where an object is destroyed. This ensures the timely destruction of external resources held onto by these objects.

Predictable runtime behavior
: A reference counted mechanism has predictable execution time; there are no long nondeterministic pauses. An object is removed at the exact moment it becomes garbage.

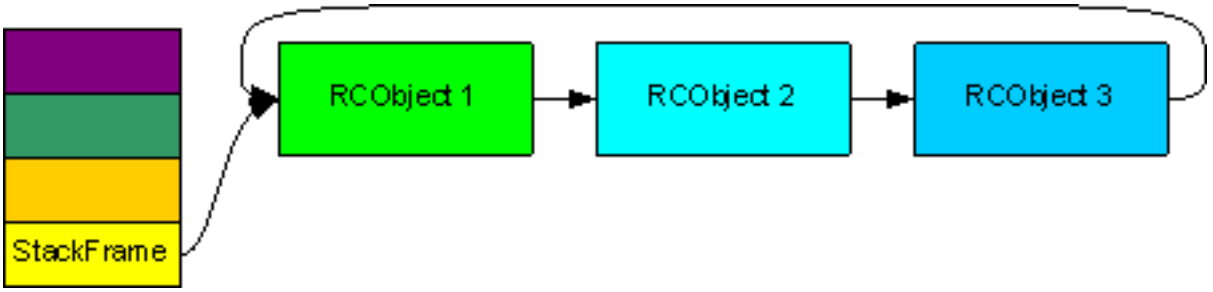
Of course, nothing is perfect:

Reference counts take up space
: Every reference counted object needs to have a field containing the reference count. In most cases this implies an overhead of 4 bytes per object. For environments in which a large number of small objects is the norm, this can amount to quite a bit of overhead.

Reference counting takes time
: For every time a reference counted object is assigned to another reference variable or passed to a function, the reference count needs to be updated. The time spent doing this kind of housekeeping is likely to be greater than the amortized time spent doing collections in a tracing collector.

Unable to deal with cycles
: A simple reference counting mechanism cannot deal with the case where an object direcly or indirectly contains a reference to itself:

Figure 2.1. A reference count cycle



For this reason, environments using reference counting either prohibit reference cycles altogether or supplement reference counting with a tracing garbage collector.

Chapter 3. The CLR and garbage collection

3.1. The CLR

The *Common Language Runtime* is an effort by Microsoft to create an execution engine and a class library and framework that allows a wide variety of languages to interoperate in a virtually seamless manner. The CLR is generally considered to be the successor to Microsoft's *Component Object Model* (COM) technology. [1]

One of the defining features of the CLR is the support for multiple languages. Components written in one language can be consumed directly by components written in another language as long as they both target the *Common Language Subset(CLS)*. The CLS represents a subset of the features of the CLR. In .NET parlance, a *CLS consumer* is a language that can utilize all the functionality in the CLS. Similarly, a *CLS producer* is a language that can produce code consumable by .NET consumers. Most .NET languages are both CLS consumers and producers.

The CLR is standardized by the *European Computer Manufacturers Association* (ECMA) and the *International Organization for Standardization* (ISO). As an open standard, it can be implemented by anyone under *Reasonable And Non Discriminatory* (RAND) terms. At the time of writing, implementations of the CLR include Microsoft .NET, Mono[10], Portable .NET[11] and the Shared Source CLI[12].

3.2. Garbage collection in the CLR

The ECMA standard that defines the Common Language Runtime is rather silent on the subject of garbage collection. Volume 1, Architecture[8] only mentions the term “garbage collection” once, and even then very briefly. The detailed specification[9] of the *Base Class Library* does however contain the following note:

[1]Not surprisingly, given this position as a successor to COM, the Microsoft implementation of the CLR has comprehensive support for interoperating with COM, including support for both consuming and producing COM components.

> The garbage collector is responsible for tracking and reclaiming objects allocated in managed memory. Periodically, the garbage collector performs a garbage collection to reclaim memory allocated to objects for which there are no valid references. Garbage collections happen automatically when a request for memory cannot be satisfied using available free memory. A garbage collection consists of the following steps: During a collection, the garbage collector will not free an object if it finds one or more references to the object in managed code. However, the garbage collector does not recognize references to an object from unmanaged code, and may free objects that are being used exclusively in unmanaged code unless explicitly prevented from doing so.

Even so, the lack of a detailed specification of the memory management model in the main specification implies a considerable flexibility in how a CLR-conforming implementation manages memory. However, all the existing implementations of the CLR use a tracing garbage collector, as described in the previous chapter.

3.3. Finalization

A large number of .NET classes hold onto resources beyond the actual memory they occupy in the managed heap. For example, objects of the .NET `System.IO.FileStream` type own an operating system file handle. Likewise, `System.Windows.Forms.Control` objects (with descendants) hold onto a window handle (a `HWND`). `System.Drawing.Brush` objects keep a reference to a Windows brush object (a `HBRUSH`).

These resources are not available in infinite supply. The operating system has a limited number of them, and it is important that code that acquires such a resource also releases it in a timely fashion.

3.3.1. `Finalize()`

To ensure these kinds of resources can be released, the CLR `Object` type declares a virtual method called `Finalize()`. Types can override this method and use it to release whatever unmanaged resources they are holding on to. The garbage collector will call this method before the object is finally

destroyed and its memory released back to the managed heap. A typical Finalize() implementation might look like this:

Example 3.1. A class with a finalizer

```
public class Brush
{
   public Brush()
   {
      this.handle = Win32.CreateBrush();
   }

   protected override void Finalize()
   {
      Win32.CloseHandle( this.handle );
   }

   // implementation omitted

   private IntPtr handle;
}
```

In C# and C++, two of the languages that run on the CLR, the `Finalize()` method is declared using the same syntax as C++ destructors. The finalizer from the above example would then look like this (in C#):

Example 3.2. Finalizer destructor syntax

```
~Brush()
{
  Win32.CloseHandle( this.handle );
}
```

3.3.2. The finalizer queue

When a GC heap is collected, objects that have a finalizer are *not* collected right away. Instead they are put in a special data structure called the *finalizer queue*.

A background thread, the *finalizer thread*, starts up and visits all the objects in the queue, invoking their `Finalize` methods. If the act of finalizing them doesn't revive them (make them accessible again from some other object), they are again marked for garbage collection and picked up by the collector on its next run. If they are resuscitated, they will be placed back on the finalizer queue, awaiting the next garbage collection in which they are collected.

The behavior just described causes objects with a finalizer to have a longer life than objects without one. If you have a large number of finalizable objects in your system, performance will obviously suffer, since these objects will have a much longer lifespan; i.e. they will always survive at least one garbage collection. The external resources these objects hold on to will also be held for a longer time than necessary, since the finalizer won't necessarily be run instantly after the object has been collected.

One other thing about finalization is that there is no 100% guarantee that it will happen. In some cases, the program might exit without having run all of its finalizers. When a program is exiting, the runtime gives a certain amount of time for remaining finalizers to run. When the time runs out, the finalizer thread is killed off and the runtime (and the program with it) is shut down anyway). There is only a single finalizer thread, so any one of the finalizers could prevent the remaining ones from running by taking too long to execute.

3.3.3. The IDisposable pattern

As the previous section describes, relying on the finalizer to take care of your external resources has some drawbacks: it is non-deterministic in that you never know when (or even if) it will run, and it causes objects to linger and hold on to memory for a longer time than necessary. To support a more deterministic way of releasing said resources, the designers of the .NET framework provided the *IDisposable pattern.*

`System.IDisposable` is an interface defining a single method, `Dispose()`:

Example 3.3. The `IDisposable` interface

```
namespace System
{
  public interface IDisposable
  {
    void Dispose();
  }
}
```

Types that own unmanaged resources will implement this interface. The user of such a type can then call `Dispose()` explicitly when he is done with the object, causing the resource to be freed instantaneously. For types where another name for this method would be more semantically correct, one can provide another method with the appropriate name and have it call `Dispose()` (or the other way around). For example, the `FileStream` class implements `IDisposable`, but provides a `Close()` method intended for the consumer to use. Quite a few of the types in the Base Class Libraries implement `IDisposable`, but any application programmer designing types with non-trivial finalization requirements are also encouraged to make use of this interface.

A strategy of implementing *only* `IDisposable` can be a bit dangerous, though. You have no guarantees that the user will ever bother calling `Dispose()`, even if you write it in bold red type in the documentation (programmers can't read, and even if they could, they wouldn't want to). Most such types will therefore also implement a finalizer, in case the `Dispose()` method is never called.

Obviously, having *both* `Dispose()` and the finalizer being called would be pretty wasteful. Yes, you would dispose of the unmanaged resource, but the problem of having finalizable objects linger for a much longer time than necessary would remain. Fortunately, you can tell the garbage collector *not* to finalize a specific object if it's already been disposed. The most common implementation of `IDisposable` will call the `GC.SuppressFinalize()` method once it has disposed of the unmanaged resource:

Example 3.4. Suppressing finalization

```
public class Foo : IDisposable
{
    public Foo()
    {
        this.resource = CreateResource();
    }

    ~Foo()
    {
        DoDispose();
    }

    public void Dispose()
    {
       DoDispose();
       GC.SuppressFinalize( this );
    }

    protected virtual void DoDispose()
    {
        FreeResource( this.resource );
    }

    // ...
    // implementation omitted
    // ...


    [DllImport( "os.dll")]
    private extern IntPtr CreateResource();

    [DllImport( "os.dll")]
    private extern void FreeResource( IntPtr resource );

   private IntPtr resource;

}
```

When `IDisposable.Dispose()` is called in the `Foo` class in the above example, it ensures that the object is removed from the finalizer queue. This avoids the problems associated with finalizers, in which the presence of a

finalizer on a type causes the object to linger for a much longer time than otherwise necessary.

`DllImport`

The `DllImport` attribute applied on the `CreateResource()` and `FreeResource()` methods in Example 3.4, “Suppressing finalization” indicates to the runtime that these are not implemented in managed code. Instead they are routines provided in dynamic libraries (`os.dll` in this case). This is yet another example of how versatile the CLR metadata mechanism is.

A typical use of the `Foo` class would be like this:

Example 3.5. Using an `IDisposable` object

```
Foo foo;
try
{
    foo = new Foo();

    // use foo here
}
finally
{
    foo.Dispose();
}
```

Putting the `Dispose()` call in a finally ensures that the object will be cleaned up even if the code in the try block throws an exception.

The `using` construct in C# makes the IDisposable pattern even easier to use. The following C# code is equivalent to the above example:

Example 3.6. The `using` statement

```
using( Foo foo = new Foo() )
{
    // use foo here
}
```

Chapter 4. The Shared Source CLI (Rotor)

4.1. ECMA 335

In August 2000, while developing their new architecture known at that time as the *Next Generation Windows Services*, Microsoft decided to submit the specification of the runtime engine and the new C# language (which at that time was known by its internal codename, “Cool”[1]) to the European Computer Manufacturers' Association for standardization. These two elements of .NET were ratified by ECMA as ECMA standards 334 [6] and 335[7] in December 2001; ECMA-334 describing the C# language and ECMA-335 describing the execution environment.

From the preface to ECMA-335:

> This Standard ECMA-335 defines the Common Language Infrastructure (CLI) in which applications written in multiple high level languages may be executed in different system environments without the need to rewrite the application to take into consideration the unique characteristics of those environments.

The CLR was also ratified by ISO as a standard, after having gone through the organization's fast track standardization process.

4.2. Rotor

The Shared Source Common Language Infrastructure (also affectionately known as *Rotor*, the internal code name for the project at Microsoft) is an implementation of the ECMA-334 and ECMA-335 standards released by Microsoft in 2002 in source form. Rotor contains source code for a large set of libraries, tools and compilers, including the JIT compiler and the virtual machine.

The source code for Rotor was derived from Microsoft's commercial implementation of .NET, but forked and modified to build and run on the

[1] Thus the t-shirts with the text “C# is Cool”

Mac OS X and FreeBSD operating systems in addition to Microsoft Windows. It is also easily portable to other operating systems and CPU architectures (it currently supports IA/32 and PowerPC CPUs). It comes with a full implementation of the Base Class Library (BCL), but libraries outside the scope of the standard such as the ASP.NET web framework and the Windows Forms GUI toolkit are not included.

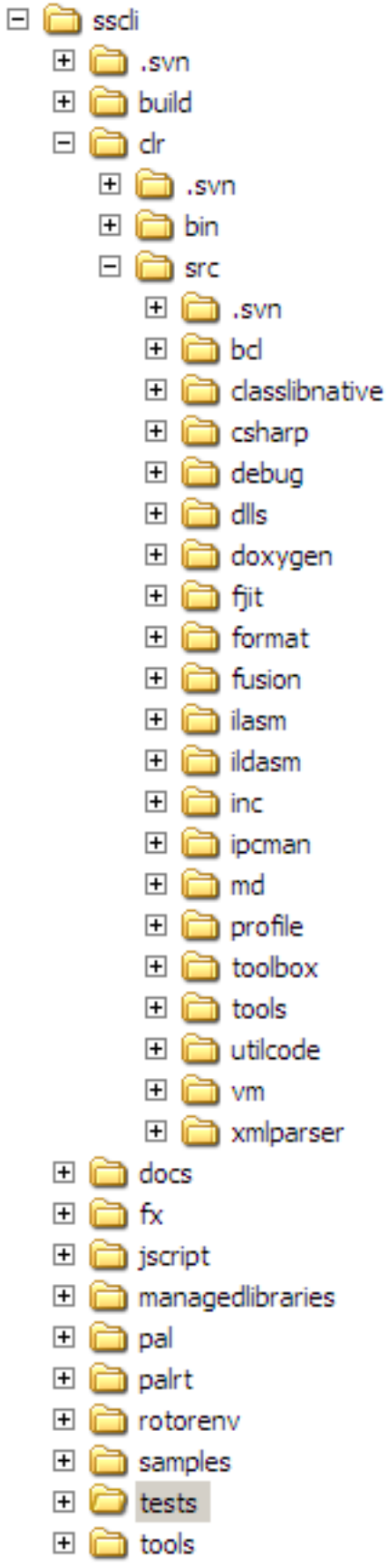
Rotor is provided under a so called “Shared Source” license. This license prohibits any form of commercial use of the libraries and tools provided. The stated purpose of Rotor is to serve as an educational tool, allowing academics to use it for research and teaching. It is also of great interest to advanced .NET programmers with an interest in what's going on under the hood. The fact that it is so close to Microsoft's commercial implementation makes it a perfect vehicle for gaining a deeper understanding of how .NET really works. For academic research it is also helpful to know that any modifications made to the SSCLI will most likely work on the commercial runtime.

Rotor also contains source code for a C# compiler, implementing the ECMA-334[6] standard. In my implementation of reference counting, I have avoided strategies that required support from the language compilers, and the implementation of the compiler will thus not be described further. It would be natural to use the Rotor C# compiler as a starting point if future work on the ideas presented here include introducing direct language support (syntactic sugar) for reference counting.

4.3. The Rotor source code

The SSCLI is a rather large piece of software (or perhaps a large collection of smaller pieces of software). The unpacked tarball takes up 115 megabytes of disk space, consisting of 10808 files spread over a large number of directories. The execution engine is written in C++, with most of the class library written in C#. There are also some assembler code files.

As a reference for later, I will here describe the most important directories in the distribution.



CLI source tree.

Most of the important code relating to the actual execution engine resides under the `clr` directory. Other interesting top level directories include:

`/build`
: The `build` directory contains the output from the build process. There can be several subdirectories beneath `build`, one for each build configuration (one of checked, fastchecked or free. Each of these subdirectories (if created as the result of a build) will contain a full implementation of the CLR, with varying support for debugging and logging.

`/pal`
: This contains the *Platform Adaptation Layer*, the SSCLI's primary way of ensuring portability across operating systems and CPU architectures. The PAL is a subset of the Win32 API, implemented both for UNIX and for Windows[2].

`/palrt`
: The `palrt` directory contains shared PAL code that has no dependencies on the underlying operating system.

`/tests`
: The `tests` directory contains two large test suites, one for the CLR and one for the PAL.

The `clr` directory contains the source for the execution engine, the JIT and the C# compiler:

`/clr/src/bcl`
: C# source for the Base Class Libraries (BCL).

`/clr/src/csharp`
: The C# compiler, written in C++.

`/clr/src/fjit`
: The Just-In-Time compiler. This is a separate component, communicating with the execution engine through a set of well-defined interfaces.

`/clr/src/inc`
: Include files that are shared among the various components.

[2]Obviously, the Win32 PAL is a very thin layer over the actual Win32 API

`/clr/src/vm`
: The execution engine, including the class loader, the garbage collector and the main entry point.

4.4. The SSCLI type system

The CLR's type system reflects the fact that it is targeted primarily (but not exclusively) towards object-oriented languages. It supports two main categories of types:

objects
: Object types are always allocated on the heap. Any representation of an object as a local variable merely consists of a pointer to the object. Objects thus have reference semantics. In addition to the instance data, an object also contains a pointer to a *method table* (used for dispatching virtual methods) and a *sync block*. The latter is used for thread synchronization, for example with the C# keyword `lock`. These two pointers give each allocated heap object an overhead of 8 bytes (on a 32 bit platform).

value types
: Value types are essentially nothing more than a sequence of bytes, and are represented as such on a program's stack. They are allocated either on the runtime stack or inline in objects that contain them. They support methods, but not inheritance, although they can implement interfaces. Since the inheritance hierarchy is never more than one level deep, all method calls can be resolved at compile time and there is no need for a method table. Additionally, they do not have a sync block, and cannot be used for thread synchronization (Technically, a compiler will allow it, but you will merely end up boxing the value type into a freshly allocated anonymous object, which is completely useless, since no other thread will be able to access the object). All primitive types, such as integers and floats, are represented in the CLR type system as value types. C#, for example, allows you to write this:

```
string s = 42.ToString();
```

Since only objects are allocated dynamically, they are the only types that are garbage collected, and thus of interest when implementing reference counting. Value types live either on the runtime stack or as part of other objects, and their lifetimes are therefore already deterministic.

4.4.1. Runtime representation of objects

At runtime, objects are represented by the `Object` class, in `clr/src/vm/object.h`. This class has a single field; a pointer to a `MethodTable` object:

Example 4.1. The `Object` class

```
class Object
{
  protected:
    MethodTable    *m_pMethTab;

    // ...
};
```

Obviously, that's not the whole story. The class `Object` has a large number of methods, and there is a “hidden” field associated with each object, located at a *negative* offset from the object itself:

Figure 4.2. Runtime layout of object types

Every time an object is allocated, the `ObjHeader` is allocated with it. If you have a pointer to an `Object`, you can access the `ObjHeader` through some pointer manipulation:

Example 4.2. Accessing an `Object`'s `ObjHeader`

```
ObjHeader   *GetHeader()
{
    return ((ObjHeader *) this) - 1;
}
```

The `ObjHeader` is used for, among other things, thread synchronization. The mechanism used here is worth noting, since I have chosen a similar approach to implement reference counting.

The `MethodTable` instance pointed to by the sole member field of `Object` is shared among all instances of a class. It determines the type identity of an object, i.e., if

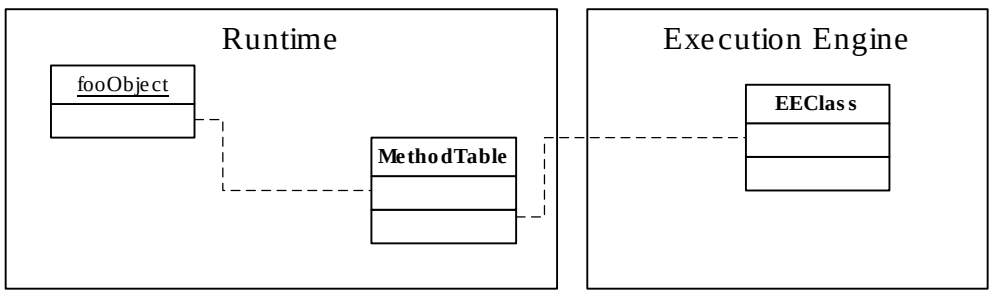
```
obj1->p_methTab == obj2->p_methTab
```

holds, `obj1` and `obj2` are of the same type.

Further, it contains data and methods to acquire information about the fields and methods associated with the type. At runtime, the `MethodTable` is used to dispatch virtual method calls (thus its name).

`MethodTable` also contains a pointer to an `EEClass` instance. Like `MethodTable` objects, `EEClass` instances also contain information about types. There is a 1-1 relationship between `EEClass` instances and `MethodTable` instances; there is one of each for every type loaded. However, while `MethodTable` contains data and operations that are designed to be accessed quickly at runtime, `EEClass` contains the data that are typically only needed by compilers and other tools that need to be able to introspect into the type system. The separation between the two allow better cache coherency. Not touching the `EEClass` instance at runtime keeps it out of the process' working set.

Figure 4.3. The relationship between an object and its `MethodTable` and `EEClass`



Chapter 5. The implementation

5.1. Introducing reference counting into the CLR

The Rotor code base is largely built around the assumption that the implementation will be garbage collected. I have decided to create a dual type system in which both reference counted types and types tracked by the current garbage collector exist side by side. This allows existing code, which may have made some assumptions about the behavior of the memory model, to work unchanged on the reference counted implementation. Of course, this also means that existing code does not gain any of the benefits of reference counting.

5.1.1. Telling the runtime about reference counted types

If the runtime is going to support both reference counted types and types tracked by a classic garbage collector simultaneously, there needs to be a way for it to tell the difference between the two types. One possible approach might be to introduce new IL instructions to allocate and initialize reference counted types. One could also add new instructions to explicitly manage the reference counts. Having such instructions would give a compiler targeting the CLR a large amount of flexibility to perform a variety of optimizations, given its superior knowledge of object lifetimes.

I have decided against this approach because of its rather invasive nature, and because of the massive amount of changes required to the runtime. It would also require all compilers targeting the runtime to be modified to emit the new instructions, something which is clearly not feasible considering the large amount of existing compilers out there targeting the CLR. A less invasive approach would be far more desirable.

What is needed in this case is a way to tell the runtime that "hey, this type should not be tracked by the tracing garbage collector". A specific type could be labelled in this way and the runtime would treat the type as a special case, tracking it using reference counting rather than the garbage collector. This would ideally not require any changes to existing compilers, allowing even these to take advantage of the new reference counting mechanism.

Fortunately, the CLI already provides a mechanism that allows us to annotate types (and methods, fields, parameters etc) with arbitrary metadata: custom

attributes. Since these are part of the standard, all existing CLI languages already know how to produce and consume custom attributes.

Metadata in the CLR

The CLR's type system, the *Common Type System*, is centered around the concept of *metadata*. From Partition I of ECMA-335 [8]:

> New types – value types and reference types – are introduced into the CTS via type declarations expressed in metadata. In addition, metadata is a structured way to represent all information that the CLI uses to locate and load classes, lay out instances in memory, resolve method invocations, translate CIL to native code, enforce security, and set up runtime context boundaries. Every CLI PE/COFF module (see Partition II) carries a compact metadata binary that is emitted into the module by the CLI-enabled development tool or compiler. Each CLI-enabled language will expose a language-appropriate syntax for declaring types and members and for annotating them with attributes that express which services they require of the infrastructure. Type imports are also handled in a language-appropriate way, and it is the development tool or compiler that consumes the metadata to expose the types that the developer sees.

In other words, metadata is the mechanism in which the runtime itself and other tools obtain information about the types and their attributes and operations. It doesn't stop there, though: it also allows developers to create their own *custom* metadata, in the form of *attributes*.

An attribute is a type deriving from the `System.Attribute` class. It can be applied to classes, structs, methods, fields and parameters, as well as to assemblies. They can contain explicit information in the form of properties on the type, or they can implicitly convey information merely by being present on some element. Which they are applied to. For example, the `Serializable` attribute indicates to the runtime that a type can be serialized:

Example 5.2. The `Serializable` attribute

```
[Serializable]
public class Foo
{
  // ...
}
```

In the C# language, attributes are expressed using square brackets in front of the element they are applied to. Example 5.2 shows the `Serializable` attribute in use. It is declared like this:

Example 5.3. Using the `Serializable` attribute

```
[AttributeUsage(AttributeTargets.Class |
 AttributeTargets.Struct |
   AttributeTargets.Enum | AttributeTargets.Delegate)]
public sealed class SerializableAttribute : Attribute
{
  // ...
}
```

Another attribute with special meaning to the runtime is `System.Runtime.InteropServices.FieldOffsetAttribute`. It is used when the fields of a class or struct are to be placed at specific offsets (for example for interoperating with unmanaged code):

Example 5.4. The `FieldOffset` attribute

```
[StructLayout(LayoutKind.Explicit)]
public class SYSTEM_INFO
{
  [FieldOffset(0)] public ulong OemId;
  [FieldOffset(4)] public ulong PageSize;
  [FieldOffset(16)] public ulong ActiveProcessorMask;
  [FieldOffset(20)] public ulong NumberOfProcessors;
  [FieldOffset(24)] public ulong ProcessorType;
  // ...
}
```

It is interesting to note that declaring custom attributes also requires the use of attributes to constrain which kinds of types they can be applied to. We see in Example 5.2, “The `Serializable` attribute” that `Serializable` can be applied to classes, structs, enums and delegates.

I have therefore decided to use such a custom attribute to signify that a given type should be reference counted rather than garbage collected. Since this attribute is rather central to the runtime, I have placed it in the `System` namespace, as part of the core library which gets implicitly referenced by all .NET assemblies: corlib. The implementation can be found in `clr/src/bcl/system/referencecountedattribute.cs` and looks like this:

Example 5.4. The implementation of `ReferenceCountedAttribute`

```
namespace System
{
    /// <summary>
    /// An attribute that indicates to the runtime that the
 applied-to type
    /// should be handled by reference counting, not M&S
 garbage collection.
    /// </summary>
    [AttributeUsage(AttributeTargets.Class, Inherited=true)]
    public class ReferenceCountedAttribute : Attribute
    {
        public ReferenceCountedAttribute()
        {
        }
    }
}
```

This attribute can then be applied onto a class like this [1]:

Example 5.5. Using the `ReferenceCounted` attribute

```
[ReferenceCounted]
public class Foo
{
   // ...
}
```

The attribute does not carry any information beyond its presence on a class. It is restricted to classes (reference types) only; as I have mentioned before,

[1] Note that C# allows you to omit the `Attribute` suffix to the attribute type when applying it. All attribute types are required to have this suffix in the name, though.

value types are allocated either on the stack or as part of reference types, and thus already have predictable life times.

As can be inferred from the `AttributeUsage` attribute applied on `ReferenceCountedAttribute`, I have chosen to let the attribute be inherited to subclasses. As an example, let's say we have the following classes:

Example 5.6. `ReferenceCounted` and inheritance

```
[ReferenceCounted]
public class Base
{
   // ...
}

public class Derived : Base
{
   // ...
}
```

In my implementation, instances of both `Derived` and `Base` will be treated by the execution engine as reference counted types.

5.2. Modifying the class loading process

Now, with that attribute in place, the information about whether a given class should be reference counted gets embedded into the assembly as part of the metadata associated with the type. A tool can reflect on the type and use the presence of the `ReferenceCountedAttribute` attribute on a class to infer that it should be reference counted. .NET code can also query for this attribute on a class at runtime.

Obviously, having tools be able to extract this information is nice and all, but ultimately worthless unless the execution engine itself can detect the attribute and act on it.

5.2.1. The SSCLI class loader

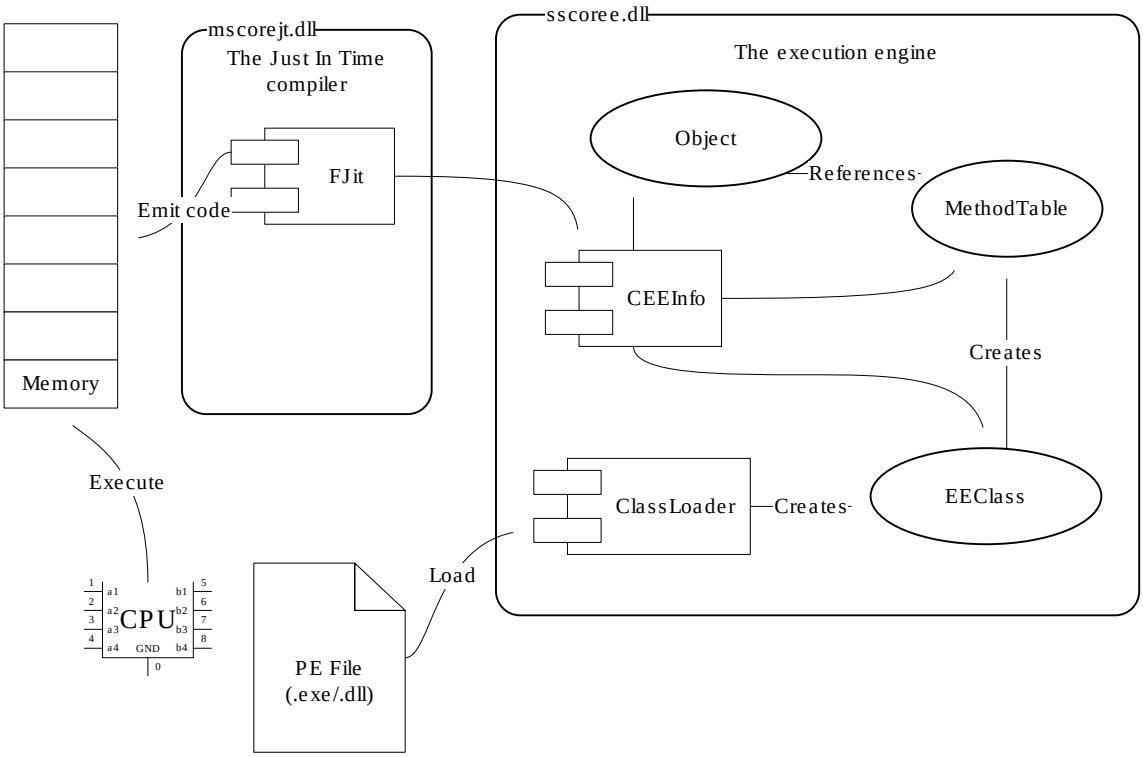
Loading a new type into the runtime is a quite involved process, and involves building the `EEClass` and `MethodTable` instances associated with the type as described in the previous chapter.

When a managed executable is run by the clix executable launcher, the `sscoree` dynamic library is loaded and the `_CorExeMain2` function is called. This function initializes the runtime and eventually ends up calling the `ClassLoader::ExecuteMainMethod` method. `ExecuteMainMethod` figures out which type contains the entry point for the assembly, loads it and then executes it.

By the time it has gotten that far, though, the `ClassLoader` has already loaded the classes that are central to the execution engine, such as `Object`, `Array` and `String`. The main work horse when it comes to loading classes is the `ClassLoader::LoadTypeHandleFromToken` method. It takes in a *metadata token* for the requested type (extracted from the assembly metadata) and builds an `EEClass` instance (and by extension, a `MethodTable` instance). The `EEClass`/`MethodTable` pair is then used by the runtime to create instances of the type.

The class loading process, as well as the runtime interaction between the JIT compiler and the rest of the execution engine, is shown in Figure 5.1, “The type loading mechanism”.

Figure 5.1. The type loading mechanism



5.2.2. Detecting reference counted types

In order to determine whether a type is reference counted, the runtime needs to check whether the `ReferenceCounted` attribute is present on that type.

5.2.2.1. The SSCLI metadata APIs

The SSCLI provides an API that allows you to manipulate, emit and inspect the metadata for an assembly or a module. This API is public and intended for use by writers of compilers and other tools, but is also used extensively internally in the execution engine.

The metadata API is COM-based and the relevant interfaces can be found in `/clr/src/inc/cor.h`. The interface used for metadata inspection is `IMetaDataImport`. Internally in the runtime, the name `IMDInternalImport` is used. As far as I have been able to tell, this is the same interface, and I have been able to use the documentation for `IMetaDataImport` in my work with `IMDInternalImport`.

I implemented a global function `IsReferenceCounted()` that takes a metadata token for the type and determines whether the `ReferenceCountedAttribute` attribute is applied to it. The implementation resides in `/clr/src/vm/nstruct.cpp` and looks like this:

Example 5.7. The `IsReferenceCounted()` function

```
HRESULT IsReferenceCounted(IMDInternalImport
 *pInternalImport, mdTypeDef cl )
{
    const void *pData;
    ULONG cbBlob;

    // we only need to know if there is such an attribute on
 the type
    HRESULT hr = pInternalImport-
>GetCustomAttributeByName( cl,
        "System.ReferenceCountedAttribute", &pData, &cbBlob);

    return hr;
}
```

The function returns `S_OK` if the attribute exists and `S_FALSE` if it doesn't. Any other return value is treated as an error.

`EEClass` contains a series of bitflags describing various attributes of the type in question. I added a new flag indicating a reference counted class (in `/clr/src/vm/class.cpp`:

Example 5.8. Flags in `EEClass`

```
VMFLAG_ISSINGLEDELEGATE                 = 0x10000000,
VMFLAG_ISMULTIDELEGATE                  = 0x20000000,
VMFLAG_PREFER_ALIGN8                    = 0x40000000, // Would
 like to have 8-byte alignment
VMFLAG_REFERENCECOUNTED                 = 0x80000000,

// ...

inline BOOL EEClass::IsReferenceCounted()
{
    return (m_VMFlags & VMFLAG_REFERENCECOUNTED);
}

inline void EEClass::SetReferenceCounted()
{
    m_VMFlags |= VMFLAG_REFERENCECOUNTED;
}
```

I also added similar flags and accessors on `MethodTable`. Obviously, this is a duplication of information since it could also be accessed through the `MethodTable`'s pointer to the `corresponding` EEClass, but there is ample precedent in the existing code for doing it this way. I can only assume the idea is to avoid accessing the `EEClass` at runtime if possible, for performance reasons.

Now, the class loader merely has to call `IsReferenceCounted` and set the flag accordingly in `ClassLoader::LoadTypeHandleFromToken()` (`/clr/src/vm/clsload.cpp`):

Example 5.9. Determining whether a type is reference counted

```
// is the type reference counted?
hr = IsReferenceCounted(pInternalImport, cl);

BOOL fIsReferenceCounted = (hr == S_OK);
if(FAILED(hr) && hr != S_FALSE)
{
    m_pAssembly->PostTypeLoadException(pInternalImport, cl,
 IDS_CLASSLOAD_BADFORMAT, pThrowable);
    return hr;
}

// ...

if (fIsReferenceCounted)
    pClass->SetReferenceCounted();
```

5.3. The reference counted heap

The reference counted objects cannot share the same heap as the garbage collected objects use. The objects on the garbage collected heap are routinely moved and/or discarded as part of a garbage collection sweep, a mechanism which doesn't take into account the needs of the reference counted objects.

The large object heap

Not all objects are placed on the garbage collected heap even in the existing SSCLI implementation. Large objects, in the SSCLI defined as objects bigger than 85,000 bytes, are allocated and managed on a separate heap. This heap, the large object heap, is further subdivided into two: a section for objects that contain internal pointers to other objects and a section for those without such pointers. This is due to the large cost of scanning such large objects for pointers. The garbage collector queries the `MethodTable` instance for an object to determine whether it contains internal pointers.

The cost of constantly moving objects as big as those on the large object heap would be prohibitive, so the garbage collector does not use compaction for these objects. Instead they are managed by a more traditional malloc-style heap.

It is then obvious that another heap implementation is necessary to fill the needs of the reference counting mechanism. I have decided here to reuse the malloc-style allocator used for the large object heap. While it may not be optimized for the needs of allocating a lot of small objects, it is well encapsulated from the rest of the implementation and can be swapped out for a more efficient implementation at a later time. Investigating such implementations might serve as a good topic for further work on this research.

The implementation of the heap used for the large object heapis located in `/clr/src/vm/gmheap.cpp`. I created a class `ReferenceCountedHeap` in `clr/src/vm/refcount.[cpp|h]`, which encapsulates this implementation and provides the operations necessary to support allocation and deallocation of reference counted objects:

Example 5.10. The `ReferenceCountedHeap` class

```
class ReferenceCountedHeap
{
public:
    ReferenceCountedHeap()
    {;}

    HRESULT Initialize();
    Object* Alloc(DWORD size);
    void Free(void* ptr);
private:
    gmallocHeap* m_Heap;
};
```

`gmallocHeap` here is the malloc implementation.

The runtime maintains a global instance of `ReferenceCountedHeap`, accessed through the pointer `g_pRCHeap` which is declared in `clr/src/vm/vars.hpp`. This matches the way the runtime has a global variable for the garbage collected heap, `g_pGCHeap`. An accessor function in `refcount.cpp` is used to allocate objects from the reference counted heap:

Example 5.11. `RCAllocateObject`

```
OBJECTREF RCAllocateObject( MethodTable* pMT )
{
    THROWSCOMPLUSEXCEPTION();

    // allocate memory for it
    Object* orObject = (Object*)g_pRCHeap->Alloc(pMT-
>GetBaseSize());

    // verify that the memory is zeroed out.
    _ASSERTE( ! orObject->HasSyncBlockIndex() );

    // make sure it has a method table
    orObject->SetMethodTable(pMT);

    RCLogAlloc(pMT, orObject);

    return( ObjectToOBJECTREF(orObject) );
}
```

This global instance also needs to be initialized. I have created a function `InitializeReferenceCountedHeap()` which is called from `InitializeEE()`, the main initialization function of the execution engine. The implementation merely creates an instance of `ReferenceCountedHeap` and calls its `Init() method.`[2]

5.4. Allocating reference counted objects

The JIT compiler is a separate component that communicates with the execution engine proper through the `ICorJitInfo` interface (`/clr/src/inc/corjit.h`). This interface is implemented by the `CEEInfo` class in `/clr/src/vm/jitinterface.cpp`. Whenever the JIT compiler needs to emit a call to allocate an object, it calls `CEEInfo::getNewHelper()`. This method investigates the `MethodTable` of the object that needs to

[2] I have, as much as possible tried to use the same coding conventions as are used by the existing code. Personally, I would rather have performed any initialization in the constructor, but I don't see it as my role to impose my personal idioms on an existing codebase. When in Redmond, do as the Redmondians.

be allocated and returns a constant corresponding to an appropriate helper function.

I created a new constant, `CORINFO_HELP_NEWREFCOUNTEDOBJECT` and injected this code into `CEEInfo::getNewHelper()`:

Example 5.12. `CEEInfo::getNewHelper()`

```
// is this a reference counted object?
if (pMT->IsReferenceCounted())
{
    return CORINFO_HELP_NEWREFCOUNTEDOBJECT;
}

// ...
```

`pMT` is here a pointer to the `MethodTable` object for the class the runtime needs to allocate an instance of. `IsReferenceCounted()` is the accessor method I created earlier.

The JIT compiler then calls `CEEInfo::getHelperFtn()` which maps the constant to a function pointer through a table. A call to this function is emitted into the generated native code. `CORINFO_HELP_NEWREFCOUNTEDOBJECT` maps to the function `JIT_NewRefCountedObject()`, which does some housekeeping and then calls the `RCAllocateObject()` function described earlier.

5.5. Keeping track of reference counts

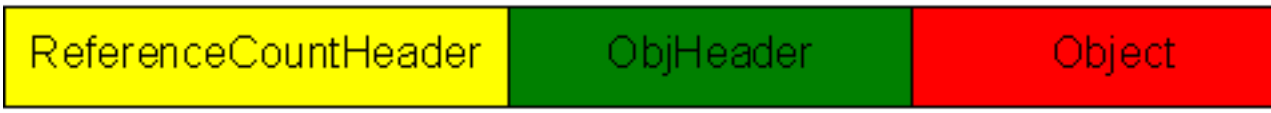
One important question remains unanswered at this point: Where do we store the reference count for an object? The first solution that comes to mind is to introduce a new field in the `Object` class. However, this would mean that *all* objects, reference counted or not, have to pay the cost of this extra field. Since the purpose of this whole exercise is to find a way for both reference counting and mark and sweep garbage collection to coexist, this is clearly unacceptable.

Another solution might be to create a subclass of `Object`, `ReferenceCountedObject`, containing this field. However, a large

number of places in the SSCLI codebase makes assumptions about the binary layout of an object, and expects the instance data to start at a given offset from the object address. Creating such a subclass with the reference count as an instance field would throw off such code. Even if it was practical to patch all locations in the code which made these assumptions, this would make a hybrid model much harder to implement.

The solution I eventually landed on was to replicate the mechanism used to allocate and place the `ObjHeader`; i.e., placing it *before* the actual `Object` instance. Obviously, the `ObjHeader` is already located in front of `Object`, so I need to locate it before that. This means that the actual runtime layout of a reference counted object on the reference counted heap ends up looking like this:

Figure 5.2. Runtime layout of reference counted types



5.5.1. `ReferenceCountHeader`

The reference count of a reference counted object is managed by an instance of the class `ReferenceCountHeader`. It has a design that resembles that of COM's `IUnknown`:

Example 5.13. The `ReferenceCountHeader` class

```
class ReferenceCountHeader
{
public:
    ReferenceCountHeader() : m_RefCount(0)
    {;}

    LONG AddRef();
    LONG Release();
    LONG ReferenceCount()
    {
        return m_RefCount;
    }

    Object* GetObject();
private:
    void Finalize();
    LONG m_RefCount;
};
```

The semantics of `AddRef()` and `Release()` should be obvious in this context, even to people without COM experience. The implementation of `GetObject()` is worth taking a look at, though. It uses some pointer arithmetic to retrieve the pointer to the actual `Object` instance associated with this object:

Example 5.14. `ReferenceCountHeader::GetObject()`

```
Object* ReferenceCountHeader::GetObject()
{
    return (Object*) ((ObjHeader*)(this + 1) + 1);
}
```

Obviously, the size of the `ReferenceCountHeader` also needs to be taken into account when allocating object instances[3]:

[3]The *`size`* parameter here already incorporates the size of the `ObjHeader` instance.

Example 5.15. `ReferenceCountedHeap::Alloc()`

```
Object* ReferenceCountedHeap::Alloc(DWORD size)
{
    // we need some space for our accounting needs too
    DWORD realSize = size + sizeof(ReferenceCountHeader);

    // get the space for it on the heap
    BYTE* newAlloc = (BYTE*)m_Heap->Alloc(realSize);
    memclr((BYTE*)newAlloc, realSize);

    // account for the space allocated for the ObjHeader and
 the
    // ReferenceCountHeader
    return (Object*) (newAlloc + sizeof(ReferenceCountHeader)
 + sizeof(ObjHeader));
}
```

5.5.2. `GC.ReferenceCount`

The `ReferenceCount()` accessor method in `ReferenceHeader` was added to serve as the underlying foundation for a new method on the `GC` class, `GC.ReferenceCount`. The `GC` class is part of the BCL and contains various methods to control the garbage collector, such as explicitly starting a garbage collection sweep, suppressing and reenabling finalization of objects and discovering which generation an object is currently part of.

The `GC.ReferenceCount()` method allows a .NET developer to query the runtime for the current reference count of a reference counted object. It is implemented as an internal call into the runtime:

Example 5.16. `GC.ReferenceCount()`

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
private static extern int nativeReferenceCount(Object o);

public static int ReferenceCount(Object o)
{
    if (o == null)
        throw new ArgumentNullException("o");
    return nativeReferenceCount(o);
}
```

The `MethodImplOptions.InternalCall` argument to the `MethodImplAttribute` attribute here indicates that the real implementation of the method can be found in native code inside the execution engine. The runtime maps the call through a table into a method on the `GCInterface` class. This method merely calls the `ReferenceCount()` method on the `ReferenceCountHeader` associated with the object, after verifying that it actually is a reference counted object (`clr/src/vm/comutilnative.cpp`):

Example 5.17. `GCInterface::ReferenceCount`

```
FCIMPL1(int, GCInterface::ReferenceCount, Object *obj)
{
    THROWSCOMPLUSEXCEPTION();

    if (!obj->GetMethodTable()->IsReferenceCounted())
        FCThrow(kInvalidOperationException);

    return obj->GetReferenceCountHeader()->ReferenceCount();
}
```

5.6. Modifying the JIT compiler

There is now a mechanism in place to allocate and free reference counted objects, as well as incrementing and decrementing their reference counts. The next step is to modify the JIT compiler in such a way that the reference count for an object is implicitly managed. Unlike COM in C++, the mechanism introduced here should not require the developer to manually call `AddRef()` or `Release()`.

The intent is to make the use of reference counted objects as transparent as possible, while still providing a very deterministic life time and finalization for such objects. Take the following example:

Example 5.18. Allocating a large number of objects in a loop

```
for( int i = 0; i < BigConstant; i++ )
{
  SomeType obj = new SomeType();
  obj.SomeOperation()
}
```

Here, a new instance of the `SomeType` class is created for each iteration of the loop. In the traditional CLR environment, the memory for the allocated objects would not be recycled and the memory use would continue to grow, possibly triggering a garbage collection. If the object also has non-trivial finalizer requirements, the memory use would be even worse, since the objects would not be cleaned up immediately in a garbage collection, instead having to wait for the finalizer thread to deal with them.

The goal of the implementation here is to ensure that in the example above, there will never be more than a single instance of `SomeType` in existence at any given iteration of the loop. If `SomeType` has a finalizer, it should be executed synchronously at the end of the loop. This is as opposed to the asynchronous behavior of the current CLR finalization mechanism.

The mechanism I have chosen is to have all reference counted objects be created with an initial reference count of 0. When they are assigned to a local variable, a parameter or a class field, their reference count gets incremented by one. If the local variable, parameter or class field already holds a reference to a reference counted object, the reference count of that object gets decremented by one. If they hold no reference, they contain null.

I proceeded to create two JIT helper functions, JIT_AddRef and JIT_Release (`/clr/src/vm/jitinterface.cpp`):

Example 5.19. `JIT_AddRef()`

```
HCIMPL1(void, JIT_AddRef,  Object* obj)
{
    if (obj == NULL)
        return;

    HELPER_METHOD_FRAME_BEGIN_NOPOLL();

    _ASSERTE(obj->GetMethodTable()->IsReferenceCounted());

    obj->GetReferenceCountHeader()->AddRef();

    HELPER_METHOD_FRAME_END();
}
HCIMPLEND
```

`JIT_Release()` is similar. Calling `JIT_AddRef()` or `JIT_Release()` on a null pointer is defined as a no-op.

5.6.1. The JIT compiler

In the SSCLI, the JIT compiler is a separate component from the execution engine proper. It resides in the directory `/clr/src/fjit`. The F in fjit presumably stands for “fast”, although the JIT in the SSCLI does not seem to be the same as that in the commercial .NET implementation, and is not very optimized.

The implementation of the JIT compiler mostly consists of a large switch statement over the various IL opcodes. The switch calls various helper methods, which in turn emits the native code through several layers of increasingly processor-specific macros. The extensive use of the C++ macro mechanism here makes working with the code a lot harder than it should be, since the macros are not very friendly to debuggers and it is sometimes very hard to see what code is actually being executed. Working with this part of the code was one of the hardest parts of implementing reference counting, and it often took me days to figure out how to do some of these things.

In my implementation I have focused solely on the x86 part of the JIT compiler.

5.6.2. Handling local variables

In CIL, local variables are handled by the `ldloc` and `stloc` instructions. They load or store the value of a local variable from or to the IL execution stack. The IL execution stack does not actually exist in the emitted native code, but the semantics of it are mostly emulated by the processor stack. For example, a `ldloc` usually translates into something like this (on the Intel architecture):

Example 5.20. Assembly emitted for the `ldloc` instruction

```
mov         eax,dword ptr [ebp-10h]
push        eax
```

In other words, the local variable is retrieved from the method stack frame and pushed onto the stack. The JIT does not push the local variable onto the stack in all cases, though. If it is to be reused instantly, for example in some calculation, it is merely kept in the CPU register. The JIT constantly keeps track of whether the *Top Of Stack* (TOS) is on the CPU stack or in a register. On the IA/32 architecture, the register used for the TOS is always `eax`.

Emitting code for the `stloc` instruction is handled by the `FJit::compileDO_STLOC` method. Here I call two helper methods which conditionally emits calls to `AddRef()` or `Release()` depending on whether the local variable is a reference counted object or not:

Example 5.21. `FJit::compileDO_STLOC`

```
// make sure to call .Release() on the old object in the var
maybeReleaseArgOrVar(varInfo);

trackedType = varInfo->type;
//trackedType.toNormalizedType();
TYPE_SWITCH_PRECISE(trackedType,emit_STVAR, (varInfo-
>offset));

// emit a call to increase the reference count if the new
 object is reference counted
maybeAddRefArgOrVar(varInfo);
```

`maybeAddRefArgOrVar()` looks like this:

Example 5.22. `FJit::maybeAddRefArgOrVar()`

```
FJitResult FJit::maybeAddRefArgOrVar(stackItems* varInfo) {
    if (isRefCounted(varInfo->type))
    {
        callInfo.reset();
        TYPE_SWITCH_PRECISE(varInfo->type, emit_LDVAR,
 (varInfo->offset));
        emit_tos_arg( 1, INTERNAL_CALL );
        emit_callhelper_I4(jitInfo-
>getHelperFtn(CORINFO_HELP_ADDREF));
    }

    return FJIT_OK;
}
```

Here I actually emulate the semantics of the `ldloc` instruction to load the value of the variable so it can be passed on the CPU stack for the call to the helper function.

`isRefCounted()` is a helper function which calls into the execution engine to determine whether the local is a reference counted object. Doing this at JIT time instead of at runtime has some interesting implications, a topic which I will revisit later.

5.6.3. Parameters

Parameters are just another type of local variables, and the `starg` instruction to store a value in a parameter is handled in an almost identical way (thus the name `maybeAddRefArgOrVar()`) to `stloc`. The method handling the emitting of native code in this case is `FJit::compileDO_STARG`.

However, parameters also have another aspect when it comes to reference counting: when an object is passed as a parameter to another method, it needs to have its reference count incremented. Further, this reference count needs to be decremented at the end of the method.

JIT-compiling a single method is handled by the `FJit::compileMethod` method. At the beginning I insert a call to this emitParameterAddRefCalls():

Example 5.23. `FJit::emitParameterAddRefCalls()`

```
FJitResult FJit::() {
    int paramCount = methodInfo->args.numArgs;
    stackItems* paramInfo;

    for(int i = 0; i < paramCount; i++ )
    {
        paramInfo = &argsMap[i];
        OpType type = paramInfo->type;

        // is this a reference counted type?
        if (type.isRef() && jitInfo-
>isReferenceCounted(type.cls()))
        {
            callInfo.reset();
            TYPE_SWITCH_PRECISE(type, emit_LDVAR, (paramInfo-
>offset));
            emit_tos_arg( 1, INTERNAL_CALL );
            emit_callhelper_I4(jitInfo-
>getHelperFtn(CORINFO_HELP_ADDREF));
        }
    }

    return FJIT_OK;
}
```

Here, a call to `AddRef()` is emitted for every reference counted parameter passed to the JIT-compiled method.

A similar function called at the end of `FJit::compileMethod` emits calls to `Release()` on every parameter *and* local variable. This, in addition to the way objects have their reference count decremented every time a local is overwritten with a new value, ensures that objects are deallocated and finalized as soon as they go out of scope. As mentioned earlier, objects referenced by local variables have their reference counts incremented to 1 when they are initially assigned.

5.6.4. Class fields

5.6.4.1. Assignments

While dealing with local method/function variables and parameters was a pretty straightforward task, assignments to class fields also needed to be intercepted and dealt with. Unfortunately, this did not turn out to be quite as simple as the work on locals. The code in the JIT compiler dealing with this was rather complex, and the way the JIT compiler is implemented, with numerous layers of increasingly CPU-specific C++ macros, tended to obfuscate matters a great deal.

I spent a lot of time trying to decipher exactly what the JIT macros were doing in order to inject code to call `AddRef()` and `Release()` whenever class fields are assigned to. In addition to trying to understand the macros, another source of difficulty lay in the implicit dependencies between the way the various IL instructions were handled by the JIT.

The instruction for field assignment in CIL is `stfld`. The semantics of `stfld` are as follows (with regard to the IL execution stack, which must not be confused with the native CPU stack):

1. An object reference is pushed onto the stack (this step is skipped for assignments to static fields).

2. The value that is to be assigned to the field is pushed onto the stack.

3. `stfld` *`field`* is executed. *`field`* here is a metadata token for the field that is to be assigned to.

For assigning a value to a non-static field of an object, the following IL is generated by the C# compiler:

Example 5.24. Simple field assignment in IL

```
ldloc.0       // load a local variable onto the stack
ldc.i4.s  42  // load the value '42' onto the stack
stfld int32::TestField::Field // store the top-of-stack into
 this field
```

This parallells what the code emitted by the JIT compiler does with regards to the native CPU stack. [4]. On the IA/32 architecture, the JIT compiler emits something like this for the above IL:

Example 5.25. Emitted x86 code for field assignment:

```
06FE0039  mov         eax,dword ptr [ebp-10h]
06FE003C  push        eax
06FE003D  mov         eax,2Ah
06FE0042  push        eax
06FE0043  mov         eax,4
06FE0048  push        eax
06FE0049  mov         eax,6D819C4h
06FE004E  call        eax
06FE0050  add         esp,0Ch
```

This assembler code pushes the address of the object and the value to be assigned to the field onto the CPU stack, followed by the offset into the object where the field is located. A helper function which performs the actual field assignment is then called.

The problem is that, by the time the JIT compiler encounters the `stfld` instruction, all the code up to where the field offset is pushed has already been emitted by the handlers for the other IL instructions. However, I also need access to the values that are on the stack in order to do several things:

1. I need to call `Release()` on the value previously stored in the field, if it is a reference to a reference counted object.

2. The original semantics of the `stfld` instruction needs to be preserved.

3. I need to call `AddRef()`
   on the new object reference stored in the field.

The first part is simple; I can emit code to do the equivalent of the `ldfld` instruction to get at the previous object reference in the field. That will leave

[4] I have, in my work on the JIT compiler, exclusively focused on supporting the IA/32 architecture. Since the JIT compiler is fairly well abstracted from the actual CPU architecture through its layers of macros, my changes should also work on PowerPC CPUs, but I have not tested this configuration.

the address of this object on the top of the native stack, and I can emit a call to `Release()`. However, this needs to be followed by a call to the original helper function for `stfld`, and that leaves me with a problem: The stack frame is no longer set up for this call. And I have yet another function call to emit after that!

The approach I eventually landed on was this: I first pop off all the arguments that have previously been put on the stack when reaching the `stfld` instruction. I then set up stack frames for an `ldfld`, `stfld` followed by yet another `ldfld`. The two `ldfld`s are used to get at the object stored in the field in order to call `Release()` and `AddRef()` on it. All this is handled by the `FJit::doEmitSTFLD_REF` helper method:

Example 5.26. FJitResult FJit::doEmitSTFLD_REF

```
pop_register(CALLEE_SAVED_1, 0); // offset
pop_register(ARG_1, 0); // RHS object
pop_register(CALLEE_SAVED_2, 0); // LHS object

// push for the second LDFLD call
push_register(CALLEE_SAVED_2, 0); // object
push_register(CALLEE_SAVED_1, 0); // offset

// push for the STFLD call
push_register(CALLEE_SAVED_2, 0); // LHS object
push_register(ARG_1, 0); // RHS object
push_register(CALLEE_SAVED_1, 0); // offset

// push for the first LDFLD call
push_register(CALLEE_SAVED_2, 0); // object
push_register(CALLEE_SAVED_1, 0); // offset

// get the object stored in the slot
emit_LDFLD_REF(isStatic);

// call Release() on it
callInfo.reset();
emit_tos_arg(1, INTERNAL_CALL);
emit_callhelper_I4(jitInfo-
>getHelperFtn(CORINFO_HELP_RELEASE));

// now the actual STFLD
emit_STFLD_REF((isStatic));

// get the new object in that slot
emit_LDFLD_REF(isStatic);

// AddRef() it
callInfo.reset();
emit_tos_arg(1, INTERNAL_CALL);
emit_callhelper_I4(jitInfo-
>getHelperFtn(CORINFO_HELP_ADDREF));
```

The above code is simplified and only applies to non-static fields; the stack frames are slightly different for static fields. The actual code takes this into account and deals with static field assignments properly.

5.6.4.2. Releasing class fields

When an object of a reference counted type is destroyed, it should call `Release()` on all of its reference counted fields (if it has any). It would also be beneficial if this behavior could be achieved for objects of “normal” types; types tracked by the tracing garbage collector.

Achieving the former should be relatively simple; just go through the values of any reference counted fields at the time of destruction and call `Release()` on them directly. Doing the latter requires some more thought. The main problem is that the garbage collector really isn't - it does not *collect* garbage as much as it collects non-garbage and dumps the rest. This unfortunately (for me) means that objects that are garbage are not visited by the collector before they are disposed of, leaving no way to decrement the reference counts on any fields held by the object.

The approach I have chosen is to treat objects with reference counted fields as if they have a finalizer. The objects will then be placed on the finalizer queue which gives us a way to to intercept them and deal with them before they are disposed of. When the finalizer is eventually called, it will iterate over all the reference counted fields and decrement their reference counts one by one. This solution has the added benefit of being usable for both types of objects; for objects controlled by the tracing GC the finalizer is called by the finalizer thread, and for reference counted objects the finalizer is called by the implementation of `Release()`.

5.6.4.2.1. Discovering reference counted fields

In order to mark objects of types that have reference counted fields as finalizable, it is necessary to discover exactly which types of objects this condition applies to. The natural place for performing such discovery is in the class loading mechanism.

In the CLR, the binary layout of a type isn't decided by the front end compiler. This is in contrast to how traditional language compilers work, where the binary layout of everything is hardcoded into the resulting executable. Instead, the CLR lays out fields at runtime when a type is initially loaded. This gives

the execution engine a large amount of flexibility to choose a layout that is optimal for the environment in which the program is to run.[5]

Laying out fields is handled by the `EEClass::InitializeFieldDescs()` method. I initially assumed it would just be a matter of checking the types of the fields and querying their `EEClass` instance as to whether they are reference counted or not. Alas, it turned out not to be that simple. At the time a type is loaded, you have no guarantee that the types of all of its fields are loaded as well. This makes sense, since loading the types of all fields would cause a cascade of type loads. Instead, the class loading mechanism does without exact type information, relying only on information about field sizes to determine a type's binary layout.

Fortunately, it *is* possible to extract a metadata token for the type of a field from the metadata token for the field itself. Given a metadata token for the type, I can use the same approach as I take to initially discover reference counted types. To extract this information from the field one needs to query its *signature*. In the CLR, a signature is a packed structure of binary data, which needs to be parsed in order to obtain the desired information. The SSCLI provides the `SigPointer` class to parse a signature. Given a field signature that represents a reference to an object, the metadata token for the object type is the third item. When the metadata token has been acquired, I can merely delegate to the `IsReferenceCounted()` function I wrote earlier that determines whether the `ReferenceCounted` attribute is applied to it.

[5]This obviously does not apply to structs that have explicit layout information associated with them. In the CLR, explicit layout of struct fields can be achieved using the `StructLayout` attribute.

Example 5.27. The `IsRCField()` function

```
HRESULT IsRCField(IMDInternalImport* pInternalImport,
 mdFieldDef fd)
{
  ULONG size;
  PCCOR_SIGNATURE sig = pInternalImport->GetSigOfFieldDef(fd,
 &size);

  SigPointer sigptr(sig);

  // the first part of the sig defines the calling
 convention
  _ASSERTE(sigptr.GetData() == IMAGE_CEE_CS_CALLCONV_FIELD);

  // then comes the type
  CorElementType corType = (CorElementType)sigptr.GetData();

  // we only deal with class objects
  if (corType == ELEMENT_TYPE_CLASS)
  {
      // next part of the signature is the metadata token for
 the field
      mdTypeDef type = sigptr.GetToken();
      return IsReferenceCounted(pInternalImport, type);
  }
  else
      return S_FALSE;
}
```

This function, `IsRCField()`, is called `InitializeFieldDescs()` for each field in a type. When a reference counted field is found, a flag is set in the `EEClass` and the `MethodTable` that indicates that the type has one or more reference counted fields. This flag is then checked by the tracing garbage collector when objects of the type are allocated in order to determine if the new object should be placed on the finalizer queue. If the flag is set, it gets placed on the queue regardless of whether it has other finalizer requirements. It is then up to the `MethodTable::CallFinalizer()` method to determine whether an object needs to perform traditional finalizer tasks, decrement reference counts on fields or both.

For a reference counted type, `CallFinalizer()` calls the `MethodTable::FinalizeRefCountedFields()` helper method.

This method iterates over all the fields of the object and calls `Release()` on any non-null reference counted object:

Example 5.28. `MethodTable::FinalizeRefCountedFields`

```
while ((curField = fdIterator.Next()) != NULL)
{
  // get the class for this type and check if it has RC
 fields
  EEClass* fieldType = curField->FindType().AsClass();
  if (fieldType->IsReferenceCounted())
  {
      OBJECTREF fieldObj = NULL;
      curField->GetInstanceField(obj, (LPVOID)&fieldObj);

      // release this object
      if (fieldObj != NULL)
      {
          fieldObj->GetReferenceCountHeader()->Release();
      }
  }
}
```

Chapter 6. Evaluating the results

6.1. Performance

6.1.1. The `Timer` class

To perform the benchmarks, I wrote a class called `Timer` that uses the `QueryPerformanceCounter` and `QueryPerformanceFrequency` Win32 API calls. These provide an extremely accurate timer. However, the use of these calls means that the `Timer` class is not portable.

6.1.2. Handling a large number of allocations

I wrote a benchmark that allocated a very large number of objects in a loop for then to immediately let them go out of scope. I ran this loop with both a reference counted type and a type tracked by the tracing garbage collector:

Example 6.1. Large number of allocations

```
public static void DoBenchmark()
{
  Timer t = new Timer();

  t.Start();
  for( int i = 0; i < NumIterations; i++ )
  {
    ReferenceCounted rc = new ReferenceCounted();
    rc.Method();
  }
  t.End();
  Console.WriteLine( "Reference counted: {0}", t.Interval );

  t.Start();
  for( int i = 0; i < NumIterations; i++ )
  {
    TracingGC gc = new TracingGC();
    gc.Method();
  }
  t.End();
  Console.WriteLine( "Tracing GC: {0}", t.Interval );
}
```

Both the objects are the same size in terms of instance fields.

With *NumIterations* set to 100 000, and running a free[1] build of the SSCLI, I would get results indicating that the reference counting mechanism was about twice as fast as the tracing mechanism.

Table 6.1. Results of benchmark 1

Run	Reference counting	Tracing GC
1	0,20s	0.48s
2	0.27s	0.43s
3	0.26s	0.45s
4	0.24s	0.38s
5	0.26s	0.49s
6	0.26s	0.55s
7	0.24s	0.52s
8	0.25s	0.40s
9	0.27s	0.51s
10	0.25s	0.41s

Raising *NumIterations* to ten millions did not change the relationship between the two mechanisms in any significant way - reference counting was still twice as fast as the tracing garbage collector.

By running a checked[2] build and turning on logging for the reference counting mechanism, I can see that there are only two objects allocated on the reference counted heap at any given time, and that the same two locations on this heap are reused over and over. There is no pressure on the reference counted heap in this case, and its size never grows over `sizeof(ReferenceCounted)*2`.

The same does not apply to the tracing collector. Each new object is allocated to a new address on the GC heap,and it takes a long time before the memory belonging to an object is finally reclaimed.

[1] A “free” build is one which is compiled with optimizations and with all logging features turned off.

[2] A “checked” build is one where all logging and debugging features are turned on, and the runtime is compiled without optimizations.

6.1.3. Assignments to local variables

The next benchmark shows what an impact the constant incrementing and decrementing of reference counts has. This test merely juggles a single object between various local variables:

Example 6.2. Assignments to local variables

```
private static void DoBenchmark()
{
  Timer t = new Timer();

  t.Start();
  ReferenceCounted rc = new ReferenceCounted();
  for( int i = 0; i < NumIterations; i++ )
  {
      ReferenceCounted rc2 = rc;
      ReferenceCounted rc3 = rc2;
      rc = rc3;
      rc3 = rc;
  }
  t.End();
  Console.WriteLine( "Reference counted: {0}", t.Interval );

  t.Start();
  TracingGC gc = new TracingGC();
  for( int i = 0; i < NumIterations; i++ )
  {
      TracingGC gc2 = gc;
      TracingGC gc3 = gc2;
      gc = gc3;
      gc3 = gc;
  }
  t.End();
  Console.WriteLine( "Tracing GC: {0}", t.Interval );
}
```

This benchmark only creates a single object from each type, so it does not stress the allocator mechanism. Further, none of the types have a finalizer.

Results from running this benchmark under a checked build shows that the tracing collector is approximately 17-18 times faster than the reference counted mechanism:

Table 6.2. Results of benchmark 2

Run	Reference counting	Tracing GC
1	8.44s	0.47s
2	8.48s	0.46s
3	8.63s	0.48s

6.1.4. Field assignments

This benchmark is quite similar to the previous one, only using fields instead of local variables:

```
private static void DoBenchmark()
{
  Timer t = new Timer();

  t.Start();
  rc = new ReferenceCounted();
  for( int i = 0; i < NumIterations; i++ )
  {
    rc2 = rc;
    rc3 = rc2;
    rc = rc3;
    rc3 = rc;

  }
  t.End();
  Console.WriteLine( "Reference counted: {0}", t.Interval );

  t.Start();
  gc = new TracingGC();
  for( int i = 0; i < NumIterations; i++ )
  {
    gc2 = gc;
    gc3 = gc2;
    gc = gc3;
    gc3 = gc;
  }
  t.End();
  Console.WriteLine( "Tracing GC: {0}", t.Interval );
}

private static ReferenceCounted rc, rc2, rc3;
```

```
private static TracingGC gc, gc2, gc3;
```

Here, the tracing garbage collector is only about 4 times faster than the reference counting mechanism:

Table 6.3. Results of benchmark 3

Run	Reference counting	Tracing GC
1	10,30s	2,54s
2	10,36s	2,80s
3	10,55s	2,75s

6.2. Missing features

6.2.1. Reference counted arrays

As of now, there is no support for reference counting in arrays at all. Array objects are not reference counted, and reference counted objects placed in arrays are not properly tracked. This is obviously a serious setback, and one that would have to be dealt with before the implementation could be used in a real environment.

6.2.2. `object` references pointing to reference counted objec

One serious implication of the hybrid implementation is that it is possible to refer to reference counted objects through references that aren't typed as a reference counted type. In the CLR, the `Object` class is the supertype of all other types, and a reference to `Object` can be used to point to all objects. Consider the following situation:

Example 6.3. An `Object` reference pointing to a reference counted object

```
public void Method()
{
  this.Create();
  obj.Foo();
}

private RefCounted Create()
{
  RefCounted rc = new RefCounted();
  this.obj = rc;
}

private Object obj;
```

Since the *`obj`* field is typed as `Object`, assigning the reference counted object to it does not increment its reference count. At the end of the `Create()` method, the only reference to the object goes out of scope, and the object is destroyed. Since the *`obj`* field now points to a destroyed object, the call to `Foo()` blows up.

There is no obvious solution to this problem. The brute force way would be to emit code that checks whether an object is reference counted for each and every field, parameter and local variable assignment. However, this would cause massive bloat of the emitted code, and having to perform this call for every single assignment would seriously degrade runtime performance. It is safe to assume that this is not a feasible way of dealing with the issue.

A more practical solution might be to just disallow assignments of reference counted types to references not typed as such altogether. Such a check could be performed in the JIT compiler, and would not impact runtime performance like the previous alternative would.

However, this leaves us in the undesirable situation where the JIT compiler rejects code that is valid according to the various language compilers already in existence. Although this check happens in the JIT compiler, from the user's perspective this is not distinguishable from what happens at runtime. It might be hard to convince users that code that compiles by a compiler that conforms to the standard can be rejected by the JIT compiler at run/load-time.

This leads naturally to a third option: modify existing compilers to treat reference counted types as a special case, disallowing unsafe assignments. Unfortunately, this is as unpractical as the first alternative. There is a large amount of compilers out there, and expecting them all to change in order to support reference counting is not realistic.

6.3. Possible enhancements

6.3.1. Inlining calls to `AddRef()` and `Release()`

The current implementation implements reference counting through emitting function calls in the Just In Time compiler to a helper function, which then in turn calls either `AddRef()` or `Release()`.

Since incrementing and decrementing reference counts is a quite frequent operation, there is a lot of overhead involved in setting up two call frames and executing the calls. The implementation of `AddRef()` is quite simple:

Example 6.4. The implementation of `AddRef()`

```
LONG ReferenceCountHeader::AddRef()
{
    return InterlockedIncrement(&m_RefCount);;
}
```

`AddRef()` would be fairly simple to implement by making the JIT compiler emit code to perform the increment directly. If the address of the object is in the `eax` register, the JIT could emit the following IA/32 code:

Example 6.5. Implementing `AddRef()` inline

```
    lock xadd dword ptr[eax-offset], 1
```

It can be argued that this would require the JIT compiler to have more knowledge of the binary layout of the runtime objects than it currently has,

and that doing this inline would violate the encapsulation that exists between the JIT and the rest of the execution engine. We remember from Figure 5.1, “The type loading mechanism” that the JIT and the execution engine exists in two separate libraries, and that the JIT compiler only communicates with the execution engine through the `ICorJitInfo` interface (which is implemented by the `CEEInfo` class).

However, for something that's so performance critical as this, I think allowing the JIT to have a little more intimate knowledge of the object internals would be worth the cost.

`Release()`, however, would be harder to inline. The implementation currently looks like this:

Example 6.6. The implementation of `Release()`

```
LONG ReferenceCountHeader::Release()
{
  LONG refcount;

  if ((refcount = InterlockedDecrement(&m_RefCount)) == 0)
  {
    RCLogFree(GetObject()->GetMethodTable(), GetObject());

    // finalize if we have a finalizer
    Finalize();

    // free memory associated with the object
    g_pRCHeap->Free(this);
  }
  return refcount;
}
```

Inserting all of that into the generated code at every spot where an object goes out of scope would cause serious code bloat, something that would probably affect performance in a negative way. However, if the contents of the `if` block was factored out to a separate function, code similar to the following could be emitted by the JIT compiler:

Example 6.7. Implementing `Release()` inline:

```
    lock xadd dword ptr[eax-offset], -1
    jnz notzero
    push eax
    call FreeObject
notzero:
    ....
```

Here, the `FreeObject` function performs the equivalent of the `if` block in the current implementation. Four instructions inserted for every object release would most likely be acceptable, especially considering there is already some overhead with regards to getting the old values of fields, parameters and local variables.

6.3.2. Replacing the allocator

The memory allocator used by the reference counted heap in the current implementation of reference counting is the same as that which is used for the large object heap in the tracing garbage collector. This allocator is obviously not tuned for the needs of quickly allocating a large number of small objects.

Further research could be vested into the possibility of either enhancing this allocator with an eye to making it more suitable for the needs of the reference counted heap.

6.4. Conclusion

Through my work on this paper and the accompanying implementation, I have shown that it is indeed feasible to introduce reference counting as an additional memory management mechanism in the common language runtime. While the hybrid model does pose some unique problems, these are not insurmountable.

Reference counting could be the mechanism of choice for types where prompt finalization is of paramount importance. This includes classes that hold references to scarce operating system resources, such as file handles. Reference counting is also a likely candidate for scenarios in which a large number of objects need to be allocated quickly, but where the references to individual instances quickly go out of scope. In this case, a tracing GC may allow quite a large number of live objects to exist before the memory pressure gets big

enough to trigger a collection. If these objects also have a finalizer, the memory allocated for the objects will not be freed until the finalizers have been run.

The benchmarks show that the reference counting mechanism is considerably slower than the tracing garbage collector for operations like assigning objects to local variables and fields. However, in real code such assignments are a relatively rare event, mitigating the extra cost of reference counting to some extent.

Appendix A. The source code

A.1. Subversion repository

The source for the SSCLI with my changes incorporated are maintained using the Subversion version control system. The Subversion command line client is available for a large number of platforms, including Windows, and can be obtained from http://subversion.tigris.org

A.1.1. Checking out the source

Using the **svn** command line client, the source code can be obtained from http://ankhsvn.com/svn/masters/trunk/sscli/:

```
C:\>svn checkout http://ankhsvn.com/svn/masters/trunk/sscli/
```

The repository can also be browsed in a regular web browser by pasting the above URL into the address field.

A.1.2. Viewing the log

Log messages for the changes made against the original SSCLI source code can be viewed using the **svn log** command:

```
C:\sscli>svn log
```

A.1.3. Unified diff

The differences between the original SSCLI and my implementation can be viewed in unified diff format:

```
C:\sscli>svn diff -r 1:HEAD
```

A.2. Statistics

In the process of implementing reference counting in the CLR, I have added 1704 lines to the source and removed 269. This does not include project files and other auxilliary files.

Appendix B. Testing

B.1. The SSCLI test suites

The SSCLI comes with a rather large battery of tests. These are divided into two categories:

PAL test suite
: This suite tests the Platform Adaptation Layer, the SSCLI's primary portability layer.

SSCLI quality test suite
: This suite tests pretty much everything else. For a large part, it consists of C# classes that stress various features of the execution engine.

B.2. The SSCLI test harness

The tests in the SSCLI quality suite are driven by a test harness written in Perl. The main driver is **rrun.pl** which resides in the `tests\` directory. To run all the tests:

```
C:\sscli\tests>perl rrun.pl
```

Specific tests can also be run by specifying the name of the test as an argument to **rrun.pl**. The test names are defined in files called `rsources` which exist in every directory in the test directory tree.

B.3. Reference counting tests

The development of the reference counted mechanism was entirely driven by tests. No feature was added before I had a failing test.

The reference counting tests are located in the `tests\refcounting` directory. These tests can be run by executing **rrun.pl** from this directory.

The tests are as follows:

testbasic
: This simple tests merely verifies that the SSCLI still behaves in a sane way after introducing the `ReferenceCounted` attribute.

testdestructor
: This test verifies that the destructor of a reference counted object is called when its reference count reaches zero.

massiveallocations
: This test checks that the reference counted heap can deal with a large number of big objects.

testgcreferencecount
: This tests the new `ReferenceCount` method on the `System.GC` class.

testparameterpassing
: This tests that reference counts are correctly incremented and decremented for parameters of methods.

testparameterpassing2
: Another test for method parameters.

testreferencecount1
: This test tests reference counts for local variables.

testfields
: This tests reference counts for assignments to instance fields of class objects.

teststaticfields
: This tests reference counts for assignments to static fields of classes.

testrcfinalizer
: Another test that verifies finalizers are called when a reference counted object is destroyed.

testconcurrency
: This verifies that reference counts are correctly maintained even when a large number of threads access and manipulate the same reference counted objects.

Bibliography

[1] *The Mythical Month*. Essays on Software Engineering. Frederick Brooks. 1975.

[2] *No Silver Bullet*. Essence and Accidents of Software Engineering. Frederick Brooks. Computer Magazine1987.

[3] *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I [http://www-formal.stanford.edu/jmc/recursive.html]*. April 1960. Communications of the ACM.

[4] *www.python.org [http://www.python.org]*.

[5] *Evaluating Java for Game Development [http://www.rolemaker.dk/articles/evaljava/]*. March 4, 2002.

[6] *Standard ECMA-334 C# Language Specification [http://www.ecma-international.org/publications/standards/Ecma-334.htm]*. 2002.

[7] *Standard ECMA-335 Common Language Infrastructure (CLI) [http://www.ecma-international.org/publications/standards/Ecma-335.htm]*. 2002.

[8] *ECMA-335: CLI Partition I - Architecture [http://msdn.microsoft.com/net/ecma/]*. 2002.

[9] *ECMA-335: Class Library Detailed Specifications [http://msdn.microsoft.com/net/ecma/]*. 2002.

[10] *http://www.mono-project.com/* .

[11] *http://www.southern-storm.com.au/portable_net.html [http://www.southern-storm.com.au/portable_net.html]*.

[12] *http://msdn.microsoft.com/net/sscli [http://msdn.microsoft.com/net/sscli]* .