Garbage Collection Performance Best Practices

In this section, we will provide a summary of best practices for interacting with the .NET garbage collector. We will examine multiple scenarios which demonstrate aspects of these best practices, and point out pitfalls that must be avoided.

Generational Model

We reviewed the generational model as the enabler of two significant performance optimizations to the naïve GC model discussed previously. The generational heap partitions managed objects by their life expectancy, enabling the GC to frequently collect objects that are short-lived and have high collection likelihood. Additionally, the separate large object heap solves the problem of copying large objects around by employing a free list management strategy for reclaimed large memory blocks.

We can now summarize the best practices for interacting with the generational model, and then review a few examples.

- Temporary objects should be short-lived. The greatest risk is temporary objects that creep into generation 2, because this causes frequent full collections.
- Large objects should be long-lived or pooled. A LOH collection is equivalent to a full collection.
- References between generations should be kept to a minimum.

The following case studies represent the risks of the mid-life crisis phenomenon. In a monitoring UI application implemented by one of our customers, 20,000 log records were constantly displayed on the application's main screen. Each individual record contained a severity level, a brief description message and additional (potentially large) contextual information. These 20,000 records were continuously replaced as new log records would flow into the system.

Because of the large amount of log records on display, most log records would survive two garbage collections and reach generation 2. However, the log records are not long-lived objects, and shortly afterwards they are replaced by new records which in turn creep into generation 2, exhibiting the mid-life crisis phenomenon. The net effect was that the application performed hundreds of full garbage collections per minute, spending nearly 50% of its time in GC code.

After an initial analysis phase, we had reached the conclusion that displaying 20,000 log records in the UI is unnecessary. We trimmed the display to show the 1,000 most recent records, and implemented pooling to reuse and reinitialize existing record objects instead of creating new ones. These measures minimized the memory footprint of the application, but, even more significantly, minimized the time in GC to 0.1%, with a full collection occurring only once in a few minutes.

Another example of mid-life crisis is a scenario encountered by one of our web servers. The web server system is partitioned to a set of front-end servers which receive web requests. These front-end servers use synchronous web service calls to a set of back-end servers in order to process the individual requests.

In the QA lab environment, the web service call between the front-end and back-end layers would return within several milliseconds. This caused the HTTP request to be dismissed shortly, so the request object and its associated object graph were truly short-lived.

In the production environment, the web service call would often take longer to execute, due to network conditions, back-end server load and other factors. The request still returned to the client within a fraction of a second, which was not worth optimizing because a human being cannot observe the difference. However, with many requests flowing into the system each second, the lifetime of each request object and its associated object graph was extended such that these objects survived multiple garbage collections and creep into generation 2.

It is important to observe that the server's ability to process requests was not harmed by the fact requests lived slightly longer: the memory load was still acceptable and the clients didn't feel a difference because the requests were still returned within a fraction of a second. However, the server's ability to scale was significantly harmed, because the front-end application spent 70% of its time in GC code.

Resolving this scenario requires switching to asynchronous web service calls or releasing most objects associated with the request as eagerly as possible (before performing the synchronous service call). A combination of the two brought time in GC down to 3%, improving the site's ability to scale by a factor of 3!

Finally, consider a design scenario for a simple 2D pixel-based graphics rendering system. In this kind of system, a drawing surface is a long-lived entity which constantly repaints itself by placing and replacing short-lived pixels of varying color and opacity.

If these pixels were represented by a reference type, then not only would we double or triple the memory footprint of the application; we would also create references between generations and create a gigantic object graph of all the pixels. The only practical approach is using value types to represent pixels, which can bring down the memory footprint by a factor of 2 or 3, and the time spent in GC by several orders of magnitude.

Pinning

We have previously discussed pinning as a correctness measure that must be employed to ensure that the address of a managed object can safely be passed to unmanaged code. Pinning an object keeps it in the same location in memory, thereby reducing the garbage collector's inherent ability to defragment the heap by sweeping objects around.

With this in mind, we can summarize the best practices for using pinning in applications that require it.

- Pin objects for the shortest possible time. Pinning is cheap if no garbage collection occurs while the object is pinned. If calling unmanaged code that requires a pinned object for an indefinite amount of time (such as an asynchronous call), consider copying or unmanaged memory allocation instead of pinning a managed object.
- Pin a few large buffers instead of many small ones, even if it means you have to manage the small buffer sizes yourself. Large objects do not move in memory, thus minimizing the fragmentation cost of pinning.
- Pin and re-use old objects that have been allocated in the application's startup path instead of allocating new buffers for pinning. Old objects rarely move, thus minimizing the fragmentation cost of pinning.

- Consider allocating unmanaged memory if the application relies heavily on pinning. Unmanaged memory can be directly manipulated by unmanaged code without pinning or incurring any garbage collection cost. Using unsafe code (C# pointers) it is possible to conveniently manipulate unmanaged memory blocks from managed code without copying the data to managed data structures. Allocating unmanaged memory from managed code is typically performed using the `System.Runtime.InteropServices`.Marshal class. (See Chapter 8 for more details.)

Finalization

The spirit of the finalization section makes it quite clear that the automatic non-deterministic finalization feature provided by .NET leaves much to be desired. The best piece of advice with regard to finalization is to make it deterministic whenever possible, and delegate the exceptional cases to the non-deterministic finalizer.

The following practices summarize the proper way to address finalization in your application:

- Prefer deterministic finalization and implement `IDisposable` to ensure that clients know what to expect from your class. Use `GC.SuppressFinalize` within your Dispose implementation to ensure that the finalizer isn't called if it isn't necessary.
- Provide a finalizer and use Debug.Assert or a logging statement to ensure that clients are aware of the fact they didn't use your class properly.
- When implementing a complex object, wrap the finalizable resource in a separate class (the `System.Runtime.InteropServices`.SafeHandle type is a canonical example). This will ensure that only this small type wrapping the unmanaged resource will survive an extra garbage collection, and the rest of your object can be destroyed when the object is no longer referenced.

Miscellaneous Tips and Best Practices

In this section, we will briefly examine assorted best practices and performance tips that do not belong to any other major section discussed in this chapter.

Value Types

When possible, prefer value types to reference types. We have examined the various traits of value types and reference types from a general perspective in Chapter 3. Additionally, value types have several characteristics that affect garbage collection cost in the application:

- Value types have the most negligible allocation cost when used as local stack variables. This allocation is associated with the extension of the stack frame, which is created whenever a method is entered.
- Value types used as local stack variables have no deallocation (garbage collection) cost—they are deallocated automatically when the method returns and its stack frame is destroyed.
- Value types embedded in reference types minimize the cost of both phases of garbage collections: if objects are larger, there are less objects to mark, and if objects are larger, the sweep phase copies more memory at each time, which reduces the overhead of copying many small objects around.
- Value types reduce the application's memory footprint because they occupy less memory. Additionally, when embedded inside a reference type, they do not require a reference in order to access them, thereby eliminating the need to store an additional reference. Finally, value types embedded in reference types exhibit locality of access—if the object is paged in and hot in cache, its embedded value type fields are also likely to be paged in and hot in cache.
- Value types reduce references between generations, because fewer references are introduced into the object graph.

Object Graphs

Reducing the size of the object graph directly affects the amount of work the garbage collector has to perform. A simple graph of large objects is faster to mark and sweep than a complex graph of many small objects. We have mentioned a specific scenario of this earlier.

Additionally, introducing fewer local variables of reference types reduces the size of the local root tables generated by the JIT, which improves compilation times and saves a small amount of memory.

Pooling Objects

Object pooling is a mechanism designed to manage memory and resources manually instead of relying on the facilities provided by the execution environment. When object pooling is in effect, allocating a new object means retrieving it from a pool of unused objects, and deallocating an object means returning it to the pool.

Pooling can provide a performance boost if the allocation and deallocation costs (not including initialization and uninitialization) are more expensive than managing the lifetime of the objects manually. For example, pooling large objects instead of allocating and freeing them using the garbage collector might provide a performance improvement, because freeing large objects that are frequently allocated requires a full garbage collection.

Note Windows Communication Foundation (WCF) implements pooling of byte arrays used for storing and transmitting messages. The System.ServiceModel.Channels.BufferManager abstract class serves as the pool façade, providing the facilities for obtaining an array of bytes from the pool and for returning it to the pool. Two internal implementations of the abstract base operations provide a GC-based allocation and deallocation mechanism side-by-side with a mechanism which manages pool of buffers. The pooled implementation (as of the time of writing) manages multiple pools internally for varying sizes of buffers, and takes the allocating thread into account. A similar technique is used by the Windows Low-Fragmentation Heap, introduced in Windows XP.

Implementing an efficient pool requires taking at least the following factors into consideration:

- Synchronization around the allocation and deallocation operations must be kept to a minimum. For example, a lock-free (wait-free) data structure could be used to implement the pool (see Chapter 6 for a treatment of lock-free synchronization).
- The pool should not be allowed to grow indefinitely, implying that under certain circumstances objects will be returned to the mercy of the garbage collector.
- The pool should not be exhausted frequently, implying that a growth heuristic is needed to balance the pool size based on the frequency and amount of allocation requests.

Most pooling implementations will also benefit from implementing a least recently used (LRU) mechanism for retrieving objects from the pool, because the least recently used object is likely to be paged in and hot in CPU cache.

Implementing pooling in .NET requires hooking the allocation and deallocation requests for instances of the pooled type. There is no way to hook the allocation directly (the new operator cannot be overloaded), but an alternative API such as Pool.GetInstance can be used. Returning an object to the pool is best implemented using the Dispose pattern, with finalization as a backup.

An extremely simplified skeleton of a .NET pool implementation and a matching poolable object base is shown in the following code:

```
public class Pool<T> {
  private ConcurrentBag<T> pool = new ConcurrentBag<T>();
  private Func<T> objectFactory;

  public Pool(Func<T> factory) {
    objectFactory = factory;
  }
  public T GetInstance() {
    T result;
    if (!pool.TryTake(out result)) {
     result = objectFactory();
    }
    return result;
  }
  public void ReturnToPool(T instance) {
    pool.Add(instance);
  }
}

public class PoolableObjectBase<T> : IDisposable {
  private static Pool<T> pool = new Pool<T>();

  public void Dispose() {
    pool.ReturnToPool(this);
  }
  ~PoolableObjectBase() {
    GC.ReRegisterForFinalize(this);
    pool.ReturnToPool(this);
  }
}

public class MyPoolableObjectExample : PoolableObjectBase<MyPoolableObjectExample> {
  ...
}
```

Paging and Allocating Unmanaged Memory

The .NET garbage collector automatically reclaims unused memory. Therefore, by definition, it cannot provide a perfect solution for every memory management need that may arise in real-world applications.

In previous sections, we examined numerous scenarios in which the garbage collector behaves poorly and must be tuned to provide adequate performance. An additional example that can bring the application down to its knees involves using the garbage collector where physical memory is insufficient to contain all objects.

Consider an application that runs on a machine with 8GB of physical memory (RAM). Such machines are soon to become extinct, but the scenario easily scales to any amount of physical memory, as long as the application is capable of addressing it (on a 64-bit system, this is a lot of memory). The application allocates 12GB of memory, of which at most 8GB will reside in the working set (physical memory) and at least 4GB will be paged out to disk. The Windows working set manager will ensure that pages containing objects frequently accessed by the application will remain in physical memory, and pages containing objects rarely accessed by the application will be paged out to disk.

During normal operation, the application might not exhibit paging at all because it very rarely accesses the paged-out objects. However,

when a full garbage collection occurs, the GC must traverse every reachable object to construct the graph in the mark phase. Traversing every reachable object means performing 4GB of reads from disk to access them. Because physical memory is full, this also implies that 4GB of writes to disk must be performed to free physical memory for the paged-out objects. Finally, after the garbage collection completes, the application will attempt accessing its frequently used objects, which might have been paged out to disk, inducing additional page faults.

Typical hard drives, at the time of writing, provide transfer rates of approximately 150MB/s for sequential reads and writes (even fast solid-state drivers don't exceed this by more than a factor of 2). Therefore, performing 8GB of transfers while paging in from and paging out to disk might take approximately 55 seconds. During this time, the application is waiting for the GC to complete (unless it is using concurrent GC); adding more processors (i.e. using server GC) would not help because the disk is the bottleneck. Other applications on the system will suffer a substantial decrease in performance because the physical disk will be saturated by paging requests.

The only way to address this scenario is by allocating unmanaged memory to store objects that are likely to be paged out to disk. Unmanaged memory is not subject to garbage collection, and will be accessed only when the application needs it.

Another example that has to do with controlling paging and working set management is locking pages into physical memory. Windows applications have a documented interface to the system requesting that specific regions of memory are not paged out to disk (disregarding exceptional circumstances). This mechanism can't be used directly in conjunction with the .NET garbage collector, because a managed application does not have direct control over the virtual memory allocations performed by the CLR. However, a custom CLR host can lock pages into memory as part of the virtual memory allocation request arriving from the CLR.

Static Code Analysis (FxCop) Rules

Static code analysis (FxCop) for Visual Studio has a set of rules targeting common performance and correctness problems related to garbage collection. We recommend using these rules because they often catch bugs during the coding phase, which are cheapest to identify and fix. For more information on managed code analysis with or without Visual Studio, consult the MSDN online documentation.

The GC-related rules that shipped with Visual Studio 11 are:

- Design Rules—*CA1001*—Types that own disposable fields should be disposable. This rule ensures deterministic finalization of type members by their aggregating type.
- Design Rules—*CA1049*—Types that own native resources should be disposable. This rule ensures that types providing access to native resources (such as `System.Runtime.InteropServices`.HandleRef) implement the Dispose pattern correctly.
- Design Rules—*CA1063*—Implement IDisposable correctly. This rule ensures that the Dispose pattern is correctly implemented by disposable types.
- Performance Rules—*CA1821*—Remove empty finalizers. This rule ensures that types do not have empty finalizers, which degrade performance and effect mid-life crisis.
- Reliability Rules—CA2000—Dispose objects before losing scope. This rule ensures that all local variables of an IDisposable type are disposed before they disappear out of scope.
- Reliability Rules—*CA2006*—Use SafeHandle to encapsulate native resources. This rule ensures that when possible, the SafeHandle class or one of its derived classes is used instead of a direct handle (such as System.IntPtr) to an unmanaged resource.
- Usage Rules—*CA1816*—Call `GC.SuppressFinalize` correctly. This rule ensures that disposable types call suppress finalization within their finalizer, and do not suppress finalization for unrelated objects.
- Usage Rules—*CA2213*—Disposable fields should be disposed. This rule ensures that types implementing IDisposable should in turn call Dispose on all their fields that implement IDisposable.
- Usage Rules—*CA2215*—Dispose methods should call base class dispose. This rule ensures that the Dispose pattern is implemented correctly by invoking the base class' Dispose method if it is also IDisposable.
- Usage Rules—*CA2216*—Disposable types should declare finalizer. This rule ensures that disposable types provide a finalizer as a backup for the scenario when the class user neglects to deterministically finalize the object.