Garbage Collection Flavors

The .NET garbage collector comes in several flavors, even though it might appear to the outside as a large and monolithic piece of code with little room for customization. These flavors exist to differentiate multiple scenarios: Client-oriented applications, high-performance server applications, and so on. To understand how these various flavors are really different from each other, we must look at the garbage collector's interaction with the other application threads (often called *mutator threads*).

Pausing Threads for Garbage Collection

When a garbage collection occurs, application threads are normally executing. After all, the garbage collection request is typically a result of a new allocation being made in the application's code—so it's certainly willing to run. The work performed by the GC affects the memory locations of objects and the references to these objects. Moving objects in memory and changing their references while application code is using them is prone to be problematic.

On the other hand, in some scenarios executing the garbage collection process concurrently with other application threads is of paramount importance. For example, consider a classic GUI application. If the garbage collection process is triggered on a background thread, we want to be able to keep the UI responsive while the collection occurs. Even though the collection itself might take longer to complete (because the UI is competing for CPU resources with the GC), the user is going to be much happier because the application is more responsive.

There are two categories of problems that can arise if the garbage collector executes concurrently with other application threads:

- *False negatives*: An object is considered alive even though it is eligible for garbage collection. This is an undesired effect, but if the object is going to be collected in the next cycle, we can live with the consequences.
- *False positives*: An object is considered dead even though it is still referenced by the application. This is a debugging nightmare, and the garbage collector must do everything in its power to prevent this situation from happening.

Let's consider the two phases of garbage collection and see whether we can afford running application threads concurrently with the GC process. Note that whatever conclusions we might reach, there are still scenarios that will require pausing threads during the garbage collection process. For example, if the process truly runs out of memory, it will be necessary to suspend threads while memory is being reclaimed. However, we will review less exceptional scenarios, which amount for the majority of the cases.

SUSPENDING THREADS FOR GC

Suspending threads for garbage collection is performed at *safe points*. Not every set of two instructions can be interrupted to perform a collection. The JIT emits additional information so that the suspension occurs when it's safe to perform the collection, and the CLR tries to suspend threads gracefully—it will not resort to a blatant SuspendThread Win32 API call without verifying that the thread is safe after suspension.

In CLR 2.0, it was possible to come up with a scenario where a managed thread entangled in a very tight CPU-bound loop would pass around safe points for a long time, causing delays of up to 1500 milliseconds in GC startup (which, in turn, delayed any threads that were already blocked waiting for GC to complete). This problem was fixed in CLR 4.0; if you are curious about the details, read Sasha Goldshtein's blog post, "Garbage Collection Thread Suspension Delay" (`http://blog.sashag.net/archive/2009/07/31/garbage-collection-thread-suspension-delay-250ms-multiples.aspx`, 2009).

Note that unmanaged threads are not affected by thread suspension until they return to managed code—this is taken care of by the P/Invoke transition stub.

Pausing Threads during the Mark Phase

During the mark phase, the garbage collector's work is almost read-only. False negatives and false positives can occur nonetheless.

A newly created object can be considered dead by the collector even though it is referenced by the application. This is possible if the collector has already considered the part of the graph that is updated when the object is created (see Figure 4-7). This can be addressed by intercepting the creation of new references (and new objects) and making sure to mark them. It requires synchronization and increases the allocation cost, but allows other threads to execute concurrently with the collection process.

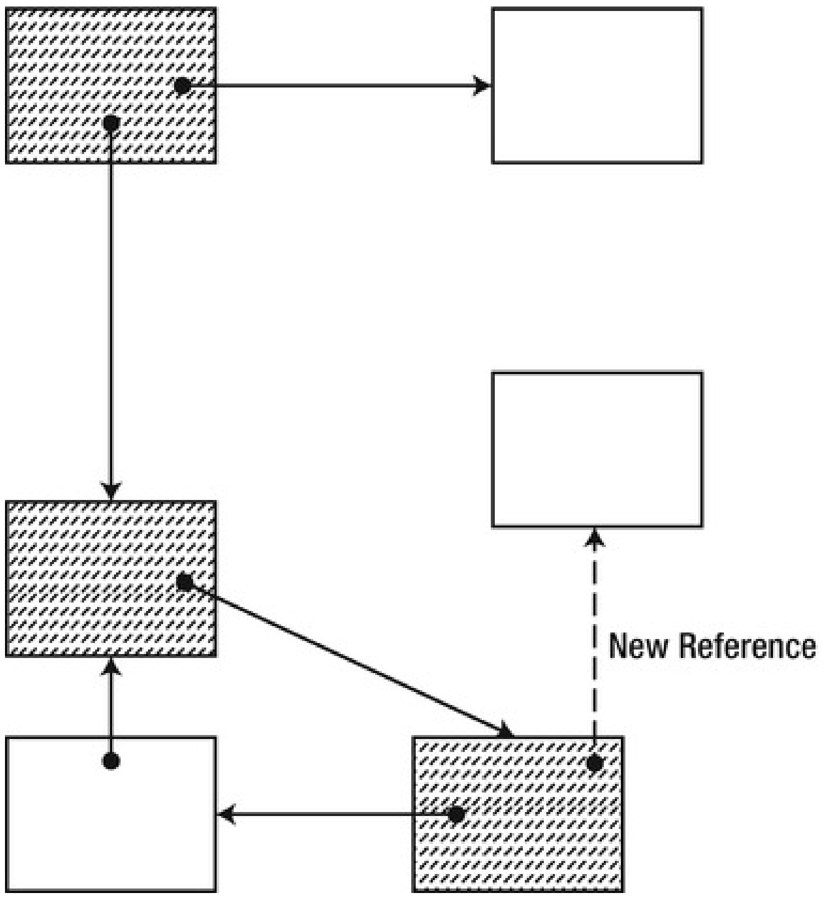

Figure 4-7 . An object is introduced into the graph after that part of the graph was already marked (dashed objects were already marked). This causes the object to be wrongly assumed unreachable

An object that was already marked by the collector can be eligible for garbage collection if during the mark phase the last reference to it was removed (see Figure 4-8). This is not a severe problem that requires consideration; after all, if the object is really unreachable, it will be collected at the next GC cycle—there is no way for a dead object to become reachable again.

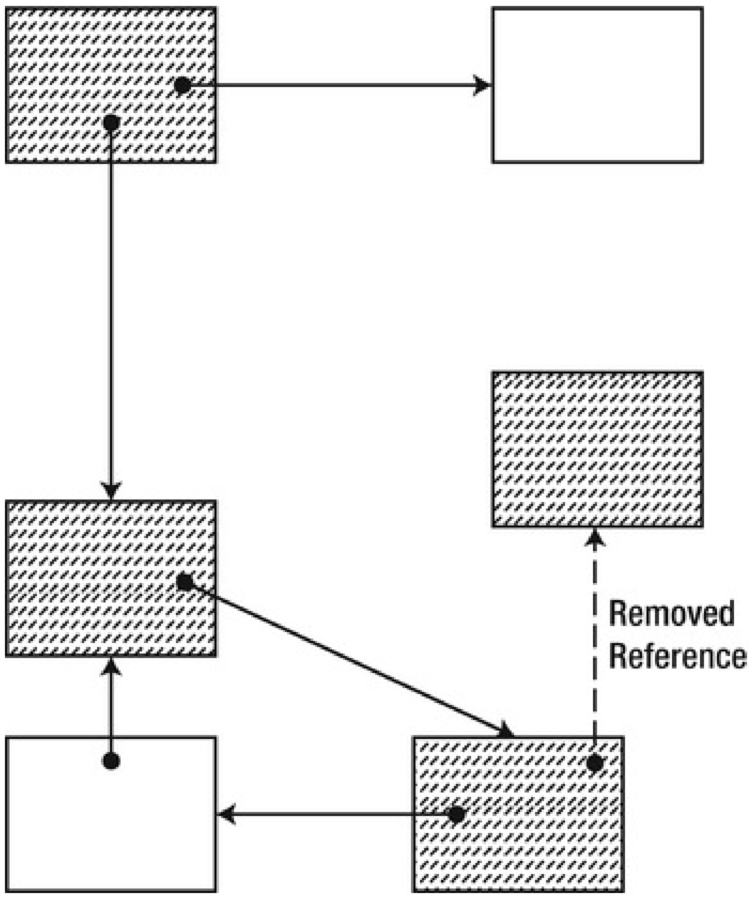

Figure 4-8 . An object is removed from the graph after that part of the graph was already marked (dashed objects were already marked). This causes the object to be wrongly assumed *reachable*

Pausing Threads during the Sweep Phase

During the sweep phase, not only do references change, but objects move around in memory. This poses a new set of problems for application threads executing concurrently. Among these problems:

- Copying an object is not an atomic operation. This means that after part of the object has been copied, the original is still being modified by the application.
- Updating references to the object is not an atomic operation. This means that some parts of the application might be using the old object reference and some other parts might be using the new one.

Addressing these problems is possible (Azul Pauseless GC for JVM, `http://www.azulsystems.com/zing/pgc`, is one example), but has not been done in the CLR GC. It is simpler to declare that the sweep phase does not support application threads executing concurrently with the garbage collector.

Tip To determine whether concurrent GC can provide any benefit for your application, you must first determine how much time it normally spends performing garbage collection. If your application spends 50% of its time reclaiming memory, there remains plenty of room for optimization. On the other hand, if you only perform a collection once in a few minutes, you probably should stick to whatever works for you and pursue significant optimizations elsewhere. You can find out how much time you're spending performing garbage collection through the % Time in GC performance counter in the .NET CLR Memory performance category.

Now that we've reviewed how other application threads might behave while a garbage collection is in progress, we can examine the various GC flavors in greater detail.

Workstation GC

The first GC flavor we will look into is termed the *workstation GC*. It is further divided into two sub-flavors: *concurrent workstation GC* and *non-concurrent workstation GC*.

Under workstation GC, there is a single thread that performs the garbage collection—the garbage collection does not run in parallel. Note that there's a difference between running the collection process *itself* in parallel on multiple processors, and running the collection process concurrently with other application threads.

Concurrent Workstation GC

The *concurrent workstation* GC flavor is the *default* flavor. Under concurrent workstation GC, there is a separate, dedicated *GC thread* marked with `THREAD_PRIORITY_HIGHEST` that executes the garbage collection from start to finish. Moreover, the CLR can decide that it wants some phases of the garbage collection process to run concurrently with application threads (most of the mark phase can execute concurrently, as we have seen above). Note that the decision is still up to the CLR—as we will see later, some collections are fast enough to warrant full suspension, such as generation 0 collections. One way or another, when the sweep phase is performed, all application threads are suspended.

The responsiveness benefits of using the concurrent workstation GC can be trumped if the garbage collection is triggered by the UI thread. In that case, the application's background threads will be competing with the garbage collection, for which the UI is waiting. This can actually *lower* the UI responsiveness because the UI thread is blocked until GC completes, and there are other application threads competing for resources with the GC. (See Figure 4-9.)

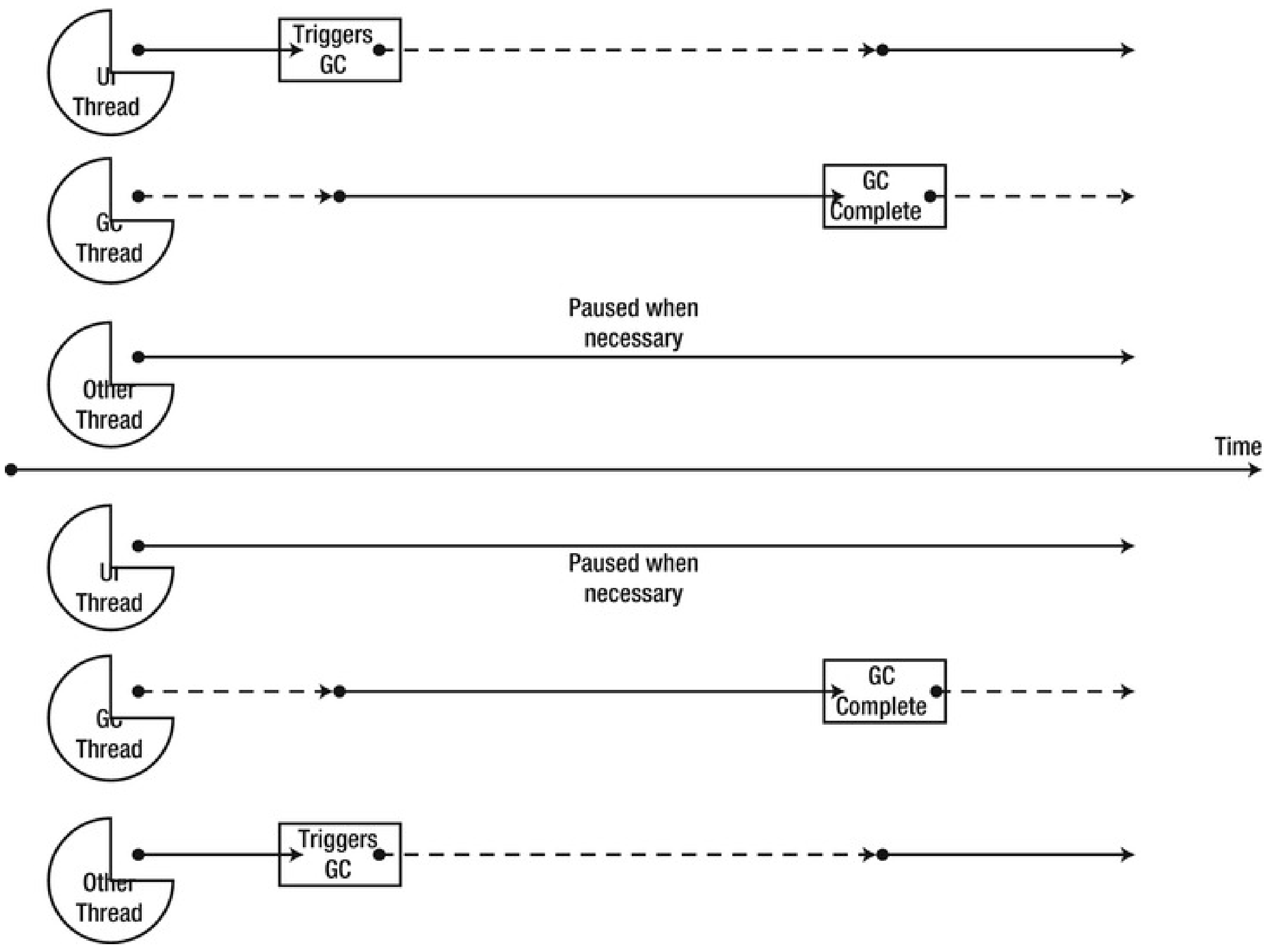

Figure 4-9 . The upper part shows concurrent GC when the UI thread triggers the collection. The lower part shows concurrent GC when one of the background threads triggers the collection. (Dashed lines represent blocked threads.)

Therefore, UI applications using concurrent workstation GC should exercise great care to prevent garbage collections from occurring on the UI thread. This boils down to performing allocations on background threads, and refraining from explicitly calling `GC.Collect` on the UI thread.

The default GC flavor for all .NET applications (save ASP.NET applications), regardless of whether they run on a user's workstation or a powerful multiprocessor server is the concurrent workstation GC. This default is rarely appropriate for server applications, as we will see shortly. As we have just seen, this is not necessarily the appropriate default for UI applications either, if they tend to trigger garbage collections on the UI thread.

Non-Concurrent Workstation GC

The *non-concurrent workstation* GC flavor, as its name implies, suspends the application threads during both the mark and sweep phases. The primary usage scenario for non-concurrent workstation GC is the case mentioned in the previous section, when the UI thread tends to trigger garbage collections. In this case, non-concurrent GC might provide better responsiveness, because background threads won't be competing with the garbage collection for which the UI thread is waiting, thus releasing the UI thread more quickly. (See Figure 4-10.)

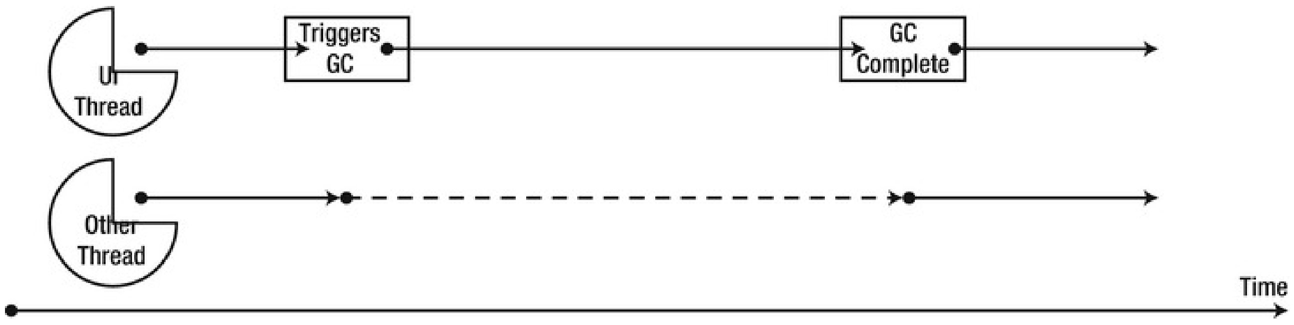

Figure 4-10 . The UI thread triggers a collection under non-concurrent GC. The other threads do not compete for resources during the collection

Server GC

The *server* GC flavor is optimized for a completely different type of applications—server applications, as its name implies. Server applications in this context require high-throughput scenarios (often at the expense of latency for an individual operation). Server applications also require easy scaling to multiple processors – and memory management must be able to scale to multiple processors just as well.

An application that uses the server GC has the following characteristics:

- There is a separate managed heap for each processor in the affinity mask of the .NET process. Allocation requests by a thread on a specific processor are satisfied from the managed heap that belongs to that specific processor. The purpose of this separation is to minimize contention on the managed heap while allocations are made: most of the time, there is no contention on the next object pointer and multiple threads can perform allocations truly in parallel. This architecture requires dynamic adjustment of the heap sizes and GC thresholds to ensure fairness if the application creates manual worker threads and assigns them hard CPU affinity. In the case of typical server applications, which service requests off a thread pool worker thread, it is likely that all heaps will have approximately the same size.

- The garbage collection does not occur on the thread that triggered garbage collection. Instead, garbage collection occurs on a set of dedicated GC threads that are created during application startup and are marked `THREAD_PRIORITY_HIGHEST`. There is a GC thread for each processor that is in the affinity mask of the .NET process. This allows each thread to perform garbage collection in parallel on the managed heap assigned to its processor. Thanks to locality of reference, it is likely that each GC thread should perform the mark phase almost exclusively within its own heap, parts of which are guaranteed to be in that CPU's cache.
- During both phases of garbage collection, all application threads are suspended. This allows GC to complete in a timely fashion and allows application threads to continue processing requests as soon as possible. It maximizes throughput at the expense of latency: some requests might take longer to process while a garbage collection is in progress, but overall the application can process more requests because less context switching is introduced while GC is in progress.

When using server GC, the CLR attempts to balance object allocations across the processor heaps. Up to CLR 4.0, only the small object heap was balanced; as of CLR 4.5, the large object heap (discussed later) is balanced as well. As a result, the allocation rate, fill rate, and garbage collection frequency are kept similar for all heaps.

The only limitation imposed on using the server GC flavor is the number of physical processors on the machine. If there is just one physical processor on the machine, the only available GC flavor is the workstation GC. This is a reasonable choice, because if there is just a single processor available, there will be a single managed heap and a single GC thread, which hinders the effectiveness of the server GC architecture.

Note Starting from NT 6.1 (Windows 7 and Windows Server 2008 R2), Windows supports more than 64 logical processors by using *processor groups*. As of CLR 4.5, the GC can use more than 64 logical processors as well. This requires placing the `< GCCpuGroup enabled = "true" />` element in your application configuration file.

Server applications are likely to benefit from the server GC flavor. However, as we have seen before, the default GC flavor is the workstation concurrent GC. This is true for applications hosted under the default CLR host, in console applications, Windows applications and Windows services. Non-default CLR hosts can opt-in to a different GC flavor. This is what the IIS ASP.NET host does: it runs applications under the server GC flavor, because it's typical for IIS to be installed on a server machine (even though this behavior can still be customized through Web.config).

Controlling the GC flavor is the subject of the next section. It is an interesting experiment in performance testing, especially for memory-intensive applications. It's a good idea to test the behavior of such applications under the various GC flavors to see which one results in optimal performance under heavy memory load.

Switching Between GC Flavors

It is possible to control the GC flavor with CLR Hosting interfaces, discussed later in this chapter. However, for the default host, it is also possible to control the GC flavor selection using an application configuration file (App.config). The following XML application configuration file can be used to choose between the various GC flavors and sub-flavors:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <gcServer enabled="true" />
    <gcConcurrent enabled="false" />
  </runtime>
</configuration>
```

The `gcServer` element controls the selection of server GC as opposed to workstation GC. The `gcConcurrent` element controls the selection of the workstation GC sub-flavor.

.NET 3.5 (including .NET 2.0 SP1 and .NET 3.0 SP1) added an additional API that can change the GC flavor at runtime. It is available as the `System.Runtime.GCSettings` class, which has two properties: IsServerGC and LatencyMode.

`GCSettings.IsServerGC` is a read-only property that specifies whether the application is running under the server GC. It can't be used to opt into server GC at runtime, and reflects only the state of the application's configuration or the CLR host's GC flavor definition.

The LatencyMode property, on the other hand, takes the values of the `GCLatencyMode` enumeration, which are: Batch, Interactive, LowLatency, and SustainedLowLatency. Batch corresponds to non-concurrent GC; Interactive corresponds to concurrent GC. The `LatencyMode` property can be used to switch between concurrent and non-concurrent GC at runtime.

The final, most interesting values of the GCLatencyMode enumeration are `LowLatency` and `SustainedLowLatency`. These values signal to the garbage collector that your code is currently in the middle of a time-sensitive operation where a garbage collection might be harmful. The LowLatency value was introduced in .NET 3.5, was supported only on concurrent workstation GC, and is designed for short time-sensitive regions. On the other hand, `SustainedLowLatency` was introduced in CLR 4.5, is supported on both server and workstation GC, and is designed for longer periods of time during which your application should not be paused for a full garbage collection. Low latency is not for the scenarios when you're about to execute missile-guiding code for reasons to be seen shortly. It is useful, however, if you're in the middle of performing a UI animation, and garbage collection will be disruptive for the user experience.

The low latency garbage collection mode instructs the garbage collector to refrain from performing full collections unless absolutely necessary—e.g., if the operating system is running low on physical memory (the effects of paging could be even worse than the effects of performing a full collection). Low latency does not mean the garbage collector is off; partial collections (which we will consider when discussing generations) will still be performed, but the garbage collector's share of the application's processing time will be significantly lower.

USING LOW LATENCY GC SAFELY

The only safe way of using the low latency GC mode is within a constrained execution region (CER). A CER delimits a section of code in which the CLR is constrained from throwing out-of-band exceptions (such as thread aborts) which would prevent the section of code from executing in its entirety. Code placed in a CER must call only code with strong reliability guarantees. Using a CER is the only way of guaranteeing that the latency mode will revert to its previous value. The following code demonstrates how this can be accomplished (you should import the `System.Runtime.CompilerServices` and `System.Runtime` namespaces to compile this code):

```
GCLatencyMode oldMode = GCSettings.LatencyMode;
RuntimeHelpers.PrepareConstrainedRegions();
try
{
    GCSettings.LatencyMode = GCLatencyMode.LowLatency;
    //Perform time-sensitive work here
}
finally
{
    GCSettings.LatencyMode = oldMode;
}
```

The amount of time you want to spend with a low latency GC mode must be kept to a minimum—the long-term effects once you exit the low latency mode and the GC aggressively begins to reclaim unused memory can hinder the application's performance. If you don't have full control of all allocations taking place within your process (e.g. if you're hosting plug-ins or have multiple background threads doing independent work), remember that switching to the low latency GC mode affects the entire process, and can cause undesired effects for other allocation paths.

Choosing the right GC flavor is not a trivial task, and most of the time we can only arrive at the appropriate mode through experimentation. However, for memory-intensive applications this experimentation is a must—we can't afford spending 50% of our precious CPU time performing garbage collections on a single processor while 15 others happily sit idle and wait for the collection to complete.

Some severe performance problems still plague us as we carefully examine the model laid out above. The following are the most significant problems:

- *Large Objects*: Copying large objects is an extremely expensive operation, and yet during the sweep phase large objects can be copied around all the time. In certain cases, copying memory might become the primary cost of performing a garbage collection. We arrive at the conclusion that objects must be differentiated by size during the sweep phase.
- *Collection Efficiency Factor*: Every collection is a full collection, which means that an application with a relatively stable set of objects will pay the heavy price of performing mark and sweep across the entire heap even though most of the objects are still referenced. To prevent a low collection efficiency factor, we must pursue an optimization which can differentiate objects by their *collection likelihood*: whether they are likely to be collected at the next GC cycle.

Most aspects of these problems can be addressed with the proper use of generations, which is the topic we cover in the next section. We will also touch on some additional performance issues that you need to consider when interacting with the .NET garbage collector.