WHITE PAPER

Optimizing memory in .NET applications

Deepen your understanding of how a .NET application uses memory, and what an application developer can do to improve memory management for better performance and more reliable applications.

The Microsoft .NET Framework and Common Language Runtime (CLR) mark a significant change in how developers build applications targeting the Windows platform. In years past, developers directly manipulated system memory, allocating, initializing, moving data around and freeing blocks of memory by address pointers. This practice tended to result in fast programs, but unfortunately introduced a wide range of highly detailed and difficult-to-debug errors.

Developers can free themselves from the error-prone tedium of managing an application's memory by using the features of the .NET Framework to do it automatically. The .NET Framework allocates memory on demand, and reclaims memory once the application is done with it. Developers can focus on solving business problems and leave memory management details to the Framework.

But nothing comes for free in writing applications. It's time-consuming to identify memory that is no longer needed, collect that memory and return it to the free memory heap. Applications that use memory poorly add to the problem, forcing the system to work harder and more often to reclaim memory. Over time, poor application memory management can also result in subtle, difficult-to-find errors that slow application performance while reducing scalability and reliability.

These types of memory problems are new and unfamiliar to most developers. In many cases, developers simply lack the experience and understanding to know when an application has a memory problem, and what if anything they can do about it. They assume that they have no control over how memory is used, allocated, and deallocated, and pay no attention to how design and implementation decisions impact memory usage.

But there are solutions, and it's simply a matter of learning where the problems can manifest themselves and why they occur. Even in a managed world, developers can help avoid pitfalls and improve the performance of their

applications, by understanding how .NET memory management works and how their applications use memory. Developer tools that assist in this understanding, and enable developers to make informed decisions on architecting and building their applications, are an essential part of creating fast and reliable .NET applications.

Working with the garbage collector

As most developers know by now, the mechanism by which the CLR reclaims memory from applications is called the garbage collector. The .NET garbage collector works by starting with roots, base object references that are not embedded inside another object. For example, static and global pointers are application roots. The garbage collector starts with these roots, and traces references to other objects on the heap. It creates a graph of all objects that are reachable from these root components.

Any object that is not in this graph is considered to be no longer in use and is added back to the heap. The garbage collector accomplishes this by walking through the heap and identifying objects that are not a part of one of these graphs. The garbage collector marks these addresses and keeps track of them until it has walked through the entire heap or some defined portion of it.

During this process, the garbage collector also compacts the heap so that fragmentation doesn't prevent an allocation due to the lack of a large enough memory block. Additionally, this compaction leaves free memory at the top of the heap, where it can be reallocated simply by moving the heap pointer. The garbage collector doesn't have to walk a linked list to find memory blocks, so allocations are fast compared to unmanaged languages.

Compaction involves relocating memory blocks down to the bottom of the heap using the memcopy function, then adjusting all of the pointers from root structures so that they refer to the new addresses. The garbage collector also must change any pointers from other objects in the application to reflect the new addresses.

For efficiency reasons, the garbage collector also uses a concept called generations in reclaiming memory. There are a total of three generations, labeled 0, 1, and 2. When objects are first allocated at the beginning of application execution, the garbage collector refers to this part of the heap as generation 0. Newly created objects always go into generation 0. These "young" objects have not yet been examined by the garbage collector.

The garbage collector checks this generation first, so it can reclaim more memory, and more quickly, than if it treated all heap memory the same. If there are references to the object when the next collection occurs, the object gets moved to generation 1. In this way, only a part of the memory heap—

the part with the greatest potential to yield free memory—needs to be checked at any one time, a strategy that can improve the performance of individual collections.

When more objects are added to the heap, the heap fills and a garbage collection must occur. When the garbage collector analyzes the heap, it builds the graph of live objects, and collects the rest. The objects that have survived a collection are now older and are considered to be in generation 1. The garbage collector maintains a table of objects and when they were accessed to identify which objects haven't been modified and therefore are eligible for garbage collection.

As even more objects are added to the heap, these new, young objects are placed in generation 0. When the time comes for the next garbage collection, the collector determines which old objects have been written to since the last collection. The garbage collector checks the references to these specific old objects to see if they refer to any new objects. If a root or object refers to an object in an old generation, the garbage collector can ignore any of the older objects' inner references, decreasing the time required to build the graph of reachable objects. Collecting newer objects first can also reduce page faulting and improve performance, because newer objects are stored contiguously in the heap.

Likewise, if an object survives a generation 1 garbage collection, it is promoted to generation 2. When a collection occurs, the three generations of heap memory—generations 0, 1, and 2—are checked in succession. If checking generation 0 reclaims enough memory, garbage collection ceases. If not, the garbage collector then checks generation 1, and finally generation 2. In practice, generation 2 objects are long-lived, and are often not collected until the application finishes and exits.

You can also work with the garbage collector to optimize memory use through weak references. Unlike strong references, the .NET Framework can garbage-collect weak references if memory is low. First, you establish a strong reference by subclassing and initializing an object, then apply a weak reference through the appropriate Framework call. If memory use remains low, the weak reference is sufficient to prevent that object from being garbage-collected. If memory becomes scarce, the garbage collector can dispose of the object and reclaim the memory.

When might you use a weak reference? One example is if you create a structure that is useful for efficiency reasons, such as a search tree. The first time it's used, you have to take the performance hit to create the tree. After that, you might want to keep it around in case the application uses it again, but not at the expense of poor performance elsewhere in the application. So with the weak reference, the .NET Framework offers the option of designating a structure that is, in effect, the first to go if memory runs short.

Garbage collection and optimization

As you might imagine, garbage collection itself is a computationally expensive process. It has the advantage of significantly improving the speed of memory allocations over unmanaged languages, because it simply allocates a block at the top of the heap and moves the heap pointer to the next free memory address. However, the garbage collection process potentially rescinds that advantage during memory reclamation. Tracing object references, then compacting memory, takes significantly more time than manually freeing memory back onto the heap.

It gets worse. Multi-threaded applications add to the complexity of garbage collection. When the garbage collector starts reclaiming memory, it both gathers free objects and moves pointers on the heap. When memory requests by one thread initiate a garbage collection, the other threads can't access any other object. In effect, the entire application stops while garbage collection is occurring.

The garbage collector uses a few different mechanisms to suspend threads safely so it can perform a collection. The reason for the multiple mechanisms is to keep threads running as long as possible and to reduce overhead as much as possible. Microsoft has implemented these measures that enable the Framework to improve thread-execution efficiency in order to minimize downtime due to garbage collection and improve performance, but this still represents a complete stop of the application.

This doesn't mean, however, that you should avoid managed code to achieve high performance. The performance improvements in memory allocations, coupled with the elimination of traditional native code memory management errors, are more than enough reason to take advantage of managed applications. But it's essential to use the .NET Framework with a good understanding of how the garbage collector works and how you can use memory management strategies to improve the performance of your applications.

As you might imagine, the automatic memory management used by the .NET Framework and CLR dramatically changes the art of application development. In the past, application development had involved processing data by moving it among different memory locations by manipulating pointers to that data in memory. Today, application development means processing data by creating and customizing objects and using methods to act on the data represented by those objects. The act of manipulating that data in memory, though very real, is indirect and in many ways hidden from the developer.

This means that developers still have to worry about memory management. But the rules have changed. Rather than concentrating on the tactical mechanics of allocating, initializing, casting and freeing memory blocks of specific size and location, developers can focus on overarching strategies for using memory management to improve application performance and reliability.

Memory management gone bad

The activities of the garbage collector seem relatively straightforward and well thought-out. Details will likely change in subsequent releases of the .NET Framework, based on experience gained by Microsoft in the behaviors of actual applications, along with incremental improvements in technology.

But many problems can arise in the details. Minor implementation changes in the same application can result in substantial performance differences. A few seemingly innocuous constructs can greatly slow down an application, or cause it to "leak" objects (keep them around after they are no longer in use). Following are a few of the more significant issues brought about by the garbage collector's strategy that could negatively impact an application's execution.

Too many objects

One of the most common problems to experience is creating too many objects. Because allocating new memory with the .NET Framework is quite fast, it's easy to forget that a single line of code could trigger a lot of allocations. The problem occurs when it comes time to collect these objects. Garbage collection involves a performance penalty, and collecting a large number of unnecessary objects exacerbates the problem.

This problem typically occurs when instructions generated from code create a class of objects known as temporary objects to perform their actions. Many .NET classes create temporary objects for their return values, for temporary strings and for associated classes such as enumerators that serve a necessary but short-lived purpose. An application developer can't simply use any instruction to perform a particular action, because that construct might produce undesirable side effects.

As a simple example, consider an exercise to concatenate two strings. It might seem simple to apply the "+" operator to perform this action. However, the "+" operator causes several new string objects to be created every time text is added to the string. Instead, using the System.Text.StringBuilder class often promotes faster string concatenation without creating new objects. This type of problem can be even worse in cases where a single instruction can create many temporary objects, all of which must be garbage-collected when their work is completed.

Object leaks

Of course, even with the .NET garbage collection strategy, you can still have object leaks. An object leak occurs when a reference is made accidentally, or not removed appropriately, resulting in the object getting written to when the application is done with it. If this object is still in some way connected to a root structure, it won't be collected, even if the application is done with that object. An example of such a leak is caching an object reference in a static member variable and forgetting to release it after the end of a request. The memory reference will remain until the application completes and the heap is returned to the operating system.

This also leads to the issue of inappropriately long-lived objects. Because garbage collection is automatic, it's easy to forget that memory is still managed according to predefined rules. If an object is kept around long enough to be promoted to generation 2 of collection, it might never be collected until the application exits.

Why is this bad? Because the number of objects stored in the heap will likely keep growing while the application is running. This causes two problems. First, more heap memory extends the amount of time required to garbage-collect, slowing down the application. Second, memory is not an infinite resource. If the application runs long enough, it will generate an out-of-memory error.

Too many object references

If you create an object that refers to many other objects, it can cause a couple of different problems. First, during all collections, it will force the garbage collector to follow all of the pointers between the objects, lengthening the time needed to complete the process. The results are particularly bad if this is a long-lived object structure, because the garbage collector goes through this process for every collection if the object has been modified.

Plus, large objects can cause problems. The .NET Framework keeps large objects (more than 20,000 bytes) in a separate large object heap, so that it can manage them separately from other objects. Large objects are never compacted because shifting 20,000-byte blocks of memory down in the heap would waste too much CPU time. Yet pointers between large objects and other objects can keep large objects alive longer than they need to be. And objects that are close to the 20,000-byte cutoff will still be allocated in the general-purpose heap, and compacting them will incur a significant performance hit.

Strategies for memory management

All of this means that managing memory in the .NET Framework requires a deep level of understanding not only of your application, but also of how the Framework performs its actions. And even then it's not possible to make the

one best decision in all circumstances. Instead, application development becomes a matter of continuously weighing strategies for implementing features, balancing factors such as efficiency, ease of implementation and maintainability.

What kinds of strategies are available to help developers manage memory for more efficient applications? At the simplest level, you can force a garbage collection in your application by calling the Collect method of the System.GC object class. This provides you with the primary say as to when your application takes this particular performance hit.

This is a simple strategy, but it's far from the best you can employ. According to MSDN, "you should avoid calling any of the collect methods" because doing so might produce unexpected side effects. Your GC.Collect call, for example, might actually execute during a critical time in the application, making already-slow code even slower. Without a significant amount of experimentation and load testing, it's difficult to tell the appropriate time to invoke a garbage collection.

That doesn't mean you shouldn't invoke the garbage collector at all. But use care, by ensuring that code is executing only a single thread when invoking garbage collection, and that the code isn't actively processing managed instructions while garbage collection occurs.

Nevertheless, unless you understand what all of your application threads are doing when you call for a System.GC, it is usually best to let the system perform this task. The impact of getting it wrong is more significant than the benefit of getting it right.

The best strategy is two-fold—to understand how the .NET Framework manages memory, and to obtain a precise picture of how your application uses memory. You can then apply both types of information to design, implement, and modify your application to optimize memory use.

Applying memory analysis

The problem is that you need information on how memory is being used, and how memory usage changes as you make changes to your code. The .NET Framework provides some information that should help you write more efficient code. Much of this information is contained within Perfmon counters, which you can examine while your application is running to get an overall picture of how memory is managed and what effect it has.

The Perfmon counters that are useful for evaluating memory management include the percent of time spent in the garbage collector, the generational heap sizes and bytes promoted between memory generations, and the large object heap size. These can help you spot trends in your code, such as too many large objects or too many bytes promoted to higher generations.

But the Perfmon counters by themselves are inadequate. One problem is that they aren't necessarily application-specific. In other words, while you can select the instances you want to see counters on, those instances might not clearly correspond to applications, processes, or threads you want to see. And you can't start and stop Perfmon memory counters to view specific activities or to do comparisons among those activities.

Perfmon counters also lack the level of detail you need to analyze memory use and make decisions. They don't give you any indication as to why the application is spending so much time using the garbage collector, for example, and they don't tell you which objects are temporary and which are long-lived. The information you get is primarily summary data, and that doesn't enable you to identify individual objects and the memory associated with them.

Instead, what is needed to examine .NET Framework memory accurately is an interactive, real-time memory analysis capability that can track individual objects in memory over time. One such product, Compuware DevPartner Studio, incorporates memory analysis on the .NET Framework that enables developers to investigate potential and actual memory problems, obtain detailed information on object behaviors and their effects on memory, and determine strategies for using memory efficiently in managed applications.

DevPartner Studio provides three fundamental views of .NET memory—RAM (memory) footprint, temporary objects and memory leaks. You can take snapshots of all these views, in order to examine the state of memory at an instant of your choosing. It also lets you force a garbage collection, so that you can observe the effects of memory reclamation as well as determine if an application has an object leak.

Taking a RAM footprint snapshot shows you who allocated the memory, what objects it comprises and which components are holding references to it, thus preventing it from being freed. In the case of the sample application shown in Figure 1, the snapshot shows that String objects are using by far the most memory. This information might prompt you to revisit your design and implementation decisions to reduce the use of String objects.

The RAM footprint can provide still more information. You might, for example, observe that your application uses more and more memory over time while running. But from simply watching the amount of memory in use by the system, you can't tell what objects are being leaked. To find this out, you must be able to start and stop memory analysis to take snapshots of the memory state at any particular time.

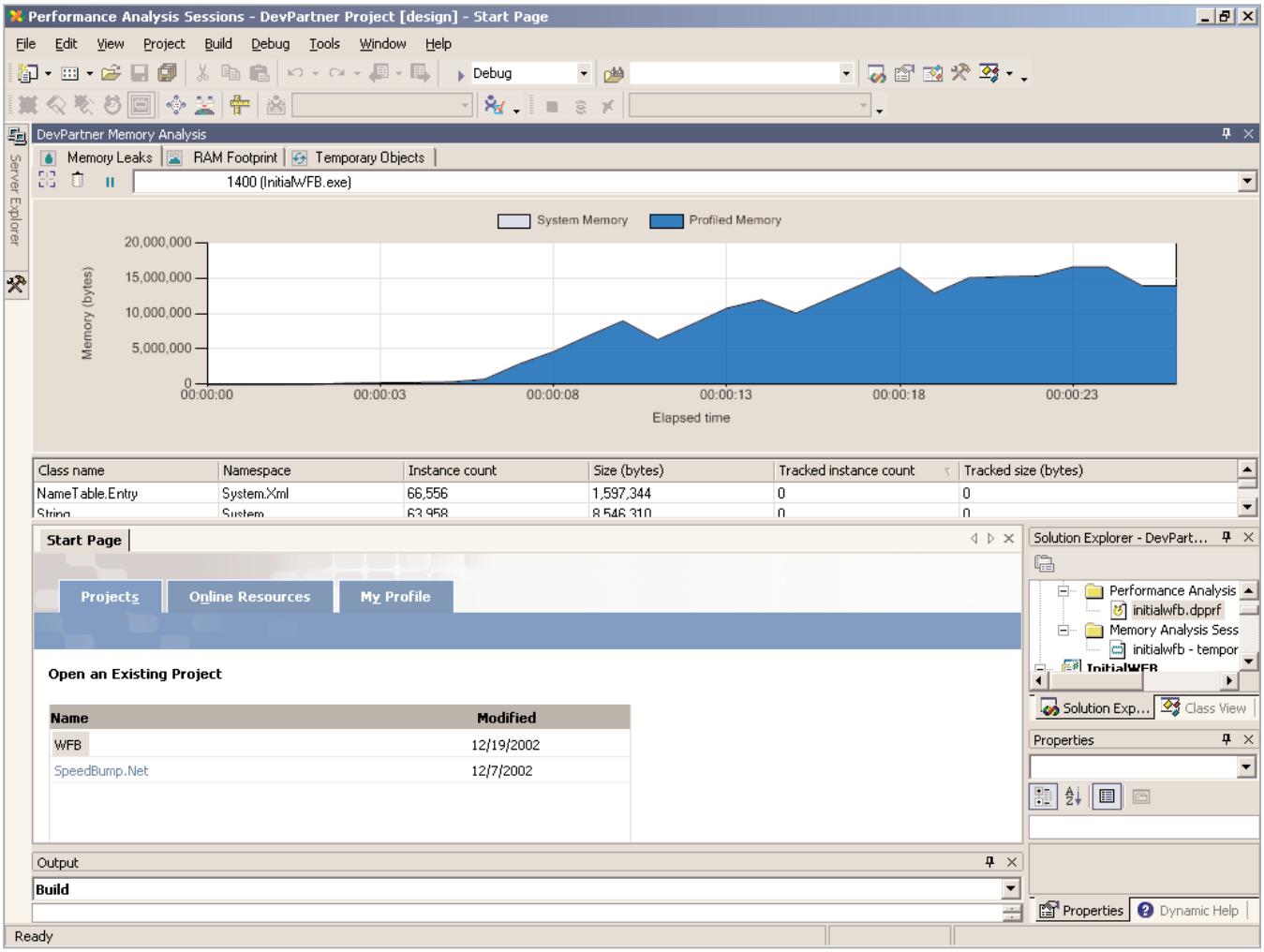

Figure 1. The RAM footprint shows dynamically how much memory is allocated and by whom. This figure shows the working set being allocated at application launch.

You can use analysis of temporary objects—the second fundamental view of .NET memory—to look for unusual or inefficient behavior that creates large numbers of temporary objects or large-sized temporary objects. These problems tend to be easy to fix, in that they typically require changing the construct that is creating the temporary objects, or changing the times those objects are being created.

DevPartner Studio lets you see the objects that allocate the most memory, along with the methods that use the most memory (see Figure 2). Further, you can drill down to examine the methods, how many times they are called, whom they are calling and who is calling them (see Figure 3). The call graph provides a visual display of this information that enables you to see at a glance how these calls occur. This provides you with a precise path on when and why these methods are called.

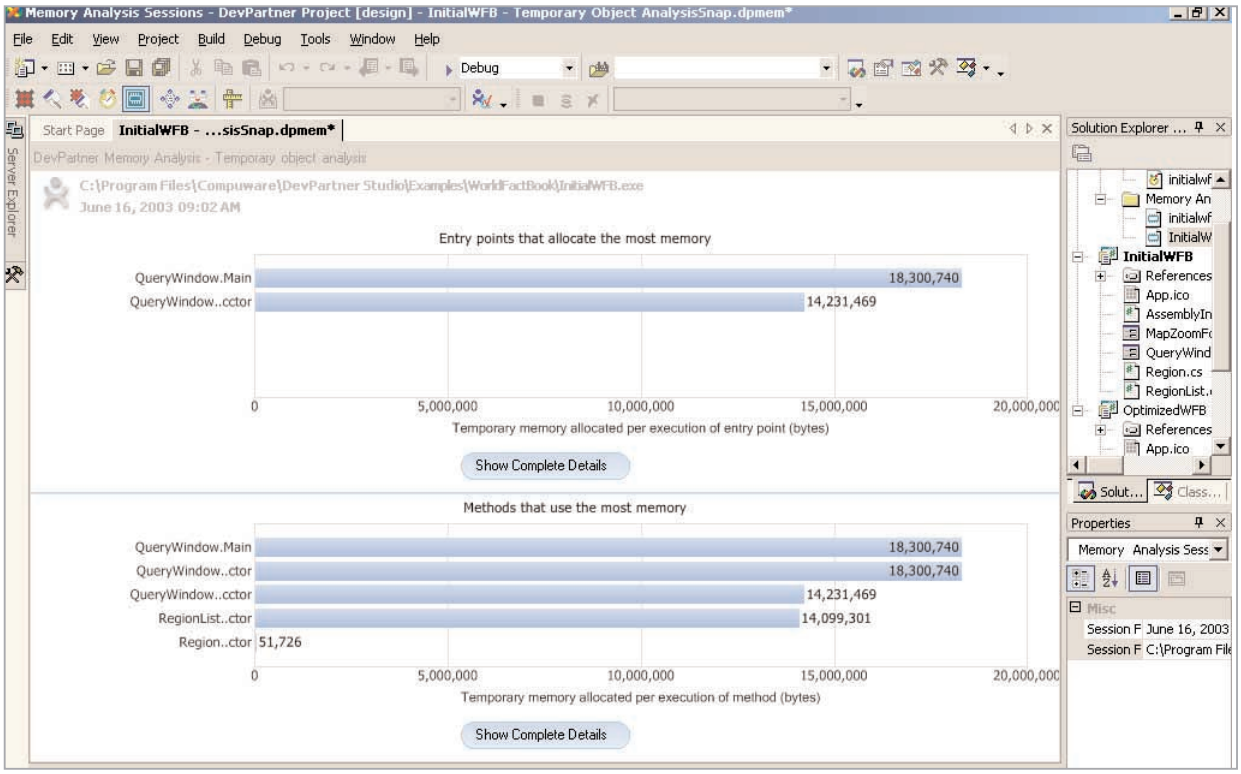

Figure 2.
The high-level temporary object view enables you to see at a glance which objects you should focus on to reduce memory use.

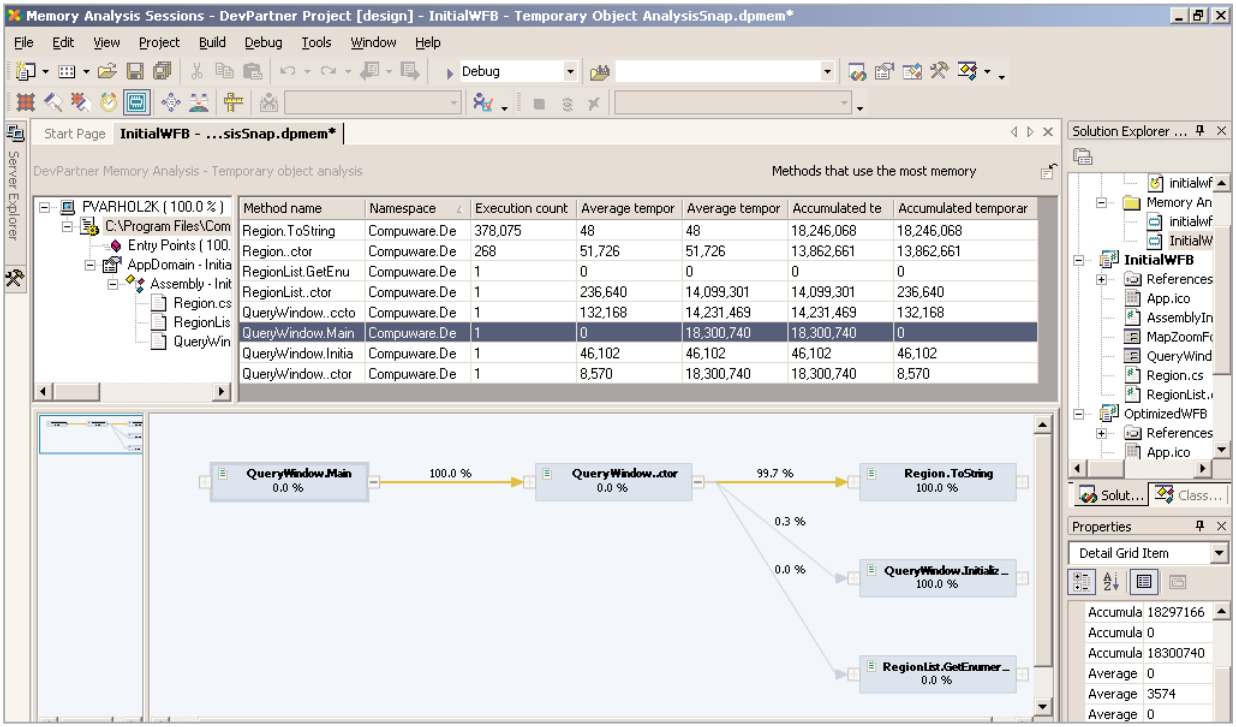

Figure 3.
The detailed temporary object view lets you see how objects and methods are being called, and whom they are calling.

The last important view of .NET memory is that of memory leaks. Objects can leak memory because references aren't released promptly—or aren't released at all—and their effects can lead to poor performance and even application failures.

When the memory leaks snapshot loads, you can examine where the leaking objects were allocated and find out which objects still hold references to them, thus preventing them from being garbage-collected. DevPartner Studio tracks memory allocations among objects to determine which are not releasing their instances over time (see Figure 4). You can use this information to determine which objects are leaking memory from your application and when.

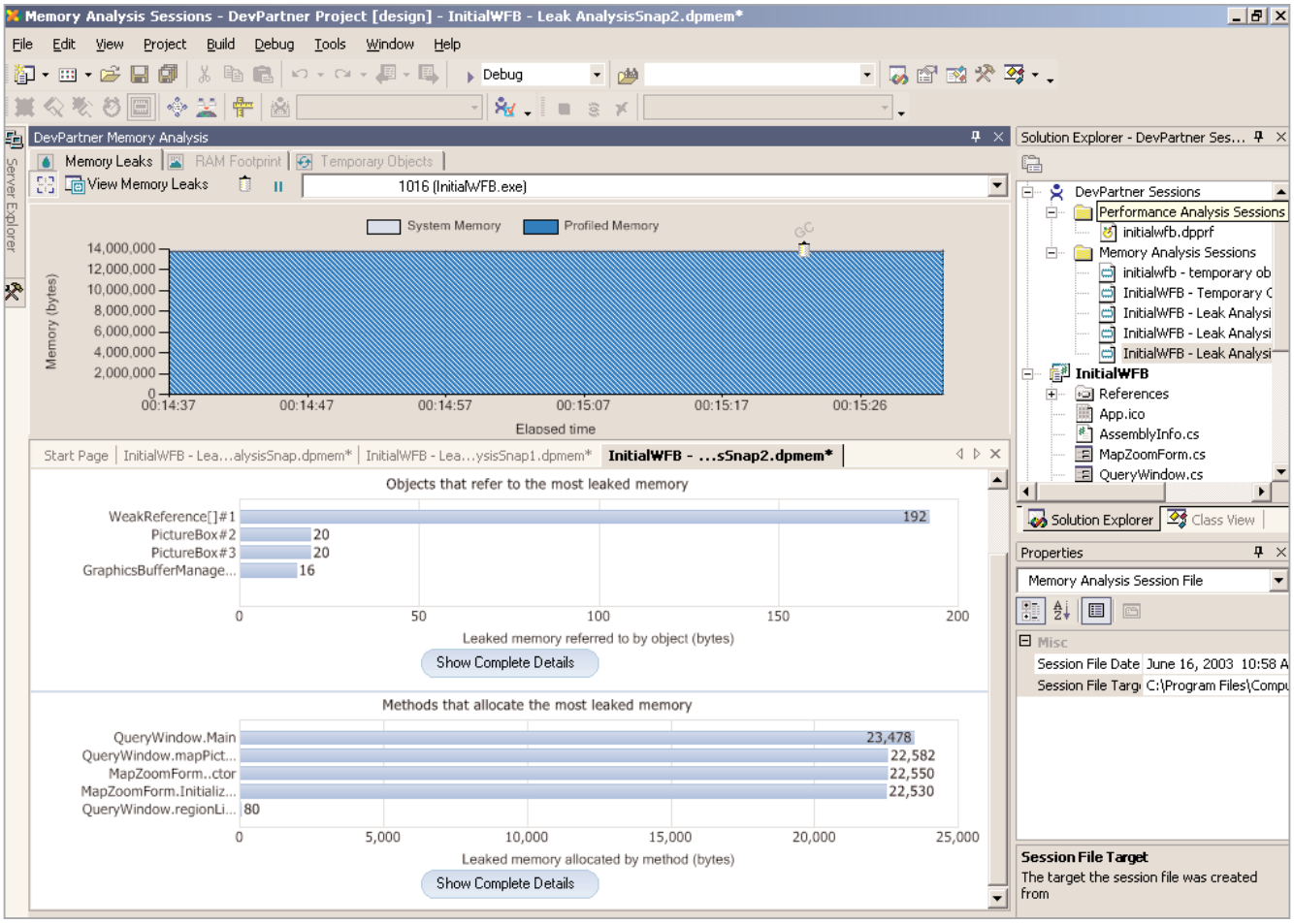

Figure 4. The memory leaks view shows objects being allocated, along with the number of instances created and released, letting you track which objects are not releasing instances.

Compuware products and professional services—delivering quality applications

Compuware is a leading global provider of software products and professional services which IT organizations use to develop, integrate, test and manage the performance of the applications that drive their businesses. Our software products help optimize every step in the application life cycle—from defining requirements to supporting production service levels—for web, distributed and mainframe platforms. Our services professionals work at customer sites around the world, sharing their real-world perspective and experience to deliver an integrated, reliable solution.

Please contact us to learn more about how our comprehensive products and services can help your organization improve productivity, create higher quality applications and ensure performance in production.

Making the most of .NET memory

Moving to .NET doesn't mean you don't have memory management issues. The problem is that these types of issues are unfamiliar to most developers, consequently making .NET development more difficult and error-prone than its memory management model suggests. Until developers can apply the principles of .NET memory management to their advantage, applications have a greater potential for poor performance, lack of scalability and memory errors.

Part of the solution, as discussed earlier, is for developers to gain a comprehensive understanding of how the .NET Framework manages memory. You can use this type of information to apply development strategies that help you optimize the use of memory in your applications.

But understanding alone is not enough. A strategy that works well for one application might not apply to others. It's critically important to understand how memory is used in individual applications, both at the summary level and for individual objects. It's also essential to study memory usage and garbage collection over time, dynamically viewing the changes in memory use and the effects of garbage collection at different times.

DevPartner Studio memory analysis provides one of the most comprehensive ways available for analyzing .NET memory. Thanks to its multiple views of memory use at the summary level, the ability to watch dynamic changes in memory over time, the ability to drill down into more detailed views of individual objects, and the availability of call graphs to analyze the relationships between objects, DevPartner Studio is an essential tool for developing fast and reliable .NET applications.

COMPUWARE®

www.compuware.com

6/03