Strategies for Memory Profiling

By Chris Farrell, memory management expert and trainer

OK, so your application has a memory problem. The first thing to remember is: don't panic, because it's definitely fixable. Having spent the last few years visiting customer sites to find and fix .NET memory and performance issues, I quickly discovered that all you need to be effective is a great tool and a clear strategy.

Two of the biggest memory problems I typically come across are memory leaks and excessive memory usage. Unsurprisingly, memory leaks get most attention, but an application that uses excessive memory can cause big problems too.

In this article, I'm going to talk about both issues, and how you can use ANTS Memory Profiler™ from Red Gate to resolve them. I'll look at:

- Why memory leaks matter
- Why excessive memory usage matters
- How to find a memory leak
- How to check an application's memory usage

Why should I care about memory leaks?

Memory leaks are feared by developers because the assumption is that they can't happen in .NET; and then, when they do, that they are impossible to fix. Neither of these is true; you just need to know how to approach fixing memory leaks.

The profile of a leaking application is typically:

- Memory usage slowly increases over time
- Performance degrades
- Application will freeze/crash requiring a restart
- After restart, the application runs without problems again, and the cycle repeats.

Memory leaks occur when a reference is retained to an object of which instances are created frequently. Finding a memory leak is simply a matter of finding the object that is being retained, and then determining which reference in your code is causing it to be retained.

The basic strategy I describe in this article will help you find memory leaks. You can also use the technique on a more regular basis as a validation check, to ensure completed units of code are leak-free. Being able to identify when code isn't leaking is even more useful – because, most of the time, that's the case.
Why should I care about excessive memory usage and fragmentation?

Reducing the overall memory footprint can help an application co-exist with other applications on the desktop or server.

It's always worth checking your application is using memory efficiently and caching large objects only when necessary. I once had to analyze an ASP.NET application which kept failing, putting the developers under a lot of pressure. Memory analysis identified the problem as a simple session-state caching issue. Whenever a user logged on to the system, the application retrieved a large data set from the database, containing security information for the user, which was placed in session state. The session-state Timeout was also raised to 90 minutes. As this was an application hit by thousands of users an hour, the server couldn't cope. The worst thing about this memory leak was that the data set wasn't actually used again after it was cached.

The reason I've described this example is not because I think storing state is wrong; it's because I think you should always ensure that the state you are maintaining is valid and necessary, especially if it is significantly large. Checking your application's memory footprint will highlight issues you don't realize are there.

Typical symptoms of an application with an excessive memory footprint include:

- Application is slow to load and runs slowly
- When the application is loaded other applications run slowly.

In this article, I show how to use ANTS Memory Profiler to find the unnecessary memory bloat in an application. We will also explore how allocating objects of > 85 KB can cause large object heap fragmentation and lead to an out-of-memory exception, when it seems there is plenty of memory available on the heap.

Finding a memory leak

A well-behaved application transaction (e.g. UI sequence or server process call) should perform its actions and then clean up after itself. The net effect on the running application should ideally be zero after completion.

Memory-leak detection is all about recording the application "at rest" and then again after a test transaction is executed. Objects left behind should be investigated further and, if new objects of the same type are created after every iteration, it should raise suspicions that there's a memory leak.

If you suspect your application has a leak, you need to identify the most likely application transactions (use-cases, processes) that may be responsible. Next, run each transaction a number of times, taking a memory snapshot with ANTS Memory Profiler at the end of each test. By comparing snapshots and cleverly using filters to eliminate the "application noise," you'll be able to focus in on the memory leak.

When you are trying to find a memory leak it's crucial that you have a clear strategy and stick to it; otherwise it's easy to be distracted by irrelevant data and follow the wrong path.

To find a memory leak you will need:

- Documented test transaction, including any necessary clean-up steps. For example, a sequence of web service or network calls or a sequence of user-interface actions ("Enter x ..."); "Click y ..."
- ANTS Memory Profiler.

© Red Gate Software Ltd
16 August 2010
Basic strategy for finding a memory leak

Here are the steps we're going to take:

1. Start your application and take memory snapshots.
2. Take a baseline snapshot.
3. Run the test process.
4. Filter out the noise.
5. Analyze the results.
6. Investigate suspicious classes.
7. Investigate instances of a suspicious class.
8. Identify the cause of the memory leak.

To illustrate the basic strategy, I've written an application containing a memory leak I've seen many times at customer sites. For "background noise," I've added a timer that runs every 5 seconds, querying a remote database. This should leave lots of "red herring" objects on the heap whenever I take a memory snapshot, and will closely mimic what happens in a real-world application. The key skill to learn is how to decide what's significant ... and what's irrelevant.

1. Start your application and take memory snapshots
   • Open ANTS Memory Profiler, either from within Visual Studio or from the Red Gate Programs folder, and start profiling your application.

Figure 1: ANTS Memory Profiler start-up window
2. Take a baseline snapshot

- Click the Take Memory Snapshot button to take your first snapshot.

ANTS Memory Profiler forces a garbage collection, then takes a snapshot of every object still left in memory, and displays the results.

This first or “baseline” snapshot is crucial because it captures the memory state before we have executed the test transaction. Comparing back to this baseline will help identify possible memory leaks in the subsequent test transactions. Figure 2 shows the baseline snapshot in my test.

Figure 2: Snapshot summary window
3. Run the test process

We’re now ready to begin the test:

- Run your test transaction, including any clean-up steps, and then take a memory snapshot.
- Repeat this at least twice more.

At the end of the test process you will have at least four snapshots (including our baseline). The greater the number of test transactions the better because, if there is a memory leak, it will continue to get bigger each time you run the test transaction – and that will make it easier to see the issue.

*Figure 3: Comparing snapshots*

The results summary in Figure 3 shows our second snapshot (snapshot 2) with a comparison against the baseline snapshot (snapshot 1) on the right.

4. Filter out the noise

Before we start to look at the results in detail, it’s a good idea to filter out some of the irrelevant noise. Filtering out the noise reduces the chances of finding false positives, and cuts down on the amount of work we need to do. ANTS Memory Profiler has some great filters to help.

From the **Standard object filters**:

- Choose **Comparing snapshots – Only new objects**.
  
  We only need to see the new objects left behind since the previous snapshot, as they are possible leak candidates.
From the **Advanced object filters**:

- In the **Kept in memory only by GC roots of type** filter, clear the **Finalizer queue** check box. This eliminates objects that are waiting for finalization. These objects are waiting for the .NET finalization thread to run, have no other references, and will disappear from the trace anyway. Filtering these objects out removes unnecessary noise from our trace.

- From the **Objects which are/are not GC roots**, select **Objects which are not GC roots**. A memory leak is extremely unlikely to be a GC root.

### 5. Analyze the results

During the test process we captured a number of snapshots, including the baseline. We now need to analyze each snapshot and look at the classes that survived garbage collection.

To do this, let’s look at the list of classes that use the most memory:

- Click the **Class List** button. The class list displays all of the classes with new object instances allocated since the previous snapshot. It can be useful to sort the list by descending class size; click the top of the **Live Size** column to do this.

*Figure 4: The class list, showing classes left in memory*
The class list is where the real work is done. We could just work through each class in turn and look at each object instance and its references. That would take some time, so we’re going to improve our chances of hitting the right class straight away, by comparing each snapshot to the previous one and looking for same classes that always appear.

We performed the same actions in each repeat of the test process, so any memory leak should be consistent, and a leaking class should appear in each snapshot. If the same class does appear in each of the snapshots, it goes to the top of my priority list for investigation.

To compare snapshots, use the snapshot toolbar.

*Figure 5: The snapshot toolbar*

In Figure 6, *snapshot 3* and *snapshot 2* are compared. *Order*, *Tick*, and *string* are consistently appearing in the snapshots, so they go straight to the top of my priority list for further investigation.

*Figure 6: Viewing the class list and comparing the snapshots*

Once you have your suspicions about a class, you can often confirm its leak profile by watching the memory leak get worse over the snapshots in the class list. Compare each snapshot against the baseline, starting with *snapshot 2*; your suspicions will be confirmed if the *Live Size* for your leaking class increases each time.
6. Investigate suspicious classes

Every class in the class list is there because it has instances that survived garbage collection. We need to find out whether we can trace any of those instances back to references in our code. If we can, then we can either fix the problem, or confirm that it is intended behavior and won’t cause a problem.

To do this, we need to look at each class in detail. I recommend starting with your list of suspicious classes first, and only then work through the class list, starting at the top (ordered by descending Live Size) until you find a memory leak.

To do this:

- Select the class in the class list, then click the Class Reference Explorer button.

Figure 7: The class reference explorer

Figure 7 shows the class reference explorer for the Order class, a graph of all of the classes that have object instances with references to Order classes. We can use this graph to trace backwards through reference paths to find which of our own classes, if any, reference the Order class:

- Click on a class to expand its referencing class (leftwards).

In Figure 7, notice MemoryAnalysis.Tick contains a percentage value (100%). This indicates that instances of Tick classes are responsible for 100% of the references to instances of Order classes.
As we are trying to trace back to find if it is one of our own classes, all we need to do is keep following the class path backwards, following the path with the highest percentage reference contribution, until we come to a class we recognize.

If you get to a GC root when you are expanding references to a class leftwards, move on to the class with the next highest percentage value, and expand references to that class leftwards.

7. Investigate instances of a suspicious class

When we reach a class we recognize, we can investigate instances of the class:

- Right-click on the source class, and select *Show Instance List for <class> on this path.*
  A list of the actual object instances of *Order* is displayed.

*Figure 8: The object instance list*

The object instance list in Figure 8 lists all of the object instances of our potentially leaking class referenced by our own code.

From here, comparing against the baseline snapshot and walking upwards through the snapshots (from 2 to 4) should confirm whether there’s a memory leak. If there is one, we’d expect to see the leaked objects building up in the list after each snapshot.
8. Identify the cause of the memory leak

Finally, we need to identify what in the source code is holding the reference that creates the memory leak, so that we can fix it:

- Select one of the objects in the list, and click the **Object Retention Graph** button.

Figure 9 shows the object retention graph for *Order*, and the object references keeping the selected object in memory. It also provides the variable names!

*Figure 9: The object retention graph*

By working up the object retention graph from *MemoryAnalysis.Order* we can see that *MemoryAnalysis.CurrencyManager* is ultimately holding a reference to our selected object. The reference is caused by an instance variable *m_Currency* of type *MemoryAnalysis.CurrencyManager* from an event source *OnPriceUpdate*.

This is one of the most common .NET memory leaks I have come across, and it's in a technique many .NET developers use.
My application is a simple currency trader. It creates a Currency Order and adds it to a Deal List. It uses a CurrencyManager class that constantly updates the currency price via an OnPriceUpdate event. Each Order subscribes to the OnPriceUpdate event using its Order.OnTick method.

```csharp
Order newOrder = new Order("EURUSD", DealType.Buy, Price, PriceTolerance, TakeProfit, StopLoss);
newOrder.OnPlaced += OrderPlaced;
m_Currency.OnPriceUpdate += newOrder.OnTick;
m_PendingDeals.Add(newOrder);
```

When the price is right an Order completes; it calls the OnPlaced event which is handled by the OrderPlaced method.

```csharp
void OrderPlaced(Order placedOrder)
{
    m_PendingDeals.Remove(placedOrder);
    m_ActiveDeals.Add(placedOrder);
}
```

The OrderPlaced method is missing one line which is needed to avoid a memory leak.

```csharp
m_Currency.OnPriceUpdate -= placedOrder.OnTick;
```

Memory leak found!

In many cases, there are alternative ways to find a leak using ANTS Memory Profiler. For example, in this case we could have found the memory leak using the Kept in memory only by event handlers filter.

Finding a leak when you don't know what transactions are causing it

If you're not sure which test transactions may be responsible, then it's a good idea to monitor the executing application running within ANTS Memory Profiler. Watch the timeline and, when memory usage starts to increase, take a number of snapshots. The number and interval between depends on the application's behavior. It's likely the application will behave the same way repeatedly, so get used to the memory usage spike, and take a set number of snapshots over this time frame. Then use the same techniques outlined above to find out if there is a memory leak.

Making sure the memory leak is fixed

This is the easy bit, because all you need to do is repeat the process and make sure the class behavior has disappeared. Run exactly the same test process and snapshots, and compare the traces with the pre-fix traces.

Recognizing when you don't have a memory leak

As I mentioned earlier, a memory leak is usually consistent in that the same sequence of actions causes the memory leak. So, when you run a test, as long as you repeat the test exactly, you can validate that it doesn't have a memory leak by comparing the class list between snapshots, as described in the Basic strategy for finding a memory leak section. Leaking classes should consistently appear in each snapshot. If no classes appear consistently, it's unlikely you have a memory leak. The classes that do appear should be investigated further. If you find no references back to code using the class reference explorer and the object retention graph, then you are not causing a memory leak.
Checking an application’s memory usage

Investigating an application’s memory footprint
To determine an application's or test transaction's memory footprint, you need to find out which class instances survive from beginning to end.

1. Take a baseline snapshot.
2. Wait for the memory trace graph in the timeline to stabilize.
3. Take another snapshot.
5. Click the Class List button to view the class list data.

Figure 10: Memory footprint snapshot

Figure 10 gives a summary of the large allocated classes after the application has loaded.
To investigate this further, click the **Class List** button.

**Figure 11: Class list for memory footprint**

Investigate the classes with the largest **Live Size**:

- Select the class with the largest footprint (in this case `byte[]`).
- Click the **Class Reference Explorer** button.
The class reference explorer allows us to trace backwards through the class references to find the classes that have object instances that are responsible for the majority of the references. If we can ultimately trace these references back to our own classes, then we then can decide whether to optimize the code to remove the references and thereby reduce the footprint.

Just as with memory leak detection:

- Follow the class path with the highest % contribution until you reach a class you recognize.
- Right-click on the source class and select **Show Instance List for <class name> on This Path**. A list of object instances for this class is displayed.
Each one of the objects in Figure 13 is adding to the footprint; to investigate their origin, select the top one and view the object retention graph.
The object retention graph in Figure 14 illustrates that the `byte[]` array is being held in memory by a `Generic List` within my own `MemoryAnalysis.Form1` class with a variable name of `m_LOH`. I have all the information I need from here to make a decision about this collection and whether its contribution to the application footprint is justified.
Investigating large object heap fragmentation

If your application allocates objects larger than 85K, they will be allocated onto the large object heap (LOH). Unlike the small object heap (for objects <=85K) the LOH doesn’t remove the gaps left between objects caused when an object is garbage collected. This can lead to memory fragmentation and to an out of memory exception when there is still space on the heap (in the gaps!).

The snapshot summary gives an overview of the memory state of the large object heap. You can use this to identify symptoms of fragmentation. Three statistics are provided:

- **Free space on all .NET heaps** (total memory reserved)
- **Largest free block** (largest reserved memory block)
- **Max. size of new object.**

If the **Free space on all .NET heaps** statistic is large and the **Largest free block** is small, this can indicate a fragmentation problem. It basically means there are lots of small chunks of fragmented memory available. It doesn’t necessarily mean there will be an out of memory exception, though. If the **Max. size of new object** is still large, then the size of the heap can still be expanded by reserving more memory, increasing the application footprint.

If **Max. size of new object** is small as well, though, an out of memory exception is on its way! This is the classic LOH fragmentation condition.

For more information, Andrew Hunter has written an excellent article on this subject: [The dangers of the large object heap](#).

**Summary**

Before you check a completed unit of work back into the source control system, I advise you to:

1. Run each of your test transactions against the memory leak basic strategy.
2. Look for anything that anything that is consistently "there" and investigate it further.
3. If there are no issues, which most of the time there won’t be, then you’re done!
4. If there are issues, fix them!

Don’t leave it until it goes into system or load testing or, even worse, production, because you can avoid a lot of work later by finding and fixing the issues now.

**About the author:**

*Chris Farrell is a development consultant, technical trainer and developer with seven years of .NET experience. He is an expert in .NET memory management and application performance optimization.*