

# Don't Fear the Garbage Collector

Chris Tacke

OpenNETCF Consulting

August, 2007

## Introduction

We here at OpenNETCF Consulting just delivered version 1.0 of a plant floor monitoring system running on an x86 device with CE 5.0. Since the application was designed for data collection and reporting, we naturally leveraged the .NET Compact Framework version 2.0 and our own Smart Device Framework to meet their functional requirements within a very aggressive schedule.

As we got to the point where we had most of the functionality nailed down, both the customer as well as us started to get a bit concerned about the overall performance of the application. As we added features the application's throughput dropped, and it was getting to the point of unacceptable.

Of course the application is very complex, with multiple processes using lots of threads. Managed and native threads routing data to and from a database through all sorts of communications layers (we'll look at the app in more detail in another article). So what is a developer to do? Where do you start to look for areas that you rework or refactor to improve overall solution performance?

One behavior that we found peculiar was a fairly consistent periodicity to the application's throughput. We have statistical counters that track message throughput of the application and we'd watch it oscillate, running up to 10 or 12 messages a seconds for a few seconds, dropping down to zero, then ramping back up over and over.

My gut feeling was that we were seeing an artifact of Garbage Collection (GC), but before jumping into the code and blindly assuming that I was correct, we needed some data to prove the theory one way or another.

## Garbage Collection

Before we dive into how we went about testing this theory, it's important to understand how we came up with the theory in the first place. What about the general behavior of the application led me to infer that this might be a root cause of the problem? To answer that question we need to take a short side trip and look at how the GC works at least at a high level. For more in-depth coverage of how the GC does what it does, take a look at my MSDN WebCast[1] as well as the blog entries from Scott Holden[2] and Steven Pratschner[3] of the .NET Compact Framework team.

When an application creates an instance of a reference-type object, the allocation must be backed by physical memory. When the object is no longer in use, that memory can be released back to the system for our application or any other to use. As managed code developers, we're protected from the ugly sausage-making task of allocating and freeing the physical memory, instead we rely on the Compact Framework to cleanly handle all of this for us.

The GC's algorithms for how it does all of this work can be a bit complex, but at a high level what it does is allocate large blocks of physical memory which it then divides into smaller "chunks" that it uses to hold our object instances. As we destroy objects, it logically frees those chunks but doesn't necessarily free the physical memory backing it. The GC's collection of these blocks is called the GC Heap.

As our application plods along creating and destroying objects, "holes" are created in the GC Heap where destroyed objects once resided. The GC keeps track of these holes and when certain thresholds or triggers are passed, it goes in and cleans them up by running finalizers, possibly shifting around the in-use blocks to collect the free space together (think of a disk defrag) and sometimes even freeing some of the allocated physical memory. This process is known as Garbage Collection or just Collection.

In order to do perform Collection, the GC has to make sure everything in the application is in a known safe and stable state. It does this by suspending all threads in the application (think of the problems you'd have if a Thread was accessing some property of a class instance while at the same time the GC was moving it in memory). It is this action - the suspension of all threads in the application - that is what many developers fear, and is what I suspected was causing the periodic performance fluctuations we were seeing. If the GC was collecting with a similar frequency as our lows in throughput performance, it would be strong evidence that it was the culprit.

### **Remote Performance Monitor**

The first step in checking the theory that the Garbage Collector was our performance-thief was to collect solid data. Fortunately Microsoft has provided managed developers a tool called Remote Performance Monitor (RPM). RPM has been around since version 1.0 of the Compact Framework, but unfortunately it doesn't seem to get much press short of a couple blog entries[4] and presentations for those fortunate enough to attend shows like MEDC.

RPM is a cool tool, even up through version 2.0 (though the 3.5 version has some features that will blow you away[5]), provided you know what to make of all of the data it spews out. Essentially what it provides is a live view of a litany of counters from the CLR. It provides the ability to save of views of the data, and even the ability to "publish" the data to PerfMon on the desktop so you can get nice trend graphs of variables over time.

To test our theory, however, we didn't need much data. We simply wanted a general sense of how much memory we were allocating in our application and how often the Garbage Collector was performing Collections and compactions.

So I fired up RPM, launched our primary data collection application through the RPM interface, and had it collect data for about a half hour while I went and had a cup of coffee. Never trust the results of short term test runs, and never analyze data without a good dose of caffeine. Here are some of the statistics from that first half-hour run.

	Category	Counter	Value
1	Loader	Total Program Run Time (ms)	1,695,518
2	Loader	Methods Loaded	5,190
3	Loader	Classes Loaded	1,623
4	Loader	Assemblies Loaded	23
5	GC	Total Bytes In Use After GC	3,310,272
6	GC	Pinned Objects	489,219
7	GC	Peak Bytes Allocated (native + managed)	4,006,896
8	GC	Objects Not Moved by Compactor	2,010,250
9	GC	Objects Moved by Compactor	1,306,894
10	GC	Objects Finalized	280,433
11	GC	Managed String Objects Allocated	2,189,191
12	GC	Managed Objects Allocated	7,138,911
13	GC	Managed Bytes In Use After GC	548,092
14	GC	Managed Bytes Allocated	440,733,748
15	GC	GC Latency Time (ms)	14,702
16	GC	GC Compactions	409
17	GC	Collections (GC)	412
18	GC	Code Pitchings	2
19	GC	Calls to GC.Collect	0
20	GC	Bytes of String Objects Allocated	184,459,896
21	GC	Bytes Collected By GC	440,063,976
22	GC	Boxed Value Types	531,631

I've numbered each line so we can run through them and discuss what each number means.

You can see from line 1 that we had a run time of 28 minutes 15.518 seconds. Lines 2-4 say that the loader loaded 5190 methods and 1623 classes out of 23 separate assemblies during this run period.

Line 5: At the time this view was captured, there was about 3.3MB in use by the application immediately following the most recent Collection.

Line 6: The GC has encountered nearly half a million pinned objects during its Collections (see line 17).

Line 7: The maximum amount of allocated memory in use at any given time through this run was just over 4MB.

Line 8: Just over 2 million objects were not moveable during compaction. This indicates a high number of pinned objects when coupled with Lines 9 and 16.

Line 9: A cumulative total of 1.3 million objects were moved during Compactions (line 16).

Line 10: 280 thousand items had to be Finalized. Any item with a finalizer survives two GC Collections before its GC Heap allocation is completely freed [5]. This seems like a lot, but it's not evident immediately whether these are from our application or from assemblies were using (e.g. SQL CE or the CF itself). If this were causing a problem, I'd expect the GC heap to be having size problems, which I don't think is the case, so it doesn't cause me any undue concern.

Line 11: We've allocated over 2.1 million strings in 28 minutes. While not overly alarming, it is the most suspect thing so far. That's a rate of about 1290 string allocations per second. This is a potential source of garbage and looking at the code to try to reduce this rate might be a good idea.

Line 12: We've allocated over 7.1 million managed object in 28 minutes. Again this isn't overly alarming, but it does mean that we're allocating objects at a rate of over 4200 per seconds. We might want to investigate places in the code where objects are created instead of reused. It's also interesting that roughly 30% of our total allocations have been strings (see line 11). It doesn't mean a lot by itself, but it's a data point to remember as I profile other applications.

Line 13: We had 548kB of managed objects still live after the last GC. When we couple this statistic with line 5 (bytes in use after GC) we see that a large amount of our allocations appear to be unmanaged. Due to the architecture of this application, that doesn't surprise me, so it's not a cause for concern.

Line 14: In 28 minutes we've allocated over 440MB of data. Wow. That's a lot of allocation.

Line 15: We've had a GC latency of almost 15 seconds. More on this one later.

Lines 16-17: We've had 412 Collections and 409 Compactions. That means we're collecting on average once every 4.1 seconds, and nearly every Collection also requires a compaction. This means we're generating a lot of garbage, and the GC Heap is quite fragmented.

Line 18: We've only had to pitch code twice. This tells me we're not having any low-memory conditions.

Line 19: Our code never calls GC.Collect. Nothing more to be said.

Line 20: Coupled with line 11 (bytes of string objects) this tells me that our string objects average about 26 bytes each. Considering that we're Unicode and that managed strings also carry a 4-byte length with them, each string is, on average, less than 10 characters. That's a lot of small strings. Again, seeing if we can reuse string might be worthwhile.

Line 21: This number is really close to Line 14 (managed bytes allocated). This tells me that we're probably not leaking and memory, we're just churning a whole lot of it.

Line 22: This says we have a boxing rate of over 300 boxing operations a second. While not distressing, we might want to at least look around and see if there's some obvious place that we're getting implicit boxing that we don't need.

So what do all these statistics means when looked at in aggregate? I see a whole lot of allocation and collection. This system is designed to run 24/7, so if we extrapolate out these numbers to a day of running we would then estimate that we'd churn (allocate and release) nearly 23 Gigabytes of data and Collect over 21,000 times. Sounds a bit frightening in those terms doesn't it?

The real gem of information here, though, is line 15 - GC Latency. This is the total amount of time that was spent by the system doing those 412 Collections and it's only 14.7 seconds. That means each collection is really taking only about 35 milliseconds. To bring that even further into perspective, we spent 14.7 seconds out of over 28 minutes collecting, so or about 0.87% of our execution time.

So while the raw numbers look big, we're actually spending less than 1% of our total execution time doing collections. My first reaction to this fact was to re-run all of my calculations. My second reaction was one of amazement and reverence. The Compact Framework team has obviously done a hell of a good job refining the GC's algorithms and they deserve recognition for that. The Collection statistics for this application could have been much, much worse if they hadn't done such a good job.

Does this mean I can rest easy? Not at all. We're still churning a lot of data and creating a whole lot of objects. It's still a worthwhile exercise to go in and spend a couple hours trying to reduce the both the Collection and churn rate, but it also means that we're going to quickly have diminishing returns. Spending a days or weeks trying to eke out every last performance gain would be foolish.

In fact I did revisit the code and spent about two hours reworking some of the code to reduce allocations. It turned out that we were generating a lot of debugging information that we used for logging even when the logging was turned off (the objects were created, just never used). After cleaning this, and a few other areas up I got the GC rate down to about once every 6 seconds, or about 0.6% of run time which is nearly a 50% improvement, but the reality was that it

was still a very miniscule part of the overall run and it lead to no noticeable improvement in throughput.

## Summary

So what did we learn from this exercise, other than we needed to look elsewhere for the performance bottle necks (they turned out to be in our usage of the SQL CE database, and I'll be going into depth on what we learned there in a later article)?

Don't fear GC. The GC has an extremely important job. It is the magic box that allows us to worry about our architecture and not about memory leaks, buffer overruns and access violations. The Compact Framework team has made the GC work well and as transparent as we could hope (though I definitely don't want them to stop trying to improve it or allowing us to have more control over it). The GC allows us to develop far more stable code in a much shorter period of time, and we must accept that it's not something to be feared or avoided.

We also got a little practice in using the tools Microsoft provides to help us analyze our systems. Without tools like this we'd be left guessing about what to work on and I'd probably still be trying to reduce the garbage the application is creating.

We still need to be cognizant of the garbage we create, but maybe in the future the first thing that comes to mind when we have a performance problem shouldn't be the Garbage Collector or the runtime. Instead maybe we should instead ask ourselves what mistake we made in our own architecture or implementation. After all even the best system can't compensate for poor code, and none of us are immune to making mistakes no matter how much experience we have.

## References

- [1] <http://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032318790&Culture=en-US>
- [2] <http://blogs.msdn.com/scotttholden/archive/2004/12/28/339733.aspx>
- [3] <http://blogs.msdn.com/stevenpr/archive/tags/GC/default.aspx>
- [3] <http://blogs.msdn.com/stevenpr/archive/2006/04/17/577636.aspx>
- [4] Microsoft hasn't yet officially published any RPM 3.5 stuff, but when they do we'll update this reference
- [5] <Insert link to MEDC decks on Community>  
Also See Reference 1