# Memory and Performance Best Practices

Xamarin Evolve, Chapter 13

Xamarin Inc

# Overview

Mobile devices may have limited resources compared to today's desktop computers, but they still have ample memory. Therefore, effective memory management is not simply about quickly releasing memory.

You can use memory to your advantage (e.g. a more fluid user interface) by having a good caching strategy. Just be ready to release this memory, like anything you can recreate, if the operating system asks for it.

E.g. an often-used view controller, using several small and lightweight objects, might be cheap to cache and reuse. The alternative, disposing and re-creating it each time, would require more allocations (and GC time) and could lead to severe memory fragmentation. Also you risk to repeat, several times, any memory leaks or cycles your code might have.

However, you cannot implement any caching effectively unless you know the real (memory) cost of what you create. Therefore *always measure*. It's also very important to test on devices, in particular for iOS devices because the simulator does not "emulate" the memory restrictions of the devices.

iOS and OSX users can use Apple Instruments to get accurate memory measurements of their applications, whereas Android users have the Android Device Manager at their disposal.

This chapter will delve into some of the best practices for effectively using memory and provide tips to help achieve high performance.

# Memory

## Reducing Pressure on the Garbage Collector

When it comes to garbage collection pauses, you care about two things:

1. How often they occur
2. How long they take

You can influence both through the way you allocate and use objects.

The duration of a garbage collection is roughly proportional to the number of live objects (and not necessarily their size). Also, the frequency of garbage collections increases with the more memory you allocate since the last collection.

Taken together these two properties, duration and frequency, mean if you allocate lots of objects, but very few of them stay alive, you will get many short garbage collections. If, on the other hand, you allocate slowly, but most of your objects stay alive, you will get a few longer duration collections.

### Generational vs. Non-generational GC

Non-generational collectors, like Boehm, always collect the whole heap. Generational collectors, like SGen, only collect a small part of the heap, called the *nursery*, most of the time. The nursery is where new objects are allocated. The

main advantage of this scheme is that those minor collections (as opposed to major collections, which do collect the whole heap, and will still occur from time to time) take time proportional to the number of new live objects since the last collection, as opposed to the total number of live objects on the heap.

What this results in is even if you have lots of live data on the heap, once you get to a state where most of your new objects die quickly, most garbage collections will be short, minor ones.

### Avoiding Collections

The extreme case on the low end is not allocating any memory; at which point no garbage collections will occur. To still be able to dynamically allocate objects - within limits - you can pre-allocate a pool of objects that you can manage manually during your no-allocation periods.

Note that this strategy is only a good idea if your rate of heap allocation really is very low. If you're allocating from the heap at a normal rate while at the same time keeping unused objects live, they will just make your garbage collections slower. On SGen, allocating objects is also very fast, so you likely won't gain anything on that front either.

## Objects Wrapping Limited Resources

Using a garbage collector frees you from having to manage object allocation, but there are certain kinds of objects that require special considerations.

When your program allocates a new object, it can determine whether it can satisfy the allocation with the existing memory pools, or whether it needs to get more memory. Getting more memory can either be done by requesting a fresh new block from the operating system or by performing a garbage collection pass.

However, some objects wrap limited resources such as:

- Network connections
- Large blocks of memory
- File descriptors

To make efficient use of memory, it is best to explicitly release these resources and not wait for the garbage collector to be activated at some indeterminate point in the future.

### Images

For example, consider image objects that bind to native iOS, Android or OS X images. These take about 20 bytes of memory. However, one of those 20 bytes contains a pointer to the actual image data, which could be on the order of megabytes. From the garbage collector's perspective, only 20 bytes are consumed. Therefore, thousands of images could be allocated before a collection pass occurs.

To address the need to free such resources earlier, you can use the `Dispose` method in C#, which can be called either directly or implicitly by wrapping an object tin a `using` statement, as shown below:

```
using (var image = UIImage.FromFile ("image.png")){
        Draw (image);
}
```

# iOS Specific Memory Considerations

## Strong Reference Cycles in iOS

In some situations, it is possible to create strong reference cycles that could prevent objects from being collected.

For example, consider the case where an `NSObject`-derived subclass, such as a class that inherits from `UIView`, is added to an `NSObject`-derived container and is strongly referenced from Objective-C, as the following code illustrates:

```
class Container : UIView {
}

class MyView : UIView {
   object parent;
   public MyView (object parent) { this.parent = parent; }
}

var container = new Container ();
container.AddSubview (new MyView (container));
```

When the above code creates the `Container` instance, the C# object will have a strong reference to an Objective-C object. Similarly, the `MyView` instance will also have a strong reference to an Objective-C object.

### AddSubview Creates a GCHandle

The call to `container.AddSubview` will increase the reference count on the unmanaged `MyView` instance. When this happens, the Xamarin.iOS runtime creates a GCHandle to keep the `MyView` object in managed code alive, because there is no guarantee that any managed objects will keep a reference to it. In fact, as far as the C# code is concerned; the object would be gone after the `AddSubview` call were it not for the GCHandle.

### Dealing with Strong Reference Cycles

The unmanaged `MyView` object will have a GCHandle pointing to the managed object, a strong link. The managed object will contain a reference to the `Container` instance. In turn the `Container` instance will have a managed reference to the `MyView` object.

In cases where a contained object keeps a link to its container, a few options are available to deal with the cycle:

- Manually break the cycle by setting the link to the container to null.
- Manually remove the contained object from the container.
- Call Dispose on the objects.

- Avoid the cycle by using `WeakReference<T>` instances to keep a weak reference to the container.

### Event Handlers and Lambdas

Reference cycles can also occur when using event handlers and lambda syntax, as lambda expressions can reference and keep your objects alive. So be sure to you remove event handlers after you no longer use them to make the code collectible.

### Best Practices to Avoid Strong Reference Cycles

There are a few guidelines you can follow to avoid strong reference cycles, as listed below:

- `NSObject`-derived classes should not keep strong references to their parents (either avoid, or use `WeakReference<T>`)
- NSObjects should not keep strong references to other `NSObject` ancestors (either avoid, or use `WeakReference<T>`)

For additional information, see the following:

- Rules to Avoid Retain Cycles
- http://stackoverflow.com/questions/13058521/is-this-a-bug-in...
- http://stackoverflow.com/questions/13064669/why-cant-monotou...

### Disposing of Objects with Strong References

If you must have the cycle and it is difficult to remove the strong dependency, make your `Dispose` method clear the parent pointer.

For containers, implement `Dispose` to remove the contained objects as shown below:

```
class MyContainer : UIView {
	public override void Dispose ()
	{
		// Brute force, remove everything
		foreach (var view in Subviews)
			view.RemoveFromSuperview ();
		base.Dispose ();
	}
}
```

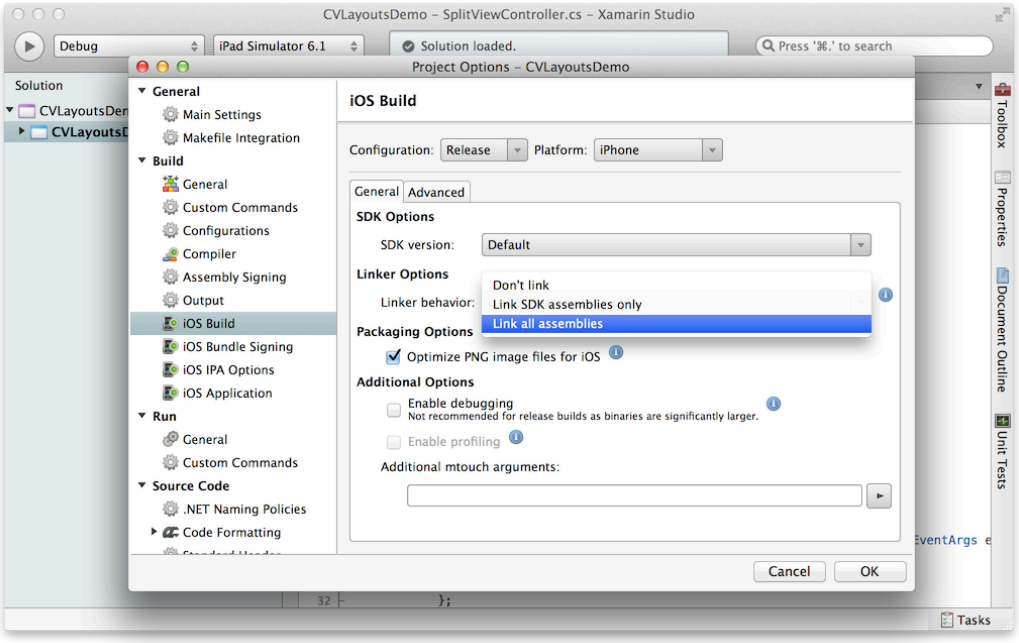
For a child object that keeps strong reference to its parent, clear the reference to the parent in the `Dispose` implementation:

```
class MyChild : UIView {
	MyContainer container;
	public MyChild (MyContainer container)
	{
		this.container = container;
	}
	public override void Dispose ()
	{
		container = null;
```

```
            }
        }
```

## Using the Linker

Use the linker on all your code. **Link all assemblies** versus the default **Link SDK assemblies**, as shown in the following screenshot from Xamarin Studio:




For bindings this will remove unused backing fields and make each instance (or bound objects) lighter, consuming less memory.

## Reducing Executable Size

In addition to runtime memory, another item to be aware of is the executable size. The following initial steps will help keep this size down:

- Ensure that your application is not being build with **Don't link**, as this would create very big applications since it would AOT *nearly* the full .NET framework that Xamarin.iOS ships.
- Make sure you're building for a single architecture (ARMv7). FAT binaries (e.g. ARMv7 and ARMv7s) are built two times and need twice the space.
- Make sure you have not enabled the Debug build (it's possible to do so in Release build since it's a checkbox). This would create larger binaries to support debugging.
- Make sure you're using the LLVM compiler. It takes more time to compile but generates better (and smaller) code.

These initial checks are pretty easy to do and are the most common reasons for getting very large binaries.

### Understanding How Applications are Built

To understand where the size comes from you need to know how applications are built.

- The main difference between the Android and iOS versions is that JIT (just-in-time compilation) is not allowed on iOS.
- This means the code must be AOT compiled (ahead-of-time compilation), which creates large executables, because IL is more compact than native code.
- If your code is generic-heavy then the final binary can get quite large since it will need to natively compile every generic possibility. Many cases can be shared, but value-types cannot.

### Additional Size Reduction Techniques

First, try to reduce your *managed* code size. The easy way to do this is the enable the linker on every assembly, i.e. Link all assemblies in your project options.

Many people think it's not worth linking their own code - because they know it will be needed at runtime (so the linker cannot remove it) or else they would not have written that code.

That's half true. The linker *might* not be able to remove most of your application code, but if you're using 3rd party assemblies they are not likely 100% used. The linker can remove that extra code (and also remove, from the SDK, all the code that is kept to support that unneeded code). The more *shared* code you have the more the linker can help you.

Less IL code means faster AOT time, which translate to faster builds and smaller final binaries (including the executable size).

Second, try to reduce your *native* size. If you're building native libraries then have a second look at them, as they will be statically (not dynamically) linked to your final binary (another rule for iOS applications). Some of them might not be worth (feature wise) their weight in your final binary and there might be *lighter* alternatives.

# Android Specific Memory Considerations

## Removing Event Handlers in Activities

When an Activity is destroyed in Dalvik it could still be alive in Mono runtime. Therefore, remove event handlers to external objects in `Activity.OnPause` to prevent the runtime from keeping a reference to an Activity that has been destroyed.

In an activity, declare your event handler(s) at a class level:

```
EventHandler<UpdatingEventArgs> service1UpdateHandler;
```

Then implement the handlers in the Activity, such as in `OnResume`:

```
service1UpdateHandler = (object s, UpdatingEventArgs args) => {
```

```
            this.RunOnUiThread (() => {
                    this.updateStatusText1.Text = args.Message;
            });
    };
    App.Current.Service1.Updated += service1UpdateHandler;
```

When the Activity exits the running state, `OnPause` is called. In the `OnPause`, implementation, remove the handlers as follows:

```
App.Current.Service1.Updated -= service1UpdateHandler;
```

## Dialogs

When using the `ProgressDialog` class (or any Dialog or Alert), instead of calling the `Hide` method when you're done with the dialog, you must call `Dismiss`. Otherwise, the dialog will still be alive and will leak the Activity by holding a reference to it.

# Performance

## Responsive User Interfaces

Modern users expect applications to be responsive to their input and not have to wait for the application to respond or experience several seconds of lag when an application is performing some task.

In general, this means that you should avoid performing intensive computations while responding to user input or user requests.

For example, if the user requests a video to be compressed, you should avoid doing this on the handler for the request. Instead you should queue a background operation to perform the task. When the task has completed, it can notify the user interface thread about the change and update the user interface as expected.

It is important to know that user-interface APIs in Android, iOS or OSX can only be invoked from the UI thread. This is because none of these platforms provide a thread-safe API. Developers that perform background operations use idioms to queue operations to be invoked on the main user interface thread when they need to update the UI.

This has a few implications:

- You should consider whether you should offer a canceling background operation in your UI. This might be useful, for example, for a long-running video encoding operation.
- You should consider whether you should automatically cancel the background task if the user changes his work context. For instance, the user might tap on the back button, rendering whatever work he requested pointless.

Typically, you would queue the operation into a background thread (more on this below). When the background thread completes the operation, it invokes an API to run some code on the main UI thread to update the interface with the results.

The code below shows conceptually how to perform background processing:

```
// Invoked when the user wants to compress a video
void CompressButtonClicked ()
{
        StartBackgroundProcessing (LongRunningTaskAndNotify);
}

void LongRunningTaskAndNotify ()
{
        LongRunningTask ();
        InvokeOnMainThread (UpdateUI);
}

void UpdateUI ()
{
        compressStatus.Text = "Done";
}
```

The above can be significantly improved by using C# lambdas, which allow most of this code to be done in a single place:

```
void CompressButtonClicked ()
{
        StartBackgroundProcessing (delegate {
                LongRunningTask ();
                InvokeOnMainThread (delegate {
                        compressStatus.Text = "Done";
                })
        });
}
```

In the above examples we use the `StartBackgroundProcessing` method to indicate that we are starting a thread on the background. The actual mechanics of starting that task in the background depends on what your application wants to achieve.

Nowadays, we recommend that developers use the features in the *Task Parallel Library* as it provides various design idioms that assist developers with their multi-threaded design. The Task Parallel Library main entry point is the `System.Threading.Tasks.Task` type. While this is the recommended method, developers can roll their own threading primitives if they want using the `System.Threading.Thread` and `System.Threading.ThreadPool` if they prefer.

## Task Parallel Library

All of Xamarin's products (Android, iOS and Mac) support the Task Parallel Library. To make things even more enjoyable, code running on the user interface thread already has a synchronization context setup to dispatch tasks on the user interface thread, which makes it natural for callbacks to invoke the code on the main UI.

### Starting a Background Operation from the UI thread

To start some background code, you can use either the method pattern or the lambda pattern. The lambda pattern has the advantage that it automatically allows you to pass contextual information by merely referencing variables from their scope.

```
// Method pattern
void Start ()
{
Task.Factory.StartNew (BackgroundTask);
}

void BackgroundTask ()
{
// My long running operation
for (int i = 0; i < 100; i++)
        UploadFile (i);
}

// Lambda pattern, notice how "n" is easily passed
void Start (int n)
{
        Task.Factory.StartNew (() => {
                for (int i = 0; i < n; i++)
                        UploadFile (i);
        }
}
```

If you are nesting tasks, see the TPL tricks blog post from Jeremie Laval.

### Invoke Code on UI thread from a Background Task

If you have some background code and you want to run code on the UI thread, you can use this cross platform code snippet to setup your task:

```
// ran on main thread, result stored to be accessible as necessary
var scheduler = TaskScheduler.FromCurrentSynchronizationContext ();

Task.Factory.StartNew (() => ExpensiveWork ()).ContinueWith (t => {
PerformUIChanges (t.Result), scheduler);
});
```

### Composing Background Tasks

You can also compose your background tasks to run one after another. Here are some examples showing common idioms:

```
// Simple
aTask.ContinueWith (t => DoSomething ());

// Passing the result from the first task to the second:
aTaskResult.ContinueWith (t => DoSomething (t.Result));

// Execute code only after all tasks have completed:
Task.WhenAll (new Task[] { t1, t2 }).ContinueWith (t => var ts = t.Result)
```

```
// Execute some background task when at least one of the provide tasks
completes
Task.WhenAny (new Task[] { t1, t2 }).ContinueWith (t => { DoSomething
()});
```

### Waiting for Background Tasks to Complete

In general, you should use continuations (listed above), but if you want to explicitly wait, you can use the following idioms:

```
// Wait for "aTtask" to complete
aTask.Wait ();

// Wait for all the tasks in the task array to complete
Task.WaitAll (new Task [] { task1, task2 })

// Wait for any of the tasks in the array to complete
Task.WaitAny (new Task [] { task1, task2 });
```

### Canceling a Background Task

If you are planning on cancelling a background task, use the following idiom:

```
var source = new CancellationTokenSource ();
var token = source.Token;
var task = Task.Factory.StartNew (() => {
token.ThrowIfCancellationRequested ();
DoSomething ();
}, token);

// Some time later, cancel:
source.Cancel ();
```

If the source is canceled before the task is scheduled, the task is not scheduled. If the source is canceled after the task has started, it is the user's responsibility to call `token.ThrowIfCancellationRequested` from the *same* token that was passed when creating the task. You can't abort a task the same way you can abort a thread.

### Wrapping .NET Async Results

Any delegate in .NET can be turned into an asynchronous invocation. The result value can be passed to `TaskFactory.FromAsync` to wrap the result from calling the `BeginInvoke` method on the delegate.

Use the `TaskExtensions.Unwrap` extension method to extract a nested task in complex flows, and `TaskCompletionSource` to model complex patterns that can't be expressed directly with `Task` internally, but when the caller needs a `Task` object.

### Exceptions in Tasks

If an exception occurs on a `Task` it must be observed at some stage, otherwise the exception will be launched on the finalizer thread, which can be hard to diagnose.

To observer the exception, just peek at the `task.Exception` property. The `task.Wait` method will throw a `TaskScheduler.UnobservedTaskException` event with an handler for a last chance to mark it observed.

A good practice is to always set up a `TaskContinuationOptions.OnlyOnFaulted` continuation to log the exception, or in the main continuation add code to log it as follows:

```
if (task.Exception != null)
        LogException (task.Exception);
```

## Additional Performance Tips

### Avoid Garbage Collection in Tight Loops

Use object pools to avoid creating garbage that would trigger the garbage collector at runtime. This is particularly relevant for games, which need to create the majority of their objects upfront.

### Use SQLite.NET

Use SQLite.NET for your ORM. The SQLite database runs on Windows, Windows Phone, Android, iOS and OS X. It's high performance, flexible, and easy to use.

### Test on All Platforms

There are slight differences between what .NET supports on the Windows Phone and Windows Tablets as well as limitations on iOS that prevent dynamic code to be generated on the flight. Either plan on testing the code on multiple platforms as you develop it, or schedule time to refactor and update the model part of your application at the end of the project.

# iOS Specific Performance Considerations

## Simulator vs. Device

Begin deploying and testing your app on an actual physical device as early as possible. Simulators do not perfectly match the behaviors and limitations of devices and so it is important to test in a real-world device scenario as early as possible.

In particular the simulator does not in any way simulate the memory or CPU restrictions of a physical device.

## Screen Refresh Rate for Games

Games tend to have tight loops to run the game logic and then update the screen. Some typical frame rates include sixty and thirty updates of the screen per second.

Some developers feel that they should update the screen as many times as possible, combining their game simulation with updates to the screen and might be tempted to go beyond sixty frames per second.

On iOS, updating the screen faster will not make a difference because the display server performs screen updates at most sixty times per second. Updating the screen faster can lead to tearing and micro stuttering. It is best to structure your

code so that screen updates are in sync with the display update. On iOS you can achieve this by using the `MonoTouch.CoreAnimation.CADisplayLink` class, which is a timer suitable for visualization and games that runs a sixty frames per second. It can be configured to skip a number of frames, but keep the number consistent.

## Core Animation Performance Hints

Avoiding transparency in Core Animations will improve its bitmap compositing performance. In general avoid transparent layers and blurred borders if possible.

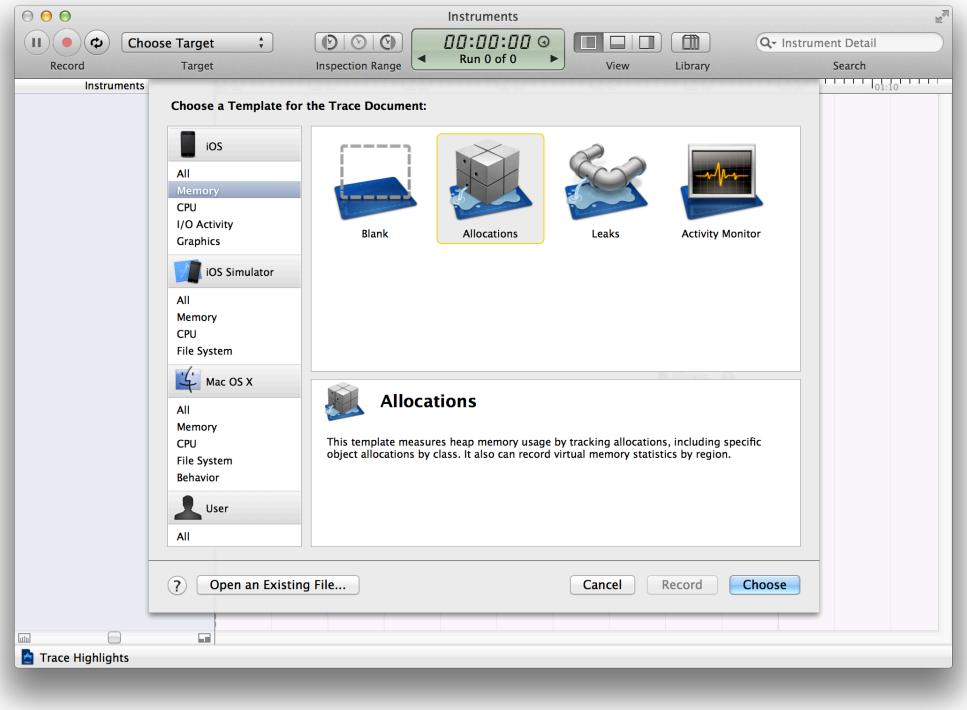
## Avoid Code Generation

Generating code dynamically with `System.Reflection.Emit` or the *Dynamic Language Runtime* must be avoided because the iOS kernel prevents dynamic code execution.

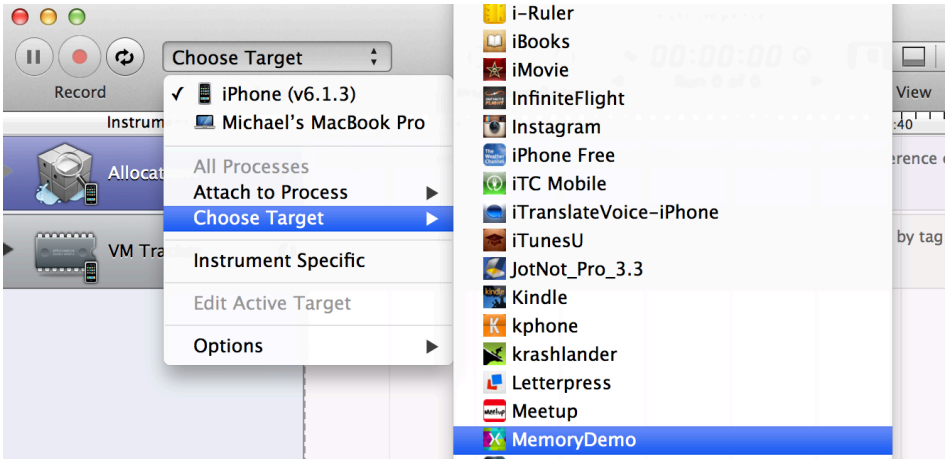
# Diagnostics Demonstration

Let's demonstrate using Instruments to diagnose a memory issue in an iOS application. We'll be working with the *MemoryDemo* sample that accompanies this chapter.

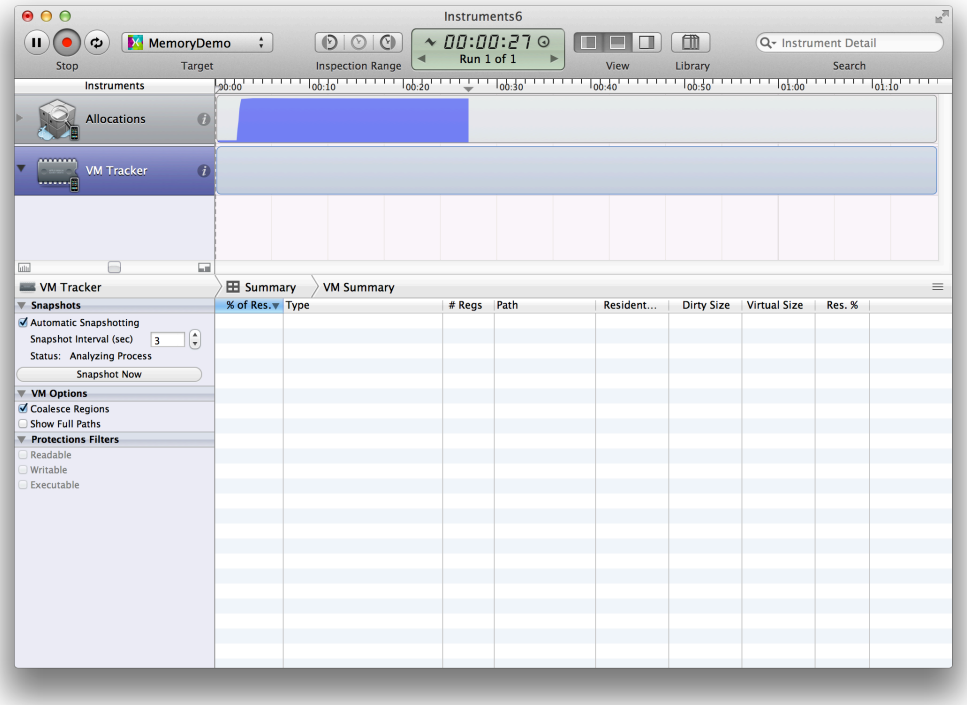
1. From Xamarin Studio, launch Instruments from the **Tools > Launch Instruments** menu item.
2. Upload the application to the device by choosing the **Run > Upload to Device** menu item.
3. Choose **the iOS > Memory > Allocations** template



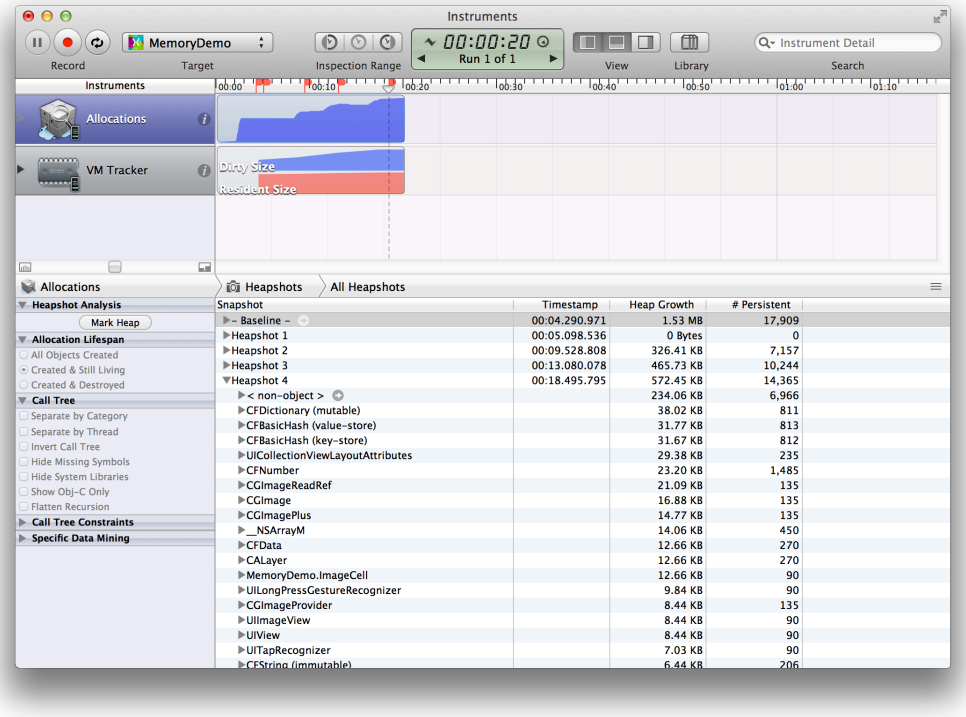

4. Select the application under the Instruments **Choose Target** menu.



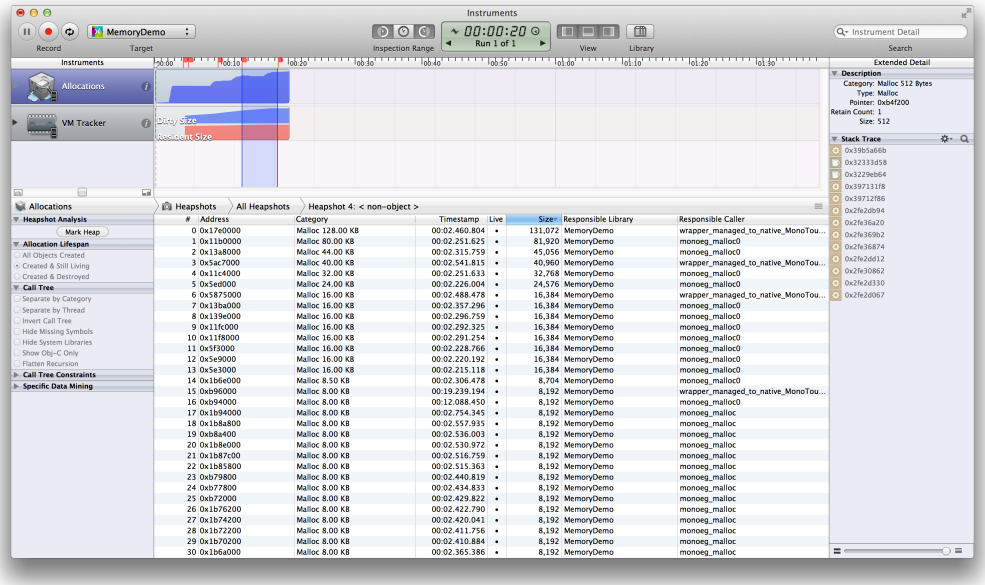

5. Select the **Record** button in Instruments, which will launch the application.
6. Under **VM Tracker**, select the **Automatic Snapshotting** checkbox.



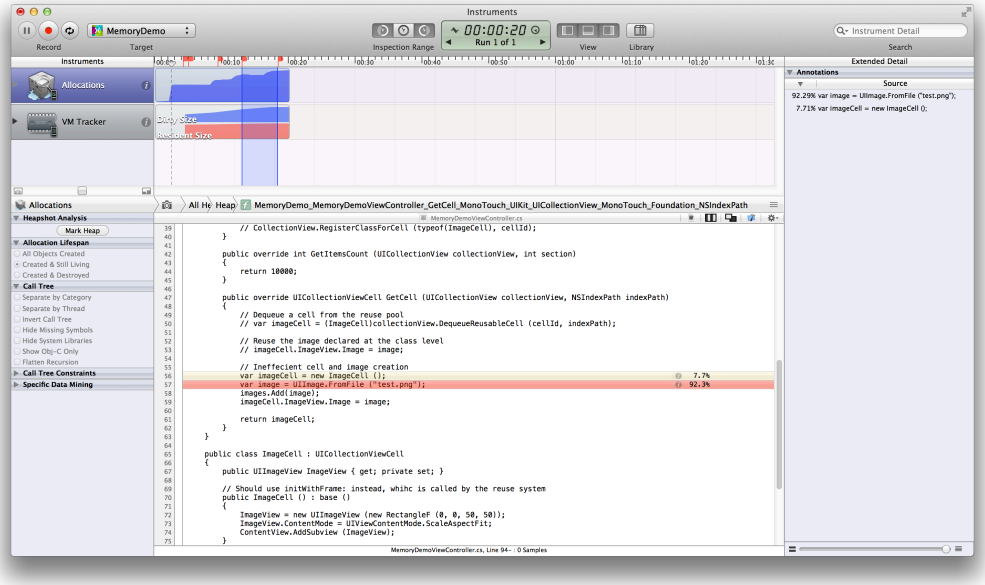

7. Select **Allocations**
8. Under **Heapshot Analysis** click the **Mark Heap** button to establish a baseline.
9. Scroll the application, then select **Mark Heap** again (repeat a few times)
10. Click the **Stop** button.
11. Expand the **Heapshot** node with the largest **Heap Growth** and sort by **Heap Growth**.
12. Notice the **<non-object>** node shows excessive memory growth. Click the arrow next to this node to see more details.




13. Sort by **Size** and display the **Extended Detail** view.




14. Clicking on the desired entry in the call stack to see the related code.




In this case, we are creating a new image for every cell and keeping each instance in a collection at the class level. Also, we are not reusing collection view cells.

Let's resolve these issues and rerun the application through Instruments.

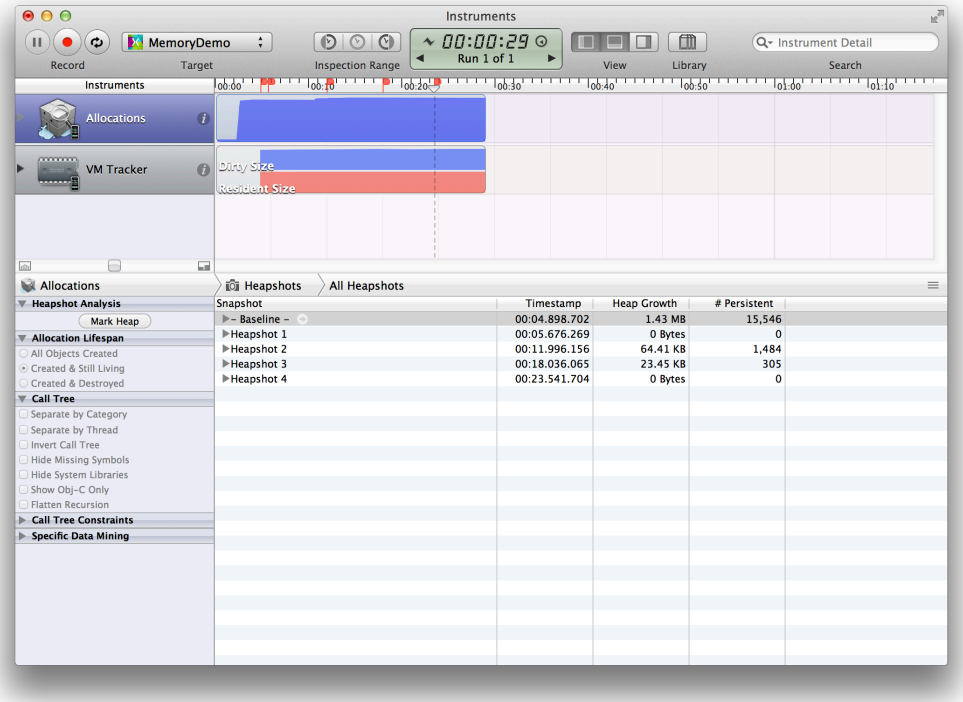
By declaring a single instance at the class level, we can reuse the image. Then we can use the cell reuse pattern with collection views to dequeue cells from a built-in pool rather than creating them every time, as shown below:

```
public override UICollectionViewCell GetCell (UICollectionView
collectionView, NSIndexPath indexPath)
{
        // Dequeue a cell from the reuse pool
        ImageCell imageCell = (ImageCell)collectionView.DequeueReusableCell
(cellId, indexPath);

        // Reuse the image declared at the class level
        imageCell.ImageView.Image = image;

        return imageCell;
}
```

Now, when we run the application, we see that memory usage is greatly reduced.




# Summary

In this chapter we discussed a variety of items to improve memory and performance. We discussed general memory issues when working in garbage collected environment, as well as platform specific items. We also discussed techniques for keeping applications performing with high responsiveness. Finally, we demonstrated how to use Instruments to diagnose memory issues.