Tizen .NET Memory Profiler

Investigating memory leaks

DEXT-Compiler Lab
01/12/2017
Memory Leaking Code

```csharp
void OnStart()
{
    timer1 = new System.Threading.Timer((arg) => {
        String str = prefixTime + System.DateTime.Now.ToString();

        callListener += (bool isResume) => {
            label.Text = (isResume ? prefixResume : prefixSleep) + str;
        }, null, 0, 1);
    }, null, 0, 1);
}
```

There is subtle memory leak in this code. Managed memory profiler helps us to find leak here.
Why another profiler?

- Native profilers
  - Where is unmanaged memory allocated?
- Managed profilers
  - What’s managed memory used for and where is it allocated?
- All profilers:
  - How much memory is used by the application?

Questions:
- What about applications with mixed managed and unmanaged code?
- How much *physical* memory is used by the application?
Mixed code

- In mixed applications, native memory lifetime can be determined by managed code.

```
Managed object --> Native memory
```

- What happens when a managed object is collected, but doesn’t free native memory?
  - Managed memory profilers show no leaks
  - Native profilers detect a memory leak, but don’t provide enough information to eliminate it
Virtual / physical memory

- Memory allocated using standard library and system calls is not application’s physical memory usage.
- Physical memory is allocated when the memory is written to (copy-on-write)
- Memory can be shared with other processes

![Virtual memory diagram]

- Page 0
- Page 1
- Page 2
- Page 3
- Page 4
- ...
- Page N

- Process 1
- Physical memory
- Process 2
### Memory Profilers

- There are a number of memory profiler tools already available

<table>
<thead>
<tr>
<th>Technology</th>
<th>Technology</th>
<th>malloc</th>
<th>managed (.NET)</th>
<th>mmap/munmap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>virtual</td>
</tr>
<tr>
<td>dotMemory</td>
<td>.NET Profiling API</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Valgrind (Massif)</td>
<td>Binary translation (slow)</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>Tizen .NET Memory Profiler</strong></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>KDE HeapTrack</td>
<td>Symbol interception (fast)</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic Analyzer (based on LeakSanitizer)</td>
<td></td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MemProf</td>
<td></td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Tizen .NET Memory Profiler can track managed memory and physical memory consumption
- Tracking managed memory and physical memory consumption is crucial for figuring out and investigating memory leaks in .NET applications
Implementation

Based on open-source KDE Heaptrack, extended to:

- Track managed objects
- Combine managed and native call stacks
- Make snapshots of a managed heap
- Show physical memory consumption
Implementation

KDE Heaptrack architecture:

- A profiler library (tracker) is preloaded into an application, or injected during runtime
- Calls to malloc/calloc/realloc/free are intercepted by the library and sent to a trace encoder
- A compact trace is written to a hard drive by the trace encoder
- Trace analyzer is used by an expert to examine the trace
Implementation

Tizen .NET Memory Profiler architecture:

- Managed profiler receives callbacks upon object allocation, GC start/stop, function enter/leave, object movement by GC, and forwards them to the tracker.
- Tracker combines native and managed call stacks, providing complete call site data for both managed and native allocations, tracks managed object allocations, movements and deallocations in the heap.
- Extended encoder writes managed allocations, call stacks and heap snapshots to the trace.
Implementation

- Calls to `mmap`, `mmap64`, `munmap` are intercepted by the tracker and their call sites are recorded.
- Tracker periodically reads `/proc/self/smaps`. Whenever a new physically mapped (private dirty) region is found, a corresponding call site is located and the data is written to the trace.
Memory Leaking Code (example 1)

```csharp
void OnStart()
{
    timer1 = new System.Threading.Timer((arg) => {
        String str = prefixTime + System.DateTime.Now.ToString();

        callListener += (bool isResume) =>
        {
            label.Text = (isResume ? prefixResume : prefixSleep) + str;
        }, null, 0, 1);
}
```

There is unnoticeable memory leak issue in this code. Managed memory profiler helps us to find leak here.
Managed memory consumption chart

Here we catch several functions, for which memory consumption continuously increases over time. This is an usual indicator of a memory leak.

Total memory consumption increases over time

Declines are because of garbage collections

Putting mouse over graph shows name of consuming function

Memory consumption of the function increases over time

Xamarin.App.addCallListener

Xamarin.App.<OnStart>b__11_0
Memory Leaking Code (example 2)

class Graph : System.IDisposable
{
    private int w, h;

    [System.Runtime.InteropServices.DllImport("native-graphics.so", EntryPoint = "Draw")]
    static extern void Draw(int width, int height);

    [System.Runtime.InteropServices.DllImport("native-graphics.so", EntryPoint = "FreeMemory")]
    static extern void FreeMemory();

    public Graph(int width, int height) { w = width; h = height; }
    public void DrawPicture() { Draw(w, h); }
    public void Dispose() { FreeMemory(); }
}

void OnDraw()
{
    Graph graph = new Graph(this.width, this.height);
    graph.DrawPicture();

    // graph.Dispose is not called; and `using` construction is not used also
}

The unmanaged memory is allocated by the `malloc` in unmanaged code called from DrawPicture.

It should be freed here by calling Dispose method.

Because the Dispose method is not called, unmanaged memory is never freed in this case.
Managed memory consumption

Total managed memory consumption doesn't increase over time. Declines are because of garbage collections, as in the first example. However, here managed part is collected without problems.
Unmanaged memory consumption

The total unmanaged memory consumption increases.

This part remains the same over time.

This part increases.

Putting mouse over this part shows us that it is the Graph.DrawPicture function.
Call stacks for memory allocations

- This chart shows us where exactly the leaking allocation came from

Unmanaged memory that is allocated through this `malloc` is never freed. It leaks.

Here is the call stack for Graph::DrawPicture function.

We see that DrawPicture is called from App::OnDraw handler.

The DrawPicture then invokes unmanaged code that allocates unmanaged memory through `malloc`.

<table>
<thead>
<tr>
<th>malloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph::DrawPicture</td>
</tr>
<tr>
<td>XamarinApplication::App::OnDraw</td>
</tr>
<tr>
<td>XamarinApplication::App::Main</td>
</tr>
<tr>
<td>&lt;unresolved function&gt;</td>
</tr>
</tbody>
</table>

3.4 MB (0.15%) contribution to peak consumption

2.9 MB (0.15%) contribution to peak consumption in malloc and below.
Summary

- The **Tizen .NET Memory Profiler** is a prototype memory profiling solution that is capable of providing the baseline necessary information for detecting and investigating memory leaks in .NET applications.

- Key features:
  - Track managed and unmanaged memory
  - Track physical memory consumption

- The solution is based on KDE HeapTrack, which is fast and stable project with more than four-years history.
Short-term plans

- Add more descriptive information about location in managed code
  - Display file names and line numbers where possible
  - Display function argument lists

- Improve profiling performance
  - Current performance slowdown is ~20-100x compared to running without profiler.
THANK YOU!
감사합니다!
СПАСИБО!
## Heap snapshot

<table>
<thead>
<tr>
<th>Instances</th>
<th>Shallow Size</th>
<th>Referenced Size</th>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>▼ 3378</td>
<td>116.4 KB</td>
<td>3.1 MB</td>
<td>[System.String]</td>
</tr>
<tr>
<td>▼ 2867</td>
<td>44.8 KB</td>
<td>2.6 MB</td>
<td>[XamarinApplication.ClockView]</td>
</tr>
<tr>
<td>▼ 1</td>
<td>16.0 KB</td>
<td>2.6 MB</td>
<td>[XamarinApplication.ClockView[]]</td>
</tr>
<tr>
<td>▼ 1</td>
<td>16 B</td>
<td>910.1 KB</td>
<td>[XamarinApplication.ClockModel]</td>
</tr>
<tr>
<td></td>
<td>16 B</td>
<td>260.0 KB</td>
<td>[XamarinApplication.App]</td>
</tr>
<tr>
<td>1</td>
<td>0 B</td>
<td>260.0 KB</td>
<td>&lt;gcroot&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0 B</td>
<td>650.0 KB</td>
<td>&lt;gcroot&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0 B</td>
<td>1.7 MB</td>
<td>&lt;gcroot&gt;</td>
</tr>
<tr>
<td>▼ 1</td>
<td>16 B</td>
<td>96 B</td>
<td>[XamarinApplication.App]</td>
</tr>
<tr>
<td>1</td>
<td>0 B</td>
<td>96 B</td>
<td>&lt;gcroot&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0 B</td>
<td>136 B</td>
<td>&lt;gcroot&gt;</td>
</tr>
<tr>
<td>▼ 3</td>
<td>5.5 KB</td>
<td>13.6 KB</td>
<td>[System.Object[]]</td>
</tr>
<tr>
<td>▼ 186</td>
<td>2.2 KB</td>
<td>452.9 KB</td>
<td>[XamarinApplication.Label]</td>
</tr>
<tr>
<td>▼ 186</td>
<td>2.9 KB</td>
<td>452.9 KB</td>
<td>[XamarinApplication.ClockView]</td>
</tr>
<tr>
<td>▼ 1</td>
<td>16.0 KB</td>
<td>452.9 KB</td>
<td>[XamarinApplication.ClockView[]}</td>
</tr>
<tr>
<td>▼ 1</td>
<td>24 B</td>
<td>452.9 KB</td>
<td>[System.Collections.Generic.List`1]</td>
</tr>
<tr>
<td>▼ 1</td>
<td>16 B</td>
<td>154.7 KB</td>
<td>[XamarinApplication.ClockModel]</td>
</tr>
<tr>
<td>1</td>
<td>0 B</td>
<td>298.3 KB</td>
<td>&lt;gcroot&gt;</td>
</tr>
</tbody>
</table>
Managed memory by type

Heaptrack - res.gz — Heaptrack GUI

File

Summary  Bottom-Up  Caller / Callee  Top-Down  Managed Heap  Flame Graph  Consumed  Instances  Allocations  Allocated  Sizes

Filter by function...  filter by file...  filter by module...

Peak  Peak instances  Leaked  Allocations  Allocated  Location

- 85.8 KB  427  21  2160.7 KB  (System.String) in ??()
- 34.1 KB  11  11  709.1 KB  (System.Char) in ??()
- 33.6 KB  10  10  33.6 KB  (System.Object) in ??()
- 15.0 KB  66  66  15.0 KB  (System.Reflection.CustomAttributeRecord) in ??()
- 9.5 KB  162  162  9.5 KB  (System.Reflection.AssemblyName) in ??()
- 8.1 KB  149  149  8.1 KB  (System.Globalization.CultureInfo) in ??()
- 7.2 KB  308  308  7.2 KB  (System.Version) in ??()
- 6.9 KB  312  312  6.9 KB  (System.Byte) in ??()
- 6.0 KB  153  153  6.0 KB  (Xamarin.Forms.Color) in ??()
- 5.8 KB  67  67  5.8 KB  (System.Int32) in ??()
- 4.0 KB  203  203  4.0 KB  (System.RuntimeType) in ??()
- 3.6 KB  62  62  3.6 KB  (Xamarin.FormsBindableProperty) in ??()
- 2.4 KB  5  5  2.4 KB  (System.Collections.Generic.Dictionary) 2 Entry in ??()
- 2.0 KB  46  46  2.0 KB  (System.RuntimeMethodHandle) in ??()
- 1.2 KB  22  22  1.2 KB  (Xamarin.Forms.Platform.Tizen.ExportRenderAttribute) in ??()
- 1.1 KB  22  22  1.1 KB  (Xamarin.Forms.DependencyAttribute) in ??()
- 1.1 KB  22  22  1.1 KB  (Xamarin.Forms.Platform.Tizen.ExportCellAttribute) in ??()
- 1.1 KB  22  22  1.1 KB  (Xamarin.Forms.Platform.Tizen.ExportImageSourceHandlerAttribute) in ??()
- 1.1 KB  10  10  1.1 KB  (System.Runtime.CompilerServices.ConditionalWeakTable) 2 Entry in ??()
- 1.1 KB  22  22  1.1 KB  (Xamarin.Forms.ExportEffectAttribute) in ??()
- 1.0 KB  33  33  1.0 KB  (Xamarin.Forms.BindablePropertyChangedDelegate) in ??()
- 1.0 KB  10  10  1.0 KB  (System.Reflex.AssemblyName) in ??()
- 692 B  27  27  788 B  (System.Type) in ??()
- 648 B  7  7  648 B  (System.Collections.Generic.HashSet) 1 Slot in ??()
- 576 B  20  20  576 B  (System.String) in ??()
- 576 B  12  12  576 B  (Xamarin.Forms.Style) in ??()
- 552 B  46  46  552 B  (System.IntPtr) in ??()
- 460 B  15  15  460 B  (System.EventHandler) in ??()
- 488 B  3  3  488 B  (System.Collections.Generic.Dictionary) 2 Entry in ??()
- 468 B  3  3  468 B  (System.Collections.Generic.Dictionary) 2 Entry in ??()
- 548 B  5  5  548 B  (System.Collections.Generic.Dictionary) 2 Entry in ??()
- 364 B  9  9  364 B  (System.String) in ??()

Backtrace
[System.String] in ??()
Calculator.CalculatorOnCreate in ...
Calculator.Tizen.Program.Create in ...
Tizen.Applications.CoreBlackend.UIControlApplication.Run in ...
Tizen.Applications.CoreApplication.Run in ...
Tizen.Applications.CoreUIApplication.Run in ...
Calculator.Tizen.Program.Main in ??()
Managed flame graph

```
[System.Char[]]
System.Text.StringBuilder
System.Text.StringBuilder [System.String]
System.IO.StreamReader.ReadToEnd
Xamarin.Forms.Xaml.XamlLoader.XamlResourceAskam
Xamarin.Forms.Xaml.XamlLoader.XamlForType
Calculator.Views.CalculatorMainPage.InitializeComponent
Calculator.Views.CalculatorMainPage..ctor
Calculator.Calculator.OnOrientationChanged Xa..
Calculator.Tizen.Program.OnCreate
Tizen.Applications.CoreBackend.UICoreBackend.OnCreateNative
Tizen.Applications.CoreBackend.UICoreBackend.Run
Tizen.Applications.CoreApplication.Run
Calculator.Tizen.Program.Main
```

256.7 KB contribution to peak consumption
Consumed managed memory