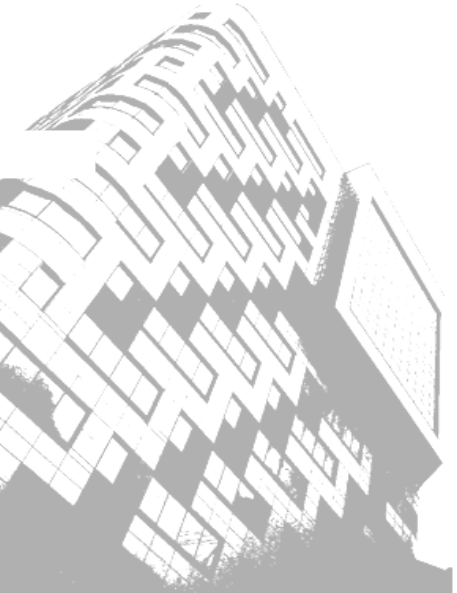
SAMSUNG

Tizen .NET Memory Profiler

Investigating memory leaks



DEXT-Compiler Lab

01/12/2017

Memory Leaking Code

```
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 subtle memory leak in this code. Managed memory profiler helps us to find leak here.

Why another profiler?

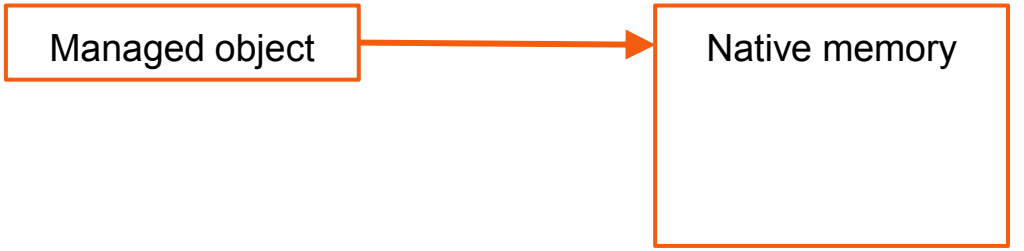
- Native profilers
  - Where is unmanaged memory allocated?
- Managed profilers
  - What's managed memory used for and where is it allocated?
- All profilers:
  - How much memory is used by the application?

Questions:

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

Mixed code

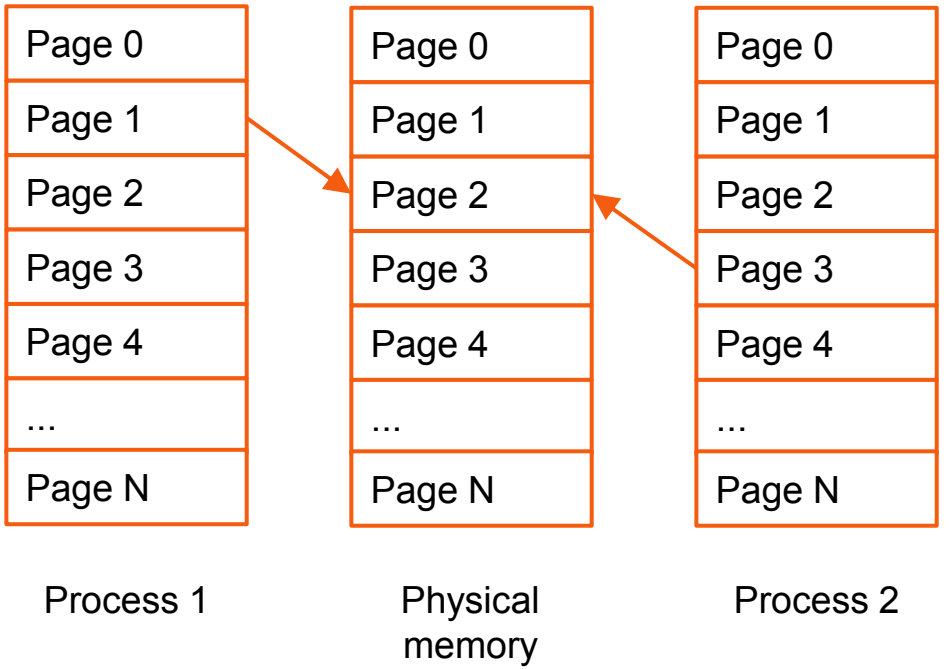
- In mixed applications, native memory lifetime can be determined by managed code.



- 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



Memory Profilers

- There are a number of memory profiler tools already available

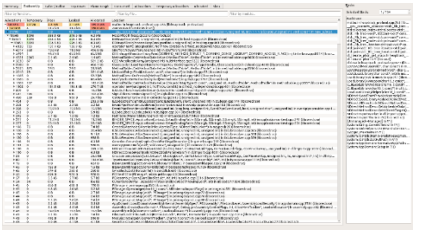
	Technology	malloc	managed (.NET)	mmap/munmap	
				virtual	physical
dotMemory	.NET Profiling API	-	+	-	-
Valgrind (Massif)	Binary translation (slow)	+	-	+	-
Tizen .NET Memory Profiler		+	+	+	+
KDE HeapTrack	Symbol interception (fast)	+	-	-	-
Dynamic Analyzer (based on LeakSanitizer)		+	-	-	-
MemProf		+	-	-	-

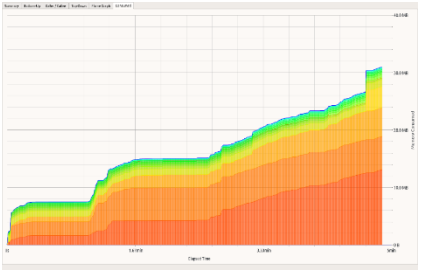
- Tizen .NET Memory Profiler can track managed memory and physical memory consumption
- Tracking managed memory and physical memory consumption is crucial for figuring out and investigating memory leaks in .NET applications

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

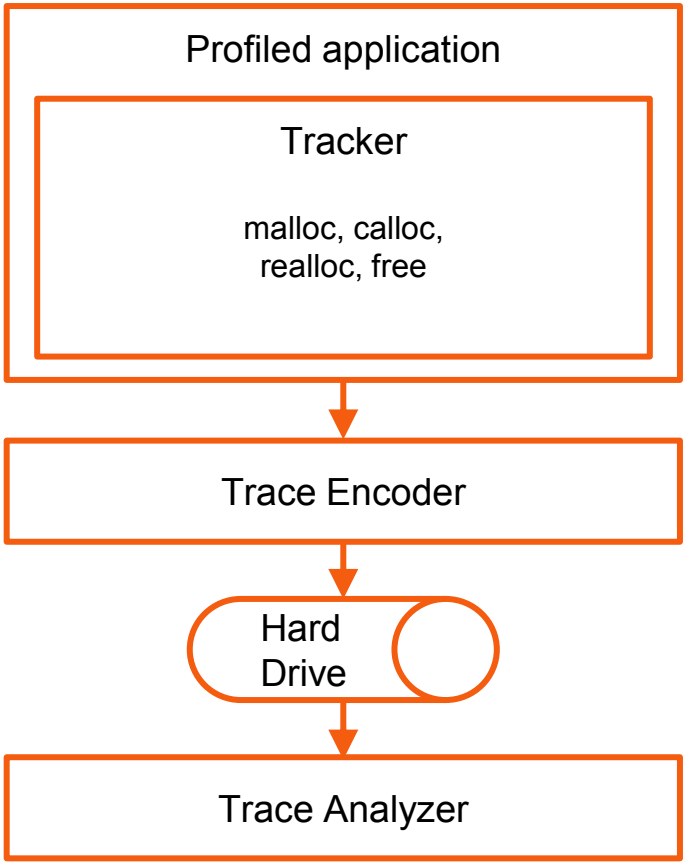




Implementation

KDE Heaptrack architecture:

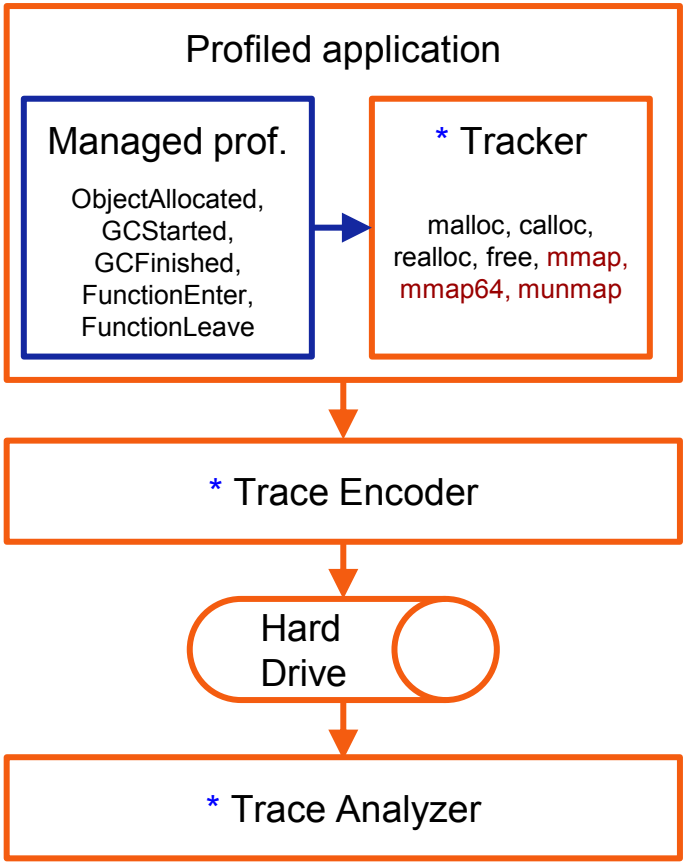
- A profiler library (tracker) is preloaded into an application, or injected during runtime
- Calls to malloc/calloc/realloc/free are intercepted by the library and sent to a trace encoder
- A compact trace is written to a hard drive by the trace encoder
- Trace analyzer is used by an expert to examine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



Implementation

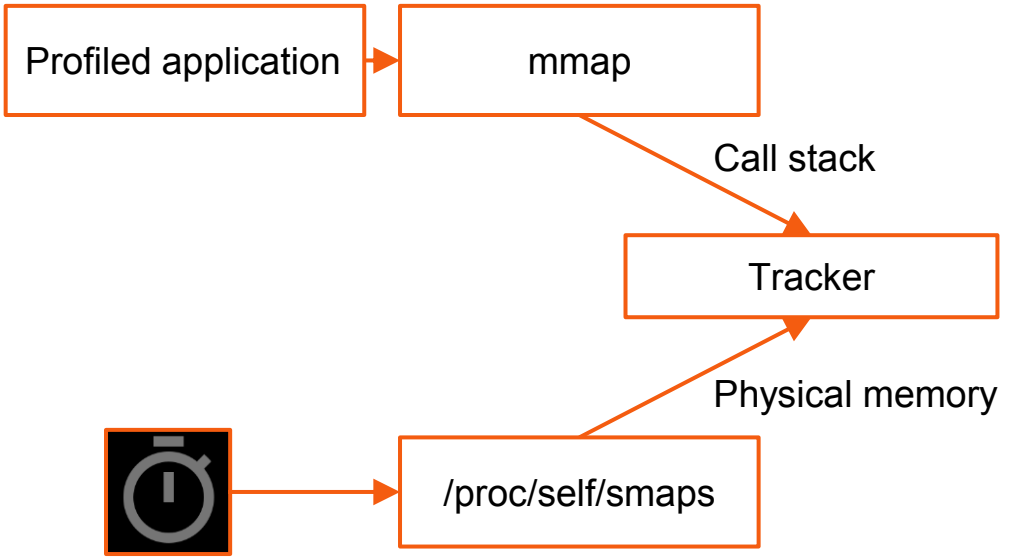
Tizen .NET Memory Profiler architecture:

- 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



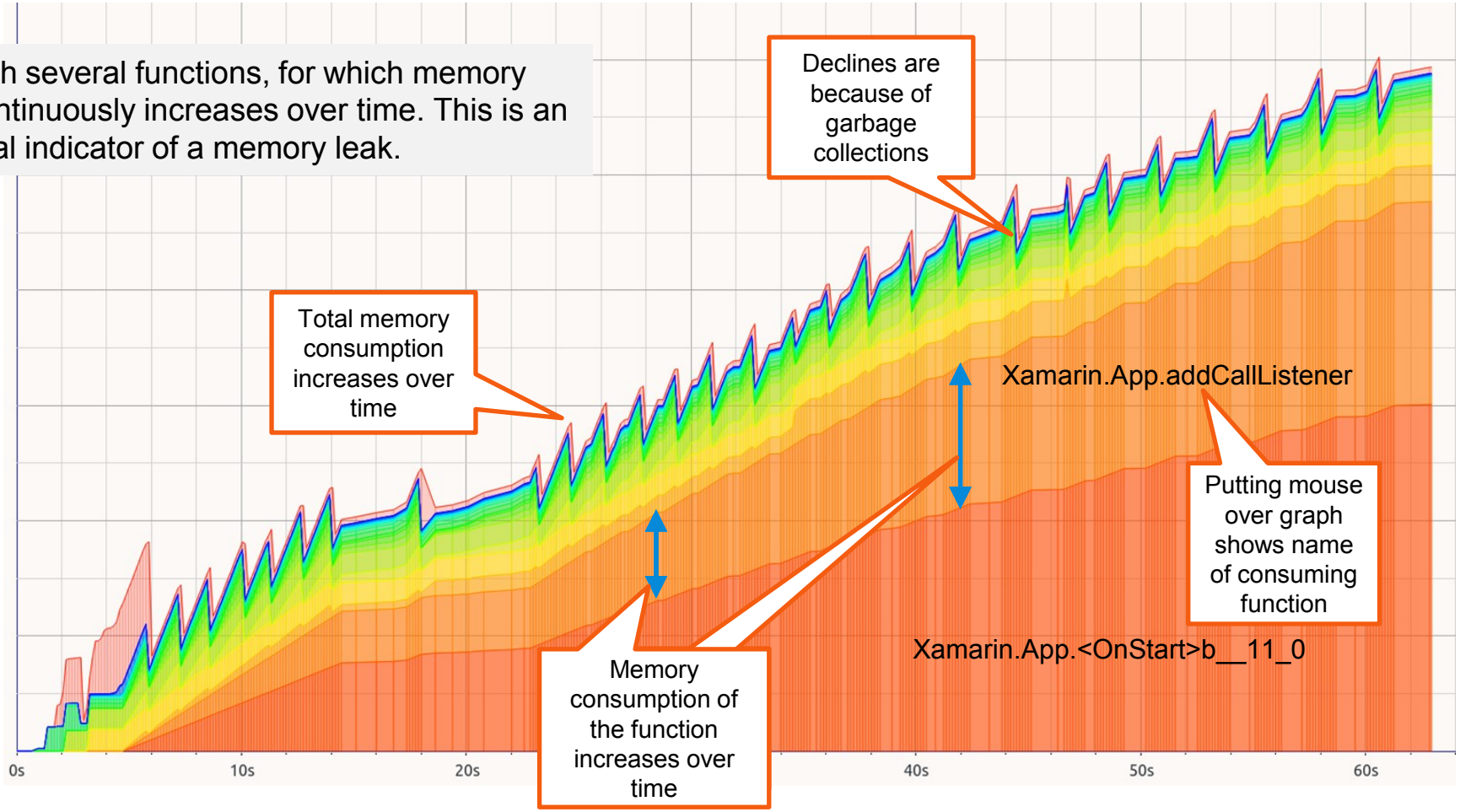
Memory Leaking Code (example 1)

```
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.



Memory Leaking Code (example 2)

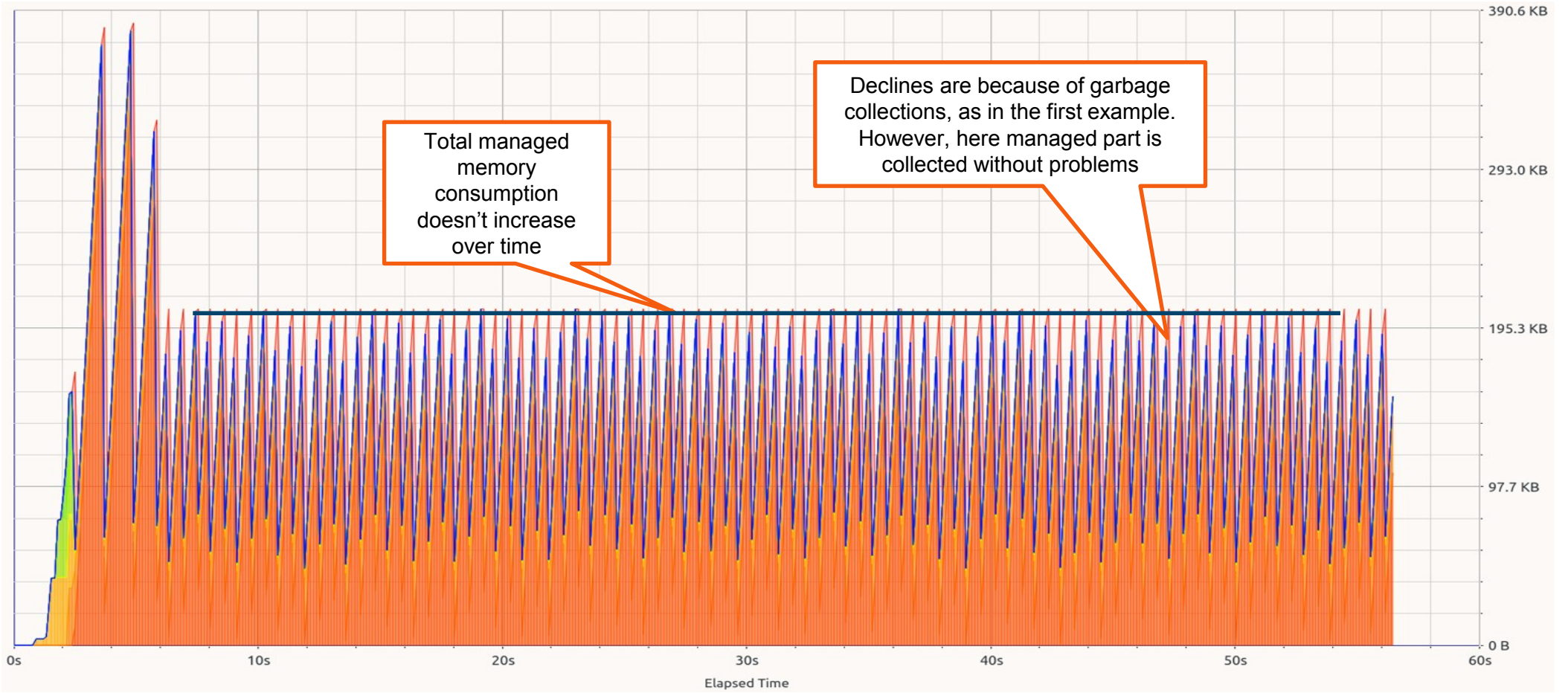
```
class Graph : System.IDisposable
{
    private int w, h;

    [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

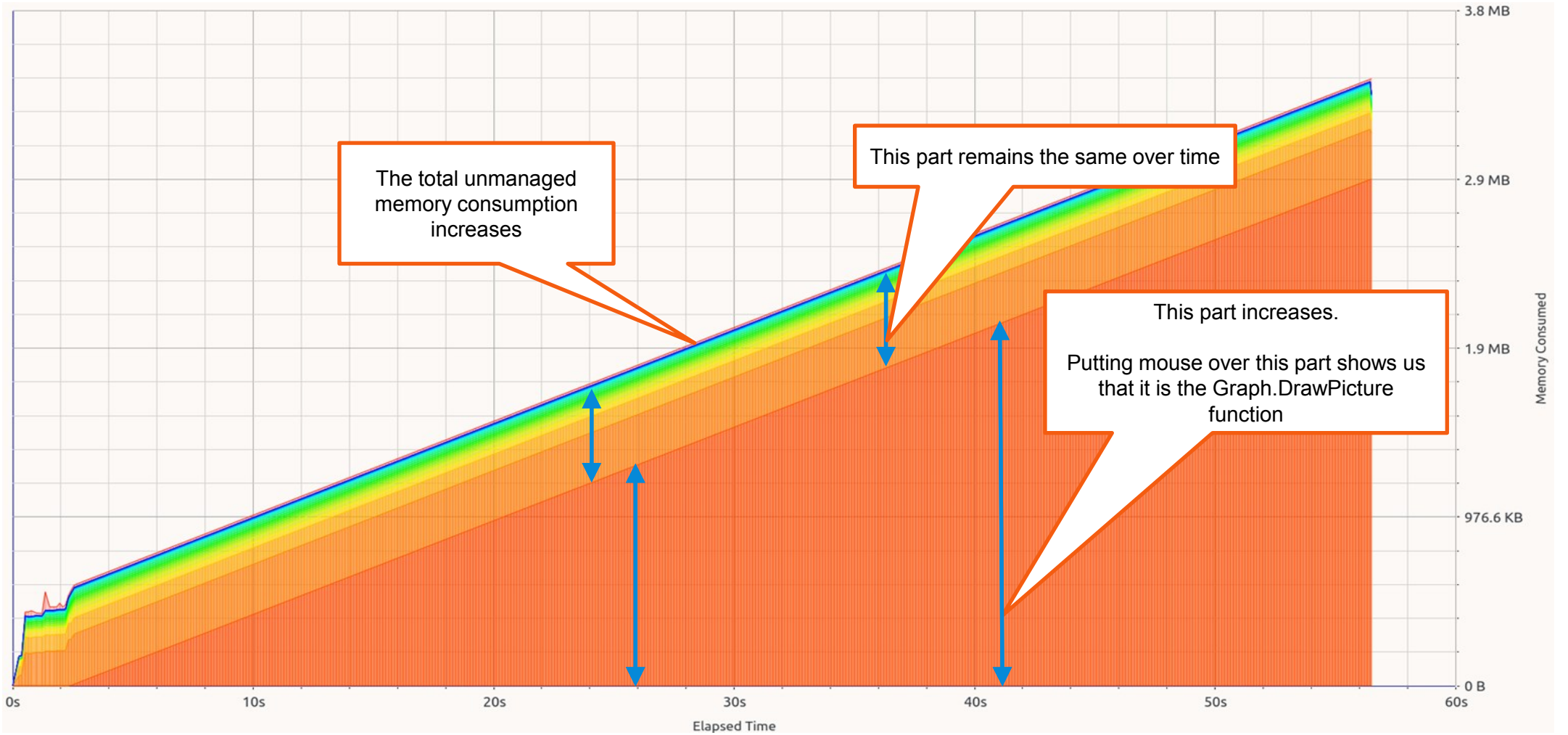
It should be freed here by calling Dispose method

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

Managed memory consumption

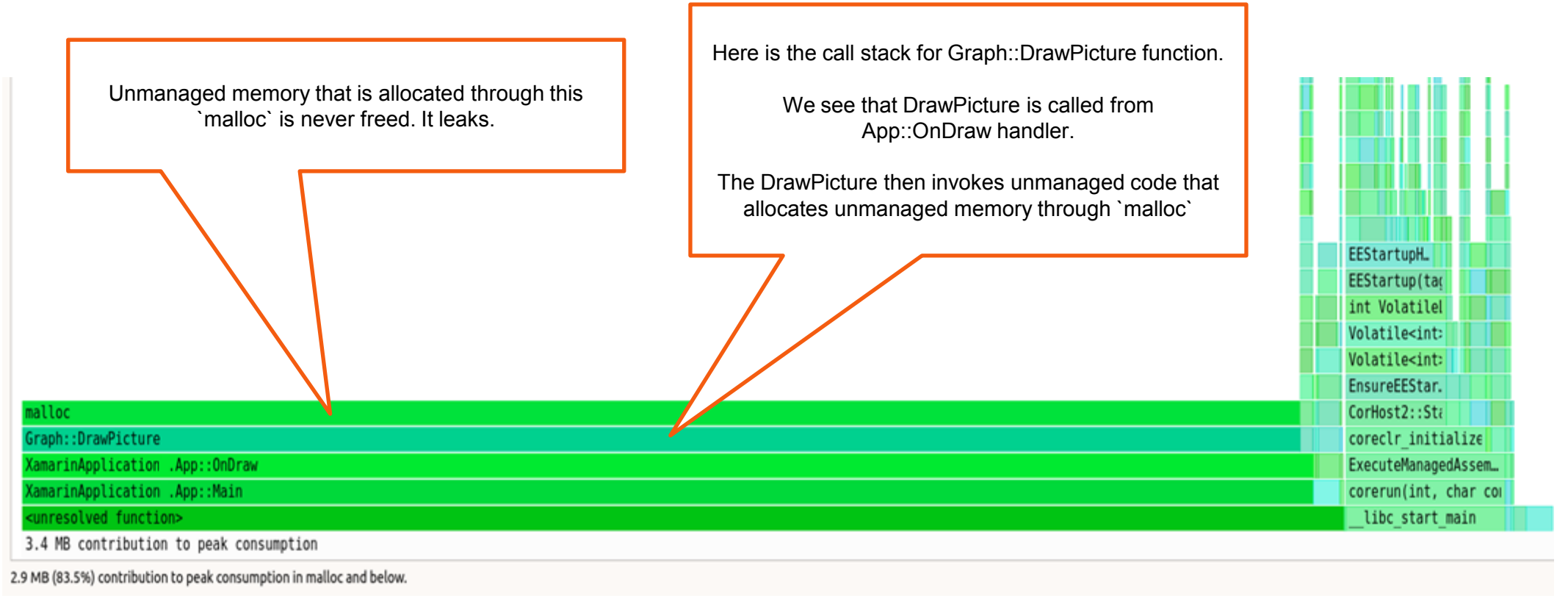


Unmanaged memory consumption



Call stacks for memory allocations

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



Summary

- The **Tizen .NET Memory Profiler** is 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!

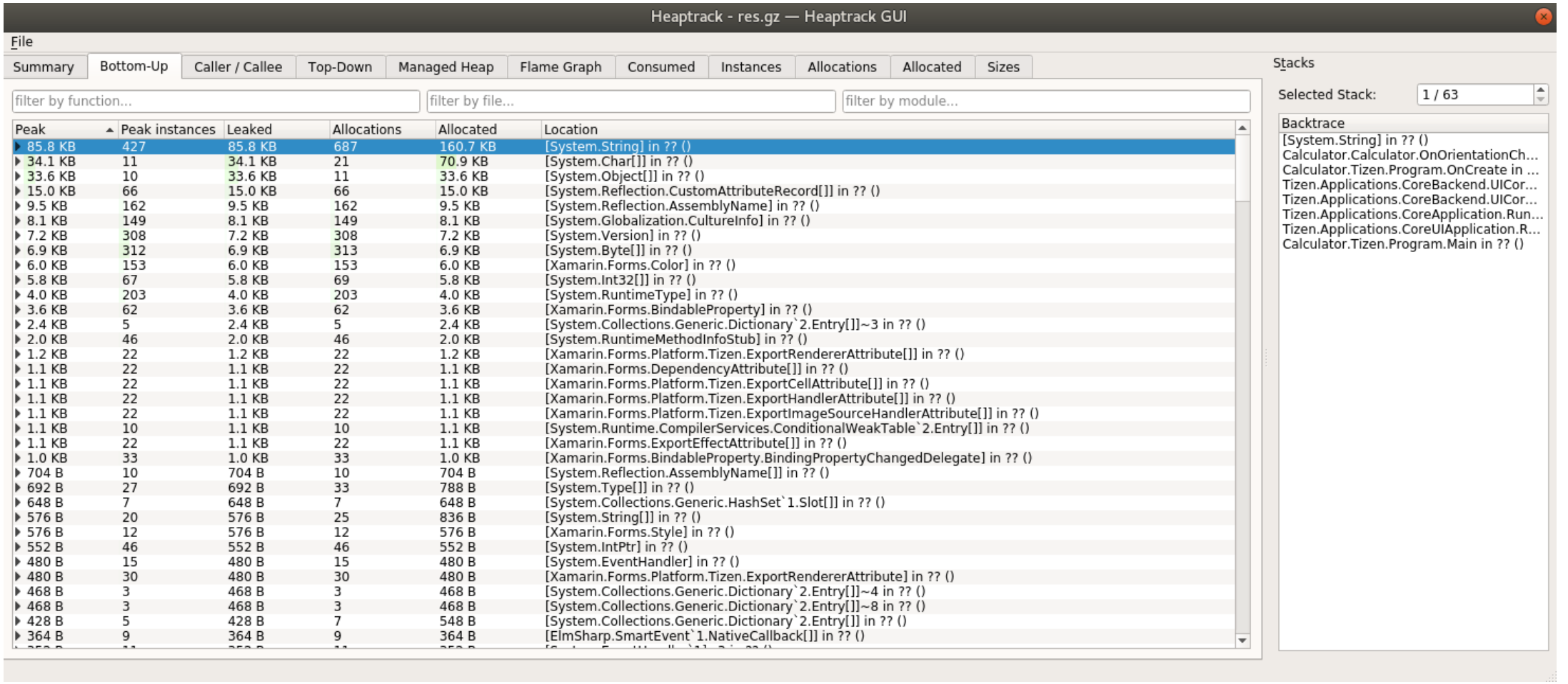
СПАСИБО!

SAMSUNG

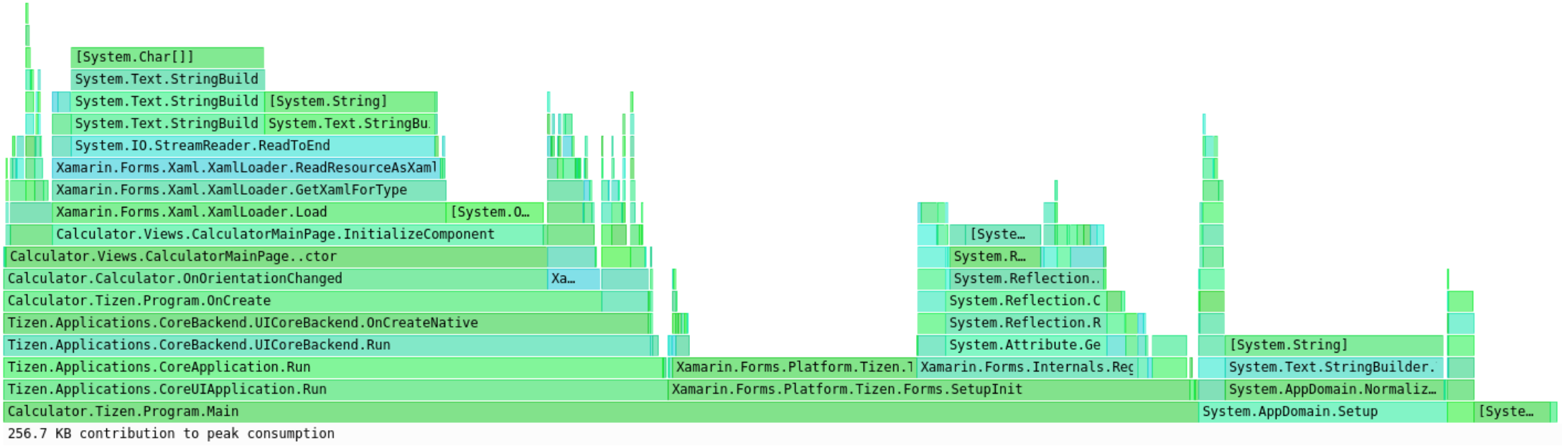
Heap snapshot

Instances	Shallow Size	Referenced Si:	Class Name
▼ 3378	116.4 KB	3.1 MB	[System.String]
▼ 2867	44.8 KB	2.6 MB	[XamarinApplication.ClockView]
▼ 1	16.0 KB	2.6 MB	[XamarinApplication.ClockView[]]
▼ 1	24 B	2.6 MB	[System.Collections.Generic.List`1]
▼ 1	16 B	910.1 KB	[XamarinApplication.ClockModel]
▼ 1	16 B	260.0 KB	[XamarinApplication.App]
1	0 B	260.0 KB	<gcroot>
1	0 B	650.0 KB	<gcroot>
1	0 B	1.7 MB	<gcroot>
▼ 1	16 B	96 B	[XamarinApplication.App]
1	0 B	96 B	<gcroot>
1	0 B	136 B	<gcroot>
▶ 3	5.5 KB	13.6 KB	[System.Object[]]
▼ 186	2.2 KB	452.9 KB	[XamarinApplication.Label]
▼ 186	2.9 KB	452.9 KB	[XamarinApplication.ClockView]
▼ 1	16.0 KB	452.9 KB	[XamarinApplication.ClockView[]]
▼ 1	24 B	452.9 KB	[System.Collections.Generic.List`1]
▶ 1	16 B	154.7 KB	[XamarinApplication.ClockModel]
1	0 B	298.3 KB	<gcroot>

Managed memory by type



Managed flame graph



Consumed managed memory