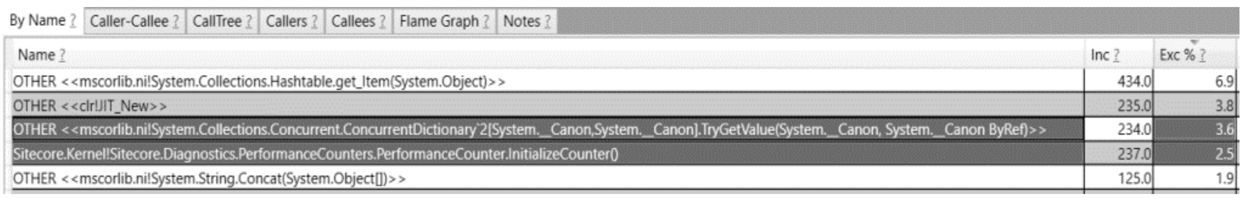
N. Mitikov, N. Guk, Yu. Honcharova

REAL-WORLD EXAMPLE OF APPLICATION PERFORMANCE ANOMALY DETECTION THROUGH MEMORY ANALYSIS

The growing complexity of modern software applications has led to the emergence of various performance-related issues that significantly degrade application performance and negatively impact user experience. The constant evolution of software applications and hardware architectures has led to a diverse range of memory behaviors, complicating the development of a one-size-fits-all solution. Given that just-in-time (JIT) compilation can effectively convert source code [1] into machine-specific instructions, it is plausible to reverse process by comprehensively capturing and analyzing the memory contents associated with the execution of the application. As memory snapshots encompass thread-associated data structures, they inherently contain valuable information pertaining to the operations executed by the respective threads, thus providing insights into the runtime behavior of an application.

The Event Tracing [2] technique, integrated into the operating system kernel, can be activated to generate detailed information that reveals the activities of each thread at millisecond granularity. Consequently, this allows for a comprehensive understanding of time distribution within the scope of various operations, thereby enabling the analysis of application performance and potential bottlenecks.

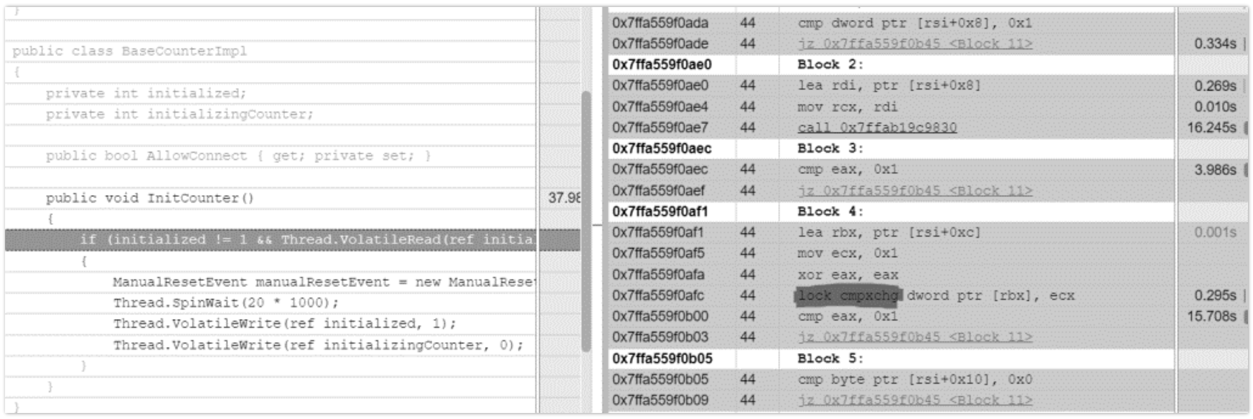
The real-life event tracing from underperforming system has revealed that 2.5% of exclusive CPU time is spent on 'InitializeCounters' method:



Name ?	Inc ?	Exc % ?
OTHER <<mscorlib.ni!System.Collections.Hashtable.get_Item(System.Object)>>	434.0	6.9
OTHER <<clr!JIT_New>>	235.0	3.8
OTHER <<mscorlib.ni!System.Collections.Concurrent.ConcurrentDictionary`2[System.__Canon,System.__Canon].TryGetValue(System.__Canon, System.__Canon ByRef)>>	234.0	3.6
Sitecore.Kernel!Sitecore.Diagnostics.PerformanceCounters.PerformanceCounter.InitializeCounter()	237.0	2.5
OTHER <<mscorlib.ni!System.String.Concat(System.Object[])>>	125.0	1.9

Fig. 1 – CPU event tracing holds Diagnostics in TOP 5 most CPU-consuming functions

Upon conducting reverse-engineering of the method body [3] from memory snapshot [4], it becomes evident that the implementation employs a thread-safe application programming interface (API) despite the absence of any technical necessity for such an approach. Upon recreating the model and processing it through Intel VTune profiler the ‘LOCK CMPXCHG’ instruction is seen to consume the most of time:



```
}

public class BaseCounterImpl
{
    private int initialized;
    private int initializingCounter;

    public bool AllowConnect { get; private set; }

    public void InitCounter()                                              37.98
    {
        if (initialized != 1 && Thread.VolatileRead(ref initia
        {
            ManualResetEvent manualResetEvent = new ManualRese
            Thread.SpinWait(20 * 1000);
            Thread.VolatileWrite(ref initialized, 1);
            Thread.VolatileWrite(ref initializingCounter, 0);
        }
    }
}
```

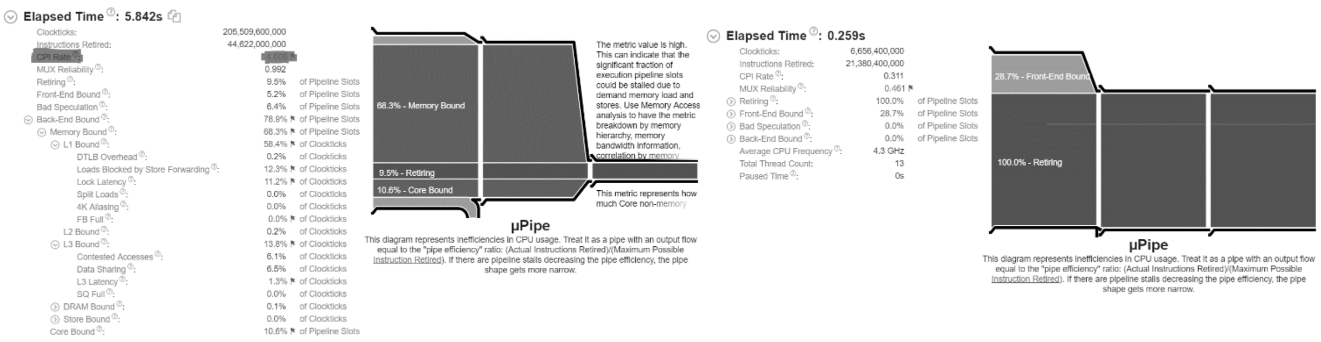
```
0x7ffa559f0ada   44   cmp dword ptr [rsi+0x8], 0x1
0x7ffa559f0ade   44   jz 0x7ffa559f0b45 <Block 11>             0.334s
0x7ffa559f0ae0        Block 2:
0x7ffa559f0ae0   44   lea rdi, ptr [rsi+0x8]                   0.269s
0x7ffa559f0ae4   44   mov rcx, rdi                             0.010s
0x7ffa559f0ae7   44   call 0x7ffab19c9830                     16.245s
0x7ffa559f0aec        Block 3:
0x7ffa559f0aec   44   cmp eax, 0x1                             3.986s
0x7ffa559f0aef   44   jz 0x7ffa559f0b45 <Block 11>
0x7ffa559f0af1        Block 4:
0x7ffa559f0af1   44   lea rbx, ptr [rsi+0xc]                   0.001s
0x7ffa559f0af5   44   mov ecx, 0x1
0x7ffa559f0afa   44   xor eax, eax
0x7ffa559f0afc   44   lock cmpxchg dword ptr [rbx], ecx        0.295s
0x7ffa559f0b00   44   cmp eax, 0x1                            15.708s
0x7ffa559f0b03   44   jz 0x7ffa559f0b45 <Block 11>
0x7ffa559f0b05        Block 5:
0x7ffa559f0b05   44   cmp byte ptr [rsi+0x10], 0x0
0x7ffa559f0b09   44   jz 0x7ffa559f0b45 <Block 11>
```

Fig. 2 – Exclusive access involving cache coherency across all cores

The execution of the LOCK CMPXCHG instruction [5] in Intel processors has a direct impact on the processor's L1 and L2 caches. When the LOCK prefix is used in conjunction with the CMPXCHG instruction, it guarantees exclusive access to the memory location involved in the operation.

To ensure mutual exclusion, the processor must enforce cache coherency across all cores which may involve invalidating or updating the cache lines in other cores' L1 and L2 caches that hold the targeted memory location. Consequently, the LOCK CMPXCHG instruction may result in increased cache coherency traffic, leading to performance implications such as increased latency and reduced throughput.

The conditions order change to firstly check if counter is allowed to be initialized resulted in over 22 times execution speed improvement from 5.8 seconds to 0.26 seconds according to Intel VTune profiler results:



```
Elapsed Time: 5.842s
    Clockticks:                                  205,509,600,000
    Instructions Retired:                         44,622,000,000
    [redacted]                                   [redacted]
    MUX Reliability:                                       0.992
    Retiring:                                              9.5%    of Pipeline Slots
    Front-End Bound:                                       5.2%    of Pipeline Slots
    Bad Speculation:                                       6.4%    of Pipeline Slots
  Back-End Bound:                                         78.9% ⚑  of Pipeline Slots
      Memory Bound:                                       68.3% ⚑  of Pipeline Slots
          L1 Bound:                                       58.4% ⚑  of Clockticks
              DTLB Overhead:                               0.2%    of Clockticks
              Loads Blocked by Store Forwarding:          12.3% ⚑  of Clockticks
              Lock Latency:                               11.2% ⚑  of Clockticks
              Split Loads:                                 0.0%    of Clockticks
              4K Aliasing:                                 0.0%    of Clockticks
              FB Full:                                     0.0% ⚑  of Clockticks
          L2 Bound:                                        0.2%    of Clockticks
          L3 Bound:                                       13.8% ⚑  of Clockticks
              Contested Accesses:                          6.1%    of Clockticks
              Data Sharing:                                6.5%    of Clockticks
              L3 Latency:                                  1.3% ⚑  of Clockticks
              SQ Full:                                     0.0%    of Clockticks
          DRAM Bound:                                      0.1%    of Clockticks
          Store Bound:                                     0.0%    of Clockticks
      Core Bound:                                         10.6% ⚑  of Pipeline Slots
```

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory

This metric represents how much Core non-memory

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

```
Elapsed Time: 0.259s
    Clockticks:                     6,656,400,000
    Instructions Retired:          21,380,400,000
    CPI Rate:                               0.311
    MUX Reliability:                        0.461 ⚑
  Retiring:                                100.0%    of Pipeline Slots
  Front-End Bound:                          28.7%    of Pipeline Slots
  Bad Speculation:                           0.0%    of Pipeline Slots
  Back-End Bound:                            0.0%    of Pipeline Slots
    Average CPU Frequency:                 4.3 GHz
    Total Thread Count:                         13
    Paused Time:                                0s
```

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Fig. 3 – Visible performance difference

Conclusion

The research was performed by combining event tracing and memory snapshot sources.

Event tracing has highlighted the most CPU-consuming methods, while memory snapshot has supplied executed code as well as thread information.

Both original candidate code versions were benchmarked by Intel VTune profiler to collect the execution statistics.

REFERENCES

1. Konrad Kokosa, Pro .NET Memory Management, For Better Code, Performance, and Scalability, 2018 ISBN 978-1-4842-4026-7
2. Brendan Gregg, Systems Performance, Enterprise and the Cloud, Second Edition, 2021 ISBN-10: 0-13-682015-8
3. Mario Hewardt, Advanced .NET debugging, 2010 ISBN 978-0-321-57889-1
4. Mark Russinovich, Aaron Margosis Troubleshooting with the Windows Sysinternals Tools, 2016 ISBN: 978-0-7356-8444-7
5. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2023
6. Intel® VTune™ Profiler User Guide