Evaluation of C++ Applications

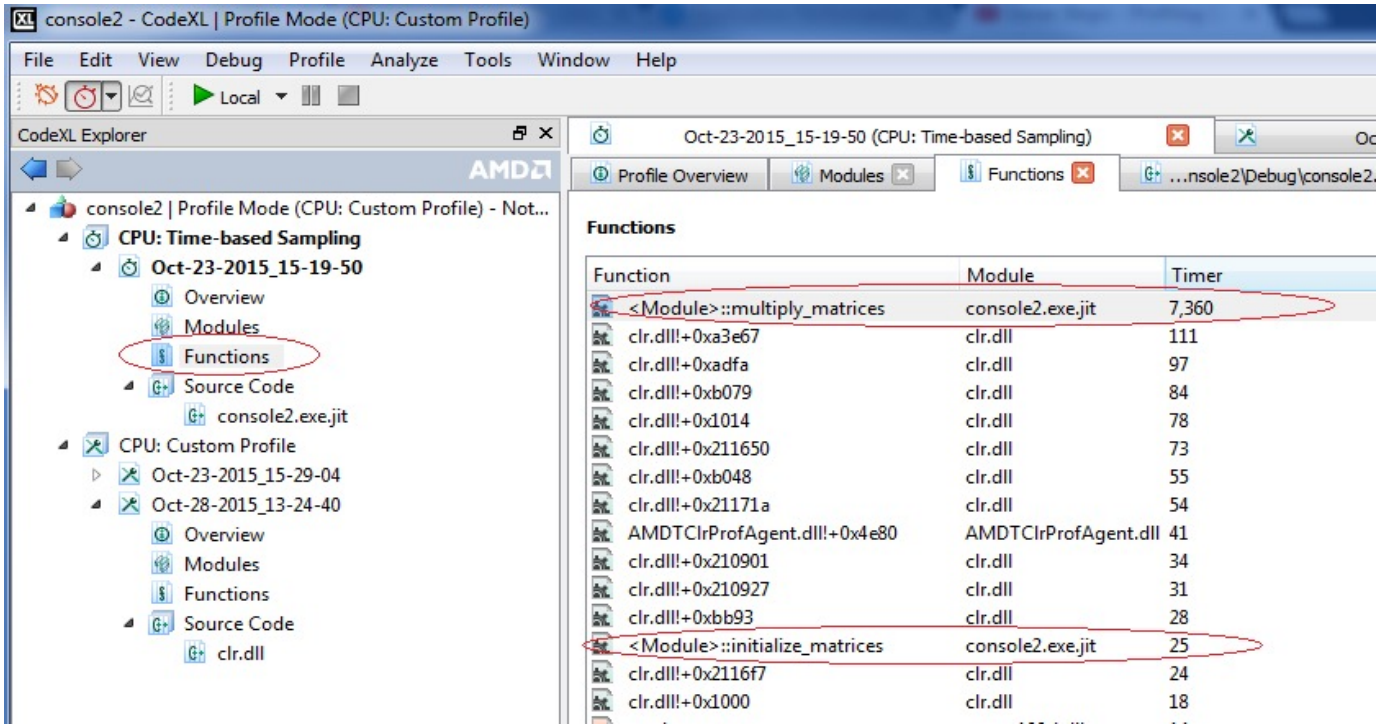
Student: Dumitrița Munteanu, Master An II, IVA

Software Performance

- Performance only meters when you don't have enough of it
- „It runs slow"
- **Symptom vs root cause**
- Performance profiling – activity that helps us discover something we didn't anticipate

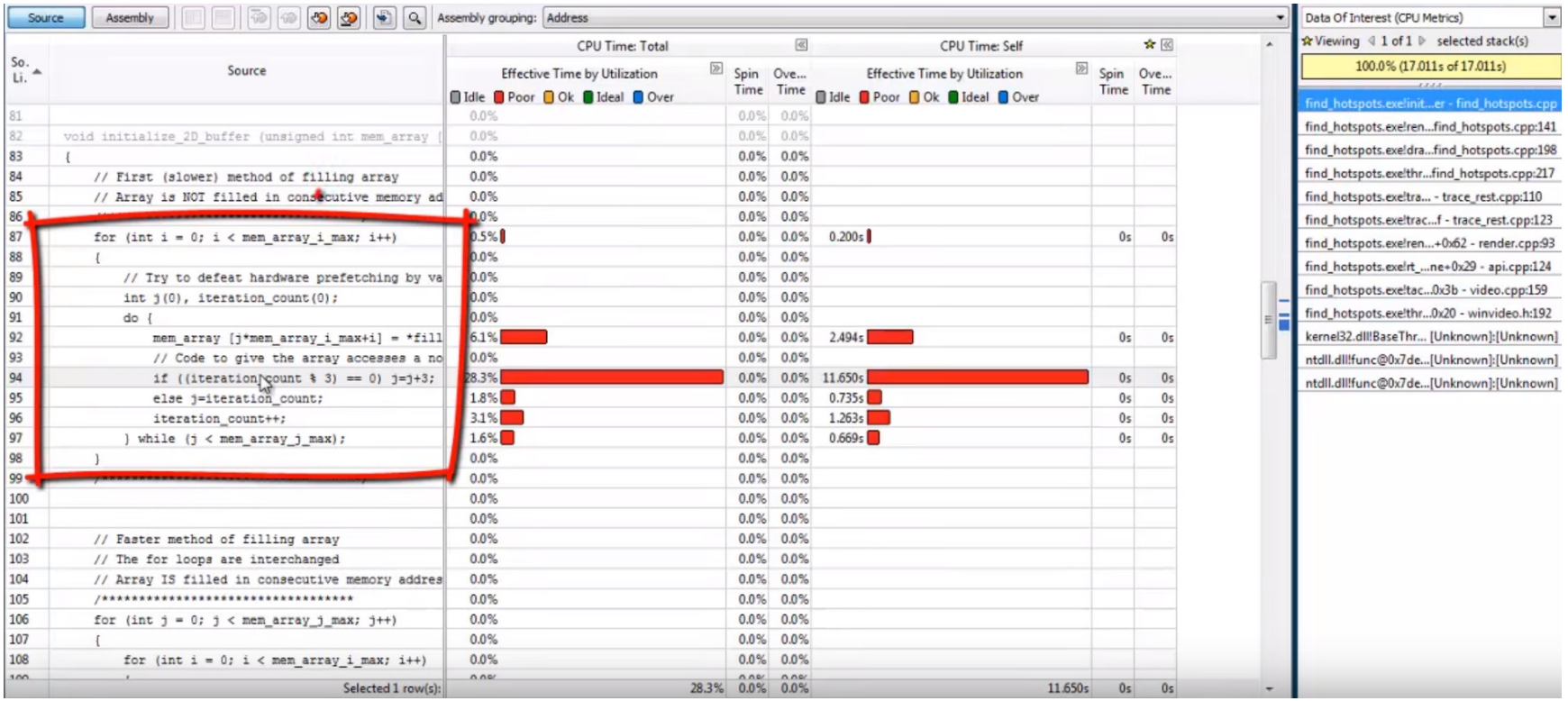
1. Identify Hot Spots

- Good start for performance analysis : identify the hot spots.
  - Which are the slowest functions?
- Run a use case and observe profiling data



Focus Optimization Efforts on Hotspots

- Highlight the line code that are being called the most



2. Number of Calls

- Some performance bottlenecks do not show up on the Top 10 slowest functions.
- Examine your loops and expensive calls.
- Take a look at the Hit Count.
- When you look at time, you get the **Hit Count** as well, but number of calls is something you can also do when you do **Code Coverage profiling**.
- Code Coverage Profiler will tell you **what was run and how many times** and most importantly **what you still didn't test.**

3. Excessive Object Allocation

- "Too much of a good thing" may not be so great.
- Run **Memory Allocation Profiler**
- Look at the objects that are dynamically allocated in heap memory
- Find **how many and what type of objects are in memory at a given point of time**.
- How much memory do they take as you run the application in a given scenario?
- See if you can maybe reduce the size of the memory that is being used.
- Find Memory Leaks

3. Excessive Object Allocation

- **Allocation Profiler** is used to find memory leaks in applications.
- It allows to determine whether the allocated memory blocks or objects remain in memory after the application execution is over.
- **Reference Count Profiler** tracks the creation and deletion of references to objects and allows to pinpoint unreleased references.
- **Resource Profiler** is used to find resource leaks and resource errors in applications.

Allocation Profiler

- **Each class has a live count value**, which if it is not zero, means that some of the objects that where created were not destroyed when the application was closed.
- Find out **where these instances were** created and determine **why they were not released**.
- **For each leak detected**, the allocation profiler collects information on the call stack i.e. the **hierarchy of functions that led to the creation of the objects that was not released**.
- Once you find the function, you can look at it's source code and find the place in which the allocated memory can be released.

3. OS Resource Usage

The size of code and data your program can declare and use under a certain OS is as well important.
Use OS Resource Usage Profiler in order to detect the unused handlers.

There are three kinds of memory limits:

- **Static code and data** - this is all the compiled code plus all static data. In C or C++, static data is generally created by variables and structs declared at "file scope", outside of a procedure.
- **Dynamic data** - this is memory that is allocated during program execution. In or C or C++ this is usually done with malloc or new;
- **Stack data** - this is memory that is allocated as a procedure is entered and deallocated when the procedure exits. In C or C++, most routine local variables and variables declared inside blocks are allocated on the stack.

Memory Limits for Applications on Windows

32-bit

- **Static data** - 2GB
- **Dynamic data** - 2GB
- **Stack data** - 1GB (the stack size is set by the linker, the default is 1MB. This can be increased using the Linker property System > Stack Reserve Size)

Note that on 32-bit Windows, the sum of all types of data must be 2GB or less. The practical limit is about 1.75GB due to space used by Windows itself

64-bit

- **Static data** - 2GB
- **Dynamic data** - 8TB
- **Stack data** - 1GB (the stack size is set by the linker, the default is 1MB. This can be increased using the Linker property System > Stack Reserve Size)

4. Database and Third Party Calls

- Connection to DB and third party calls could be a source of bottlenecks.
- **Data transfer** and **calls to third party functions** can be time intensive.
- Observe the timings on the line level.
- See **how much time was taken to execute a particular call**.
- Optimize application design around bottlenecks that you can't eliminate directly.

5. Eliminate Unused Code

- Dead application code and unused modules may accumulate access weight over time.
- Reduce the footprint of the application.
- Use a Code Coverage profiler for cleaner and leaner code.
- Code Coverage profiler can help to identify unused libraries so that they can be removed.

Conclusion

- Use Profiler Tools to find
  - Hotspots
  - Excessive functions calls
  - Excessive object allocation
  - Unnecessary OS resources in use
  - Bottlenecks in SQL statements and third party calls
  - Unused Code
- Profiles Tools
  - AMD Code XL / Code Analyst
  - VTune Intel
  - AQTime
  - Glow Code / Very Sleepy