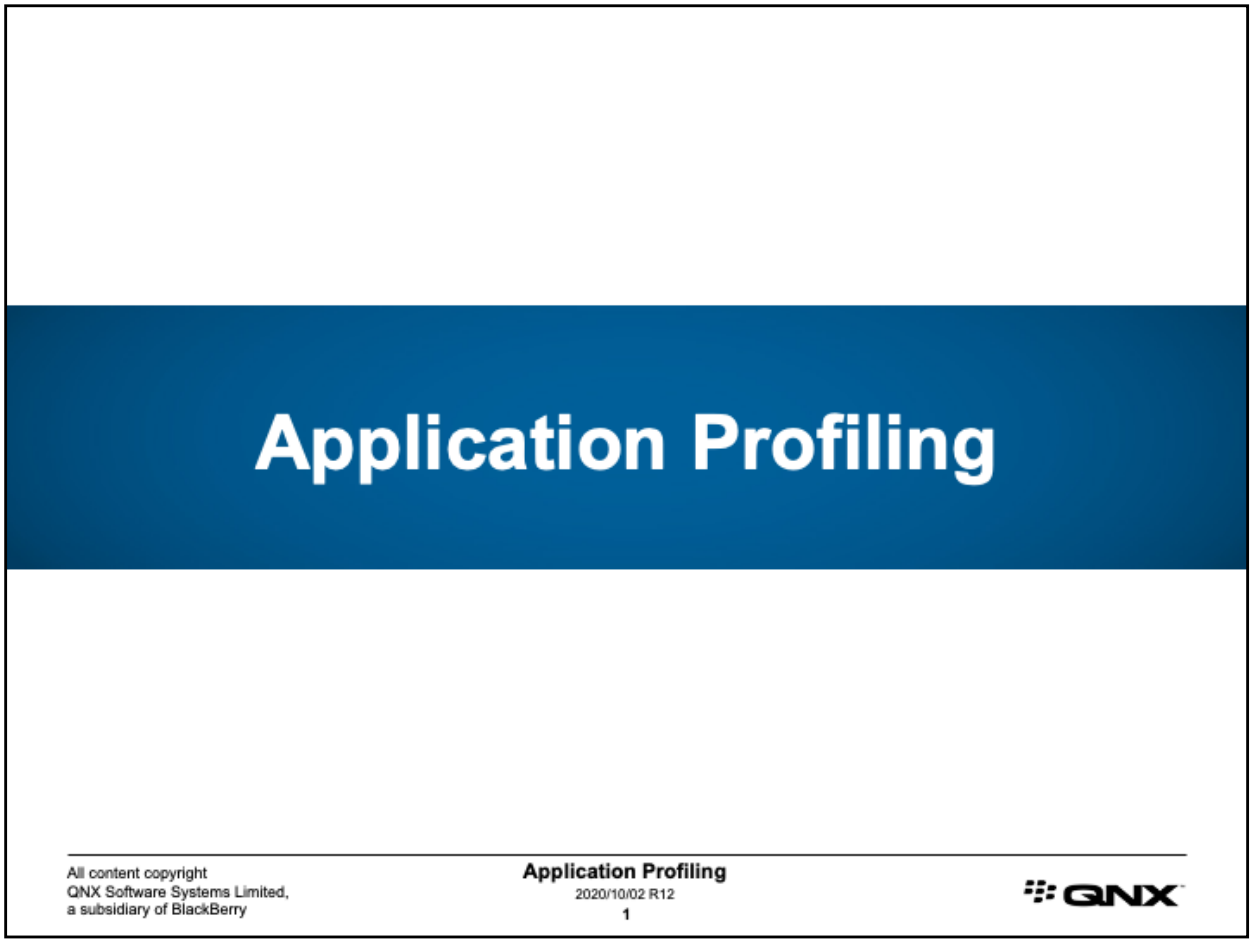


NOTES:

Introduction

You will learn:

- how application profiling is implemented
- the setup necessary before doing profiling
- how to do the profiling and analyse the results both:
  - while the application is running
  - after the application has gone away

NOTES:

Application Profiling

Topics:

- **Overview**
- **Sampling and Call Count Profiling**
- **Function Instrumentation Profiling**
- **Conclusion**

NOTES:

Overview

Application profiling:

- is a way of finding out where in your process time is being spent
- tells you:
  1. how often each function was called and by whom
  2. how long was spent in each function (sampling or instrumentation)
  3. how much CPU time was used for individual lines of code (sampling)
- this helps you:
  - tune algorithms
  - find bottlenecks
  - ...

note: sampling methods result in approximated values

NOTES:

Overview

Three types of application profiling:

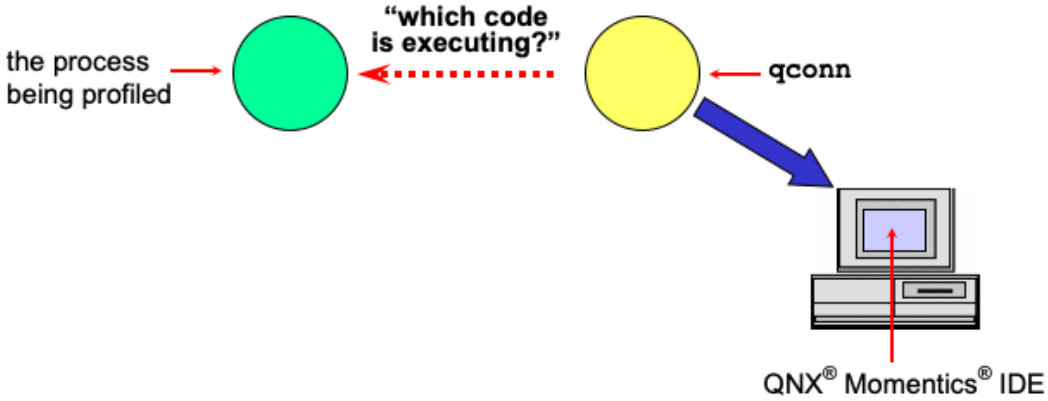
1. Statistical profiling
2. Sampling and Call Count instrumentation profiling
3. Function Instrumentation profiling

NOTES:

Overview - 1. Statistical profiling

Statistical profiling:

- **qconn** periodically samples to find out which code in your process is executing
- the longer the time you profile it, the more accurate the results



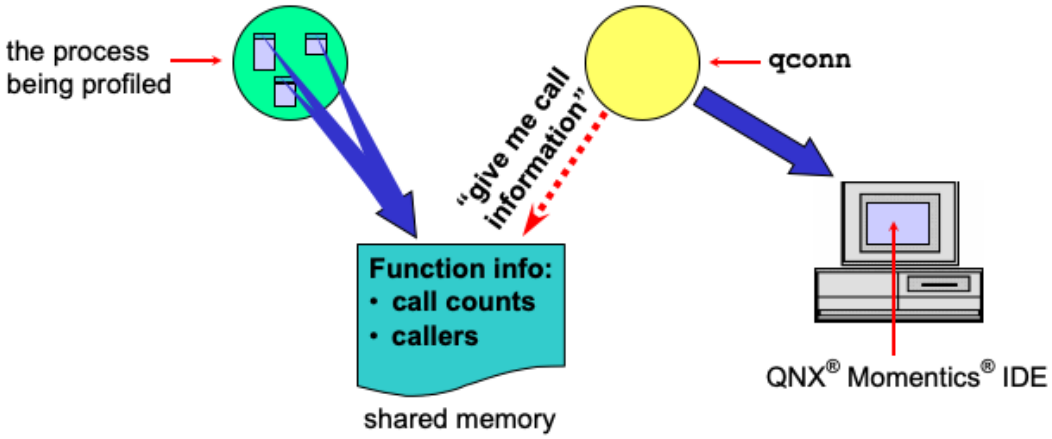
- you don't need to do anything special to your executable for this to work

NOTES:

Overview - 2. Instrumented Call Count profiling

Sampling and Call Count instrumentation:

- code inserted at the beginning of each function stores information about who called it and how often



- you must build with profiling enabled for this to work (we'll see how later)

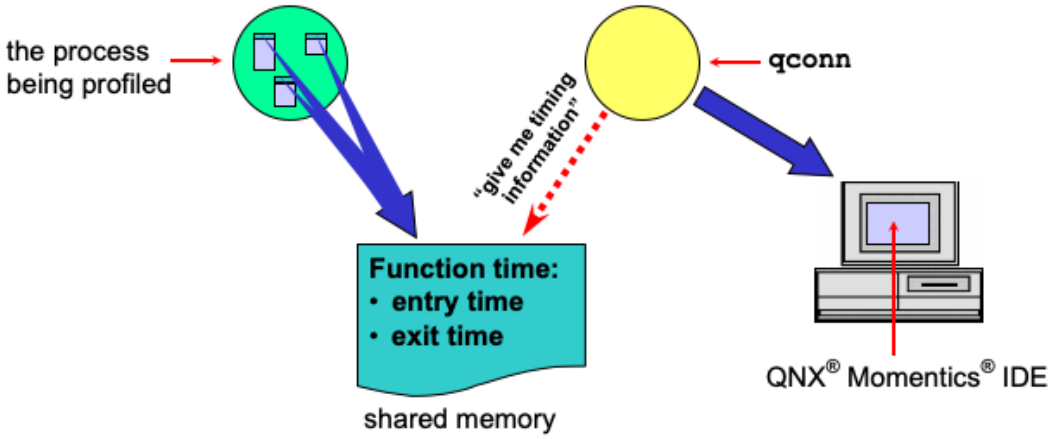
NOTES:

For call information to be gathered, you must build your process with profiling. This changes which startup code is linked in to run initially in your executable (i.e. the code that calls *main()*) and it inserts code at the beginning of each of your functions. When a function is called, this code uses some shared memory to tell **`qconn`** who called the function and increments a count of how often. **`qconn`** passes this on to the IDE for display.

Overview - 3. Instrumented Function profiling

Function Instrumentation:

– code inserted just after entrance to and just before the exit from every function call, providing precise function run time information



-must be built with special compile and link options

NOTES:

Application Profiling

Topics:

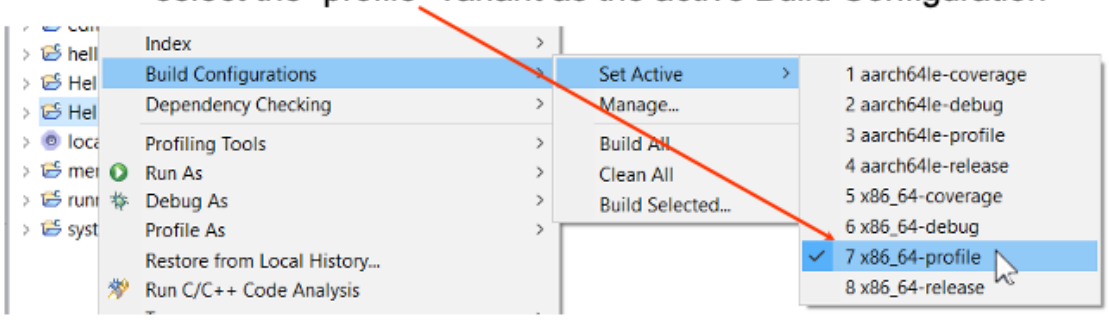
- Overview
- → Sampling and Call Count Profiling
- Function Instrumentation Profiling
- Conclusion

NOTES:

Setting up for Profiling - Call Count profiling

Building for Call Count Profiling:

- this will instrument your code
  - insert code before each function to gather call information
- for an existing QNX C/C++ Application Project:
  - select the “profile” variant as the active Build Configuration



- if using your own build environment (e.g. Makefile):
  - add the `-p` option to your compile and link command lines

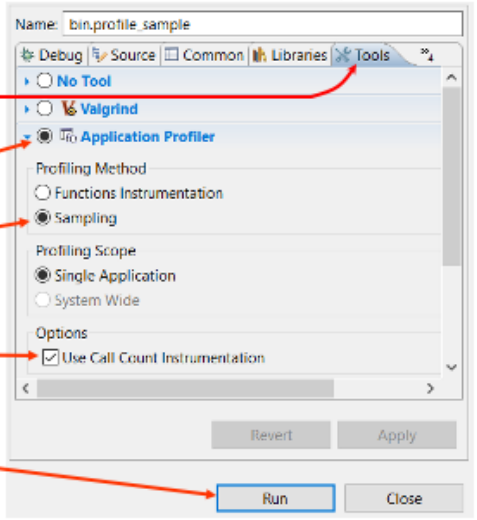
NOTES:

Setting up for Profiling - Call Count profiling

To profile while running:

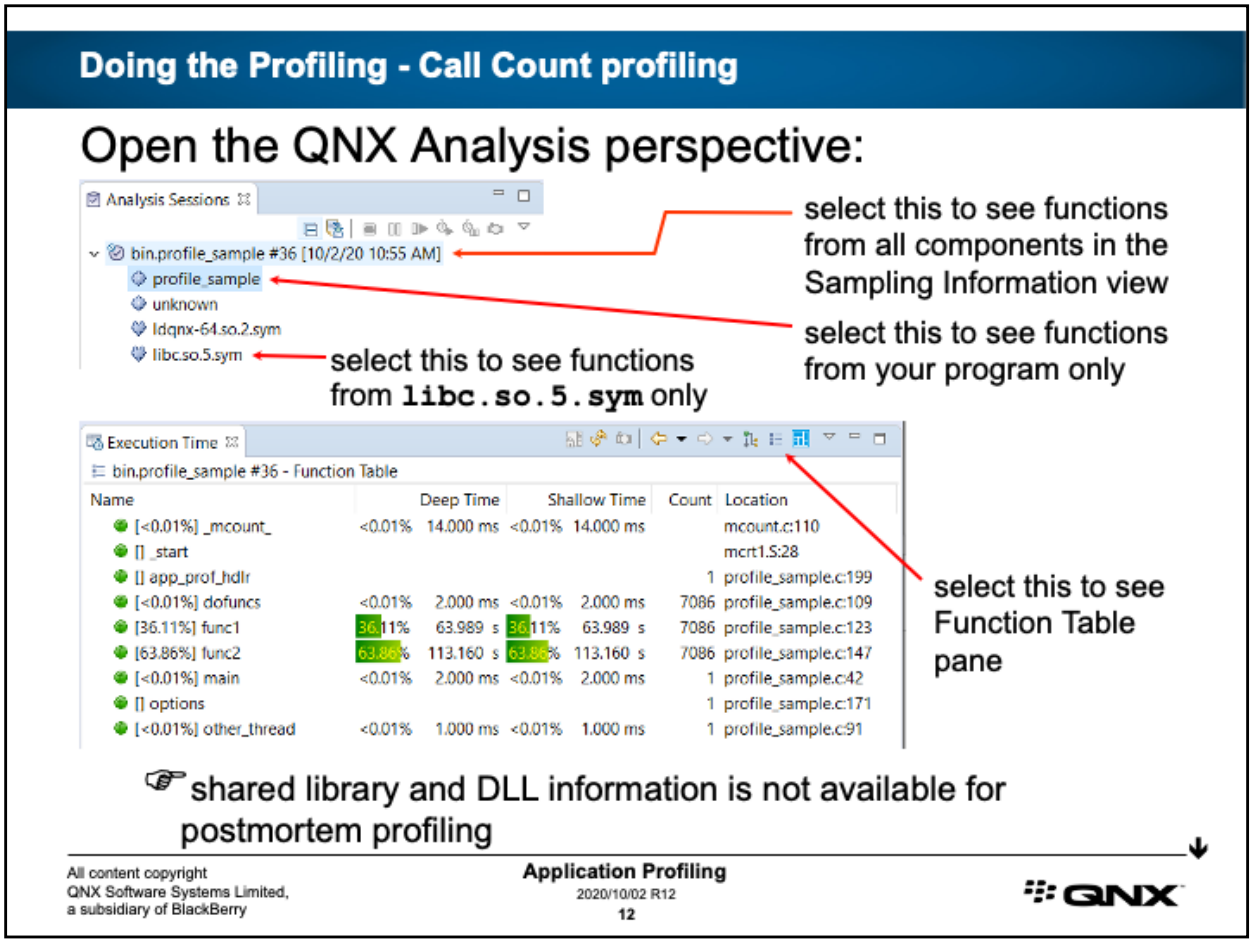
– create a C\C++ QNX Application Run Configuration:

1. in the Tools tab
2. select Application Profiler
3. select Sampling and Use Call Count Instrumentation
4. click Run



- use the debug version of the executable if you want to see sampling results per line of code

NOTES:

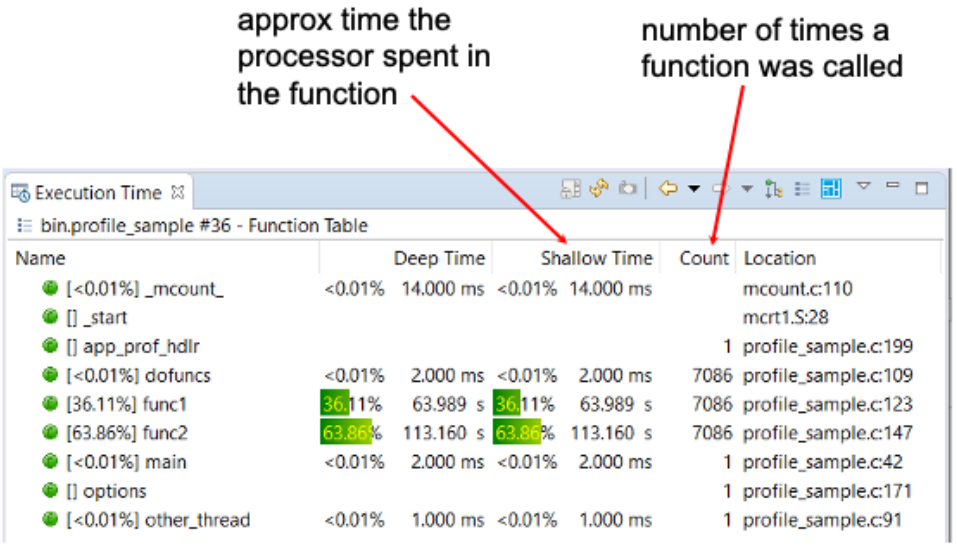


NOTES:

You may have to menu-click in or resize the Execution Time view in order to see the toolbar buttons.

Doing the Profiling - Call Count profiling

Sampling information:



Name	Deep Time		Shallow Time		Count	Location
[<0.01%] _mcount_	<0.01%	14.000 ms	<0.01%	14.000 ms		mcount.c:110
[] _start						mcrt1.S:28
[] app_prof_hdlr					1	profile_sample.c:199
[<0.01%] dofuncs	<0.01%	2.000 ms	<0.01%	2.000 ms	7086	profile_sample.c:109
[36.11%] func1	36.11%	63.989 s	36.11%	63.989 s	7086	profile_sample.c:123
[63.86%] func2	63.86%	113.160 s	63.86%	113.160 s	7086	profile_sample.c:147
[<0.01%] main	<0.01%	2.000 ms	<0.01%	2.000 ms	1	profile_sample.c:42
[] options					1	profile_sample.c:171
[<0.01%] other_thread	<0.01%	1.000 ms	<0.01%	1.000 ms	1	profile_sample.c:91

NOTES:

Deep time is not meaningful in sampling mode, so the column can be ignored.

Doing the Profiling - Call Count profiling

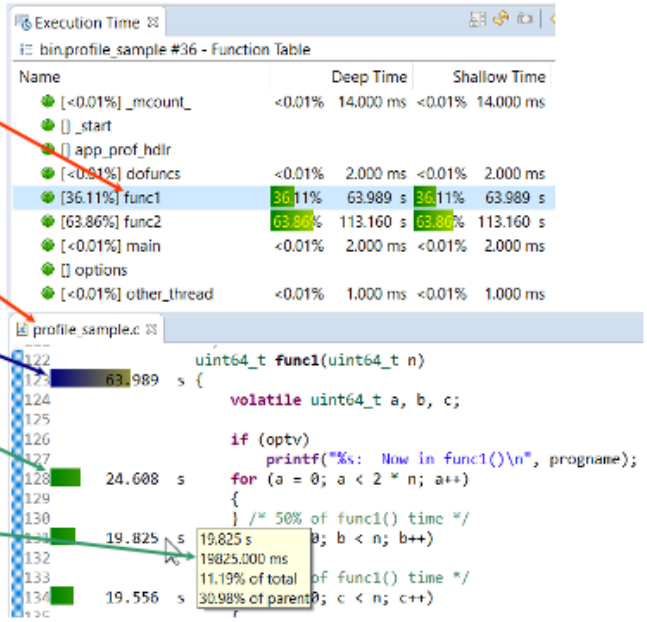
Using the editor:

double-click on a function in the Function Table and the editor appears with the function annotated

time for function relative to total time

time for line relative to total time.

hover to show time for line:
- as percent of total time
- as percent of function time



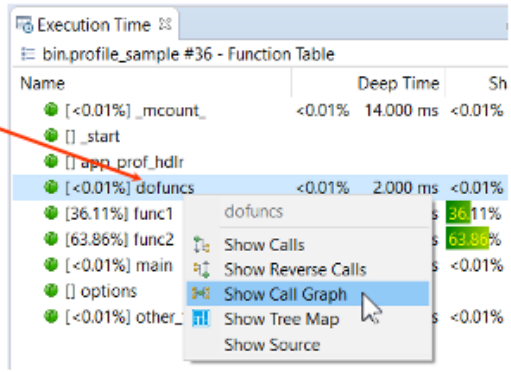
☞ you must have had debugging information in your executable for this to work

NOTES:

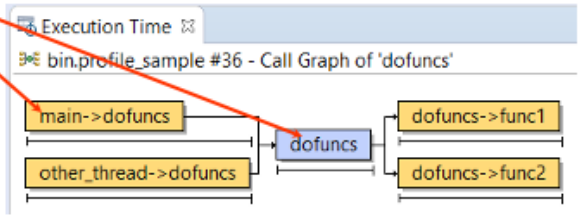
Doing the Profiling - Call Count profiling

Call Information:

right click on function and select Show Call Graph



click on the various functions to move through the call graph

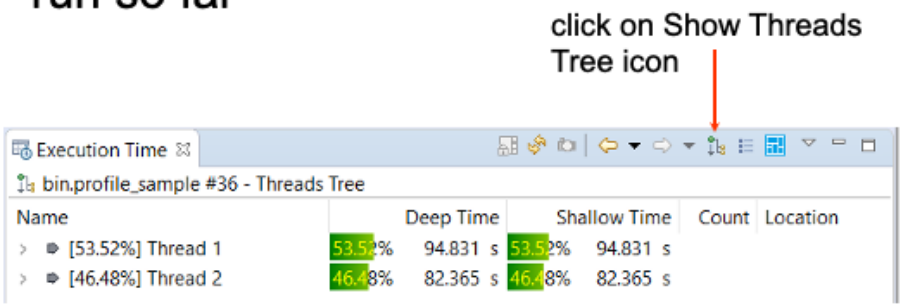


NOTES:

Doing the Profiling - Call Count profiling

Thread information:

– the Thread Info view shows the processor time and percent time that each thread has run so far

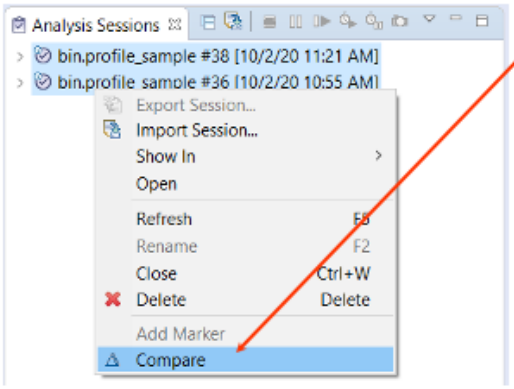


NOTES:

Doing the Profiling – Comparing Sessions

Did you improve things?

- after you've made changes, did the improve the behaviour?
- compare two sessions:
  - select two sessions, then menu → Compare:

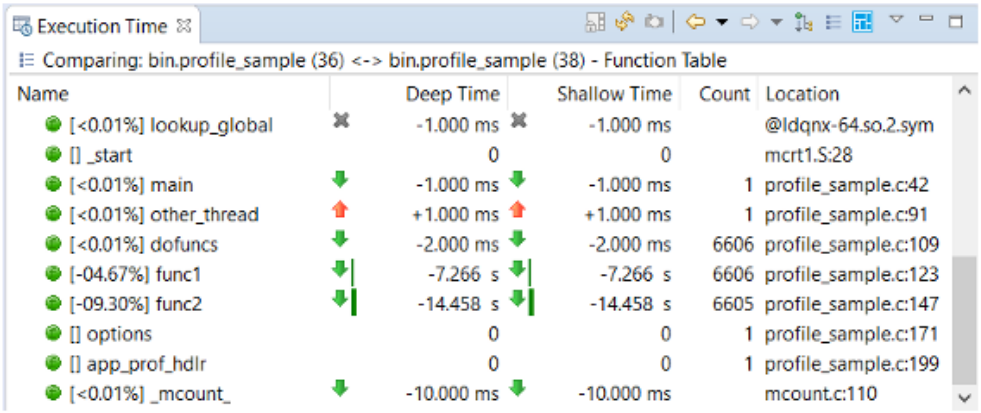


NOTES:

A comparison is only meaningful/useful if you're testing something comparable. For our exercise examples, where you may have run the program for different periods of time, this doesn't makes sense. It is more useful if you're trying to figure out how long it took to do some operation, before and after changes.

Comparing Sessions

Compared sessions look like:



- time used increased for this function
- time used decreased for this function
- function is only in the 2nd run dataset
- function is only in the 1st run dataset

NOTES:

EXERCISE

Profiling while running:

- build your **`application_profiling`** project. This will result in an executable called **`profile_sample`**
- do the steps you just learned to profile it while it runs
- examine the various views
- which function is consuming the most CPU time?
  - which lines in the function are consuming the most CPU time?

NOTES:

Application Profiling

Topics:

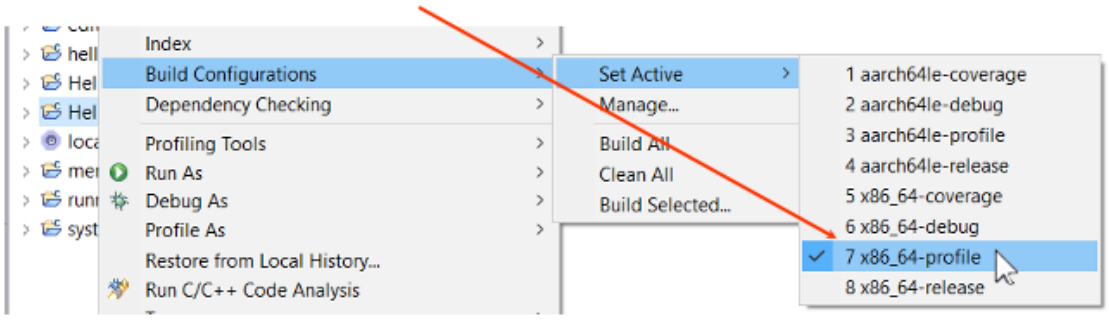
- Overview
- Sampling and Call Count Profiling
- → Function Instrumentation Profiling
- Conclusion

NOTES:

Setting up for Profiling – Function Instrumentation

Building for Function Instrumentation Profiling:

- this will instrument your code
  - by inserting code in each function to gather timing information
- for an existing QNX Executable Project:
  - select the “profile” variant as the active Build Configuration



continued...

NOTES:

Setting up for Profiling - Function Instrumentation

Building for Function Instrumentation Profiling (continued):

- if using your own build environment (e.g. Makefile):
  - add the **`-finstrument-functions`** option to your compile command line
  - add **`-lprofilingS`** option to your link command line after your objects (.o files)

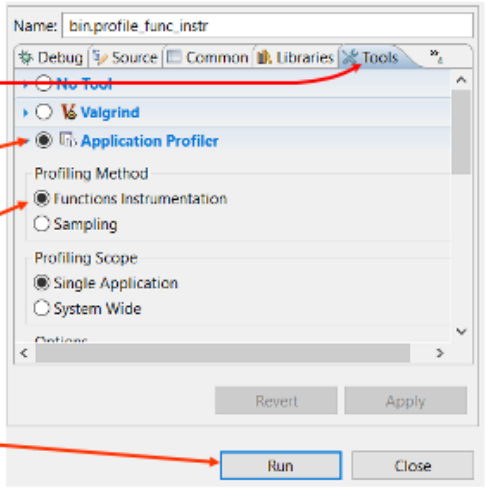
NOTES:

Setting up for Profiling – Function Instrumentation

To profile while running:

– create a C\C++ QNX Application Run Configuration:

1. in the Tools tab
2. select Application Profiler
3. select Function Instrumentation
4. click Run

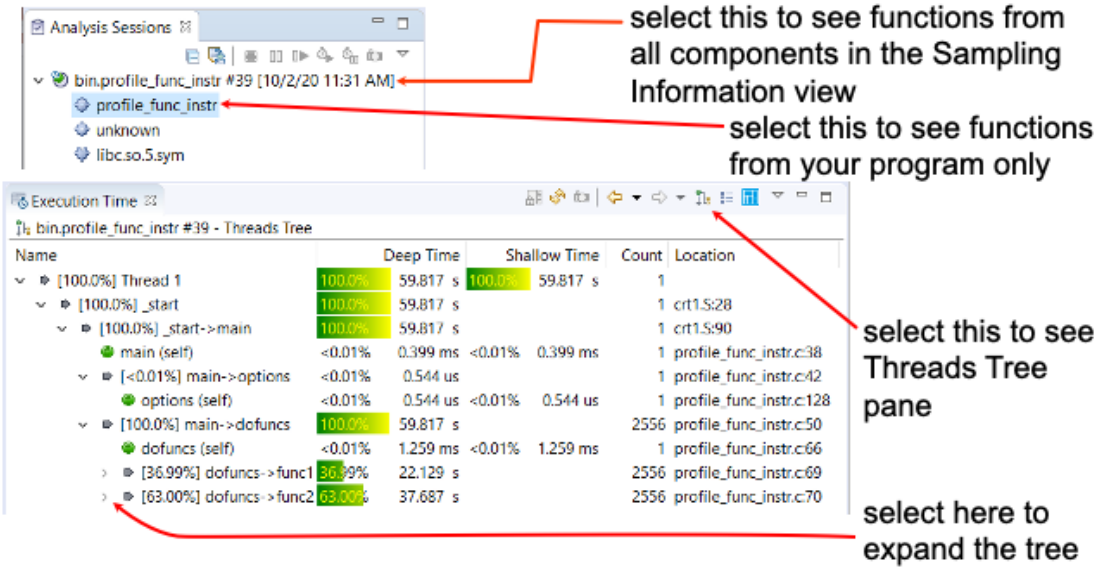


- use the debug version of the executable if you want to see sampling results per line of code

NOTES:

Doing the Profiling - Function Instrumentation

Execution Time view – Threads Tree:

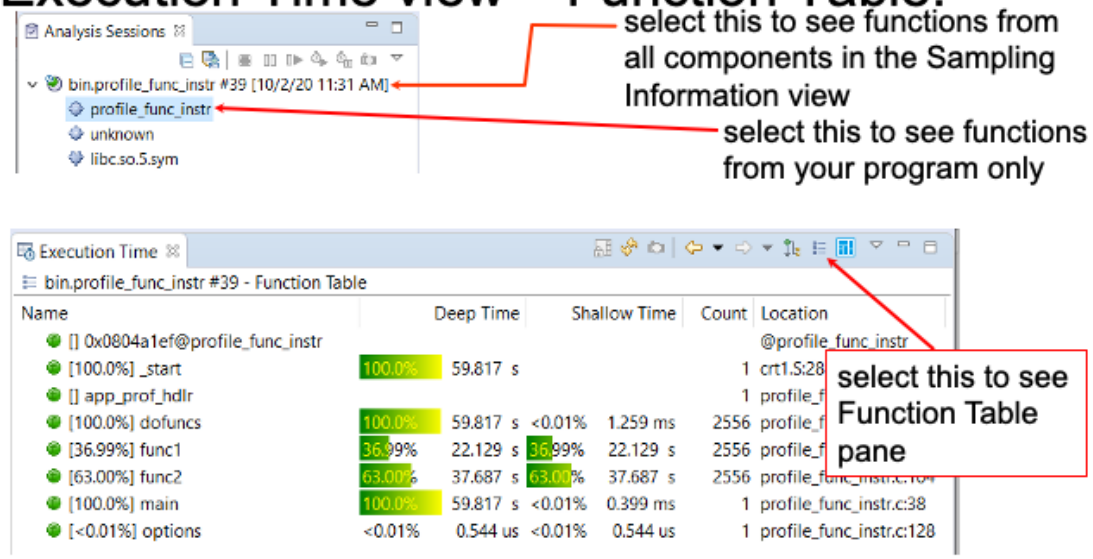


– you can see exactly where the application spends its time, and which functions are used in the process.

NOTES:

Doing the Profiling - Function Instrumentation

Execution Time view – Function Table:

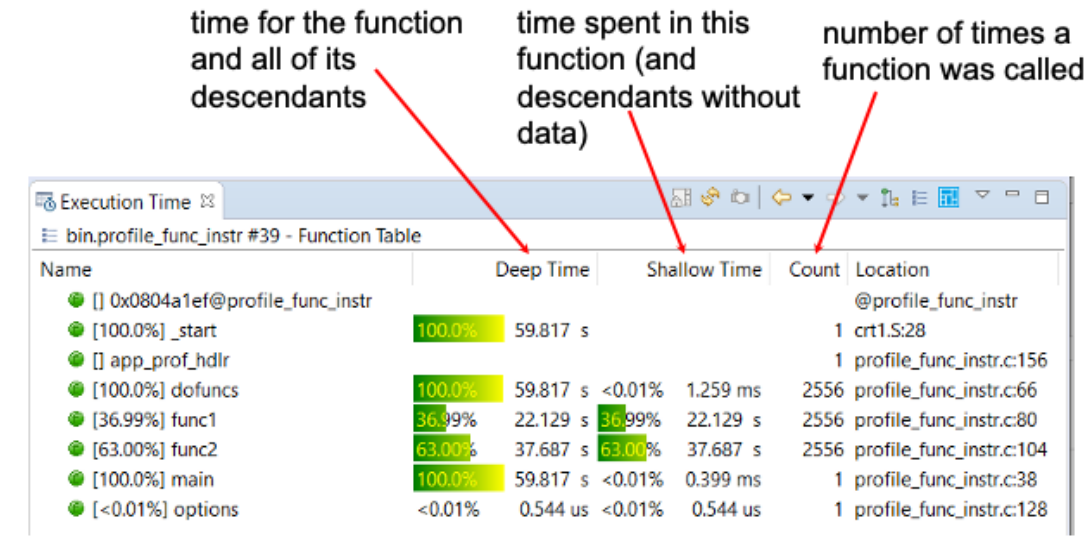


- helps you identify those functions that take the longest to complete

NOTES:

Doing the Profiling - Function Instrumentation

Sampling Information:



There's currently a known problem with Deep Time having a value less than the sum of the Shallow Times if the process is multi-threaded. For single-threaded processes, Deep Time should be correct.

NOTES:

Deep time includes is the sum of the function's own shallow time and its children's deep times.

Shallow time includes all time in the local function, and all time in functions it calls for which data is not generated, that is, ones that weren't build with function instrumentation. This generally includes all of the QNX library.

Also, shallow time can not be computed for a function that has not completed, for example, the main() function while a program is running.

Doing the Profiling - Function Instrumentation

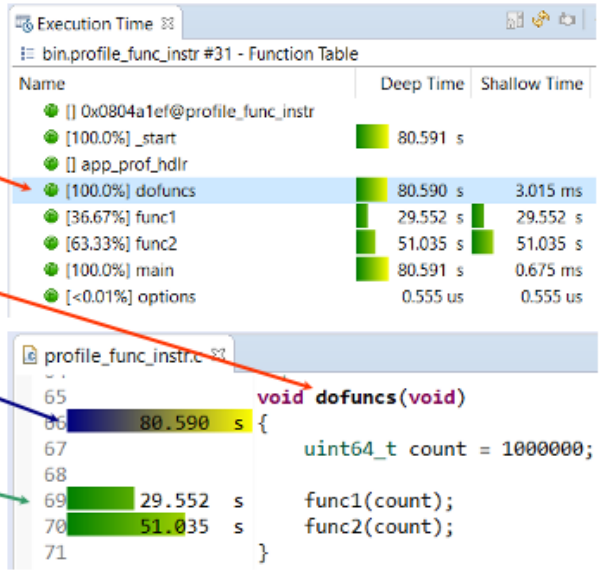
Using the editor:

double-click on a function in the Function Table and the editor appears with the function annotated

time for function relative to total time

time for the called functions relative to total time

call Information is available by right-clicking in the Execution Time view and selecting Show Call Graph

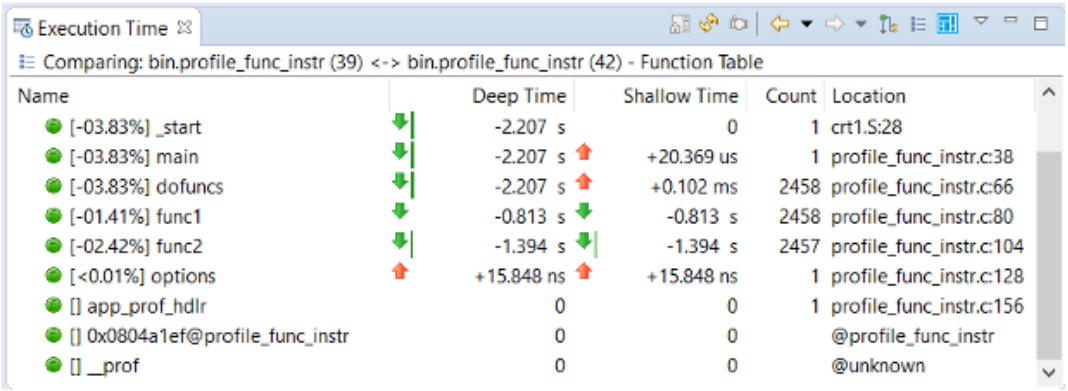


☞ you must have had debugging information in your executable for this to work

NOTES:

Function Instrumentation - Comparing

We can also compare function instrumentation results:



- time used increased for this function
- time used decreased for this function
- function is only in the 2nd run dataset
- function is only in the 1st run dataset

NOTES:

EXERCISE

Profiling while running:

- build your **`application_profiling`** project. This will result in an executable called **`profile_func_instr`**
- do the steps you just learned to profile it while it runs
- examine the various views
- which function has taken the longest to complete?
  - which function in its own code has taken longest to complete?

NOTES:

Application Profiling

Topics:

- Overview
- Sampling and Call Count Profiling
- Function Instrumentation Profiling
- → Conclusion

NOTES:

Statistical vs Instrumented Profiling

Statistical Profiling:

- sampling based
  - driven by timer interrupt
    - not useful for anything driven by timers
    - clock-tick precision (usually 1ms)
- frequency of line-of-code execution
  - per-function data is derived from per-line data

Instrumented Profiling:

- measured by instrumentation
  - *ClockCycles()* precision (usually sub-microsecond)
- elapsed-time based
  - includes time preempted or blocked
  - time to complete an operation
- only per-function data, no per-line data

NOTES:

Conclusion

You learned:

- how profiling is implemented:
  - statistical profiling is used to gather execution frequency for individual lines of code
  - instrumented profiling is used to gather call information or precise function time information
- the setup needed to do profiling
- how to do analysis:
  - while the application is running

NOTES: