

Application Profiling

All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

1



NOTES:

QNX, Momentics, Neutrino, and “Build a more reliable world” are registered trademarks in certain jurisdictions, and Qnet is a trademark of QNX Software Systems.

All other trademarks and trade names belong to their respective owners.

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:

Topics:

→ **Overview**

Sampling and Call Count Profiling

Function Instrumentation Profiling

Conclusion

NOTES:

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:

Three types of application profiling:

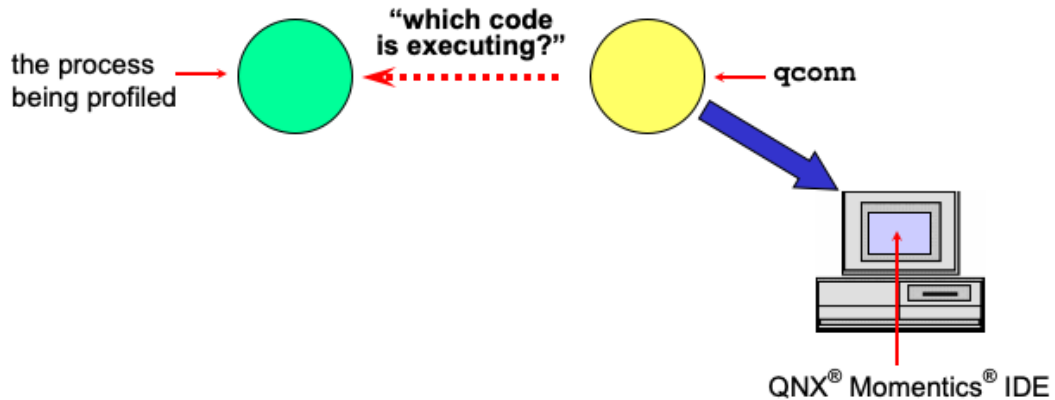
1. Statistical profiling
2. Sampling and Call Count instrumentation profiling
3. Function Instrumentation profiling

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



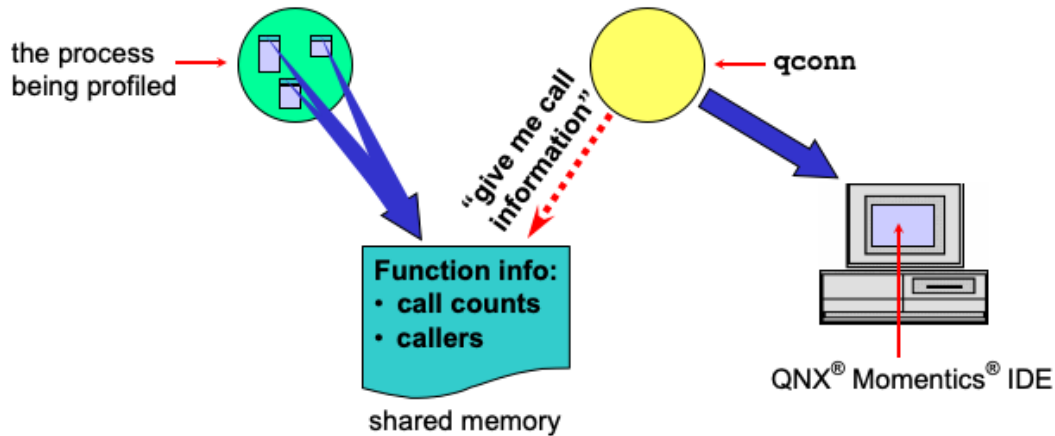
- you don't need to do anything special to your executable for this to work

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)

All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

7

QNX

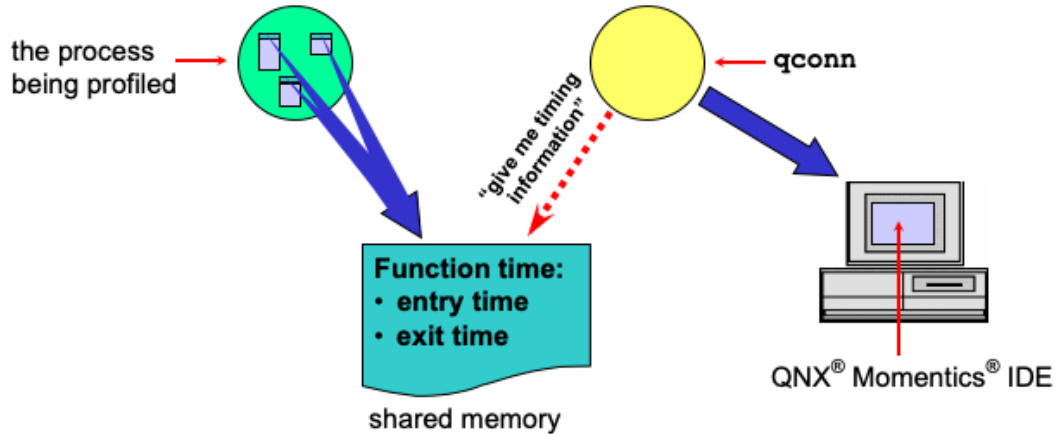
NOTES:

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

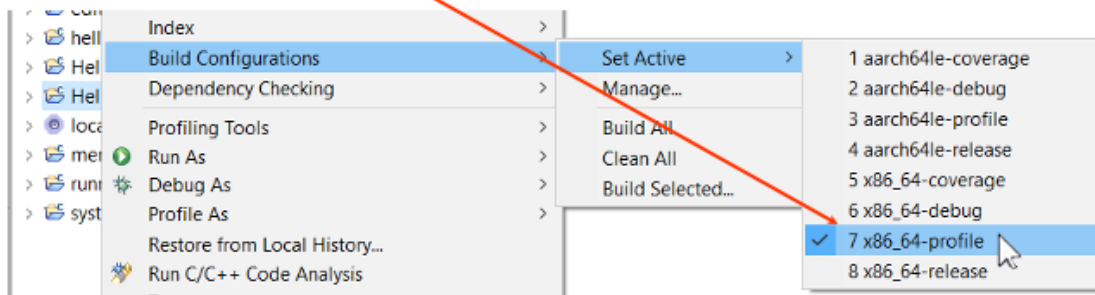
→ **Sampling and Call Count Profiling**
Function Instrumentation Profiling
Conclusion

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:

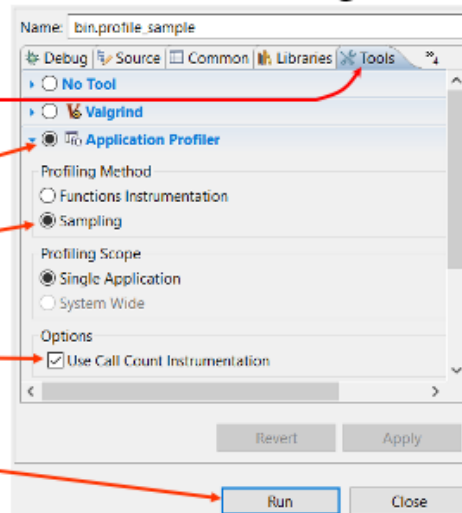
– create a C/C++ QNX Application Run Configuration:

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:

Doing the Profiling - Call Count profiling

Open the QNX Analysis perspective:

The screenshot shows the QNX Analysis perspective with two main views:

- Analysis Sessions:** A tree view showing the hierarchy of analysis sessions. Red arrows point to:
 - `bin.profile_sample #36 [10/2/20 10:55 AM]` with the text: "select this to see functions from all components in the Sampling Information view"
 - `profile_sample` with the text: "select this to see functions from your program only"
 - `libc.so.5.sym` with the text: "select this to see functions from `libc.so.5.sym` only"
- Execution Time:** A table view showing the function table. A red arrow points to the toolbar icon for the Function Table pane with the text: "select this to see Function Table pane".

The Function Table pane displays the following data:

Name	Deep Time	Shallow Time	Count	Location
[<0.01%] _mcount_	<0.01%	14.000 ms	<0.01%	mcount.c:110
[] _start				mcr15:28
[] app_prof_hdr			1	profile_sample.c:199
[<0.01%] dofuncs	<0.01%	2.000 ms	<0.01%	7086 profile_sample.c:109
[36.11%] func1	36.11%	63.989 s	36.11%	7086 profile_sample.c:123
[63.86%] func2	63.86%	113.160 s	63.86%	7086 profile_sample.c:147
[<0.01%] main	<0.01%	2.000 ms	<0.01%	1 profile_sample.c:42
[] options			1	profile_sample.c:171
[<0.01%] other_thread	<0.01%	1.000 ms	<0.01%	1 profile_sample.c:91

shared library and DLL information is not available for postmortem profiling

NOTES:

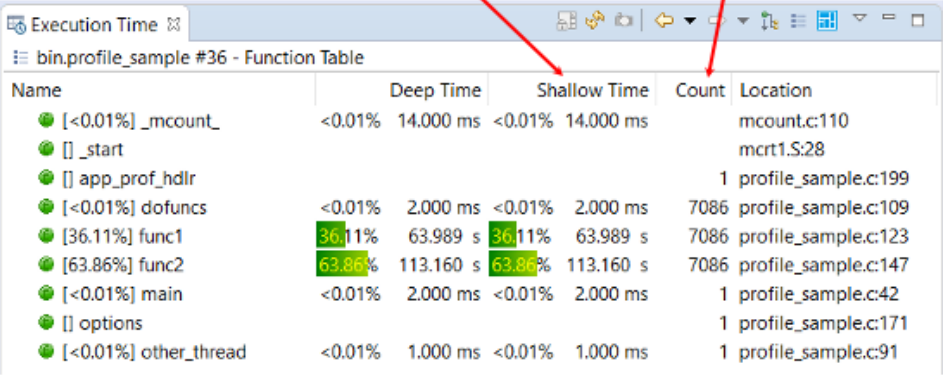
You may have to menu-click in or resize the Execution Time view in order to see the toolbar buttons.

Doing the Profiling - Call Count profiling

Sampling information:

approx time the
processor spent in
the function

number of times a
function was called



bin.profile_sample #36 - Function Table

Name	Deep Time	Shallow Time	Count	Location
[<0.01%] _mcount_	<0.01% 14.000 ms	<0.01% 14.000 ms		mcount.c:110
[] _start				mcr1.S:28
[] app_prof_hdlr			1	profile_sample.c:199
[<0.01%] dofuns	<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

All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

13



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

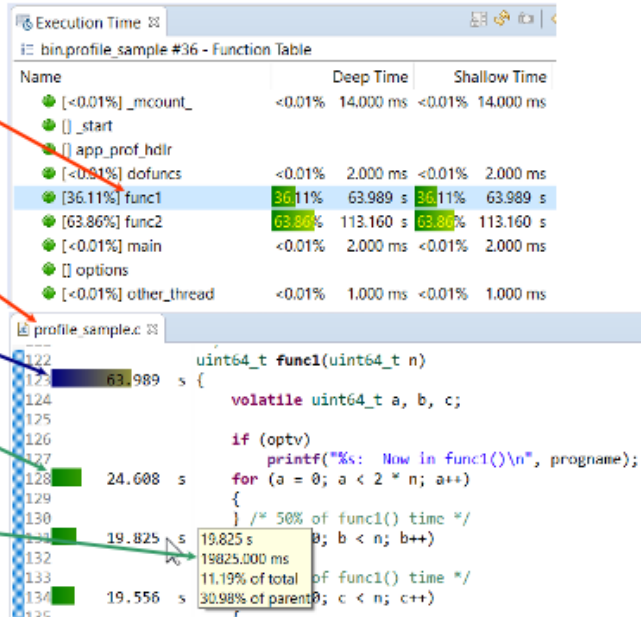
time for function
relative to total time

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

Execution Time

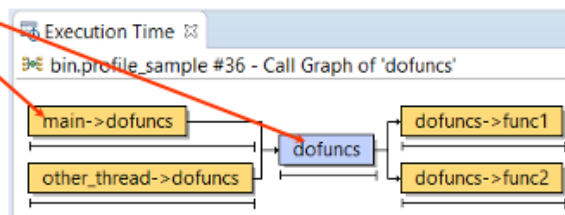
bin.profile_sample #36 - Function Table

Name	Deep Time	Sh
[<0.01%] _mcount_	<0.01% 14.000 ms	<0.01%
[] _start		
[] app_prof_hdr		
[<0.01%] dofuns	<0.01% 2.000 ms	<0.01%
[36.11%] func1		\$ 36.11%
[63.86%] func2		\$ 63.86%
[<0.01%] main		<0.01%
[] options		
[<0.01%] other_		<0.01%

Context menu for 'dofuns':

- dofuns
- Show Calls
- Show Reverse Calls
- Show Call Graph
- Show Tree Map
- Show Source

click on the various
functions to
move through the
call graph



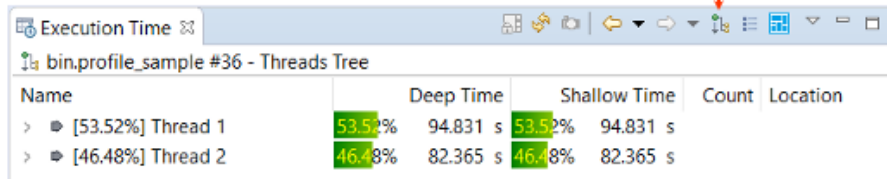
NOTES:

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

click on Show Threads
Tree icon



The screenshot shows the 'bin.profile_sample #36 - Threads Tree' window. The toolbar at the top contains various icons, including a tree icon. A red arrow points to this tree icon. The table below displays the thread information.

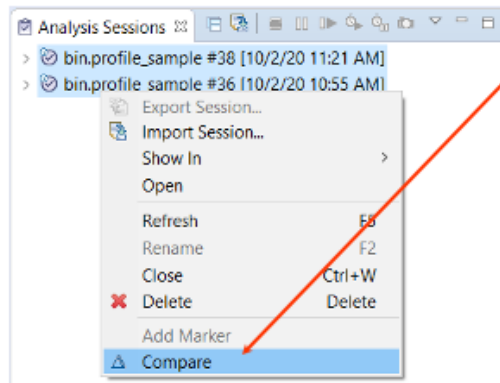
Name	Deep Time	Shallow Time	Count	Location
> [53.52%] Thread 1	53.52% 94.831 s	53.52% 94.831 s		
> [46.48%] Thread 2	46.48% 82.365 s	46.48% 82.365 s		

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:



All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

17



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 make 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:

Execution Time ⌕

Comparing: bin.profile_sample (36) <-> bin.profile_sample (38) - Function Table

Name		Deep Time	Shallow Time	Count	Location
● [<0.01%] lookup_global	⌕	-1.000 ms	-1.000 ms		@ldqnx-64.so.2.sym
● [] _start		0	0		mcr11.5:28
● [<0.01%] main	↓	-1.000 ms	-1.000 ms	1	profile_sample.c:42
● [<0.01%] other_thread	↑	+1.000 ms	+1.000 ms	1	profile_sample.c:91
● [<0.01%] dofuncs	↓	-2.000 ms	-2.000 ms	6606	profile_sample.c:109
● [-04.67%] func1	↓	-7.266 s	-7.266 s	6606	profile_sample.c:123
● [-09.30%] func2	↓	-14.458 s	-14.458 s	6605	profile_sample.c:147
● [] options		0	0	1	profile_sample.c:171
● [] app_prof_hdlr		0	0	1	profile_sample.c:199
● [<0.01%] _mcount_	↓	-10.000 ms	-10.000 ms		mcount.c:110

- ↑ 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

All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

18



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

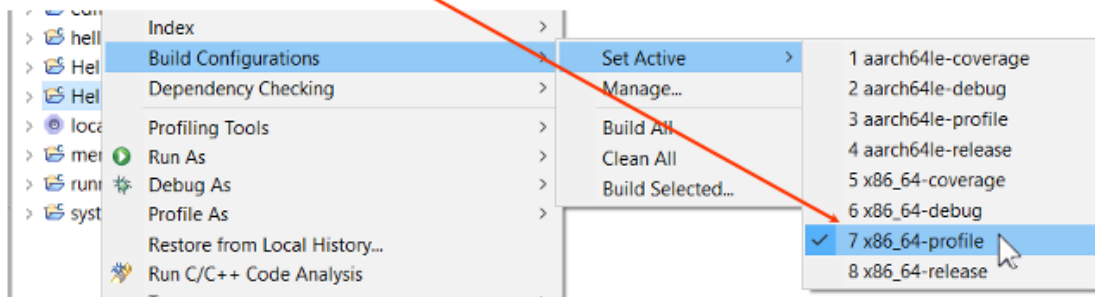
Conclusion

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:

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)

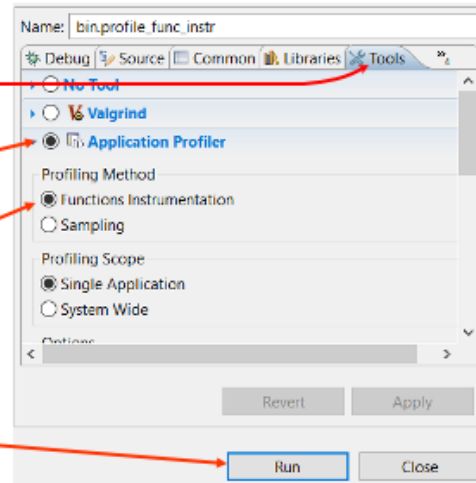
NOTES:

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:

Analysis Sessions

- bin_profile_func_instr #39 [10/2/20 11:31 AM]
 - profile_func_instr
 - unknown
 - libc.so.5.sym

Execution Time 33

bin_profile_func_instr #39 - Threads Tree

Name	Deep Time	Shallow Time	Count	Location
[100.0%] Thread 1	100.0% 59.817 s	100.0% 59.817 s	1	
[100.0%] _start	100.0% 59.817 s		1	crt1.S:28
[100.0%] _start->main	100.0% 59.817 s		1	crt1.S:90
main (self)	<0.01% 0.399 ms	<0.01% 0.399 ms	1	profile_func_instr.c:38
[<0.01%] main->options	<0.01% 0.544 us		1	profile_func_instr.c:42
options (self)	<0.01% 0.544 us		1	profile_func_instr.c:128
[100.0%] main->dofuncs	100.0% 59.817 s		2556	profile_func_instr.c:50
dofuncs (self)	<0.01% 1.259 ms	<0.01% 1.259 ms	1	profile_func_instr.c:66
[36.99%] dofunc->func1	36.99% 22.129 s		2556	profile_func_instr.c:69
[63.00%] dofunc->func2	63.00% 37.687 s		2556	profile_func_instr.c:70

select this to see functions from all components in the Sampling Information view

select this to see functions from your program only

select this to see Threads Tree pane

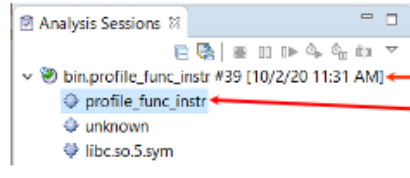
select here to expand the tree

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

NOTES:

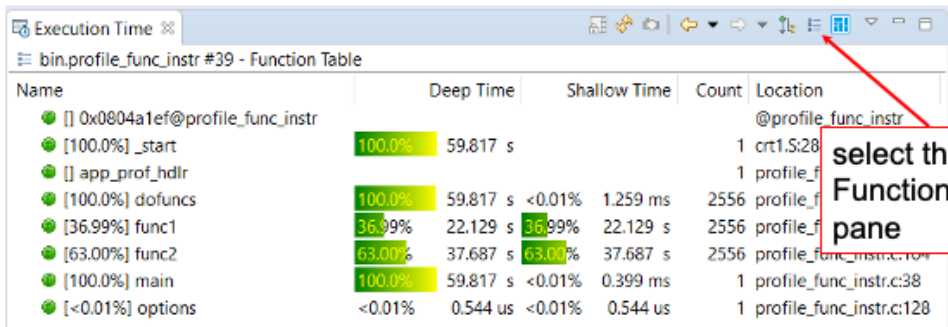
Doing the Profiling - Function Instrumentation

Execution Time view – Function Table:



select this to see functions from all components in the Sampling Information view

select this to see functions from your program only



select this to see Function Table pane

Name	Deep Time	Shallow Time	Count	Location
[0x0804a1ef@profile_func_instr				@profile_func_instr
[100.0%] _start	100.0%	59.817 s	1	crt1.S:28
[100.0%] app_prof_hdlr			1	profile_f
[100.0%] dofuns	100.0%	59.817 s	<0.01%	1.259 ms
[36.99%] func1	36.99%	22.129 s	36.99%	22.129 s
[63.00%] func2	63.00%	37.687 s	63.00%	37.687 s
[100.0%] main	100.0%	59.817 s	<0.01%	0.399 ms
[<0.01%] options	<0.01%	0.544 us	<0.01%	0.544 us

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

NOTES:

Doing the Profiling - Function Instrumentation

Sampling Information:

time for the function
and all of its
descendants

time spent in this
function (and
descendants without
data)

number of times a
function was called

Name	Deep Time	Shallow Time	Count	Location
[] 0x0804a1ef@profile_func_instr				@profile_func_instr
[100.0%] _start	100.0% 59.817 s		1	crt1.S:28
[] app_prof_hdlr			1	profile_func_instr.c:156
[100.0%] dofuns	100.0% 59.817 s	<0.01% 1.259 ms	2556	profile_func_instr.c:66
[36.99%] func1	36.99% 22.129 s	36.99% 22.129 s	2556	profile_func_instr.c:80
[63.00%] func2	63.00% 37.687 s	63.00% 37.687 s	2556	profile_func_instr.c:104
[100.0%] main	100.0% 59.817 s	<0.01% 0.399 ms	1	profile_func_instr.c:38
[<0.01%] options	<0.01% 0.544 us	<0.01% 0.544 us	1	profile_func_instr.c:128

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.

All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

26

QNX

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:

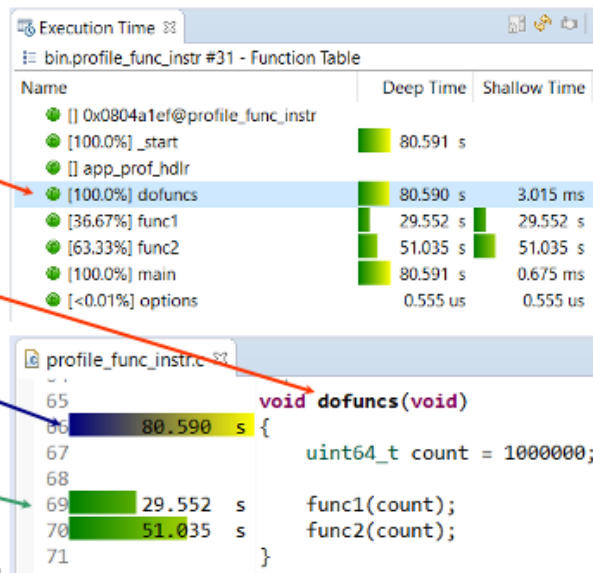
double-click on a function in the Function Table and the editor appears with the function annotated

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:

Execution Time

Comparing: bin.profile_func_instr (39) <-> bin.profile_func_instr (42) - Function Table

Name		Deep Time	Shallow Time	Count	Location
[-03.83%] _start	↓	-2.207 s	0	1	crt1.S:28
[-03.83%] main	↓	-2.207 s	+20.369 us	1	profile_func_instr.c:38
[-03.83%] dofuncs	↓	-2.207 s	+0.102 ms	2458	profile_func_instr.c:66
[-01.41%] func1	↓	-0.813 s	-0.813 s	2458	profile_func_instr.c:80
[-02.42%] func2	↓	-1.394 s	-1.394 s	2457	profile_func_instr.c:104
[<0.01%] options	↑	+15.848 ns	+15.848 ns	1	profile_func_instr.c:128
[] app_prof_hdlr		0	0	1	profile_func_instr.c:156
[] 0x0804a1ef@profile_func_instr		0	0		@profile_func_instr
[] __prof		0	0		@unknown

↑ 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

All content copyright
QNX Software Systems Limited,
a subsidiary of BlackBerry

Application Profiling

2020/10/02 R12

28



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:

Topics:

Overview

Sampling and Call Count Profiling

Function Instrumentation Profiling

→ Conclusion

NOTES:

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: