Luke Stackwalker User's Guide

Table of Contents

1 Getting Started

Luke Stackwalker is a C/C++ source code profiler. It is meant to help you to optimize your programs by showing what part of your program consumes the most CPU time. In addition to that, Luke Stackwalker can help you understand how and why the CPU intensive parts of the program get called.

Luke Stackwalker may be able to profile programs written in other compiled languages than C or C++ as long as the compiler produces debug symbol information that is compatible with Microsoft's dbghelp debugging APIs.

To profile your program with Luke Stackwalker, you need the following:

- Debug information for the program has to be available.

- You need Internet connectivity at least for the first time when you profile your program so that the debug information files for the system components used by your program can be downloaded.

2 Setting up your profiling project

First, select **Profile/Project Setup...** menu command or select the corresponding toolbar button (). The project settings wizard will be displayed:
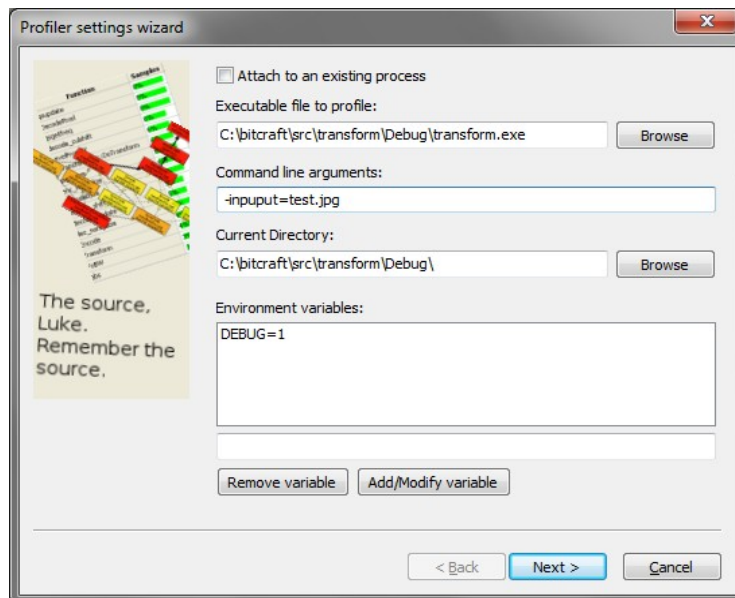
Figure 1: Profiler settings wizard page 1

On page 1, you should select the executable to be profiled, and define any command line options that Luke Stackwalker should use when launching it.

Next, you should define the directory in which the program should be launched in.

Last, you can define a list of environment variables that your program may need. The environment variables are defined in a format like VARIABLE=VALUE in the edit control above the **Add/Modify variable-**button. The variables listed here will be added to the standard environment on your computer. If an environment variable is defined in your standard environment and in the project settings, the version in the project settings overrides the system setting.

Alternatively you can specify that Luke Stackwalker should attach to an already running process instead of starting a new one. This is done by clicking the **Attach to an existing process-**checkbox.

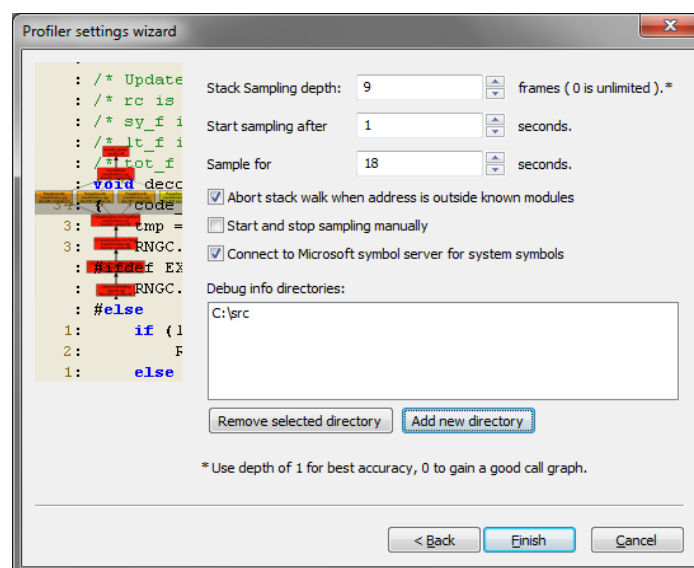Then click **Next** to go to the second page of the wizard:

Figure 2: Profiler settings wizard page 2.

On this page you can configure the following items:

- **Stack Sampling Depth**: This setting defines how deep Luke Stackwalker will examine the call stack of your program. A setting of 0 means that that the call stack is examined as deep as possible, and the call graphs of functions should be complete. A setting of 1 means that only the top-of-stack function is sampled and thus the call graphs to functions cannot be displayed. A smaller Stack Sampling Depth makes Luke Stackwalker slightly faster.

- **Start Sampling after**: This setting defines a delay in seconds from starting your program to when Luke Stackwalker begins collecting call stack sample data. Use this setting if your application does some fixed initialization that you want to avoid sampling.

- **Sample for**: This setting defines how long Luke Stackwalker collects call stack sample data from your program.

- **Abort sampling stack address is outside known modules**: This option stops walking the callstack if the return address from a function is not inside any of the known executable modules lodaed by the program. Having this option selected keeps the collected call graphs a lot cleaner in case the return address from some function cannot be correctly determined.

- **Start and stop sampling manually**: As an alternative to the two above settings, you can also manually start and stop the call stack data collection. This option is suitable for cases where the action to be profiled is also manually triggered

- **Connect to Microsoft symbol server for system symbols**:.Having this checkbox checked allows Luke Stackwalker to use the Windows symbol servers provided by Microsoft to download debug info files for system DLLs. When Luke Stackwalker has debug information available for system DLLs, it can follow the call stack from Windows system components to the source code in your program. Luke Stackwalker will cache the debug information files from the Microsoft symbol server to c:\websymbols\. If the system debug info files have been downloaded once, it is not necessary to keep this setting on; as a matter of fact, the profiling will probably even start faster if the setting is off - especially if you are working without Internet connectivity.

- **Debug info directories**: If Luke Stackwalker cannot find debug information for some part of your program, add the directories in which you have the debug info files to this list. Normally it is not necessary to use this list.

Once you have all the settings covered, press finish to close the wizard. Now you should probably save your settings using the **File/Save Project Settings...** menu command or the corresponding toolbar button ().

3 Profiling your program

When you are ready to profile your program, choose **Profile/Run** from the menu or press the toolbar button.

If you have chosen to attach to an already running program, the following dialog box is shown:
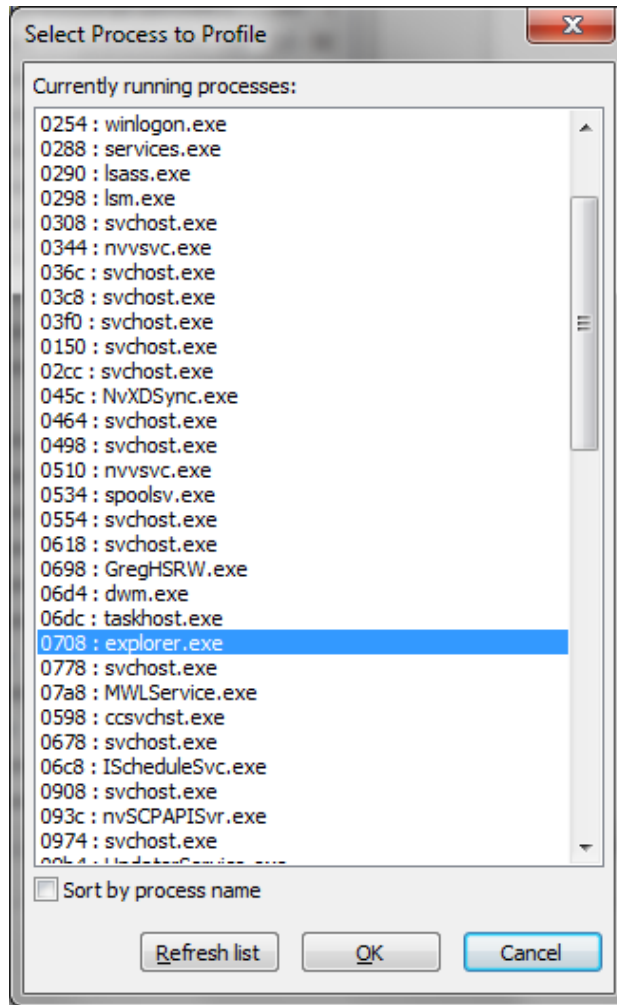
Figure 3: Select Process to Profile - dialog box

The dialog displays the processes running on the computer, with the process ID first, then the name of the program. Select the process you want to profile from this list and press OK.

Now the profiling progress dialog appears, and if you have specified a sampling start delay, the dialog will first count down that delay.

Next, Luke Stackwalker will load debug information files for the modules used by your program. The details of the debug info load process can be seen in the log window, like this:

```
C:\WINDOWS\system32\IMM32.DLL:IMM32.DLL (76390000), size: 118784, SymType: 'PDB',
PDB: 'C:\websymbols\imm32.pdb\F7A5B5DB13324153B57AAF340C77EA512\imm32.pdb'

C:\WINDOWS\system32\avgrsstx.dll:avgrsstx.dll (10000000), size: 20480, SymType: '-nosymbols-',
PDB: ''
```

When debug information has been loaded, the actual sampling will begin and the progress dialog displays how many stack samples have been collected and how much time is still remaining:
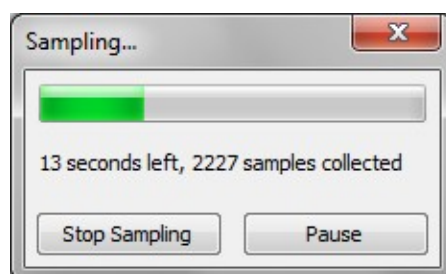

Figure 4: Profiler progress dialog

During the sampling, the log may show an occasional error message like this:

```
ERROR: SymGetSymFromAddr64, GetLastError: 126 (Address: 3EBE26EA)
```

It means that Luke Stackwalker was not able to determine the function name or source code file name and line number from an address in the call stack. Some such errors are not a problem, but if a very large amount of errors is displayed, the accuracy of the profile data may be bad.

4 Viewing the profile data

Once the profiling is finished, Luke Stackwalker displays the profile data. In the toolbar, the threads combo box allows you to select which thread's sample data is displayed. The threads are displayed in the combo box sorted by the amount of CPU time they have consumed.

By default, the samples collected from only one thread are displayed. You can select which thread's information is displayed by clicking on the thread ID in the combo box. You can also see the sum of the sample data from multiple threads by selecting those threads by pressing Ctrl+left mouse button. If you select multiple threads, the sample data of all the selected threads is summed in the profile display.
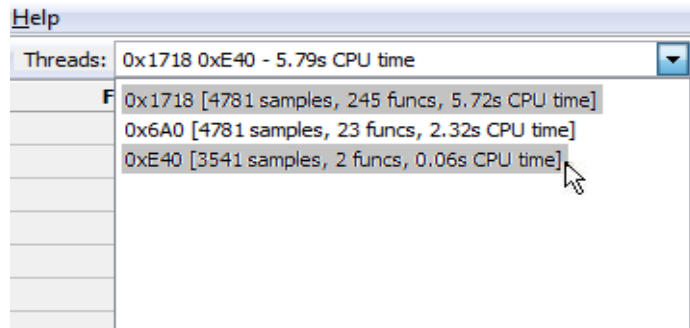

Figure 5: Selecting multiple threads from the toolbar Threads combo box

When you have selected the thread(s) to display, Luke Stackwalker displays the list of functions that have been sampled being on top of the call stack. The function list is sorted by the number of samples, the function with largest amount of samples (=most CPU time consumed) on the top.

Function	Samples	Src File	Lines	Module
CTransformDoc::DoTransform	613	C:\bitcraft\src\transform\transformDoc.cpp	526 - 651	transform
qsupdate	610	c:\bitcraft\src\transform\range\qsmodel.c	186 - 190	transform
ToBW	544	C:\bitcraft\src\transform\transformDoc.cpp	53 - 56	transform
qsgetfreq	510	c:\bitcraft\src\transform\range\qsmodel.c	158 - 159	transform
qsgetsym	476	c:\bitcraft\src\transform\range\qsmodel.c	166 - 179	transform
decode_culshift	476	c:\bitcraft\src\transform\range\rangecod.c	297 - 306	transform
enc_normalize	455	c:\bitcraft\src\transform\range\rangecod.c	156 - 180	transform
LevelFromXY	444	C:\bitcraft\src\transform\transformDoc.cpp	146 - 155	transform
DecodePixel	429	C:\bitcraft\src\transform\transformDoc.cpp	344 - 371	transform
encode_shift	395	c:\bitcraft\src\transform\range\rangecod.c	207 - 220	transform
EncodeByte	365	C:\bitcraft\src\transform\transformDoc.cpp	267 - 272	transform
Transform	328	C:\bitcraft\src\transform\transformDoc.cpp	60 - 107	transform
dec_normalize	312	c:\bitcraft\src\transform\range\rangecod.c	269 - 275	transform

Figure 6: Profile view

The profile display has the following information:

- Function name
- Either the absolute number or the percentage of all (top-of-stack) samples in the selected threads that were in this function. You can switch between these two display modes by selecting one of the commands "**Show Samples as Percentages**" or "**Show Sample Counts**" in the **View** menu.
- Source file name for the function
- First and last source line that were sampled in this function
- The name of the executable module in which the function was sampled

The profile display only displays functions that were sampled at the top of the call-stack; this means that the displayed profile only takes into account the time that was spent within the function itself, not the time spent in other functions called from within that function.

Clicking on a function in the profile display does two things:

- if the source code for the function can be found, the source file is opened in the source code view.
- the call graph to the selected function is shown in the call graph view
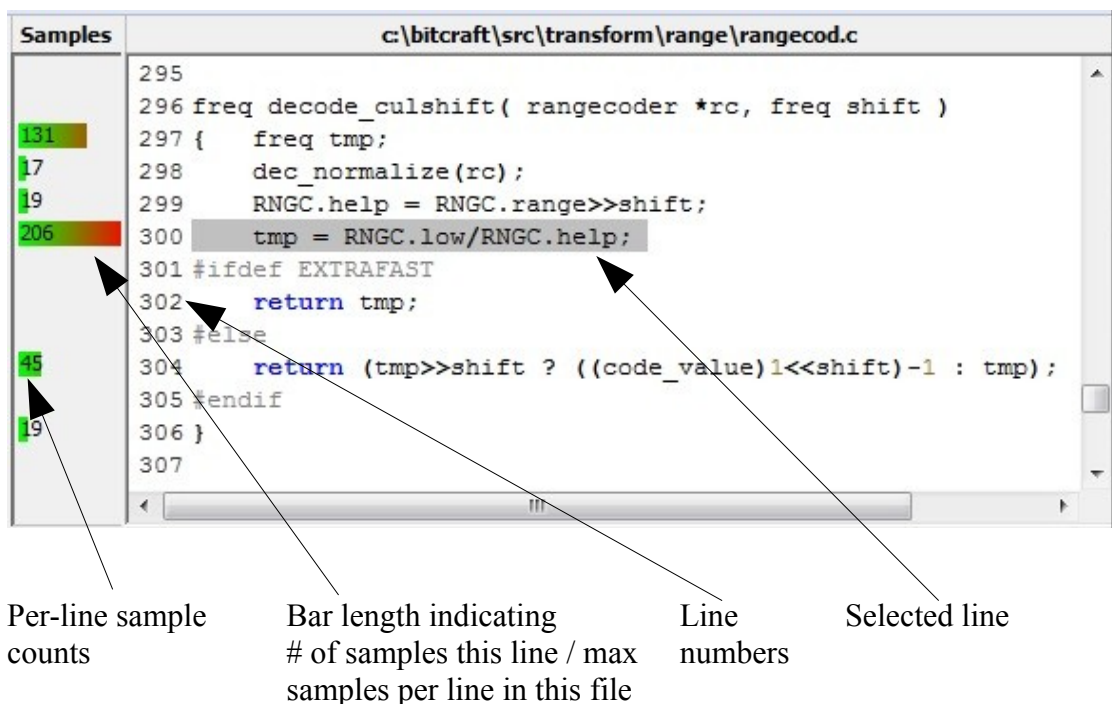
4.1 Source Code View

Figure 7: Source code view

On the left of the source code view, Luke Stackwalker shows the per source line sample counts (or percentage of total samples in this file) as numbers. The bar length shows the number of samples on the current source line relative to the maximum number of samples per source line in the current source file.

The source code line that is selected in the source display is the one with the most top-of-stack samples within that function.

Especially when profiling a program that has been compiled with full compiler optimizations on, the accuracy of the per-source code line sample counts have to be taken with a grain of salt. Optimizing compilers typically interleave machine code from multiple adjacent source lines to achieve the best possible performance, so this is likely to affect the source line sample counts.

4.2 Call Graph View

The call graph view displays all the paths in the code that led to the calling of the selected function in the profile view.

The color of the call graph nodes and the arrow line style and thickness in the call graph view indicate how many times the node or edge in the graph was sampled: Red color indicates most samples, and yellow the least. A thin, dashed line in the call arrows indicates the least amount of calls, and a thick continuous line indicates the most calls. The color coding only takes into account the currently displayed call graph. In other words, the coding is relative to the displayed call graph; also the sample counts in the caller functions count only the number of times those functions have been on the call stack calling the selected top-level function.
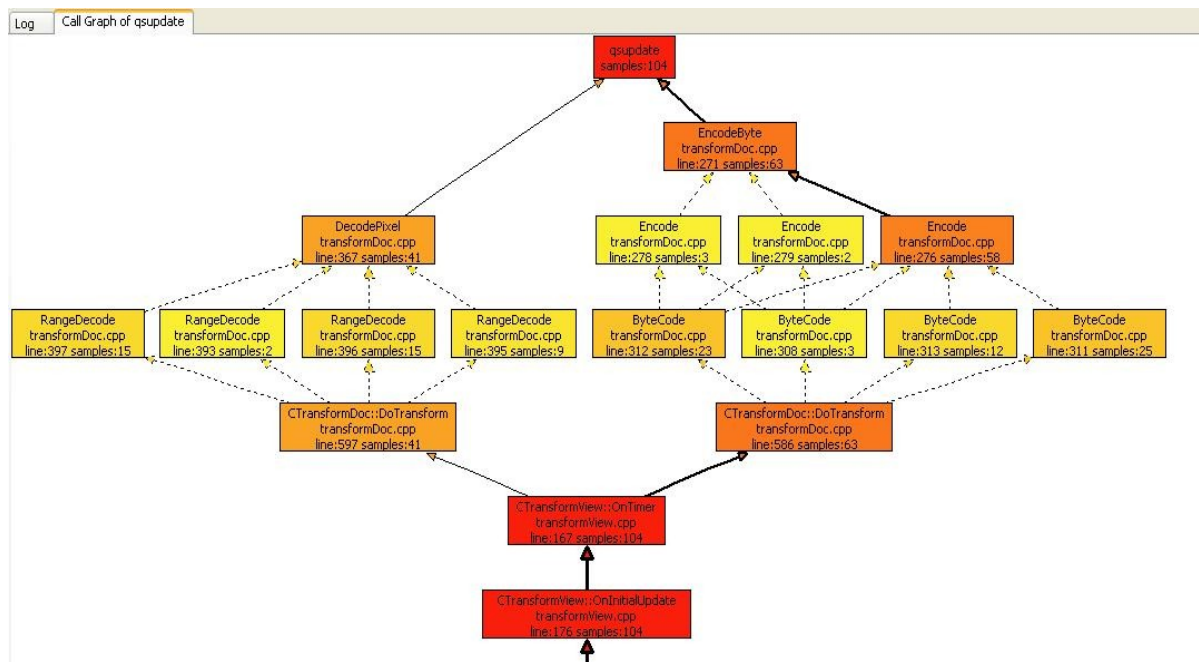

Figure 8: Call graph view

Clicking on the caller functions in the call graph opens the function into the source code view, if the source code for that function is available. The source code line that gets selected in the source code view is the line that corresponds to the call graph node clicked on.

If the call graph view does not properly display the call graph, the is usually caused by one of the following reasons:

- 'The 'Stack Sampling Depth' project setting is set to 1 (or some other small number). Luke Stackwalker only decodes the stack to the depth specified by that project setting, it also limits the call graph depth.

- Luke Stackwalker could not load the debug information for the module containing the function. It may help to explicitly specify the directory containing debug information for the

module in the 'Debug info directories' project setting.

- Compiler optimizations may have prevented Luke Stackwalker from decoding the call stack. Sometimes the best strategy in understanding the performance behavior of a program is to profile it in two steps:

    o First profile an optimized build (with a stack sampling depth of 1) to just collect the profile information.

    o Then profile a debug (non-optimized) build (with a stack sampling depth of 0 = infinite) to collect the call graph information.

You can then use the first profile to find out which functions in the program are the big CPU hot spots, and use the second profile to understand how the hot spots were called.

4.3 Abbreviating long function names

Sometimes the profile and call graph views become unnecessarily wide or even unreadable when the profile contains functions with very long names. For an example, that can happen when profiled C++ code contains a lot of STL usage. In such cases, the name abbreviation feature in Luke Stackwalker should help. Just open the Symbol Abbreviations dialog using the View/Abbreviate Name menu command, type in the function name – or a part of a name you want to abbreviate and the shorthand version for it, then press the Add/Modify button. The abbreviations mechanism is actually like a generic search/replace mechanism that works in the profile and call graph views.
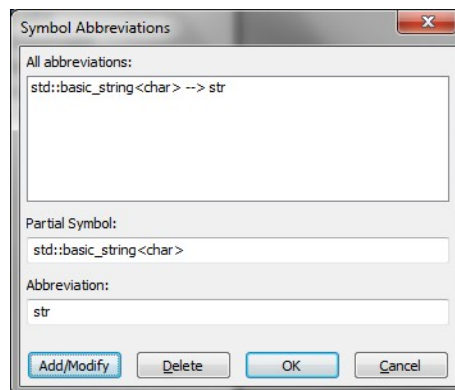

Figure 9: Abbreviations dialog

The symbol abbreviations settings are saved into the project settings file.

4.4 Ignoring a function from the profile calculation

Sometimes, the top of the profile is 'corrupted' by a function equivalent to the Sleep() windows system call that may take a large portion of the execution time of some of the selected threads, – but does not consume much CPU time. In a case like that, you probably want to see how the profile would look like if that function was not present in the program. You can do that by selecting the function in the profile view and then choosing the **View/Ignore/Count in this function**-menu command. The effect of that is demonstrated by the picture below.

Function	Samples
ZwWaitForSingleObject	1775
NtDelayExecution	123
WholePixelDistSSE2A_16	51
CPutPict::putnonintrablk	28
CMpegEnc::simplesearch	17
pred_comp_16x16_SSE2	16
QuantNonIntraSSSE3	15
v8_dct_8x8_fwd_16s	14
pred_comp_8x8	13
CMpegEnc::image_analysis_MB	10
HalfPixelDist01	9

Function	Samples
ZwWaitForSingleObject	ignored
NtDelayExecution	ignored
WholePixelDistSSE2A_16	51
CPutPict::putnonintrablk	28
CMpegEnc::simplesearch	17
pred_comp_16x16_SSE2	16
QuantNonIntraSSSE3	15
v8_dct_8x8_fwd_16s	14
pred_comp_8x8	13
CMpegEnc::image_analysis_MB	10
HalfPixelDist01	9

Figure 10: The effect of the ignoring a function in the profile view.

Selecting the same menu command again returns the function back to the profile.

4.5 Saving and loading profile data

You can save the profile data you have collected by using the File/Save Profile Data... menu command and then later load it for viewing using the File/Load Profile Data... menu command. It is often a good idea to save a baseline performance profile of your program when you start optimizing it and then save new profiles during the optimization process. That way you can compare the performance of the original program to your current working product. While Luke Stackwalker does not provide a profile comparison tool, it is easy enough to open two copies of the program and view the two profiles in them.