SpeedDemon Profiler v1.2

Documentation

Table Of Contents

I. Introduction

A. What SpeedDemon Profiler Does

SpeedDemon Profiler is a cross-platform profiling tool. It is comprised of two parts: a library that you link your program with, and a tool for viewing results. There are two types of profilers out there:

1. Those that use some form of instrumentation to intercept all of the function calls as they are being made and perform timing on those functions. This is a 'classic' profiler design.
2. Those that sample the running program's instruction pointer at random intervals to find out where the program is spending its time. This is a 'monte-carlo' profiler design.

Each profiler design has its advantages and disadvantages.

The classic profiler has the advantage that it can track the entire call graph of the program and leaves nothing to chance. The disadvantage to the classic profiler is that it generally has high overhead. In some implementations it can slow the program's run-time speed down by over 20 times.

The monte-carlo profiler is a low-overhead solution because sampling the execution of the program at random intervals is a non-intensive task. The advantages are obviously speed, but also that the non-invasiveness doesn't change the overall weighting of procedure execution time, which is important for programs that depend heavily on waiting on external, non-profiled, timesinks. For example, video games, which may rely heavily on how long it takes the graphics card to process a list of polygons. The disadvantages are that you may miss functions, and you can't get a real call graph of what was executed.

In the end, after weighing the two against each other, we opted to go with the 'classic' design for SpeedDemon Profiler. A future product may implement the second mechanism, possibly as an extension to SpeedDemon. Our implementation of the classic profiler properly handles recursive functions, multithreaded environments, abnormal terminations, and non-local execution like exceptions and longjmp() in C. Our run-time overhead is less than 4-5 times that of the original execution time when the CPU is fully utilized, making it one of the best profilers out there time-wise. We expect to do even better in future releases.

SpeedDemon records the time that each function you profile takes (F time, or Function Time), as well as the amount of time the function takes with the procedures it calls (F+C time, or Function And Children Time). These profiling results are written out to a ".spd" file, which is later opened with the *SPDReader* tool. In *SPDReader,* you can see all of the results in tabular form, and details for each functions. You can filter the results by thread, and you can focus on a subtree of the execution, showing only the cumulative times and functions underneath a chosen function in the call graph.

B. Supported Environments

Currently SpeedDemon Profiler v1.0 comes with support for profiling the following targets:

Win32 (Windows 2000, Windows XP, Windows .NET Server 2003, etc)
- X86 (Pentium and higher)

Win64 (Windows XP x64 Edition, Windows .NET Server 2003, etc)
- x64 (AMD x64)

Windows Mobile 2003 (Windows CE 4.2, Pocket PC, and SmartPhone)
- PXA255 and PXA263 (PXA25x and PXA26x series)
- PXA270 (PXA27x series)

Windows Mobile 2005 (Windows CE 5.0, Pocket PC, and SmartPhone)
- PXA255 and PXA263 (PXA25x and PXA26x series)
- PXA270x (PXA27x series)

The libraries may also work on other ARM or XScale architectures, and on other version of Windows CE, and possibly on Windows 95/98/ME, though they have not been tested and are completely unsupported. Feel free to report to us through the forum your experiences testing SpeedDemon Profiler on other platforms, or if you have requests for future versions to support certain platforms.

C. Contents of the Software Package

docs/faq.pdf
    Frequently Asked Questions in PDF (Adobe Acrobat) format
docs/manual.pdf
    This manual in PDF (Adobe Acrobat) format
docs/faq.odt
    Frequently Asked Questions in ODT (OpenDocument) format
docs/manual.odt
    This manual in ODT (OpenDocument) format
docs/faq.doc
    Frequently Asked Questions in DOC (Microsoft Word) format
docs/manual.doc
    This manual in DOC (Microsoft Word) format

lib/WCE/PXA255/speeddemonlib.lib
    Profiler static library version for WCE/PXA25x/PXA26x

lib/WCE/PXA270/speeddemonlib.lib
    Profiler static library version for WCE/PXA27x

lib/Win32/X86/speeddemonlib.lib
    Profiler static library version for Win32 running on X86.

lib/Win64/x64/speeddemonlib.lib
    Profiler static library version for Win64 running on x64.

spdreader.exe
    Viewing and analysis tool for profiling results. Runs on Win32 systems.
Msvcp80.dll
msvcr80.dll
msdia80.dll
    Support libraries for *SPDReader*.

II. How To Prepare Your Program

A. Compiling

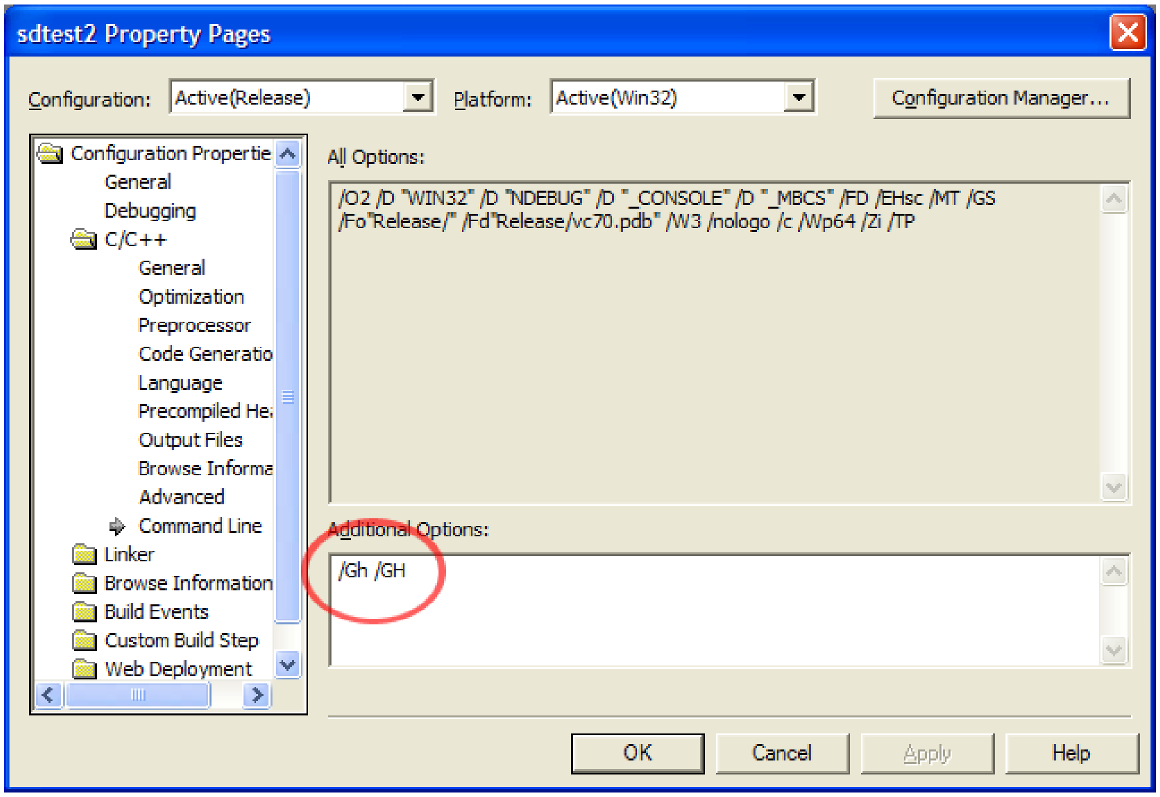
The means by which a classic profiler works is by 'instrumenting' each function or each function call site in your program. Instrumenting means that a calls to an 'entry' function and an 'exit' function are added to your program in many places and the time it takes to get from the 'entry' to the 'exit' is measured. A good profiler compensates for the amount of time it takes in the 'entry' and 'exit' functions so that time doesn't show up in the analysis.

Some profilers use post-build instrumentation, or runtime instrumentation to add their hook functions. This requires modifying the output executable either before it is run, or after it is already running. This is incompatible with several types of programs, and is error-prone. Often times, it also introduces a size restriction, and many times new compilation idioms in new versions of compilers quickly obsolete these instrumentation methods. While these methods work on medium sized programs, often times they do not work on larger programs with many shared libraries.

SpeedDemon Profiler utilizes the fact that most modern compilers support automatic instrumentation of functions (and some support instrumentation of function call sites as well). Microsoft Visual Studio 7.0 and higher support it, as well as eMbedded Visual C++ 4.0. Other compilers such as GCC support it as well. While runtime instrumentation is more convenient for the user, the unparalleled reliability of compiler-time instrumentation made it an obvious choice for SpeedDemon Profiler.

For all environments, you should build with debug symbols. For Win32, Win64 and Windows CE, this means compiling with the 'Program Database' debugging symbols option turned on in the compiler.

For Microsoft Visual Studio building a Win32 target program, enable the compiler flags **/Gh** and **/GH** in all of the projects you wish to profile. This adds calls to '_penter' and '_pexit' (function instrumention), which SpeedDemon Profiler provides for you.



Configuration: Active(Release)

Platform: Active(Win32)

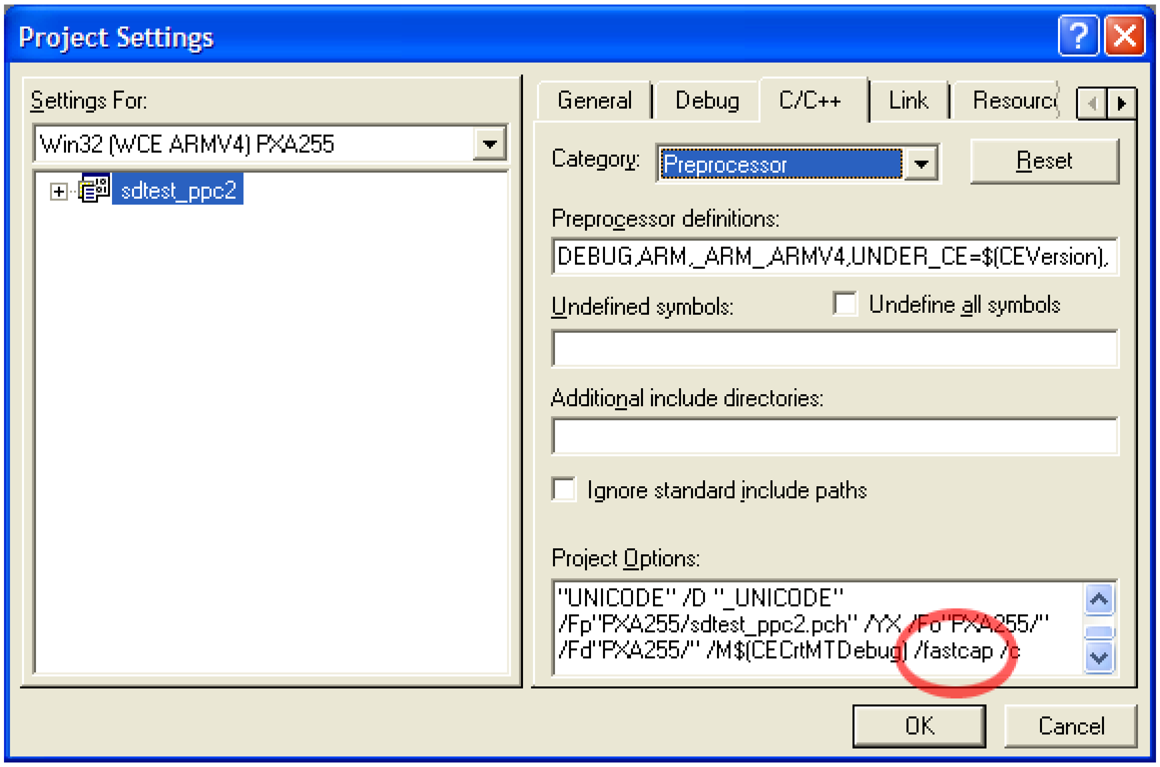
All Options:

```
/O2 /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_MBCS" /FD /EHsc /MT /GS
/Fo"Release/" /Fd"Release/vc70.pdb" /W3 /nologo /c /Wp64 /Zi /TP
```

Additional Options:

```
/Gh /GH
```

For Microsoft Visual Studio or eMbedded Visual C++ building a Windows CE target program, enable the compiler flag **/fastcap** or **/callcap** on the projects you wish to profile. Using **/fastcap** enables call-site instrumentation. Using **/callcap** enables function instrumentation. Call-site instrumentation is more accurate and is recommended anywhere it is available, as it records some information for functions that are outside of the profiling scope, such as system library calls. Refer to the compiler's documentation for more details on these features.



Settings For: Win32 (WCE ARMV4) PXA255

sdtest_ppc2

Category: Preprocessor

Preprocessor definitions:
DEBUG,ARM,_ARM_,ARMV4,UNDER_CE=$(CEVersion),

Undefined symbols:

Undefine all symbols

Additional include directories:

Ignore standard include paths

Project Options:

```
"UNICODE" /D "_UNICODE"
/Fp"PXA255/sdtest_ppc2.pch" /YX /Fo"PXA255/"
/Fd"PXA255/" /M$(CECrtMTDebug) /fastcap /c
```

B. Linking

Once you have configured your projects to compile with the instrumentation flags, calls to the SpeedDemon Profiler hook functions will be undefined externals in your program. To define them, link with the appropriate static or import library from the 'lib' folder. The static library version can be linked with as many targets as you like and adds minimal overhead per build target.

For all environments, you should build with debug symbols. For Win32, Win64 and Windows CE, this means ensuring that both the compiler and linker settings are set to generate Program Database (.pdb) debug symbols.

If you have an executable that links in several dynamic libraries, and you wish to profile the entire system, you should ensure that the static library gets linked into every single target you wish to profile. If you do not do this, you will be missing information from your profiling results.

The dynamic library version is primarily for debugging. If linking with the static library version causes any problems, try using the dynamic library version of the SpeedDemon. Future releases of SpeedDemon may eliminate this option.

One you have linked with SpeedDemon Profiler, it's time to run your program and put it through the motions. There are no other configuration tasks required to get SpeedDemon Profiler to work.

III. Running The Program

A. Execution

When you run your program after it has been compiled and linked with SpeedDemon Profiler, it will time the profiled modules as it executes. You can expect the program to run slower, by possibly up to 4-5 times where it is extremely CPU intensive. The calculations are written to the disk in the same location as the executable, or in the current working directory. The profile data is given the name of the executable file, except with the extension '.spd'. This file is overwritten every time the profiled executable runs, so you should copy the data somewhere else if you intend to run the program multiple times.

One should note that the size of the generated profiler data file is relatively small compared to that of other profilers, and that its size does not depend on how long you run your tests, only on how large the program is.

B. What The Profiler Collects

SpeedDemon Profiler collects all kinds of program and function timing information. This information is visible in the Statistics view of SPDReader, as the list columns. Specifically, the categories are:

Source – The source file and line number where the function is located.
# Of Calls – The number of times the function was called
Min. F Time – The minimum time spent per call to this function excluding its children (function calls that it makes).
Avg. F Time – The average time spent per call to this function excluding its children.
Max. F Time – The maximum time spent per call to this function excluding its children.
Min. F+C Time – The minimum time spent per call to this function including its children.
Avg. F+C Time – The average time spent per call to this function including its children.
Total F Time – The total time spent during all calls to the function, excluding its children.
Max. F+C Time – The maximum time spent per call to this function including its children.
Total F+C Time – The total time spent during all calls to the function, including its children.

Avg. F/Size – A measure of the function's time per line of code. This metric measures the amount of time spent in the function with relation to its length. In theory, long functions should take a long time, and short functions should take a short time. Hence, if this number ever gets very large, you may have tight loops in small functions that are taking up an disproportionate amount of time, and are worth a second look at the algorithm used.

All of this information is collected on a per-thread basis, and takes into account the loading and unloading of shared libraries.

IV. Using SPDReader

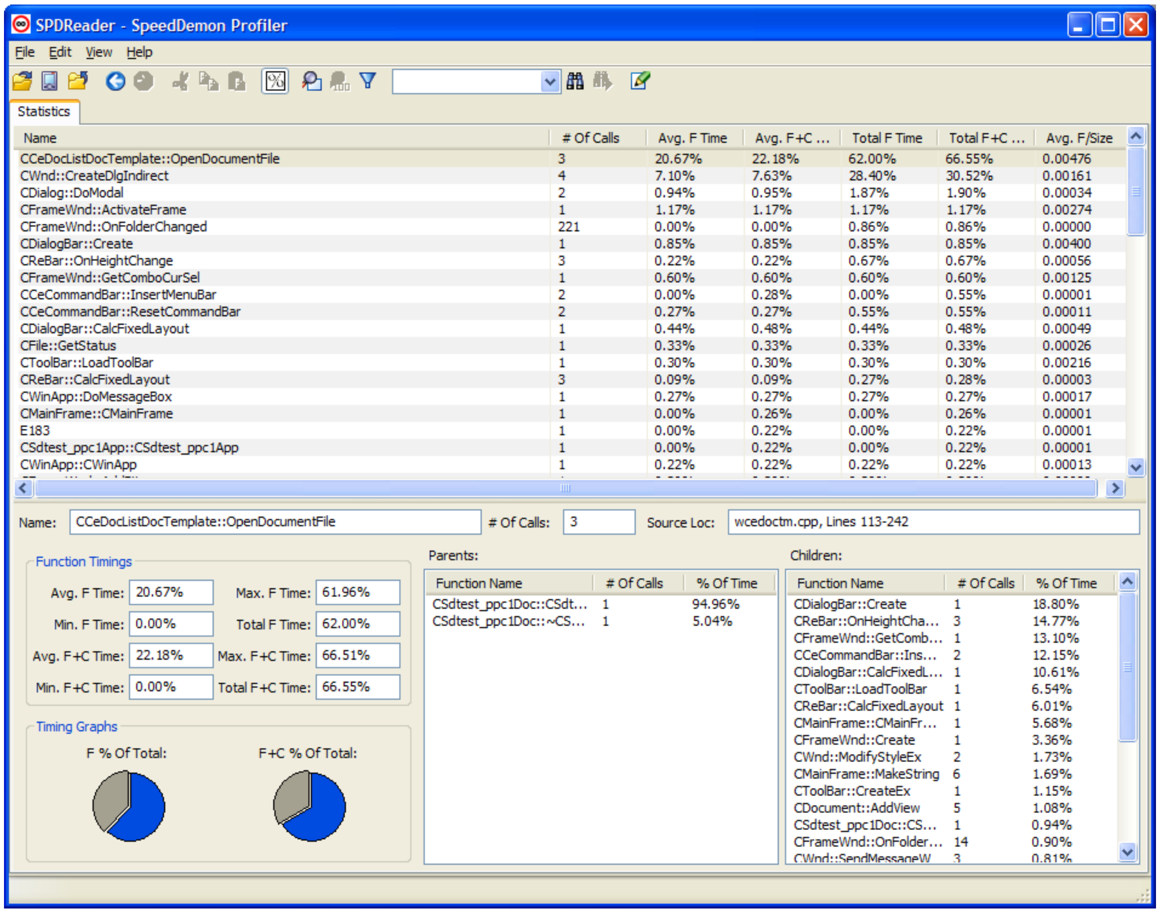
A. Opening The Data File

First, Launch SPDReader by running the *spdreader.exe* program. This is a Win32 binary and should be run on a Win32 machine, not on a handheld device. You should be running SPDReader on the same machine that compiled the program, in order for SPDReader to get at the debug symbol information, and the source code.

To start analyzing the results, you can either copy the “.spd” file over to your PC manually, from wherever it is located, or use the “Import File From Device” menu option to open the “.spd” file directly from your handheld device if that's where it is located. The “.spd” file should be located in the profiled program's working directory, or in the same directory as the executable (especially on handhelds where there is no notion of 'working directory').

When you import the data, you will be asked for a place to save the generated report file. The report file is essentially the same as the data file, except it has bound the function information from your debug symbols to the data. Which means, don't blow away your symbols or recompile your program between when you get the '.spd' file generated and when you open it with SPDReader and create the '.spr' file. The SPR file can be viewed at any time and does not require debug symbols, but the SPD file can not be imported unless the matching debug symbol files for that particular run can be found, so it's important to import right after the file is generated, and save the report somewhere

When you are done importing, or opening a saved report file, your screen should look like this:



Name	# Of Calls	Avg. F Time	Avg. F+C ...	Total F Time	Total F+C ...	Avg. F/Size
CCeDocListDocTemplate::OpenDocumentFile	3	20.67%	22.18%	62.00%	66.55%	0.00476
CWnd::CreateDlgIndirect	4	7.10%	7.63%	28.40%	30.52%	0.00161
CDialog::DoModal	2	0.94%	0.95%	1.87%	1.90%	0.00034
CFrameWnd::ActivateFrame	1	1.17%	1.17%	1.17%	1.17%	0.00274
CFrameWnd::OnFolderChanged	221	0.00%	0.00%	0.86%	0.86%	0.00000
CDialogBar::Create	1	0.85%	0.85%	0.85%	0.85%	0.00400
CReBar::OnHeightChange	3	0.22%	0.22%	0.67%	0.67%	0.00056
CFrameWnd::GetComboCurSel	1	0.60%	0.60%	0.60%	0.60%	0.00125
CCeCommandBar::InsertMenuBar	2	0.00%	0.28%	0.00%	0.55%	0.00001
CCeCommandBar::ResetCommandBar	2	0.27%	0.27%	0.55%	0.55%	0.00011
CDialogBar::CalcFixedLayout	1	0.44%	0.48%	0.44%	0.48%	0.00049
CFile::GetStatus	1	0.33%	0.33%	0.33%	0.33%	0.00026
CToolBar::LoadToolBar	1	0.30%	0.30%	0.30%	0.30%	0.00216
CReBar::CalcFixedLayout	3	0.09%	0.09%	0.27%	0.28%	0.00003
CWinApp::DoMessageBox	1	0.27%	0.27%	0.27%	0.27%	0.00017
CMainFrame::CMainFrame	1	0.00%	0.26%	0.00%	0.26%	0.00001
E183	1	0.00%	0.22%	0.00%	0.22%	0.00001
CSdtest_ppc1App::CSdtest_ppc1App	1	0.00%	0.22%	0.00%	0.22%	0.00001
CWinApp::CWinApp	1	0.22%	0.22%	0.22%	0.22%	0.00013

Name: CCeDocListDocTemplate::OpenDocumentFile # Of Calls: 3 Source Loc: wcedoctm.cpp, Lines 113-242

Function Timings

Avg. F Time:	20.67%	Max. F Time:	61.96%
Min. F Time:	0.00%	Total F Time:	62.00%
Avg. F+C Time:	22.18%	Max. F+C Time:	66.51%
Min. F+C Time:	0.00%	Total F+C Time:	66.55%

Parents:

Function Name	# Of Calls	% Of Time
CSdtest_ppc1Doc::CSdt...	1	94.96%
CSdtest_ppc1Doc::~CS...	1	5.04%

Children:

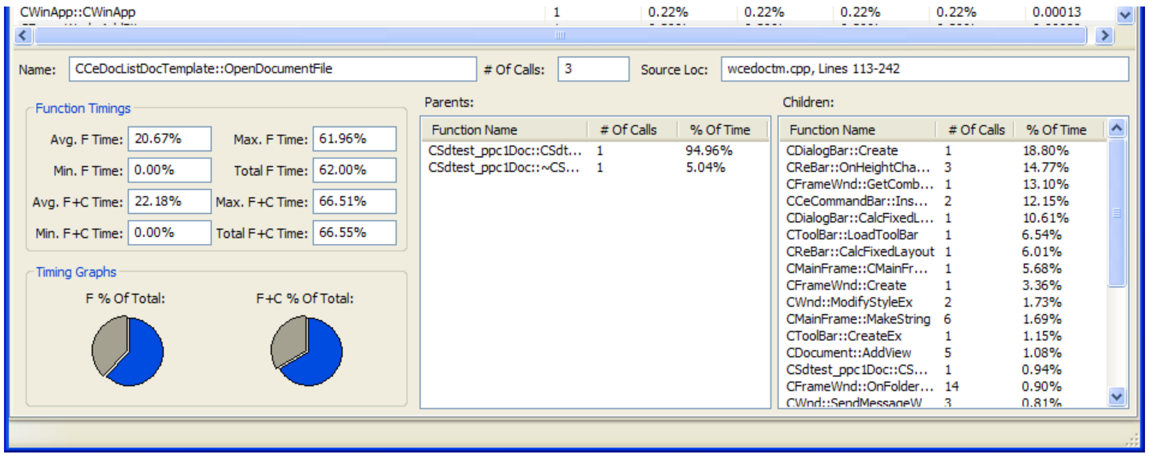
Function Name	# Of Calls	% Of Time
CDialogBar::Create	1	18.80%
CReBar::OnHeightCha...	3	14.77%
CFrameWnd::GetComb...	1	13.10%
CCeCommandBar::Ins...	2	12.15%
CDialogBar::CalcFixedL...	1	10.61%
CToolBar::LoadToolBar	1	6.54%
CReBar::CalcFixedLayout	1	6.01%
CMainFrame::CMainFr...	1	5.68%
CFrameWnd::Create	1	3.36%
CWnd::ModifyStyleEx	2	1.73%
CMainFrame::MakeString	6	1.69%
CToolBar::CreateEx	1	1.15%
CDocument::AddView	5	1.08%
CSdtest_ppc1Doc::CS...	1	0.94%
CFrameWnd::OnFolder...	14	0.90%
CWnd::SendMessageW	3	0.81%

B. Browsing The Results

The Statistics view shows you the functions that have been profiled and their various timings. You can sort the view by clicking on the list headers.

The time is displayed in decimal milliseconds by default, but by pressing the '%' icon, the 'Show Time Percentages' menu item, or hitting Ctrl-P, it will switch to being displayed as a percentage of the total profiled time.

The window below the statistics view is the function detail panel. For the function currently selected the statistics list, it shows all of the function timings as well as the list of the function's parents and children, with the relative percentage of time contributed to or from the parents or children. There are also pie charts giving a quick graphical depiction of the amount of time spent in the function.



CWinApp::CWinApp	1	0.22%	0.22%	0.22%	0.22%	0.00013

Name: CCeDocListDocTemplate::OpenDocumentFile # Of Calls: 3 Source Loc: wcedoctm.cpp, Lines 113-242

Function Timings

Avg. F Time:	20.67%	Max. F Time:	61.96%
Min. F Time:	0.00%	Total F Time:	62.00%
Avg. F+C Time:	22.18%	Max. F+C Time:	66.51%
Min. F+C Time:	0.00%	Total F+C Time:	66.55%

Timing Graphs

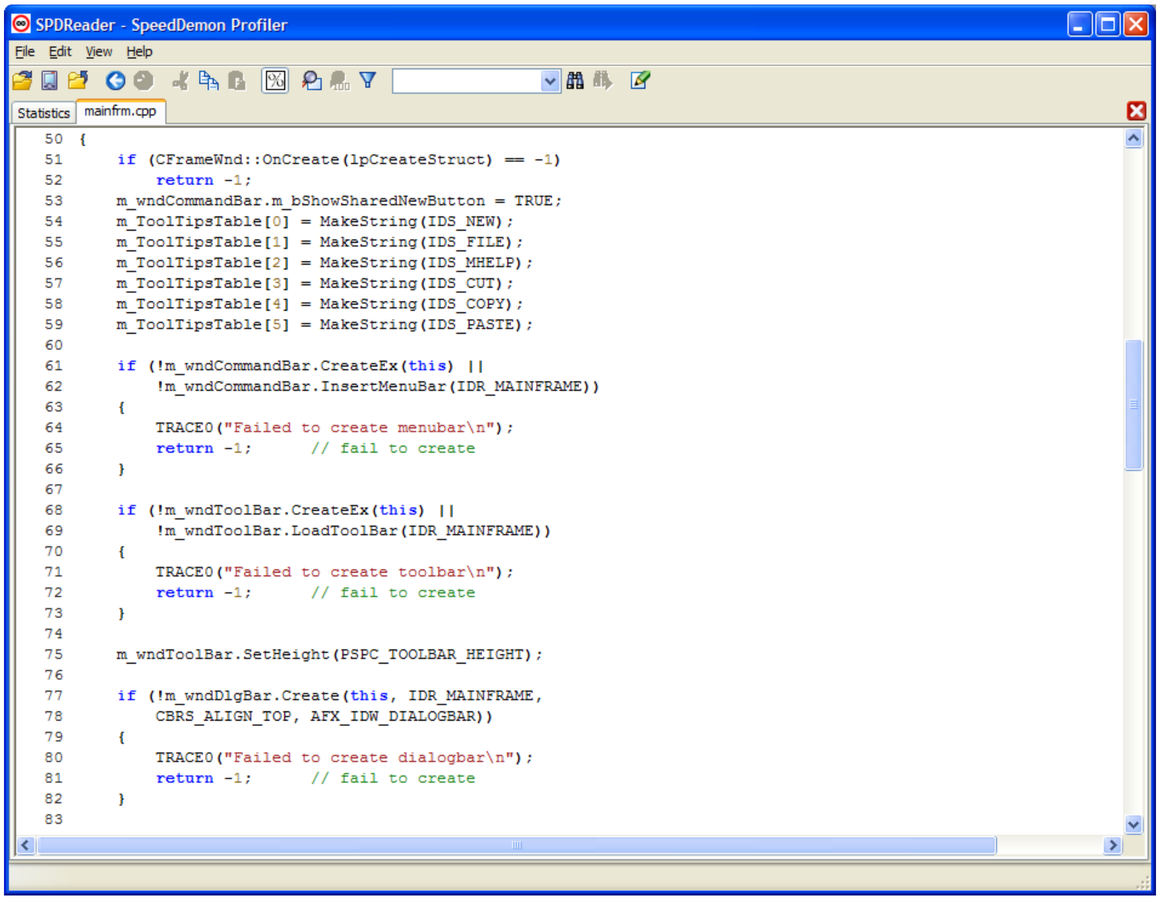
Parents:

Function Name	# Of Calls	% Of Time
CSdtest_ppc1Doc::CSdt...	1	94.96%
CSdtest_ppc1Doc::~CS...	1	5.04%

Children:

Function Name	# Of Calls	% Of Time
CDialogBar::Create	1	18.80%
CReBar::OnHeightCha...	3	14.77%
CFrameWnd::GetComb...	1	13.10%
CCeCommandBar::Ins...	2	12.15%
CDialogBar::CalcFixedL...	1	10.61%
CToolBar::LoadToolBar	1	6.54%
CReBar::CalcFixedLayout	1	6.01%
CMainFrame::CMainFr...	1	5.68%
CFrameWnd::Create	1	3.36%
CWnd::ModifyStyleEx	2	1.73%
CMainFrame::MakeString	6	1.69%
CToolBar::CreateEx	1	1.15%
CDocument::AddView	5	1.08%
CSdtest_ppc1Doc::CS...	1	0.94%
CFrameWnd::OnFolder...	14	0.90%
CWnd::SendMessageW	3	0.81%

By double-clicking on a function in the Statistics view, it will open up a syntax-colored text view containing the function's source file with the selected function highlighted. Click the corner 'X' to close the source view.



```
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    m_wndCommandBar.m_bShowSharedNewButton = TRUE;
    m_ToolTipsTable[0] = MakeString(IDS_NEW);
    m_ToolTipsTable[1] = MakeString(IDS_FILE);
    m_ToolTipsTable[2] = MakeString(IDS_MHELP);
    m_ToolTipsTable[3] = MakeString(IDS_CUT);
    m_ToolTipsTable[4] = MakeString(IDS_COPY);
    m_ToolTipsTable[5] = MakeString(IDS_PASTE);

    if (!m_wndCommandBar.CreateEx(this) ||
        !m_wndCommandBar.InsertMenuBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create menubar\n");
        return -1;      // fail to create
    }

    if (!m_wndToolBar.CreateEx(this) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }

    m_wndToolBar.SetHeight(PSPC_TOOLBAR_HEIGHT);

    if (!m_wndDlgBar.Create(this, IDR_MAINFRAME,
        CBRS_ALIGN_TOP, AFX_IDW_DIALOGBAR))
    {
        TRACE0("Failed to create dialogbar\n");
        return -1;      // fail to create
    }

```

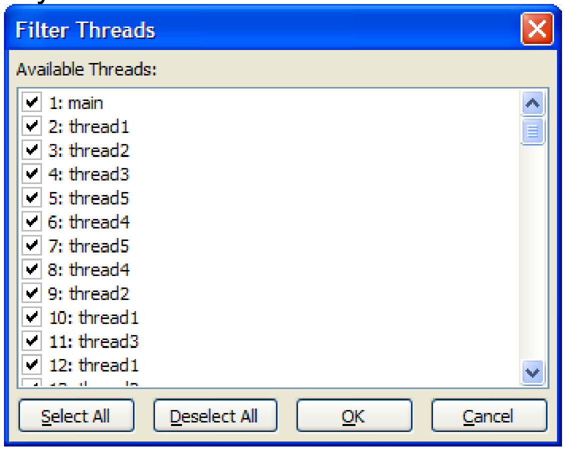
If you want to locate a particular function, there is a 'Find' and 'Find Next' functionality available from menu bar or from the toolbar. The toolbar also has a type-in 'quick find' box that you can search from. The search is case-sensitive, so ensure you are typing in what you are looking for with the correct capitalization.

C. Filters And Focusing

SPDReader can dynamically focus on a particular subtree of the profiled functions by using the 'Focus on Function' command. One can either choose it from the menu or toolbar, or right-click on the function in the Statistics view and choose it from the pop-up menu.

Focusing on a function removes all functions from the statistics panel and from the details except for those reachable from the chosen function. This function becomes the new 'root' of the profiling, and all function timings are adjusted appropriately. Focusing as implemented by SpeedDemon Profiler is more accurate than many other packages as it takes into consideration how the various functions in the subtree were reached and only includes time that resulted from being descendant from the new focus root (it is sensitive to the call stack).

One can also filter on a particular thread. This allows you to remove timing information for threads you don't care about:



- [x] 1: main
- [x] 2: thread1
- [x] 3: thread2
- [x] 4: thread3
- [x] 5: thread5
- [x] 6: thread4
- [x] 7: thread5
- [x] 8: thread4
- [x] 9: thread2
- [x] 10: thread1
- [x] 11: thread3
- [x] 12: thread1