Using Xperf to Take a Trace

Pigs Can Fly

Lets get to it! Here is how to take a basic trace then look at CPU and disk utilization. Its really simple, just three commands to turn on tracing, turn it off, and then view the trace.

First, from an elevated command prompt window, enable a basic set of the kernel events using this command:

```
xperf -on PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+INTERRUPT+DPC+CSWITCH -maxbuffers 1024
```

This command enables a set of events in the kernel and sets the maximum number of buffers to 1024. The default size for each buffer is 64K. So for this session, ETW will use up to 64MB of memory for ETW buffers. As buffers are filled with events, they are written to the log file in the background and then made available again for accepting events. By default, xperf sets the minimum number of buffers to 64. ETW will start with this many buffers and only allocate more buffers if needed. Events will only be lost if ETW cannot allocate more buffers and/or keep up with the event rate by writing data to the disk. By default, the kernel events are written to \kernel.etl on the current drive.

Next, do something interesting - it can be anything from opening Internet explorer and a web page, or compiling a program with Visual studio, to something more complex like opening three or four Microsoft Office applications and doing some work.

Run the following command when your interesting thing is done:

```
xperf -d foo.etl
```

This simple command will take 10 to 30 seconds (or possibly longer) because it merging the raw kernel event data with meta data and doing some other post processing. We call this 'stop and merge'. Here is what this command does:

1. Performs a 'run down', during which the kernel logs a set of events that describe the state of the system.

2. Turns off the kernel logger

3. Interlaces data from multiple trace files and the kernel trace.

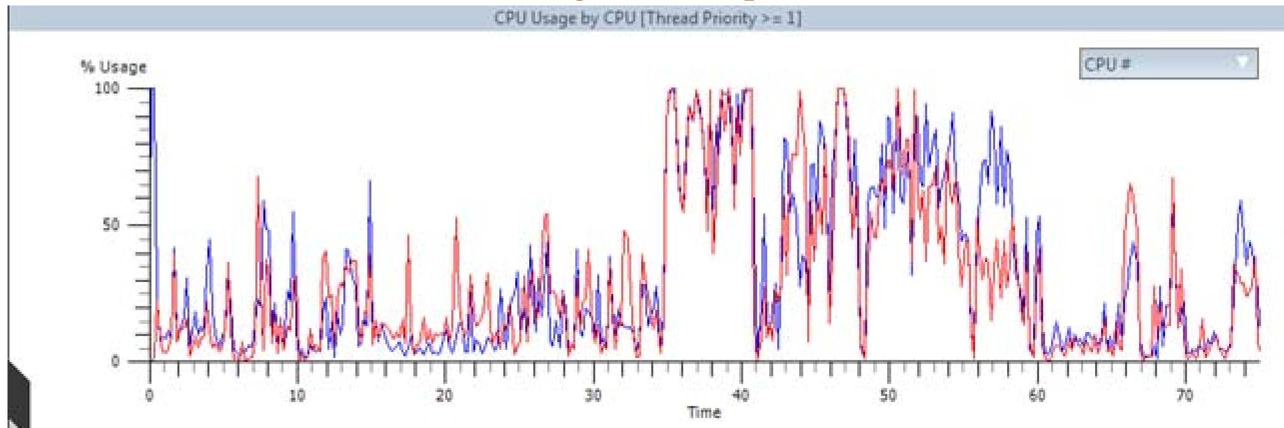
4. Adds some meta info to the trace needed for processing the trace on other systems. This data is saved in the trace as a set of synthetic events.

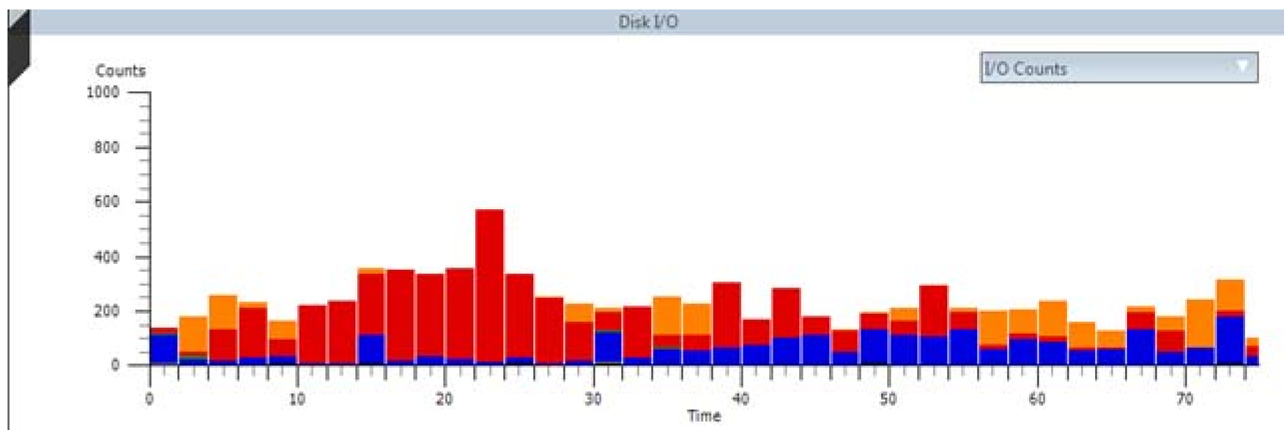
5. Saves the trace data into the file foo.etl (or the file name of your choice).

Finally, load the trace in the Performance Analyzer with the following command

```
xperf foo.etl
```

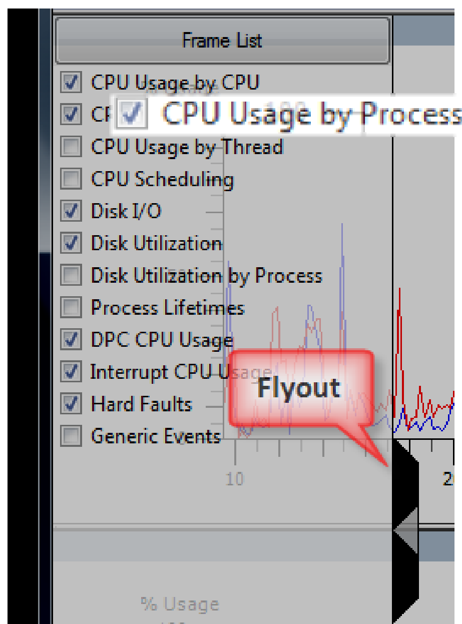
For this example, I took a trace of using Visual Studio 2008 to compile a program. Here are screen shots of the CPU Usage by CPU and for disk I/O counts.





Those are pretty interesting, but lots of things are running in the system, and I'd like to see just the CPU usage for Visual Studio itself.

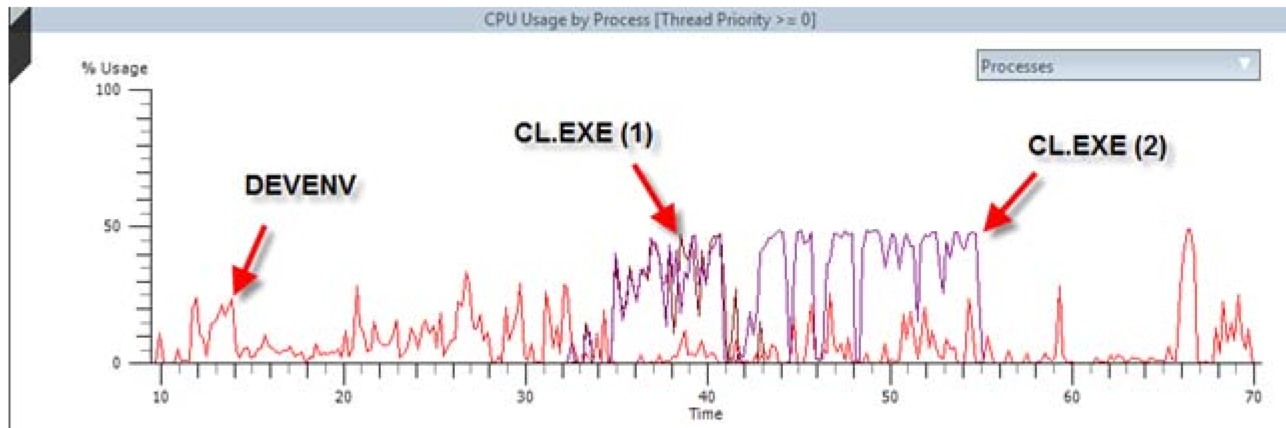
The CPU usage by process graph makes this easy, just click on the fly out control on the left of the window and select the CPU Usage by Process graph.



The fly out frame lists the graphs available for the events in the trace. If there trace doesn't contain events that are needed for a particular graph, then the graph is not shown.

Performance Analyzer will automatically save the graphs you have selected. You can change them at any time.

For my trace, the CPU usage for the DEVENV.EXE process and two CL.EXE processes looked like this.



DEVENV is the Visual Studio 2008 environment itself. The CL.EXE processes are the two compiler sessions it started, one for each CPU on my laptop.

This is a simple example that illustrates some key points:

1. The kernel events can be enabled and disabled at any time. There is no need to re-boot the system, log-out/log-in, or restart processes to use the kernel events, or any ETW event provider. ETW events from any source can be dynamically controlled at run time.

2. The xperf tools are designed for a post processing model, one where a trace is captured, then later analyzed. This is in contrast to an observational model where you watch dynamic charts, graphics, or tabular data as something occurs. The reason for this model is that ETW and the tools are designed for log time efficiency.

3. This model is also specifically designed for taking traces on one machine, then analyzing them on another machine. This ability is critical for running performance tests in a lab setting.

4. The tools let you look at both system wide activity and process specific activity.

Xperf Support for XP

"Do the xperf tools support XP or Windows Server 2003?" is a frequently ask question. The answer is no mostly, and yes for a few things.

xperf.exe can be used on Windows XP SP2, and Windows Server 2003 for turning tracing on and of, and merge kernel trace data with user mode traces into a single ETL file. These operations are simply called "trace control". NOte that the '-stackwalk' switch is not supported on XP because its kernel doesn't support capturing the stack on events, this is anew feature in the Vista kernel.

However, all operations that require trace decoding (and that's almost everything else), must be done on Vista or Windows Server 2008. This includes viewing traces in the Windows Performance Analyzer tool (xperfview.exe).

The next question is this "The xperf tool kit installer doesn't install the tools on XP or WS2003; how do I get the tools on those systems?"

The answer is simple: From a Vista or WS2008 installation copy xperf.exe and perfctrl.dll to the target system. This is all xperf needs to support trace control.

After you have generated an ETL file, you can then copy it to a Vista or WS2008 system for trace decoding.

For those of you interested in the long story....

Event Tracing for Windows was first introduced in Windows in 2000. Back then, the OS only supported a small number of events; very few other Window's components used ETW. In those days, event logging with ETW was in its infancy and the people that wrote event consumers generally also wrote the code that produced the events, or worked closely with those that did.

Back in the day, many event providers and consumer's simply used the same C/C++ data structures to produce and consume events. While simple, this sometimes broke because people wouldn't version the events correctly when the event structure changed. In short, if the producer and the consumer code wasn't kept in sync then things were busted. This got to be a real problem as ETW was used more broadly.

This problem was solved by using meta to describe events. This allowed event consumers to decode events without knowledge of the events binary format. This worked much better; it allowed the event provider author to change an event's binary format without breaking the consumer. In the XP time frame MOF files were used to describe events. For example, you can find he kernel's context switch event here.

Three things changed for Vista:

1. The entire Windows build system was updated so that every component was described by an XML based manifest. This included describing ETW events. We deprecated the MOF format and all new events were authored with XML based descriptions in their manifests using the Event Manifest Schema.

2. The use of ETW became very prevalent - many teams added event providers to their components and used them for Windows Event Logging (which is ETW based), performance work, diagnostics, and testing. For example, on my laptop, there are 985 registered ETW event providers. Use the "xperf -provider" command to see what is registered on your system.

3. Our team decided to make a major investment in ETW based tools as did other teams around Windows. This meant that meta information for events was very important as it enabled event providers and the consumers to be more decoupled and cohesive.

But, this posed one problem for us: do we fully support trace decoding on both Vista and XP? Or just on Vista? It was technically possible to keep trace decoding working on XP, but this would require shipping some Vista components with the tools because the required trace decoding infrastructure is

only present on Vista. Unfortunately, this isn't possible for all kinds of business, legal, and some technical reasons. It would have also doubled our test matrix.

After much discussion, we decided it was an easily workable compromise to support trace collection on XP, and require Vista or WS2008 for all trace decoding operations.

Using the Windows Sample Profiler with Xperf

Using the xperf tools, ETW, and the kernel sample profile interrupt all together provides a very effective and easy to use sample profiler for the analysis of both application and system wide performance. At each sample interrupt, the ETW sub-system captures the instruction pointer and the stack. This data is lazily and efficiently logged to an ETL file. Once the data is saved, it can be analyzed with Performance Analyzer.

Note: the examples in this post only works on Vista or Server 2008 32-bit; Prior operating system's do not support taking stack traces. Taking stack traces on 64-bit platforms will be the topic of another post.

Here is an example of profiling FS.EXE, a grep-like utility I've written. I use this tool for experimenting with various topics such as efficient I/O, well performing string matching algorithms, and instrumenting applications with ETW.

For this test, I put the following commands in a CMD file:

```
xperf -on PROC_THREAD+LOADER+INTERRUPT+DPC+PROFILE
-stackwalk profile
-minbuffers 16 -maxbuffers 1024 -flushtimer 0
-f e:\tmp.etl
fs.exe farglenorgin c:\coding\*.cpp *.h -s
xperf -d profile.etl
```

Since the commands are a bit long, I've separated them above and added line breaks to make them readable. Each command above should be on one line in your command file.

The first command turns on the kernel logger and enables the following events:

- PROC_THREAD flag enables the process and thread events. These mark the beginning and ending of each process and thread. The kernel provider guarantees that there will be a begin/end pair for every process and thread during the trace. Process and threads that exist before the trace was started or are still running when the trace is stopped also have these events.

- The LOADER flag enables the loader events that log when the kernel loads an image (an EXE or DLL)

- The INTERRUPT and DPC flags enable the ETW interrupt and DPC events which mark each interrupt and deferred procedure call which are routines that run at DISPATCH_LEVELt

- The PROFILE flag does two things; it turns on the systems sample profile interrupt and it enables the kernel's sample profile ETW event.

The other flags are important as well.

The -stackwalk profile parameter turns on ETW's stack walking feature for the sample profile event. Every time a sample profile event is triggered by the sample profile interrupt, ETW will capture the stack and save the data in the trace buffers.

The -minbuffers 16 parameter sets the minimum number of buffers that ETW will allocate for storing events. Note, you need at least two for each processor in you system.

The -maxbuffers 1024 parameter sets the maximum number of buffers ETW will allocate to 1024 - a total of 64MB.

The -flushtimer 0 parameter tells ETW to never flush the buffers based on timer, buffer's will only be written to disk when they are full.

The -f e:\tmp.etl parameter tells ETW to lazily write the full ETW buffers to e:\tmp.etl. This puts the log file on a different physical drive than the drive on which the experiment is running. This means that the writes that ETW uses to save the trace data do not occur on the interesting drive.

The second command simply runs the experiment. It searches for the string 'farglenorgin' in all my .CPP and .H files. I'm using a string that doesn't exist so I execute the worst case code paths in the application. Replace this command with a command to run your experiment, or a pause instruction so you can dork around with a graphical program.

The third command simply stops the kernel logger, merges the data and saves it in profile.etl.

NOTE: These commands need to be run from an elevated command prompt. Controlling ETW tracing requires administrative privileges.

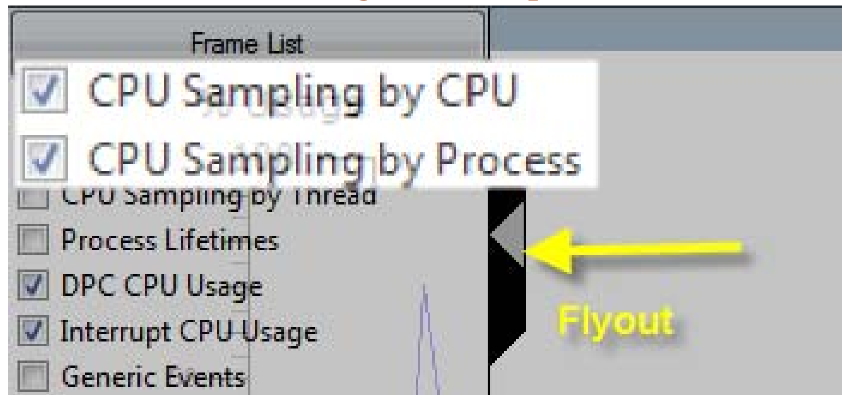
There is now one other thing to do before examining the data - setting the symbol path. Here is how I set the symbol path for this example:

```
set _NT_SYMBOL_PATH =
c:\coding\fs\release\fs;
SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
```

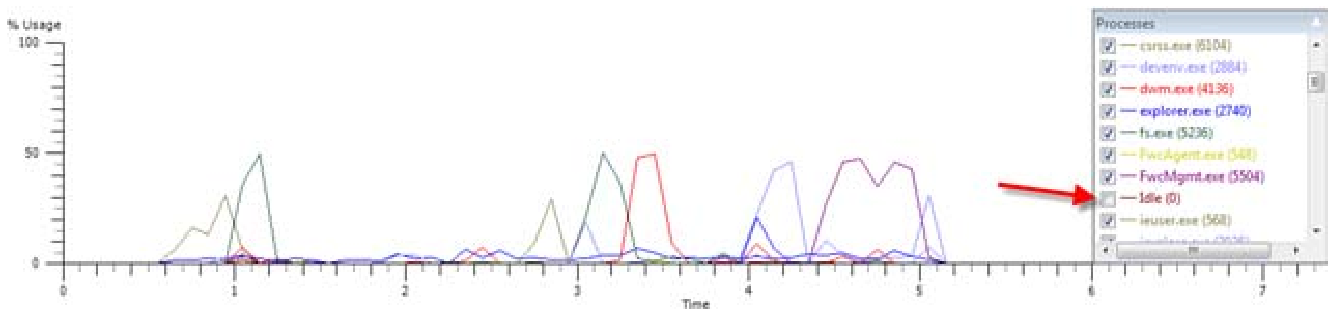
This tells the symbol decoder to look for symbols in the release build directory for FS.EXE and in the Windows public symbol server, caching the served symbols in c:\symbols. The xperf tools uses the symbol decoding libraries from the debugging tools for Windows. You can find more information on using symbols here.

Once the trace is taken and the symbol path is set, then simply open the trace in Performance Analyzer with the command "xperf profile.etl".

The CPU Sampling by Process graph is the most interesting graph for this example. To select the visible graphs, click on the flyout control on the left of the window, then select the CPU Sampling by CPU, and by Process graphs.



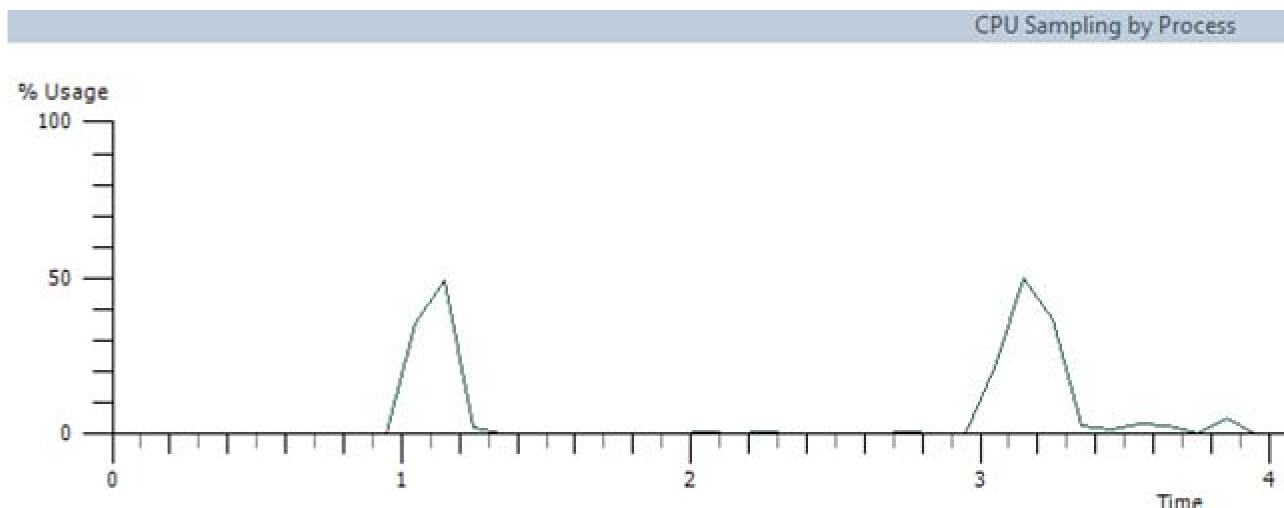
For his experiment, the CPU sampling by Process graph looks like this:



By default, all processes running during the trace are shown except the idle task (as seen above). You can change which processes are displayed or hidden by by using the check boxes in the legend drop down as shown above.

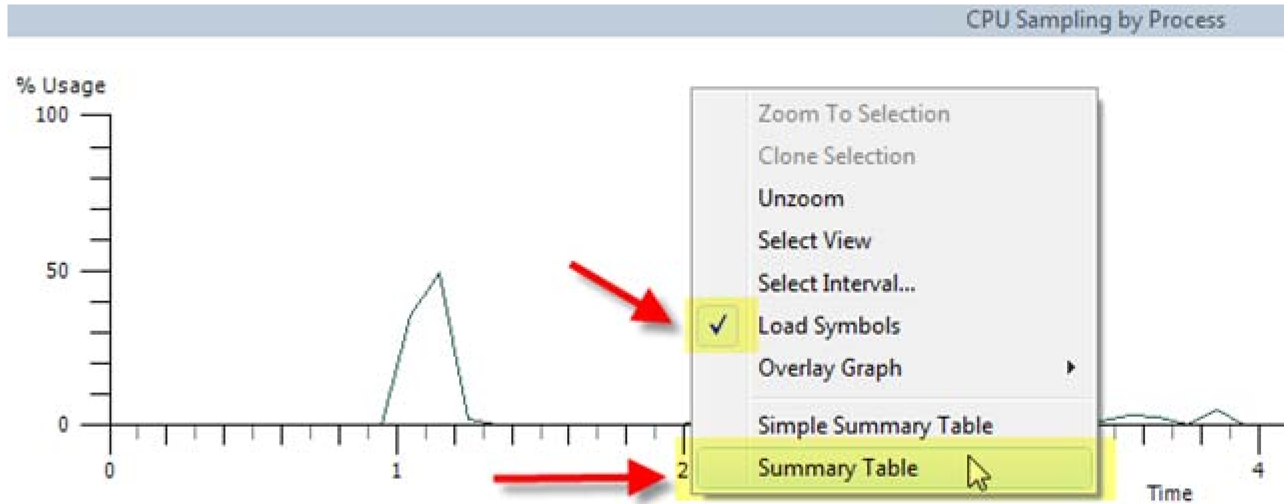
This graph illustrates an important concept about the kernel event provider and the xperf tools in general - they are specifically designed to analyze system wide and application performance data and events. For example, in the legend above, there are many processes listed, but only a few of them actually used any CPU time during the experiment.

Using the legend, you can eliminate all processes except the interesting one. Here is what the graph of CPU utilization for only FS.EXE looks like

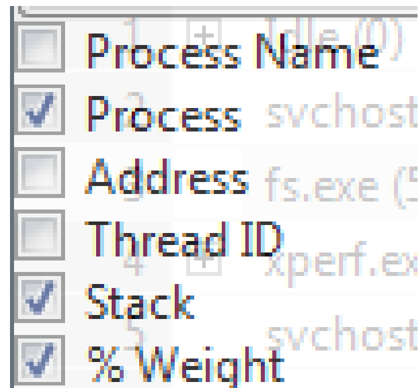


This is pretty cool as it provides a nice overview of the CPU utilization of FS.EXE, but it really doesn't tell us much about where time is being spent in the process it self.

The real power in Performance Analyzer is in its summary tables. These are tabular displays of data about a specific chart, or a region in a chart. For this experiment, I looked at the sample profile data for the entire trace. To do this, right mouse click on the CPU Sampling by Process chart, make sure that the load symbols option is set, then select the Summary table view.



Note, it will take 10 to 20 seconds for the summary table to show up. Performance Analyzer is loading symbols while this is happening. (putting symbol loading on a background thread is on our to do list...).



After the summary table pops up, click on the flyout and select the columns for display. In this case you will want the process, stack and % Weight columns (feel free to experiment with other columns).

Next, arrange the columns as follows. The columns to the left of the gold column are grouping columns. You can change the order of columns and put them to the left or right of the gold column by dragging.

profile.etl - [0 s - 7.3852695 s] - 7.3852695 s - Windows Performance Analyzer

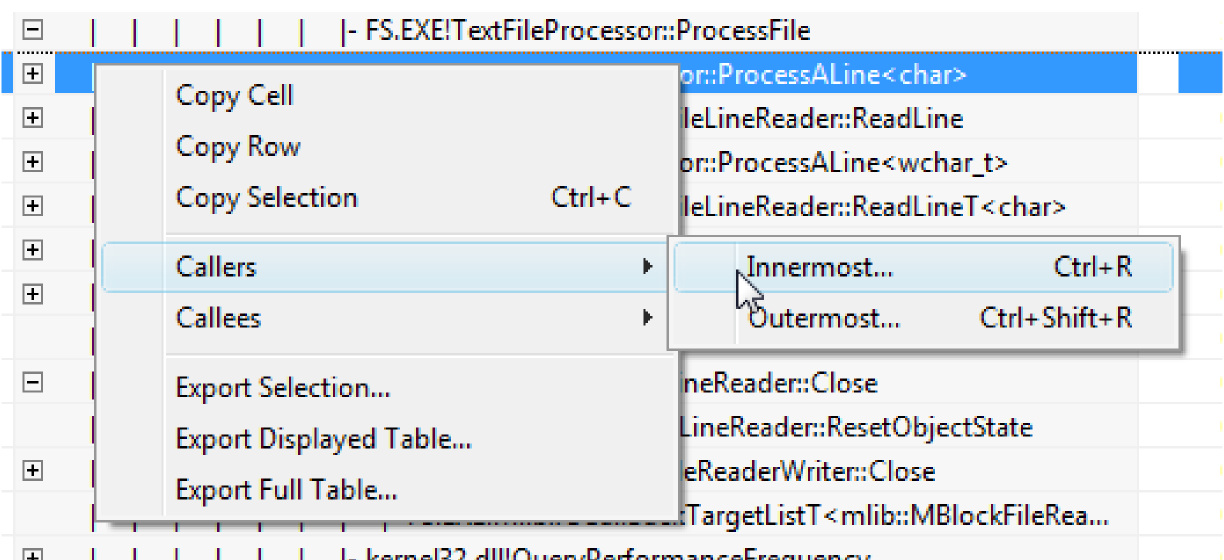
Trace Window Help

Line	Process	Stack	% Weight
1	⊞ Idle (0)		83.99

Now, you can expand the stacks for FS.EXE and see where it is spending its time. Not that this isn't by function as in some profilers but by call stack. This is much more powerful than simply knowing the functions where time is spent as it also shows you how the time consuming functions were called. Its no surprise that my find string utility spends most of its time in the following stack:

<table>
<tr><th>Line</th><th>Process</th><th>Stack</th><th>% Weight</th></tr>
<tr><td>1</td><td>Idle (0)</td><td></td><td>83.99</td></tr>
<tr><td>2</td><td>svchost.exe (860)</td><td>[Root]</td><td>3.36</td></tr>
<tr><td>3</td><td>fs.exe (5236)</td><td>[Root]</td><td>2.92</td></tr>
<tr><td>4</td><td></td><td>|- ntdll.dll!_RtlUserThreadStart</td><td>2.59</td></tr>
<tr><td>5</td><td></td><td>| ntdll.dll!__RtlUserThreadStart</td><td>2.59</td></tr>
<tr><td>6</td><td></td><td>| kernel32.dll!BaseThreadInitThunk</td><td>2.59</td></tr>
<tr><td>7</td><td></td><td>| FS.EXE!__tmainCRTStartup</td><td>2.59</td></tr>
<tr><td>8</td><td></td><td>| |- FS.EXE!wmain</td><td>2.59</td></tr>
<tr><td>9</td><td></td><td>| | |- FS.EXE!Processor::ProcessCommandLineTokens</td><td>2.56</td></tr>
<tr><td>10</td><td></td><td>| | | FS.EXE!Processor::DoScanDirectoriesForFiles</td><td>2.56</td></tr>
<tr><td>11</td><td></td><td>| | | FS.EXE!ScanForFiles</td><td>2.56</td></tr>
<tr><td>12</td><td></td><td>| | | FS.EXE!mlib::TraverseDirT<wchar_t>::traverse_dir</td><td>2.56</td></tr>
<tr><td>13</td><td></td><td>| | | |- FS.EXE!DirScanner<wchar_t>::end</td><td>2.02</td></tr>
<tr><td>14</td><td></td><td>| | | | FS.EXE!Processor::FileProcessorCallBack</td><td>2.02</td></tr>
<tr><td>15</td><td></td><td>| | | | |- FS.EXE!Processor::ProcessFiles_OutOfOrder</td><td>2.00</td></tr>
<tr><td>16</td><td></td><td>| | | | | |- FS.EXE!Processor::ProcessFile</td><td>1.39</td></tr>
<tr><td>17</td><td></td><td>| | | | | | |- FS.EXE!TextFileProcessor::ProcessFile</td><td>1.25</td></tr>
<tr><td>18</td><td></td><td>| | | | | | | |- FS.EXE!TextFileProcessor::ProcessALine<char></td><td>0.83</td></tr>
<tr><td>19</td><td></td><td>| | | | | | | | |- FS.EXE!MBoyerMoore<char>::Find</td><td>0.70</td></tr>
<tr><td>20</td><td></td><td>| | | | | | | | | |- FS.EXE!MBoyerMoore<char>::Find<itself></td><td>0.54</td></tr>
<tr><td>21</td><td></td><td>| | | | | | | | | |- FS.EXE!MStrFinder<char>::PfToUpperI</td><td>0.12</td></tr>
<tr><td>22</td><td></td><td>| | | | | | | | | |- FS.EXE!SFTraits<char>::CompareC</td><td>0.04</td></tr>
</table>

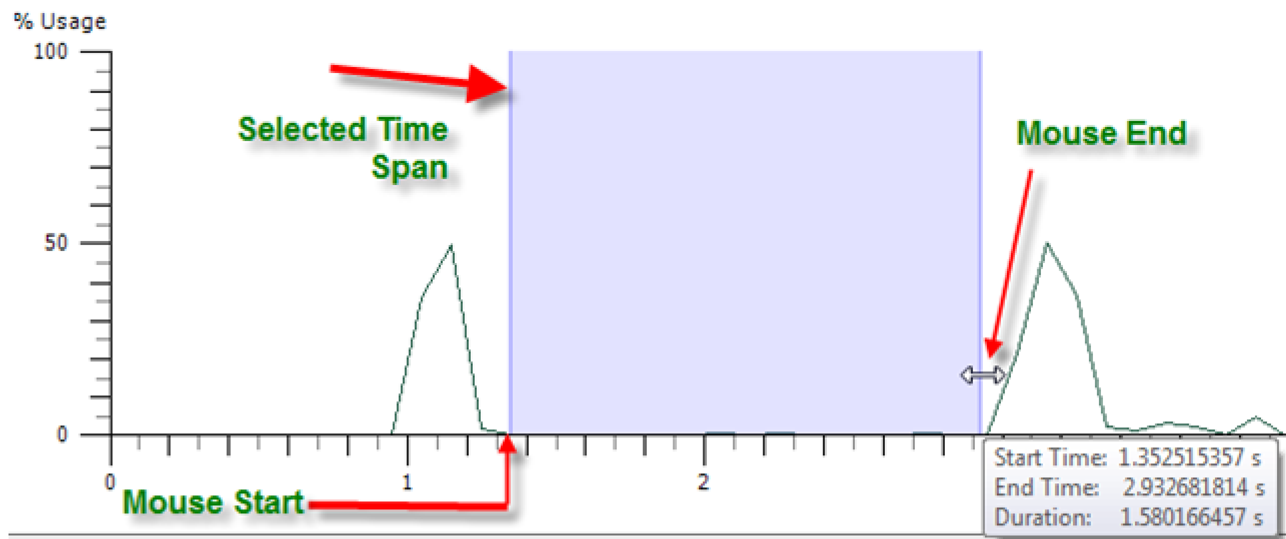
As with other sample profilers, you can look "up" and "down" the stacks from any particular point. This is commonly called a butterfly view. Right mouse click on any item in the stack column and experiment with the callers/callees and inntermost/outermost options, like this:



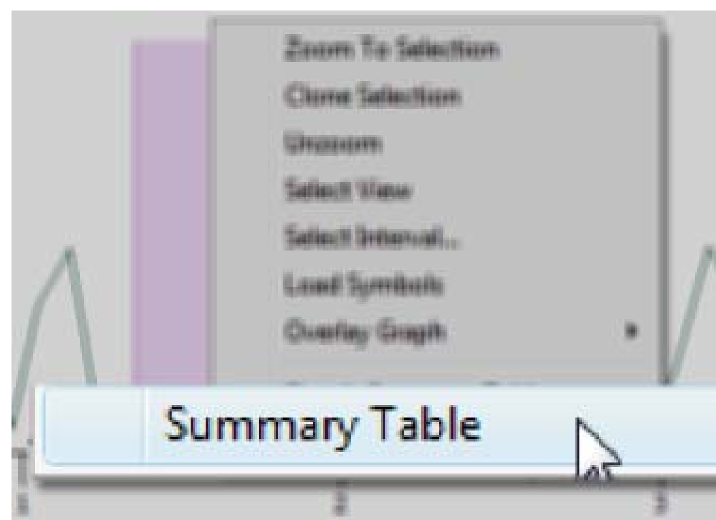
This stack trace has very simple call stacks so it isn't very useful for looking at butterfly views. But try one of your own programs and look at a butterfly stack view of a function that is called often from multiple places. Or, use the butterfly view too look at a intermediate function and see all the functions it calls, and their stacks.

The above screen shots and summary table views contain the data from the entire trace. This works ok for short traces. But for longer traces, or even short traces with a lot of detail, we often need to look at specific time spans.

For example, there are some time spans in my experiment where FS isn't using very little CPU time. I'd like to see what FS is up to in that time span. This is easily done by using the left mouse button to select a time span on the X axis and zooming the graph to that view, or look at the summary table for that span, as in this example.



Once the interesting region is selected, I simply use the right mouse button to pop up the context menu and select summary table.



Note that you can open up multiple summary tables, each from different regions of a graph, or even different graphs. This is great for making comparisons.

The new summary table window now only shows the data for selected time span in the trace. Is not surprising that FS is spending the little CPU time it is using in user mode asynchronous procedure calls.

```
[-] |- ntdll.dll!KiUserApcDispatcher                                                          0.19
[-] |   |- kernel32.dll!BaseploCompletionSimple                                               0.16
    |   |   FS.EXE!mlib::MBlockFileReaderWriter::ReadCompletionRoutineStub                    0.16
    |   |   FS.EXE!mlib::MBlockFileReaderWriter::ReadCompletionRoutine                        0.16
    |   |   FS.EXE!mlib::CCallBackTargetListT<mlib::MFileLineReader>::ExecuteCallback         0.16
    |   |   FS.EXE!mlib::MAsyncFileLineReader::MBFRWCallBack                                  0.16
    |   |   FS.EXE!mlib::CCallBackTargetListT<mlib::MFileLineReader>::ExecuteCallback         0.16
    |   |   FS.EXE!Processor::MBFRWCallBack                                                   0.16
    |   |   FS.EXE!Processor::MBFRWCallBack_OutOfOrder                                        0.16
[-] |   |   |- FS.EXE!Processor::MoveFromPrefetchedToReadyQueue                               0.13
[+] |   |   |   |- FS.EXE!mlib::MCQue<mlib::MBlockFileReaderWriter::BufferDescriptor *>::operator[]   0.06
[+] |   |   |   |- FS.EXE!Processor::MoveFromPrefetchedToReadyQueue<itself>                   0.06
[+] |   |   |- FS.EXE!mlib::fundamental_string<wchar_t,mlib::m_traits<wchar_t>,std::allocator<wc...   0.03
[+] |   |- ntkrpamp.exe!KiServiceExit                                                         0.03
[-] |- kernel32.dll!WaitForSingleObjectEx                                                     0.10
[+] |   |- ntdll.dll!ZwWaitForSingleObject                                                    0.06
    |   |- kernel32.dll!WaitForSingleObjectEx<itself>                                         0.03
```

This post illustrates some key concepts:

- The xperf tools are designed for both system wide and application specific analysis.
- Profiling with ETW is very, very light weight. While the experiment is running, the xperf tools are not even loaded - the kernel itself is collecting the data. All analysis is done as post processing tasks.
- OS based sample profiling collects both user and kernel mode stacks.
- So long as you have symbols, production code can be profiled - no special debug or instrumented builds are required.
- In this example, I started and stopped FS.exe (the experiment) between the tracing start and stop. But, since this is ETW based, sample profiling can be started and stopped at any time, without stopping or restating even a single process. You can profile anything at any time on any system.
- Stack views provide a very powerful method for analyzing where time is spent in a process.
- The general technique with Performance Analyzer is to use the graphics to identify interesting time spans in the trace, then use the summary tables to look at the data in detail.

So Just What Is In A Trace? Using The Xperf Trace Dumper

There is a lot of information in a typical kernel trace. While the Performance Analyzer tool is quite powerful and makes it easy to view a trace graphically, sometimes you just need to see what is in the trace directly. Xperf makes this easy.

First, its important to understand that a trace file (.ETL) is simply just the buffers produced by trace session written to a file. The data in an ETL file isn't pre-processed, summarized, or otherwise annotated with meta data as it comes out of the OS. Its is just the raw data that comes from a ETW session. This is because ETW is designed for log time efficiency - ETW does the absolutely minimal amount of work needed to get the trace data to a file, or other consumer.

This means that all the heavy lifting of post processing trace data happens later. With the xperf tools, there are two places where this occurs:

In the merge step, xperf takes the kernel trace and trace files and merges them into a single trace file. Xperf will merges (adds) meta data to the trace (I've got another post that provides all the detailed on merging in the works...). The result of merging is a single trace file that can be analyzed by the tools directly on the target machine, or copied to another system for analysis. Note that the merge step must happen on the system where the trace was taken (the target system).

When a trace is processed xperf using actions, or loaded into Performance Analyzer, the core trace processing components do a lot of work on the raw trace data. This includes things like mapping process IDs (PIDs) to file and process names, mapping addresses to filenames, loading symbols for address, unifying stacks, and handling 64-bit and 32-bit differences.

As you've seen in the other posts, once a trace is merged it can be viewed in the Performance Analyzer. But, xperf also allows you to see what is in the trace using the dumper action. This is easy to do:

```
xperf -i fs.etl -a dmper >fs.csv
```

The -i fs.etl specified that the input file is FS.ETL. The -a dumper parameter tells xperf to execute the dumper action. The output goes to the standard output.

There is a short cut for this as well: the dumper action is the default action so if you only specify an input file then xperf simply dumps it. For example, the following command does the same thing as the one above:

```
xperf -i fs.etl  >fs.csv
```

The resulting file is an ANSI text file where each line is one record. Each record consists of a comma delimited set of fields. The first field of each line is the name (or type) of the record.

There are some special lines and sections at the front of the file. Each record type is described by a header line. The header lines are delimited by the 'BeginHeader' and 'EndHeader' lines. Note that the line immediately after the 'EndHeader' line is unique, it doesn't have a header line. This line describes some of the characteristics of the trace such as its duration, and the pointer size.

The first field of each header line is the name (or type) of the ETW record that the header line describes. The rest of the fields are the names of each of the fields for the record type. Here is an example of the process start event header (P-Start) and a P-Start event.

```
P-Start,  TimeStamp, Process Name
( PID),  ParentPID,  SessionID,  UniqueKey, UserSid, Command Line
```

```
P-Start, 1017280, fs.exe (3004), 3608, 1, 0x86661508,
S-1-5-21-626881126-397955417-188441333-3225678,
c:\coding\fs\Release\fs\fs.exe  blflargorg c:\coding\*.cpp *.h -s
```

This event describes the start of a process. There is also a corresponding P-End event. For processes that are already running when he trace is begun, the kernel logger includes a pseudo P-Start event.

This means that every PID seen in other events will have a corresponding P-Start event in the trace before it is seen in an event.

Also note that xperf will dump events that you add to your own applications so long as you include an event manifest in your app. So, you can add your own events and use xperf to dump them in the context of all the other events you include in the trace.