Intro to ETW tracing and OSVR

Ryan A. Pavlik, Ph.D.
Senior Software Engineer, Sensics, Inc.
August-September 2015

What is Event Tracing for Windows?

aka ETW, `xperf`, Windows Performance Toolkit (WPT), Windows Performance Analyzer (WPA), …

"Event Tracing for Windows (ETW) is an efficient kernel-level tracing facility that lets you log kernel or application-defined events to a log file. You can consume the events in real time or from a log file and use them to debug an application or to determine where performance issues are occurring in the application.

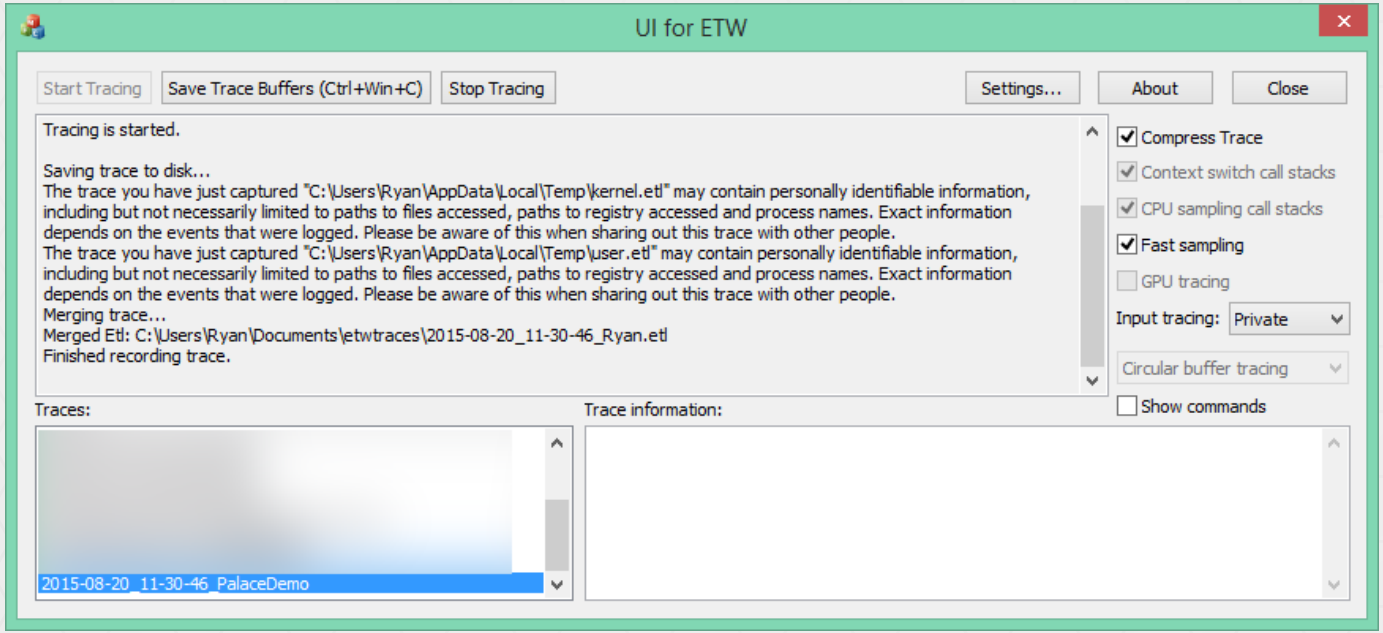
ETW lets you enable or disable event tracing dynamically, allowing you to perform detailed tracing in a production environment without requiring computer or application restarts."

Source: MSDN, "Windows Events: About Event Tracing"

What about the rest of those? `xperf` is the command-line tool to control ETW, so often used synonymously with ETW. Windows Performance Toolkit is the name given by MS to the installable package that provides user-level tools for ETW. WPT includes Windows Performance Analyzer, the GUI we'll use to display and explore tracing data, as well as other tools like WPRUI (Win. Perf. Recorder UI, a limited GUI for controlling tracing that we won't use).

What is Event Tracing for Windows really?

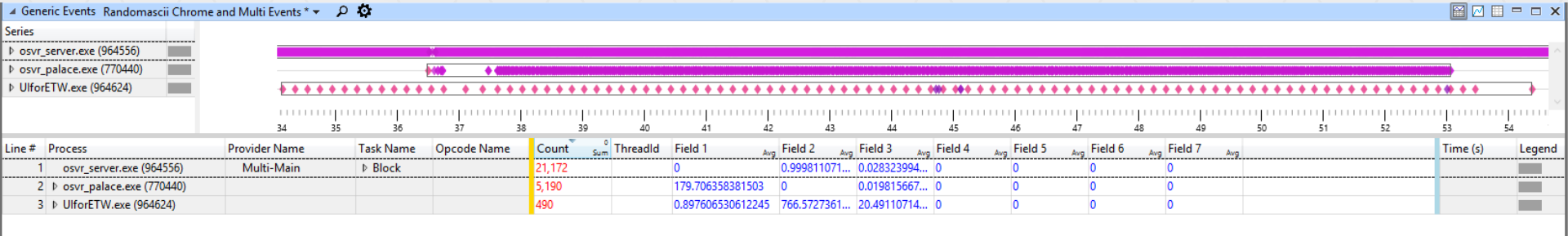
- Effectively, a collection of tools and technologies to get very in-depth performance details across an entire running environment at minimal performance impact.
- With UIforETW, an open-source tool that (among other things) automates the complex command lines to start, stop, and manage traces, it can be a great investigative tool (including high rate sampling and wait-analysis profiling) on its own: I've used it like this for some time now.
- OSVR on Windows now augments ETW traces made by UIforETW with custom events on both the client and server side.
- The following screenshots barely scratch the surface of what you can do with ETW, especially now with OSVR emitting custom events.



Settings I used for this trace: I chose to turn on fast sampling, but not GPU tracing, and traced to an in-memory circular buffer that gets saved on a keystroke. This can let you leave tracing running continuously, then after you notice unusual behavior or something you'd like to investigate, you can hit the shortcut key and save a trace of the last *x* seconds, for after-the-fact performance investigation.

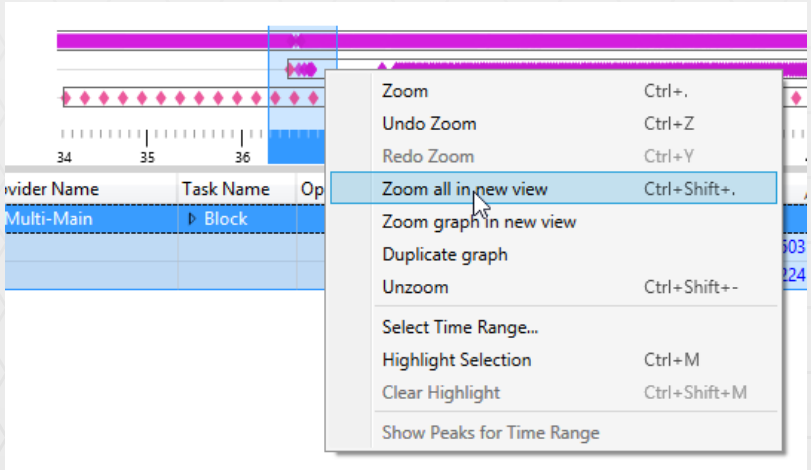
Trace opened in WPA

- Double-clicked on trace name in UIforETW. Screenshot shows only the top pane shown here: these are the “custom events” emitted by UIforETW and OSVR. Other panes show lots more data.
- UIforETW “marks” processor speed, temp, etc. at regular intervals (is thermal throttling affecting your performance?) and also marks keyboard and mouse movements (customizable - was a key press or mouse click a trigger for an unusual behavior?).

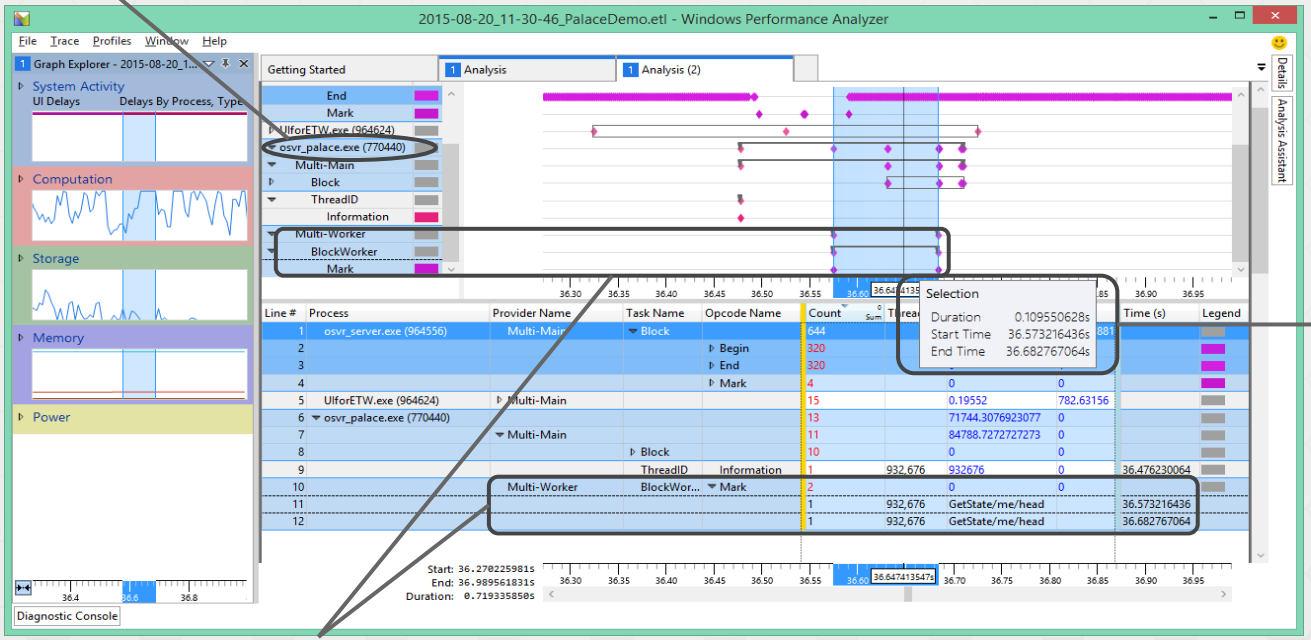


Focusing in: App startup

Select an area of interest, and “Zoom all” - usually in new view. Hold shift to snap selection ends to marks. Here, we’re looking at app startup.



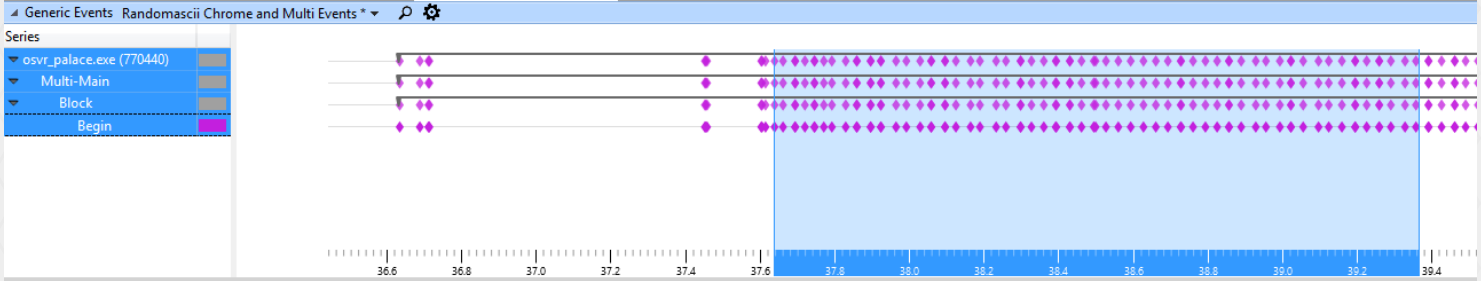
The app process is here: note the triangle is "expanded"



We created a new selection snapped at either end to those two event markers. The selection tooltip shows the duration.

These two sections correspond and show the first two "GetState" calls by the app.

Here, by expanding the providers in the top pane and zooming in, we find that the first two calls to OSVR "get state" for the head are 0.1s apart, so presumably OSVR is starting up early in the app launch process while the app is still loading. If you wanted to know what your app (or others!) was doing during that time, you could "zoom" again and use the other panes of WPA to investigate your code.

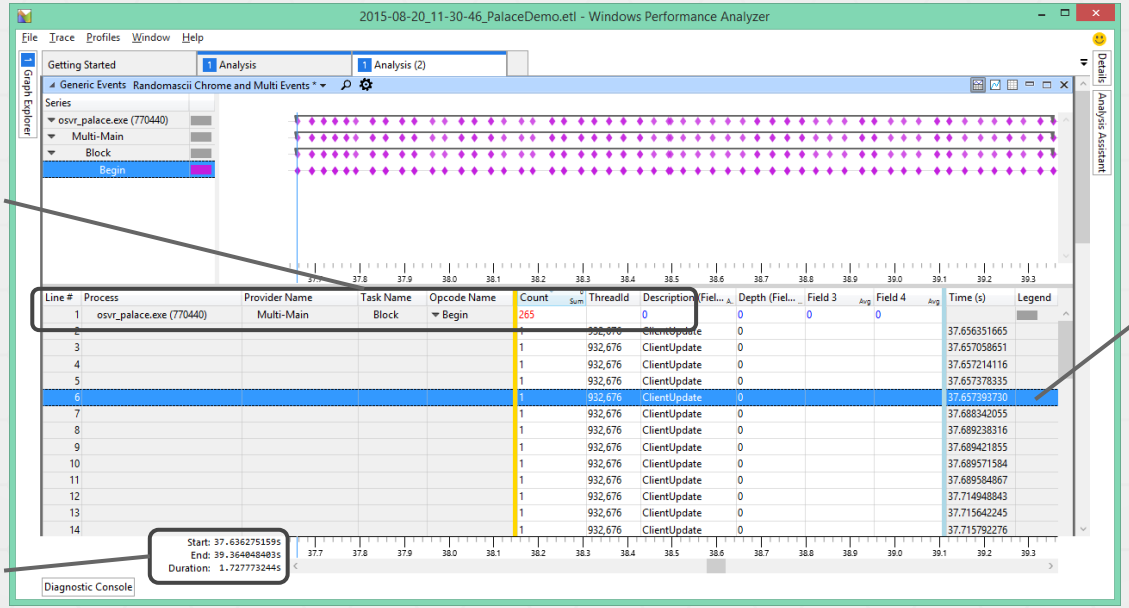


Line #	Process	Provider Name	Task Name	Opcode Name	Count Sum	ThreadId	Description (Fiel...	Depth (Fiel...	Field 3 Avg	Field 4 Avg	Field 5 Avg	Field 6 Avg	Field 7 Avg
1	osvr_palace.exe (770440)	Multi-Main	Block	Begin	466		0	0	0	0	0	0	0
2					1	932,676	ClientUpdate	0					
3					1	932,676	ClientUpdate	0					
4					1	932,676	ClientUpdate	0					
5					1	932,676	ClientUpdate	0					
6					1	932,676	ClientUpdate	0					
7					1	932,676	ClientUpdate	0					
8					1	932,676	ClientUpdate	0					
9					1	932,676	ClientUpdate	0					
10					1	932,676	ClientUpdate	0					
11					1	932,676	ClientUpdate	0					
12					1	932,676	ClientUpdate	0					
13					1	932,676	ClientUpdate	0					
14					1	932,676	ClientUpdate	0					
15					1	932,676	ClientUpdate	0					
16					1	932,676	ClientUpdate	0					
17					1	932,676	ClientUpdate	0					
18					1	932,676	ClientUpdate	0					
19					1	932,676	ClientUpdate	0					
20					1	932,676	ClientUpdate	0					
21					1	932,676	ClientUpdate	0					

We can zoom out in a number of ways: Ctrl-scroll, context (right-click) menu, etc. Right clicking on events lets you filter them. Here, we've zoomed out, filtered to just the "ClientUpdate" events, and selected some "steady-state" running after startup finishes, to zoom in again.

All tables shown in WPA show only information corresponding to the current “zoomed region”. Here, you can see in the summary row that 265 ClientUpdates took place in this view (it looks like fewer since many were very close together)

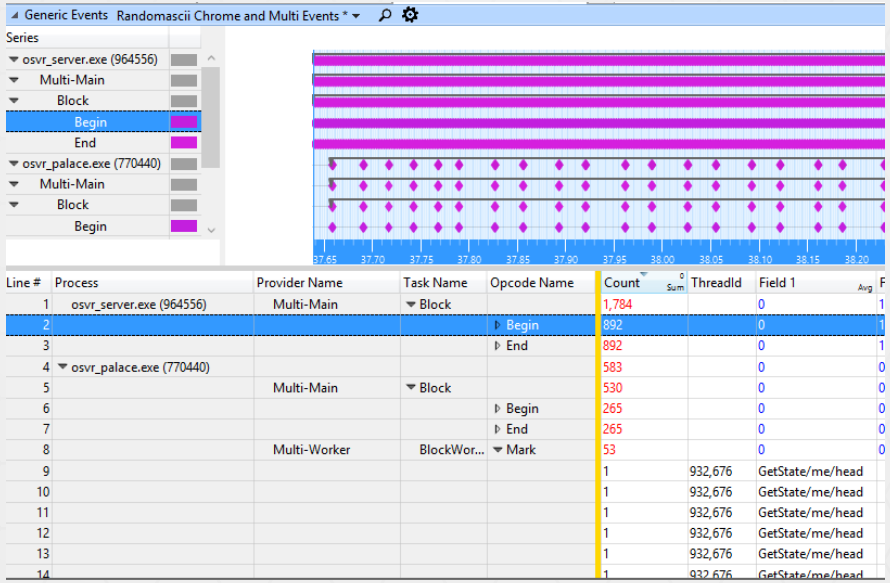
This area shows information about the current zoomed region. The duration we’re viewing is 1.72777 seconds.



All the ClientUpdate events in the period are in this table along with their timestamp relative to the beginning of the trace. (These tables can be exported to spreadsheets if you want to do more in-depth number crunching than the WPA GUI provides)

Here, we’ve zoomed in to a “steady state” portion of the trace recording, and also filtered the custom events so that only the “Begin” marks of a ClientKit update are shown. Some basic analysis reveals the app was making just over 150 calls to ClientUpdate per second on average: slower than the tracker update rate, but higher than the framerate.

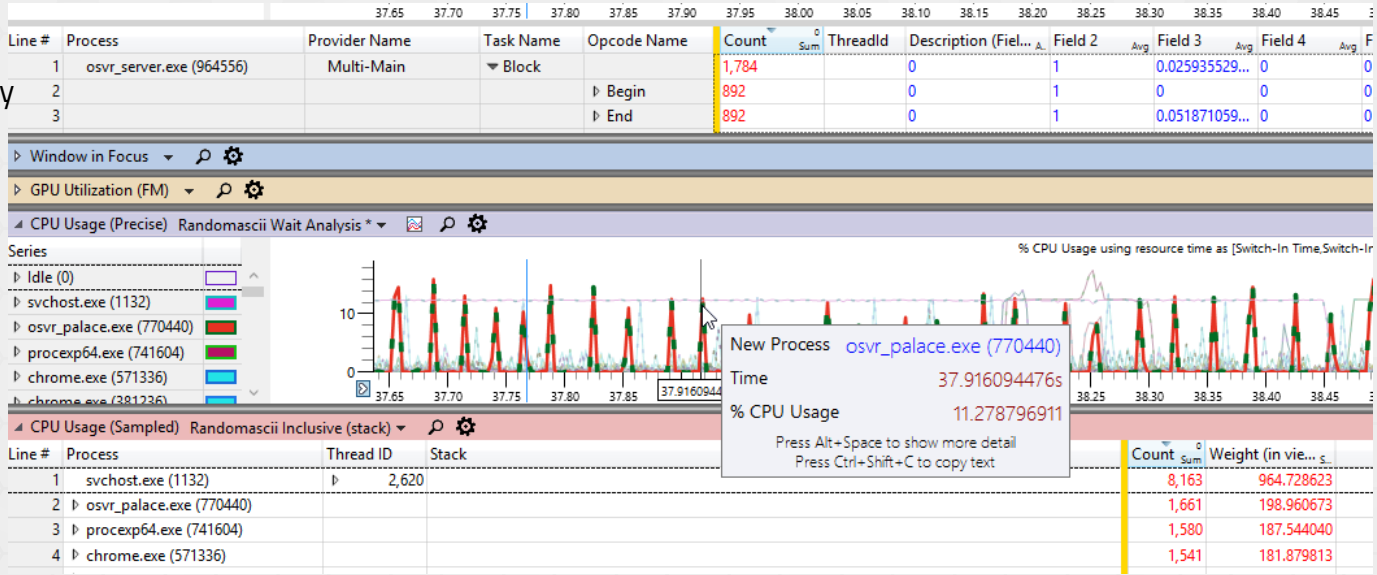
Removing the filter: in that same 1.7s time period, the server update ran 892 times, and the application got the state of /me/head 53 times. (The fact that 53 < 265 (update calls) means the app doesn't care about head pose as often as it is updating OSVR. Since this is a Unity app making those accesses from managed (.NET) code, that means it was probably a good choice to use the state interface here, since the alternative would have been >265 native-to-managed callbacks.)



Line #	Process	Provider Name	Task Name	Opcode Name	Count (Sum)	ThreadId	Field 1 (Avg)	F
1	osvr_server.exe (964556)	Multi-Main	Block		1,784		0	1
2				Begin	892		0	1
3				End	892		0	1
4	osvr_palace.exe (770440)				583		0	0
5		Multi-Main	Block		530		0	0
6				Begin	265		0	0
7				End	265		0	0
8		Multi-Worker	BlockWor...	Mark	53		0	0
9					1	932,676	GetState/me/head	
10					1	932,676	GetState/me/head	
11					1	932,676	GetState/me/head	
12					1	932,676	GetState/me/head	
13					1	932,676	GetState/me/head	
14					1	932,676	GetState/me/head	

Comparing multiple events, within and between processes

The highlighted graph line here is the app CPU usage, which shows a fairly regular spike to approximately one logical CPU on this machine.



Same time region, different panes of ETW functionality - correlated with events

Links and References

UIforETW release v1.13 and WPT 10 were used for these screenshots. More info:

- Intro blog post (Bruce Dawson): https://randomascii.wordpress.com/2015/04/14/uiforetw-windows-performance-made-easier/
- GitHub repo (click Releases for downloads): https://github.com/google/UIforETW
- Be sure to "Copy startup profiles" and "Copy symbol DLLs" from the Settings dialog when you first launch UIforETW or after updating it.
- UIforETW was initially written by Bruce Dawson, formerly of Microsoft and Valve, currently at Google working on Chrome performance (hence why there are some extra features in UIforETW that are Chrome-specific - it originated as an internal tool)
    - He's posted a lot on his blog (see above) about ETW/xperf in the past few years (filter posts by the "xperf" tag), much of it is still applicable and very useful reading (although UIforETW takes away much of the pain described in the earliest posts)
    - His role in the creation of UIforETW is also why "Randomascii" (one of his handles - see blog URL) appeared in these screenshots: he provides useful custom views for WPA in the "startup profiles" mentioned above.
- GPUView (linked from UIforETW when GPU tracing enabled) intro page: http://graphics.stanford.edu/~mdfisher/GPUView.html

Note

To avoid increasing dependencies and potential performance impacts, tracing is disabled by default in the provided binaries as of September 2015. However, corresponding tracing-enabled DLLs are now shipped in the “tracing” folder in binary snapshots.

You can directly replace the DLLs with ones built with tracing turned on to add tracing as needed. See the readme in that folder for details.

If you build from source, I’d suggest turning on tracing (as I do) - never know when you might need it.

For additional information:

- OSVR developer portal
  - http://osvr.github.io

- Sensics – Founding contributor to OSVR, experts working in VR/AR for over a decade
  - http://www.sensics.com