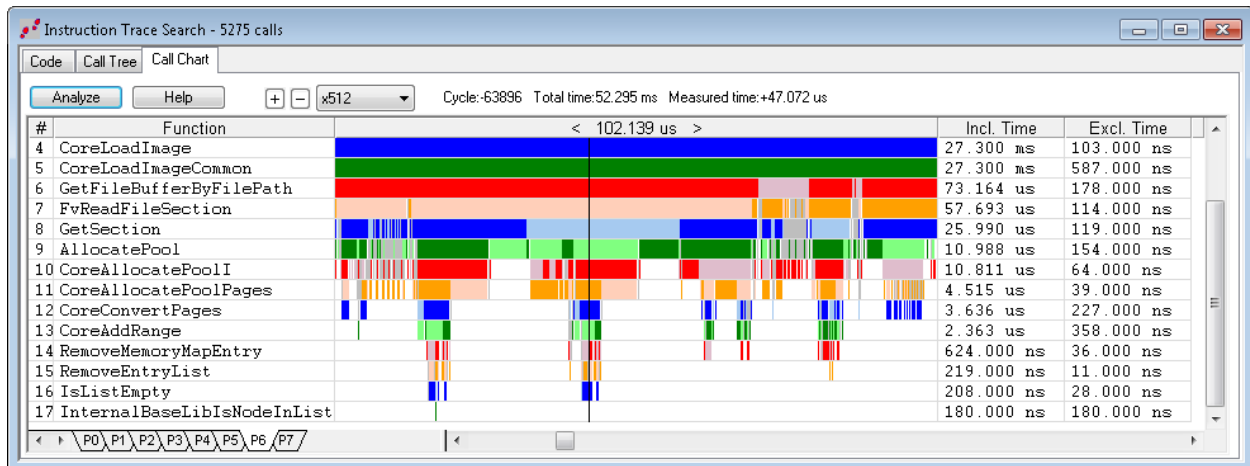


# INTEL<sup>®</sup> ADDS HIGH SPEED INSTRUCTION TRACE



**EBOOK**

**BY LARRY TRAYLOR**

*By Larry Traylor, Vice President Software Debug and Trace*



Larry Traylor co-founded Arium Corporation in 1977. Larry served as president, CEO, and chairman of the board of Arium. He was instrumental in driving that company's vision for product creation of hardware-based program debug and code trace tools. In 2013, Larry joined ASSET InterTech when Arium was acquired by ASSET. He has a BSEE from Cal-Poly Pomona.

## Table of Contents

|  |    |
|--|----|
| Overview:.....   | 4  |
| What is Intel® Doing? .....                                      | 4  |
| How Intel Processor Trace Compresses the Information .....       | 5  |
| Trace Features Used by ASSET InterTech’s SourcePoint Tools ..... | 5  |
| The Older Intel Trace Methods:.....                              | 6  |
| Use Models and Advantages for High Speed Trace .....             | 6  |
| How Trace can be Displayed in Modern Tools like SourcePoint..... | 7  |
| Call Graph Display .....   | 8  |
| Using the Statistics View to Tune Execution Times .....          | 10 |
| Other Features of SourcePoint that Make Use of Trace .....       | 11 |
| There is Even More on the Way from Intel!.....                   | 12 |
| Conclusions.....   | 12 |
| Learn More about SourcePoint™ for Intel® .....                   | 12 |

## Table of Figures

|  |    |
|--|----|
| Figure 1: A SourcePoint List Display.....            | 8  |
| Figure 2: Sample Instruction Trace of UEFI Code..... | 9  |
| Figure 3: A SourcePoint Call Chart Display.....      | 10 |
| Figure 4: SourcePoint’s Statistics View .....        | 11 |

© 2014 ASSET InterTech, Inc.

ASSET and ScanWorks are registered trademarks while the ScanWorks logo and SourcePoint are trademarks of ASSET InterTech, Inc. All other trade and service marks are the properties of their respective owners.

## Overview:

Instruction-Trace-Producing hardware is now included in newer Intel® processors and SOCs. Until recently the only instruction trace in Intel processors and SOCs had been provided by either last-branch-record registers (LBR's) or branch-trace-messages (BTM's). Neither of these features provided powerful enough trace to be of much use and certainly were much weaker than the popular competitive brands. Without good instruction-trace capabilities, some kinds of problems are very difficult to diagnose, as shown later in this paper. Intel's new trace features offer some major new capabilities for software defect diagnosis ("code debugging"). These features, when combined with full-featured debuggers like ASSET's SourcePoint™, will significantly reduce development time.

## What is Intel® Doing?

Intel has included one of two forms of this new trace in several recent SOCs. Because it was still being defined and improved, Intel chose not to enable most users with this capability on early silicon. Several new processors, currently being sampled to OEMs, have these features included and enabled in all versions of the silicon. In these processors the trace feature is available architecturally, which means all parts have it enabled. This feature, referred to as "Intel Processor Trace (PT)", is described in the following Manual:

[Intel® 64 and IA-32 Architectures Software Developer's Manual](#)  
[Volume 3 \(3A, 3B & 3C\): System Programming Guide, Chapter 36](#)  
[Order Number: 253668-051US June 2014](#)

In earlier SOCs the feature was called "real-time instruction trace" (RTIT). In some current SOCs RTIT is available. In later SOCs and other Intel processors the feature is called "Intel Processor Trace".

Information about the presence or version of a trace feature in a specific processor or SOC is "Intel confidential", so this article will only talk about the features in general. For information about a specific part, contact your ASSET InterTech representative and ensure you have the appropriate Intel NDA for the discussion.

While there are many technical differences between the two versions (RTIT and Intel PT) and also among some of the instantiations, there is plenty of common ground for discussion. For purposes of this paper, the author will call these trace features collectively “High Speed Processor Trace”. This is to differentiate it from older methods, such as Branch Trace Message/Branch Trace Store (BTM/BTS), which slowed the execution down substantially when enabled.

## **How Intel Processor Trace Compresses the Information**

Intel Processor Trace, like many trace algorithms today, limits the data it carries to time-stamped instruction-flow information. In the most compressed form, a portion of a trace stream of bytes will simply represent taken/not-taken branches in the execution stream. In this mode, each byte represents up to 6 branches, and this will usually represent 30 or more executed instructions. Along with these taken/not-taken packets there are several other packet types that include new address packets (when needed), time-stamp information, and other auxiliary packet types. Some of these packets are periodic while others are as-needed.

This format is more completely described in the referenced Intel manual. It provides a very compressed trace stream for collecting and post processing by tools like SourcePoint.

## **Trace Features Used by ASSET InterTech’s SourcePoint Tools**

One of the most important features of “High Speed Processor Trace” in Intel’s newer ICs is that it is nearly full speed. It has no significant impact on the execution speed of the program being executed. In contrast, when using BTMs with BTS (storage to memory) there was a minimum of a 60% slow down. For some code this could be much greater. This change in execution speed could often impact whether a bug does or does not occur. High Speed Processor Trace (HSPT) has no measurable impact on the experiment.

In addition, HSPT has several types of time stamp available in most of its instantiations. Using cycle-accurate timestamp, time can be measured with a resolution of the processor clock. In later instantiations, global timestamp will allow alignment with all threads and all other trace sources, including AET and other new sources.

This, highly-compressed, full-speed trace, when enabled, provides instrumentation of an operating program to allow for examination of the exact sequence of execution of instructions, including asynchronous sequences like exceptions and external interrupts.

These trace features are on par with trace methods found in other architectures and will produce the results that firmware engineers have come to expect. These features can also be used at the application level and for diagnosing system faults.

### **The Older Intel Trace Methods:**

In the ten years prior to Intel's introduction of High Speed Processor Trace, (Intel Processor Trace in the long term), Intel had only two methods of instruction trace. These were BTM to BTS and LBR.

LBR trace was based on a relatively small number of pairs of last-branch-record registers. A typical number was 8 or 16 pair. Each pair of registers would record the "from" and "to" addresses" of the last 8 or so changes of execution flow. This could typically record 40 or 50 instructions. This would rarely capture all of the last interrupt. Due to the small depth of this trace, it often did not contain the fault producing event.

The other method available was BTM to BTS. This was cumbersome to setup and use, and it slowed the execution of the processor greatly. This often altered the problem being diagnosed. Because of the difficulty in use and the speed issue, most programmers did not use this feature.

Neither of these types of trace had any notion of time stamp, so, many post-processing features could not be implemented, such as time based execution call graphs and statistics views.

### **Use Models and Advantages for High Speed Trace**

There are many powerful uses for instruction trace. These include several types of defect root-cause determination, understanding of performance issues, and gaining a quick overview of the execution of a program or process.

The classical and still most compelling use of instruction trace is in finding the interference in a particular program sequence by an asynchronous event. There is nothing more frustrating than finding the place that a program is making the wrong decision, only to discover that you have no

way of telling what altered that data object upon which the errant decision is based. In a simple case, it could be a matter of determining what code sequence called this code; call stacks can often show this. In a difficult case, the data may have been changed by code running in an interrupt that was not supposed to modify the data in question (maybe from an errant pointer?). In this case, even though the software engineer may be able to reliably trigger the debug tool on the exact errant decision, he has no way of knowing what piece of code modified the bad data or why. Using trace to see what code preceded the bad decision will usually yield the culprit in a very short amount of time. This can literally save weeks in diagnosing a blue screen or Linux “oops”.

Another very common use of trace is to actually measure which parts of code are contributing to the execution time of a function. Nothing shows this more clearly than a statistics view of the code operating at full speed. This can often directly show the engineer exactly which pieces of code can be optimized for the maximum improvement. Measuring these times using real-time trace allows making the measurement without altering the experiment.

These are just two examples of how using instruction trace can take weeks out of a development schedule. Many developers of embedded programs have been using trace for years and the number of ways it can be used to diagnose problems is almost limitless. Firmware (UEFI) development on Intel based computers can now take advantage of these silicon/tool features.

## **How Trace can be Displayed in Modern Tools like SourcePoint**

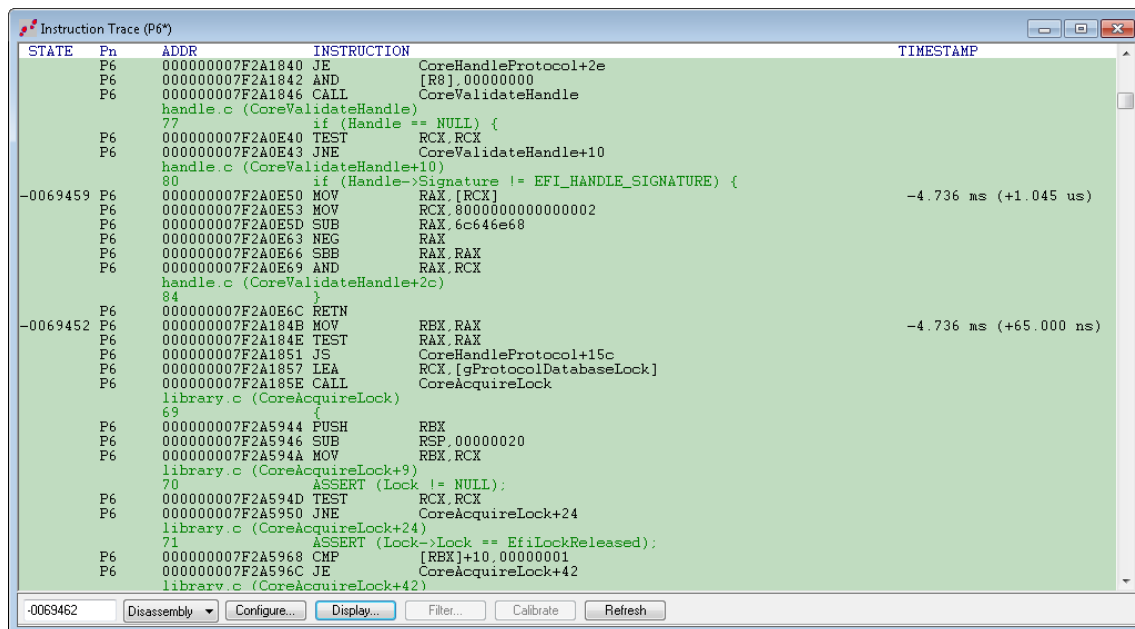
Now that High Speed Processor Trace is available in Intel chips, development engineers can take advantage of the powerful features in tools like SourcePoint. SourcePoint has many different ways it can display data collected in a trace buffer. Conventional displays that are list-based are available with many format options ranging from simple disassembly to full source display. These displays are enhanced by many features such as flyover symbol. In addition to the classical list-based displays, SourcePoint offers several trace post-processing features and displays that make it very easy to visualize code execution at a high level and then drill down to the line by line views. Modern large trace buffers, in the gigabyte range, make examining trace detail without good browsing tools impractical. SourcePoint offers several types of post-processing tools which include:



- Structured Search
- Call Charts
- Call Graphs
- Statistical Summary Displays.

Several of these will be described in detail in later sections.

The most basic trace display is the list display. In SourcePoint, the software engineer can select the items to be displayed and control the color coding to differentiate multiple trace sources. The lines are time-stamped and can be used to index into other displays such as source windows, chart windows, or other trace list displays. The lines can be assembly, source, or mixed. An example of a list display is shown in Figure 1. This display shows both assembly and source-level depiction of the executed code. This list display is very configurable by the user in SourcePoint.



| STATE    | Pn | ADDR             | INSTRUCTION                                      | TIMESTAMP              |
|----------|----|------------------|--|------------------------|
| P6       |    | 000000007F2A1840 | JE CoreHandleProtocol+2e                         |                        |
| P6       |    | 000000007F2A1842 | AND [R8],00000000                                |                        |
| P6       |    | 000000007F2A1846 | CALL CoreValidateHandle                          |                        |
|          |    |                  | handle.c (CoreValidateHandle)                    |                        |
|          |    | 77               | if (Handle != NULL) {                            |                        |
| P6       |    | 000000007F2A0E40 | TEST RCX,RCX                                     |                        |
| P6       |    | 000000007F2A0E43 | JNE CoreValidateHandle+10                        |                        |
|          |    |                  | handle.c (CoreValidateHandle+10)                 |                        |
|          |    | 80               | if (Handle->Signature != EFI_HANDLE_SIGNATURE) { |                        |
| -0069459 | P6 | 000000007F2A0E50 | MOV RAX,[RCX]                                    | -4.736 ms (+1.045 us)  |
| P6       |    | 000000007F2A0E53 | MOV RCX,8000000000000002                         |                        |
| P6       |    | 000000007F2A0E5D | SUB RAX,6c646e68                                 |                        |
| P6       |    | 000000007F2A0E63 | NEG RAX  |                        |
| P6       |    | 000000007F2A0E66 | SBB RAX,RAX                                      |                        |
| P6       |    | 000000007F2A0E69 | AND RAX,RCX                                      |                        |
|          |    |                  | handle.c (CoreValidateHandle+2c)                 |                        |
|          |    | 84               | }  |                        |
| P6       |    | 000000007F2A0E6C | RETN   |                        |
| -0069452 | P6 | 000000007F2A184B | MOV REX,RAX                                      | -4.736 ms (+65.000 ns) |
| P6       |    | 000000007F2A184E | TEST RAX,RAX                                     |                        |
| P6       |    | 000000007F2A1851 | JS CoreHandleProtocol+15c                        |                        |
| P6       |    | 000000007F2A1857 | LEA RCX,[gProtocolDatabaseLock]                  |                        |
| P6       |    | 000000007F2A185E | CALL CoreAcquireLock                             |                        |
|          |    |                  | library.c (CoreAcquireLock)                      |                        |
|          |    | 69               | {  |                        |
| P6       |    | 000000007F2A5944 | PUSH REX   |                        |
| P6       |    | 000000007F2A5946 | SUB RSP,00000020                                 |                        |
| P6       |    | 000000007F2A594A | MOV REX,RCX                                      |                        |
|          |    |                  | library.c (CoreAcquireLock+9)                    |                        |
|          |    | 70               | ASSERT (Lock != NULL);                           |                        |
| P6       |    | 000000007F2A594D | TEST RCX,RCX                                     |                        |
| P6       |    | 000000007F2A5950 | JNE CoreAcquireLock+24                           |                        |
|          |    |                  | library.c (CoreAcquireLock+24)                   |                        |
|          |    | 71               | ASSERT (Lock->Lock == EfiLockReleased);          |                        |
| P6       |    | 000000007F2A5968 | CMP [RBX]+10,00000001                            |                        |
| P6       |    | 000000007F2A596C | JE CoreAcquireLock+42                            |                        |
|          |    |                  | library.c (CoreAcquireLock+42)                   |                        |

Figure 1: A SourcePoint List Display

## Call Graph Display

When a large amount of trace has been captured, and the code base is extensive, looking at the detail of the trace is very tedious. The Call Graph display allows the SourcePoint user to look at large portions (or even all of the trace buffer) and view it in a graph showing call depth. Each



line in this graph can represent a different function at different points in time. Changes in color represent changes in a function. Each line moving downwards represents another level of call depth. A moveable cursor points to specific points on the timeline (x axis of graph). The left-hand column displays the names of the functions, at each level, at the point indicated by the cursor.

The controls above the graph allow the user to expand the graph (zoom in) at the point indicated by the cursor. Figure 2 shows an actual trace of a range of UEFI in the boot-up of an Intel based computer. This is a good illustration of the power of this viewer:

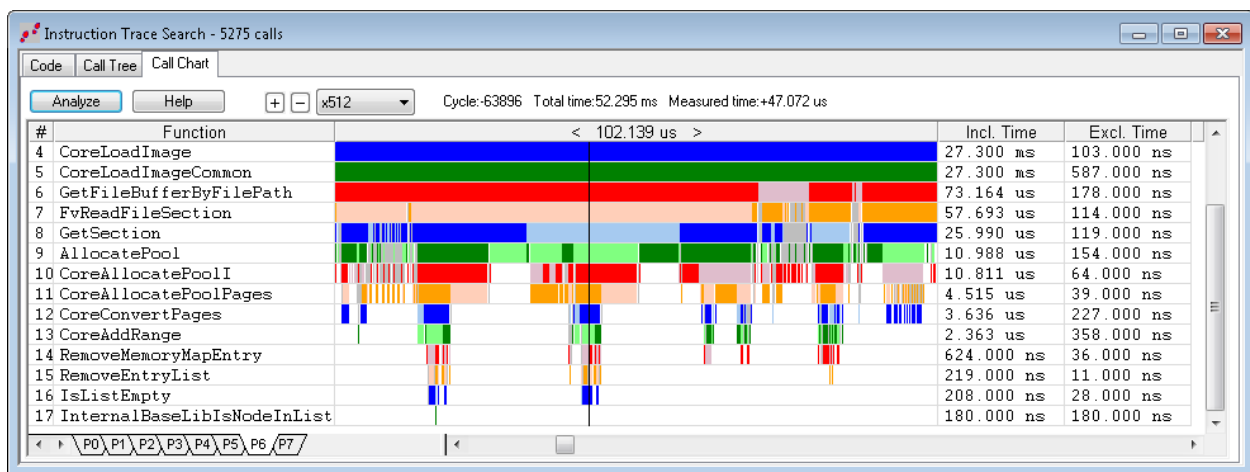


Figure 2: Sample Instruction Trace of UEFI Code

Another way of looking at the same information is with the Call Chart. In this view, specific areas can be drilled into by function name, expanding or collapsing as desired. Both of the call views can be *synchronized* to a list view so that the specific code can be examined at the point of interest. Figure 3 is a Call Chart display.

Instruction Trace Search - 5275 calls

Code Call Tree Call Chart

Analyze Help Expand All Collapse All Upper limit: 0 Lower limit: 99 Show interrupts

| Cycle  | Address  | #  | Function                 | Timestamp | Incl. Time | Excl. Time |
|--------|----------|----|--------------------------|-----------|------------|------------|
| -69462 | 7F299805 | 7  | CoreHandleProtocol       | -4.737 ms | 3.332 us   | 164.000 ns |
| -69462 | 7F2A1846 | 8  | CoreValidateHandle       | -4.737 ms | 1.045 us   | 1.045 us   |
| -69452 | 7F2A185E | 8  | CoreAcquireLock          | -4.736 ms | 165.000 ns | 13.000 ns  |
| -69452 | 7F2A5989 | 9  | CoreRaiseTpl             | -4.736 ms | 152.000 ns | 152.000 ns |
| -69441 | 7F2A1869 | 8  | CoreGetProtocolInterface | -4.736 ms | 286.000 ns | 81.000 ns  |
| -69441 | 7F2A178A | 9  | CoreValidateHandle       | -4.736 ms | 0 ns       | 0 ns       |
| -69436 | 7F2A17D0 | 9  | CompareGuid              | -4.736 ms | 205.000 ns | 205.000 ns |
| -69436 | 7F2A6BD6 | 10 | ReadUnaligned64          | -4.736 ms | 0 ns       | 0 ns       |
| -69430 | 7F2A6BE1 | 10 | ReadUnaligned64          | -4.736 ms | 0 ns       | 0 ns       |
| -69425 | 7F2A6BED | 10 | ReadUnaligned64          | -4.736 ms | 0 ns       | 0 ns       |
| -69420 | 7F2A6BF9 | 10 | ReadUnaligned64          | -4.736 ms | 0 ns       | 0 ns       |
| -69405 | 7F2A190E | 8  | AllocatePool             | -4.736 ms | 1.497 us   | 211.000 ns |
| -69402 | 7F2A7059 | 9  | CoreRaiseTpl             | -4.735 ms | 0 ns       | 0 ns       |
| -69397 | 7F2A7077 | 9  | CoreAllocatePoolI        | -4.735 ms | 1.061 us   | 256.000 ns |
| -69397 | 7F2A01CA | 10 | LookupPoolHead           | -4.735 ms | 0 ns       | 0 ns       |
| -69391 | 7F2A0231 | 10 | IsListEmpty              | -4.735 ms | 358.000 ns | 42.000 ns  |
| -69391 | 7F2A667B | 11 | InternalBaseLibIsNodeI   | -4.735 ms | 316.000 ns | 316.000 ns |
| -69371 | 7F2A0341 | 10 | RemoveEntryList          | -4.735 ms | 180.000 ns | 11.000 ns  |
| -69371 | 7F2A6735 | 11 | IsListEmpty              | -4.735 ms | 169.000 ns | 23.000 ns  |
| -69371 | 7F2A667B | 12 | InternalBaseLibIsNode    | -4.735 ms | 146.000 ns | 146.000 ns |
| -69347 | 7F2A0374 | 10 | DebugClearMemory         | -4.735 ms | 267.000 ns | 267.000 ns |
| -69346 | 7F2A6B23 | 11 | InternalMemSetMem        | -4.735 ms | 0 ns       | 0 ns       |
| -69337 | 7F2A0399 | 10 | DebugPrint               | -4.734 ms | 0 ns       | 0 ns       |
| -69328 | 7F2A7086 | 9  | CoreReleaseLock          | -4.734 ms | 225.000 ns | 225.000 ns |
| -69325 | 7F2A23D6 | 10 | CoreSetInterruptState    | -4.734 ms | 0 ns       | 0 ns       |
| -69311 | 7F2A1945 | 8  | InsertTailList           | -4.734 ms | 109.000 ns | 13.000 ns  |
| -69311 | 7F2A65B7 | 9  | InternalBaseLibIsNodeInL | -4.734 ms | 96.000 ns  | 96.000 ns  |
| -69298 | 7F2A1957 | 8  | CoreReleaseLock          | -4.734 ms | 66.000 ns  | 66.000 ns  |

Figure 3: A SourcePoint Call Chart Display

## Using the Statistics View to Tune Execution Times

Without trace, it can be very difficult to determine the execution time of the various areas in the program. Very often some programmed operation, such as booting up a computer with UEFI, takes longer than desired. It may not be obvious what portions of the code are the real culprits. SourcePoint's statistics view can be used to quickly find out exactly where the time is being spent. Figure 4 shows the statistics view in SourcePoint.

Instruction Trace Statistics

Function Profiling

Analyze Help  Include interrupts Total time: 52.295 ms Calls: 5275

| Function                           | Count | Incl. Time | Incl. % | Excl. Time | Excl. % |
|------------------------------------|-------|------------|---------|------------|---------|
| DebugPrint                         | 67    | 52.070 ms  | 99.57%  | 51.993 ms  | 99.42%  |
| BasePrintLibSPrintMarker           | 13    | 110.267 us | 0.21%   | 50.022 us  | 0.10%   |
| InternalBaseLibIsNodeInList        | 285   | 41.082 us  | 0.08%   | 41.082 us  | 0.08%   |
| DebugClearMemory                   | 42    | 30.911 us  | 0.06%   | 30.911 us  | 0.06%   |
| CompareGuid                        | 534   | 26.335 us  | 0.05%   | 25.982 us  | 0.05%   |
| CopyMem                            | 63    | 26.460 us  | 0.05%   | 24.567 us  | 0.05%   |
| BasePrintLibValueToString          | 61    | 17.728 us  | 0.03%   | 17.210 us  | 0.03%   |
| AsciiStrLen                        | 34    | 8.259 us   | 0.02%   | 8.259 us   | 0.02%   |
| CoreInstallProtocolInterfaceNotify | 4     | 36.887 ms  | 70.54%  | 7.899 us   | 0.02%   |
| CoreFreePoolI                      | 11    | 38.816 us  | 0.07%   | 4.823 us   | 0.01%   |
| CoreFindFreePagesI                 | 6     | 4.312 us   | 0.01%   | 4.312 us   | 0.01%   |
| CoreAddRange                       | 10    | 22.504 us  | 0.04%   | 4.183 us   | 0.01%   |
| CoreReleaseLock                    | 77    | 4.228 us   | 0.01%   | 4.090 us   | 0.01%   |
| CoreRaiseTpl                       | 91    | 4.090 us   | 0.01%   | 4.090 us   | 0.01%   |
| CoreAllocatePoolI                  | 27    | 60.059 us  | 0.11%   | 3.838 us   | 0.01%   |
| CoreConvertPages                   | 10    | 51.150 us  | 0.10%   | 3.596 us   | 0.01%   |
| CoreHandleProtocol                 | 19    | 21.231 us  | 0.04%   | 3.582 us   | 0.01%   |
| CoreReadImageFile                  | 43    | 13.809 us  | 0.03%   | 3.428 us   | 0.01%   |
| IsListEmpty                        | 138   | 25.486 us  | 0.05%   | 3.278 us   | 0.01%   |
| CoreLoadPeImage                    | 1     | 8.754 ms   | 16.74%  | 3.268 us   | 0.01%   |
| CoreFindProtocolEntry              | 18    | 20.290 us  | 0.04%   | 3.048 us   | 0.01%   |
| IsDevicePathEnd                    | 40    | 2.959 us   | 0.01%   | 2.851 us   | 0.01%   |
| CoreLocateHandle                   | 6     | 10.584 us  | 0.02%   | 2.664 us   | 0.01%   |
| CoreFreeMemoryMapStack             | 10    | 13.680 us  | 0.03%   | 2.417 us   | 0.00%   |
| InsertHeadList                     | 85    | 8.963 us   | 0.02%   | 2.057 us   | 0.00%   |
| InternalMemCopyMem                 | 63    | 1.893 us   | 0.00%   | 1.893 us   | 0.00%   |
| CoreAcquireLock                    | 59    | 4.327 us   | 0.01%   | 1.870 us   | 0.00%   |

Navigation: P0 P1 P2 P3 P4 P5 P6 P7

Figure 4: SourcePoint's Statistics View

Note that in all of the trace post-processing displays there is a tab per processor. The list display, which can be time-aligned to all these other displays, can be one display per processor or multiple color-coded trace displays.

The exact time-stamp features, and therefore the exact alignment features available, differ from one Intel processor to another. For exact features for a specific processor please contact your local ASSET representative.

## Other Features of SourcePoint that Make Use of Trace

In addition to the post-processing screens shown, SourcePoint has several other popular trace-processing features. Trace views may be searched in either simple textual algorithms or in structured algorithms that evaluate addresses and data and search for those. Also, all trace views contain flyover examination of data objects. Code windows and symbol windows can be opened, referencing cursor-selected objects in the trace window.

SourcePoint also contains some of the most powerful and quick symbol-finding/evaluating dialogs available in any tool. These symbol-search tools work across program modules making them extremely convenient when used in a UEFI environment.

## **There is Even More on the Way from Intel!**

This document has provided an overview of new processor instruction-trace features in existing and coming Intel processors. In addition to instruction trace, Intel processors still produce Architectural Event Trace. In the very newest processors (not yet released) there are even more types of trace available to the software engineer. ASSET InterTech can describe some of these as long as there is an appropriate three-way NDA in place. Please ask your local ASSET representative.

## **Conclusions**

Newer Intel processors and SOCs offer many trace features not available before in Intel silicon. The most significant is High Speed Processor Trace. While not all of the deployment details are in the public domain yet, there is enough information available to engage in very useful conversations. For several of these processors and SOCs, these features will be available to anyone who has access to the chips. These features, when used with a tool like SourcePoint from ASSET InterTech can greatly reduce UEFI debug time.

## **Learn More about SourcePoint™ for Intel®**

---

- [\*Find your local ASSET InterTech Sales Manager.\*](#)
- [\*SourcePoint for Intel - web page\*](#)
- [\*Software Debug and Trace eBooks\*](#)
- [\*SourcePoint for Intel - Resources:\*](#)

### *Data Sheets*

- SourcePoint for Intel Processors with UEFI Support Data Sheet
-