Intel's trace hub | New Methods for Software Debug Find Root Cause Faster!

eBook

by Larry A 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

Table of Figures

Acronyms	
BTM	Branch Trace Messages
DDR	Double Date Rate
DMI	Direct Media Interface
ME	Management Engine
MIPI	Mobile Industry Processor Interface
MMIO	Memory Mapped IO
QPI	Intel Quick Path Interconnect
SoC	System-on-a-chip
STM	System Trace Messaging
STPV2	System Trace Protocol version 2
UEFI	Unified Extensible Firmware Interface

Executive Summary

The really hard-to-find bugs, encountered toward the end of big software projects, like a UEFI port, cause the larger and harder to manage schedule slips. Trace can really help shorten the time to find these bugs. Instruction trace is too difficult to navigate given today's code base size and complexity. Trace Hub is the answer to guiding a drill down into instruction trace to root cause the bug.

Hard to find bugs are usually caused by asynchronous events. Examining the flow of instructions as they were executed (Instruction Trace) is the only way to see these bug causes clearly. That said, the instruction trace process used on today's code bases tends to produce huge trace files composed of millions to trillions of instructions. Finding the correct place to look is nearly impossible without a way to navigate at a higher (courser level). Trace Hub provides this capability by allowing the programmer to label code states in real time trace.

Programmers debugging BIOS (UEFI today) have been without trace for the last 20 years. Intel has only recently decided to put the machinery in the silicon to provide trace. Debug of UEFI (and other firmware) on Intel Architecture (IA) based systems can now be much more efficient.

The really hard bugs have the biggest effect on program schedule slippage. These same bugs are the ones most positively affected by the availability and use of trace.

Debug Overview: Where Trace Fits

Debugging firmware such as UEFI is basically the same process as debugging any other programmed system. After the code is written the following steps are taken:

(1) The code is stepped or moved through manually to see that it basically works. (code walking)
(2) When each module or system is thought to be mostly functional, the system is then run as a whole and exercised by some form of test suite or environment.
(3) Miss-behaviors (bugs) are observed in the running system. These bugs are then root caused and fixed.

There are two distinct types of debugging available to programmers (in most environments today). These are:

(A) Static Debug: This type of debug involves stopping program execution at some point by means of either a breakpoint or as the result of a step. The engineer then examines the program object values in that state. These program objects, which are accessed either directly or indirectly, are the processor registers, memory values and other hardware registers within the system. This type of debug is quite adequate for code walking as mentioned in (1) above and for easy-to-find root causes, such as those found during (2) above when the system is run as a whole and where the coding error is near the symptom in address space and execution order.

(B) Dynamic Debug: This type of debug uses some method to record the execution of a program while it is running and, after the experiment examination of the results. This is trace. In most modern application processors, for practical reasons, this is often limited to the instructions executed and not the values (data objects) that these instructions operated on. This experiment often ends in stopping the execution, but this is not required and not always possible. Dynamic debug is the only effective method of finding the root cause of bugs found in item 2 above. This type of bug involves a flaw or failure in a running system.

In an environment where there is not an operating system running yet (UEFI) or when debugging the kernel of that operating system, the tools (debug tools) that provide the functionality listed in A and B above must run on a separate computer from the one where the bugs are being found. This is because any quality debugger will need the features of an operating system like Windows, Linux or OSX in order to provide the debug environment expected. This arrangement is called remote-hosted debugging. It also usually provides hardware assisted debug, which consists of some sort of pod that interfaces between the computer hosting the debugger and the system on which the code is running (which is being debugged).

Most of today's code is written in C or C++. Therefore, there is lots of code reuse and many layers of calls from the code that is sequencing a process to the code that is doing a specific piece of work. This means that the actual IP of the code that may exhibit misbehavior (bug) will provide little information about where in the overall process the error occurred. For example, a string copy routine that was passed a bad pointer does not have a bug in it just because it attempts to write to a location that causes a bus hang. The bug is related to the bad value in the pointer passed to it. Likewise, the routine that called the string copy may have only passed on the bad pointer. This sort of layered software may be many tens of levels deep. The actual bug may be somewhere very different. Finding the actual root cause is the whole point in using trace tools. This level of complexity is why instruction trace must be augmented with System Trace (Trace Hub) in order to navigate the massive instruction trace data set.

A Brief History of Intel Trace

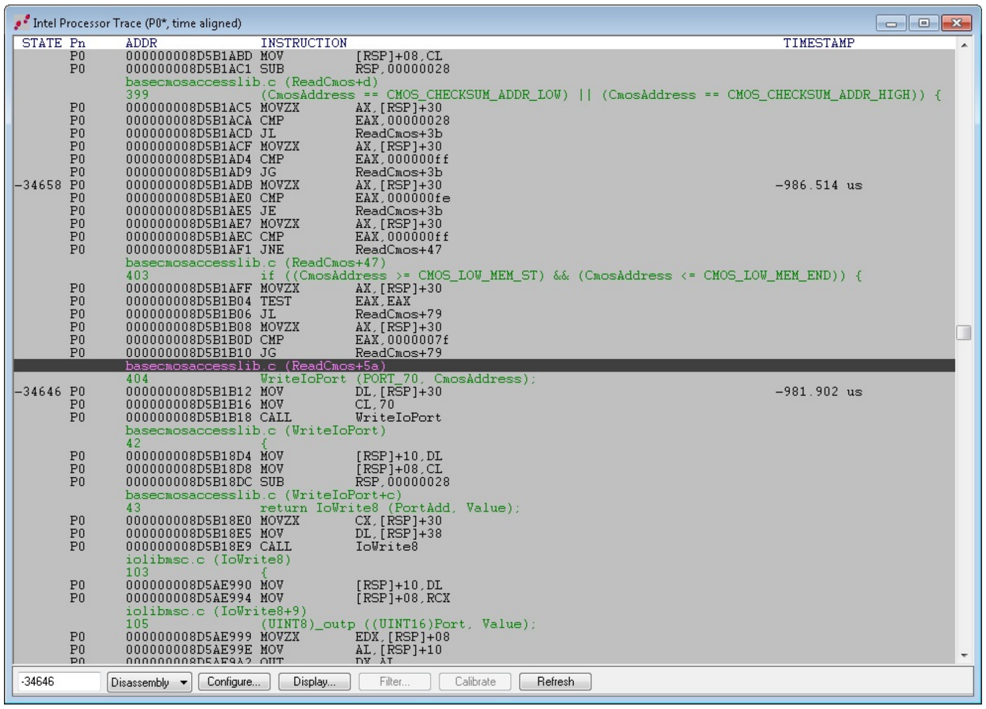
In order to understand why the new trace additions to Intel's processors are so important it is useful to look briefly at the history of trace for debugging code on microprocessors and particularly on those offered by Intel. These make up the vast majority of the processors in general purpose computers today. They, of course, all have their heritage based on Intel's 8088 microprocessor.

In the early days of the microprocessor the chips, like the 8086 and 68000, had front side busses that were ahead of any memory or cache. This meant that all of the information to trace the program a processor was running was available on the processor pins in the form of the front-side-bus. Later, in order to get around the issues of cache, Branch Trace Messages (BTMs) were added to the Intel architecture. This means that up until the mid-1990's almost all Intel processors could be traced by recording the signals on the pins of the processor and then post processing that data (and possibly the memory image). During this period almost all BIOS, on Intel based computers, was debugged through the use of front-side-bus tracing.

In the late 1990's, the speeds of the front side bus became impractical to instrument. Soon after that, the introduction of QPI and follow-on multilane serial busses made front side bus tracing impractical if not impossible. In addition, the pipelining and speculative execution made BTMs so invasive, in terms of slowing execution, that there was no longer any real useful method of capturing instruction trace on an Intel core. Intel made no strong effort to work around these issues.

During the period of time from around 1996 until 2015 most BIOS (later EFI) was debugged without the aid of trace. Developers became used to relying on other methods like printf. Intel added some minimal trace mechanisms like Last-Branch-Record (LBR) and Branch-Trace-Store (BTS). They also added non-execution trace mechanisms like Architectural Event Trace (AET). LBR trace which held 4 to 32 trace-records, was so shallow it was rarely used. BTS was so invasive and so difficult to set up it also was not used much.

A few years ago, driven by competitive pressure, Intel began the task of adding hi-performance instruction (execution) trace to its processors. An illustration of Intel Processor Trace (execution trace) is shown in Figure 1 below. Beginning in 2015 all of its new chips contained this feature. In addition the Trace Hub was added to the mix. Today, in 2015 Intel processors contain a very powerful set of trace features. The combination of Intel Processor Trace (Intel PT) and the Trace Hub, including system trace messaging (STM), provide a world class set of features that tools can be used to render very usable trace displays and processing elements.



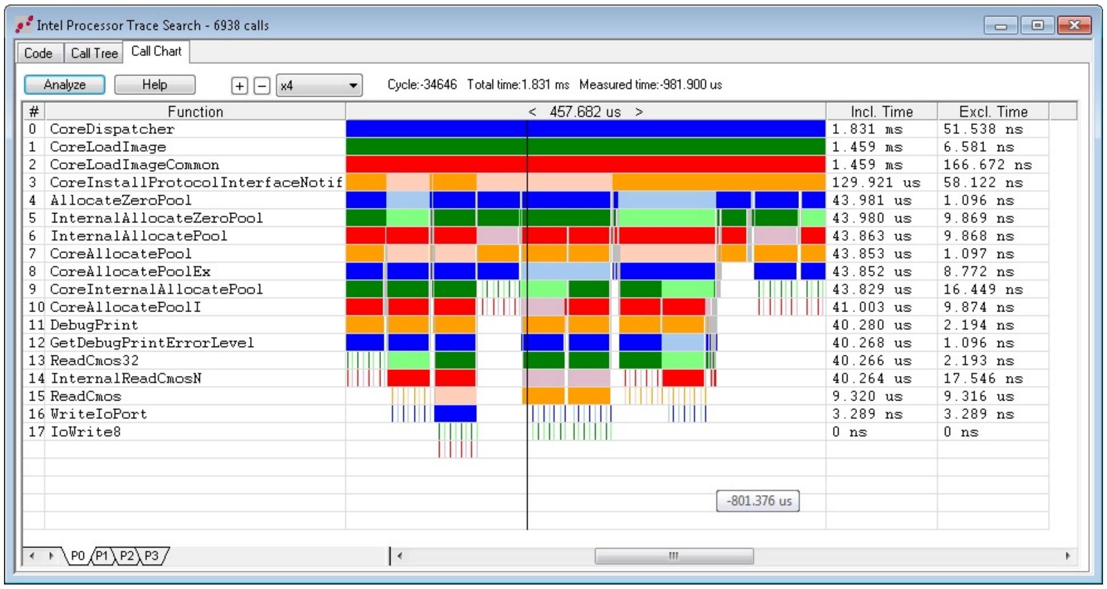
```
 STATE Pn      ADDR              INSTRUCTION                                                 TIMESTAMP
       P0      000000008D5B1ABD MOV          [RSP]+08,CL
       P0      000000008D5B1AC1 SUB          RSP,00000028
               basecmosaccesslib.c (ReadCmos+d)
               399              (CmosAddress == CMOS_CHECKSUM_ADDR_LOW) || (CmosAddress == CMOS_CHECKSUM_ADDR_HIGH)) {
       P0      000000008D5B1AC5 MOVZX        AX,[RSP]+30
       P0      000000008D5B1ACA CMP          EAX,00000028
       P0      000000008D5B1ACD JL           ReadCmos+3b
       P0      000000008D5B1ACF MOVZX        AX,[RSP]+30
       P0      000000008D5B1AD4 CMP          EAX,000000ff
       P0      000000008D5B1AD9 JG           ReadCmos+3b
-34658 P0      000000008D5B1ADB MOVZX        AX,[RSP]+30                                   -986.514 us
       P0      000000008D5B1AE0 CMP          EAX,000000fe
       P0      000000008D5B1AE5 JE           ReadCmos+3b
       P0      000000008D5B1AE7 MOVZX        AX,[RSP]+30
       P0      000000008D5B1AEC CMP          EAX,000000ff
       P0      000000008D5B1AF1 JNE          ReadCmos+47
               basecmosaccesslib.c (ReadCmos+47)
               403              if ((CmosAddress >= CMOS_LOW_MEM_ST) && (CmosAddress <= CMOS_LOW_MEM_END)) {
       P0      000000008D5B1AFF MOVZX        AX,[RSP]+30
       P0      000000008D5B1B04 TEST         EAX,EAX
       P0      000000008D5B1B06 JL           ReadCmos+79
       P0      000000008D5B1B08 MOVZX        AX,[RSP]+30
       P0      000000008D5B1B0D CMP          EAX,0000007f
       P0      000000008D5B1B10 JG           ReadCmos+79
               basecmosaccesslib.c (ReadCmos+5a)
               404              WriteIoPort (PORT_70, CmosAddress);
-34646 P0      000000008D5B1B12 MOV          DL,[RSP]+30                                   -981.902 us
       P0      000000008D5B1B16 MOV          CL,70
       P0      000000008D5B1B18 CALL         WriteIoPort
               basecmosaccesslib.c (WriteIoPort)
               42               {
       P0      000000008D5B18D4 MOV          [RSP]+10,DL
       P0      000000008D5B18D8 MOV          [RSP]+08,CL
       P0      000000008D5B18DC SUB          RSP,00000028
               basecmosaccesslib.c (WriteIoPort+c)
               43               return IoWrite8 (PortAdd, Value);
       P0      000000008D5B18E0 MOVZX        CX,[RSP]+30
       P0      000000008D5B18E5 MOV          DL,[RSP]+38
       P0      000000008D5B18E9 CALL         IoWrite8
               iolibmsc.c (IoWrite8)
               103              {
       P0      000000008D5AE990 MOV          [RSP]+10,DL
       P0      000000008D5AE994 MOV          [RSP]+08,RCX
               iolibmsc.c (IoWrite8+9)
               105              (UINT8)_outp ((UINT16)Port, Value);
       P0      000000008D5AE999 MOVZX        EDX,[RSP]+08
       P0      000000008D5AE99E MOV          AL,[RSP]+10
       P0      000000008D5AE9A2 OUT          DX,AL
```

Figure 1: Intel Processor Trace

Software/Firmware Debug: A Changing Problem

Today's software programs present a much more difficult environment in which to locate the cause of failures (bugs). There are several factors that contribute to this difficulty in finding a bug. This means that those 3 to 8 really hard-to-find bugs may extend a nine month development and test cycle by two to four months. Some of the factors that make this modern environment so difficult to locate bugs in include:

(1) The size of the code base has grown dramatically. Today's UEFI can easily be 7 MB. That is a pretty big boot loader! Code complexity grows at least exponentially with code size.
(2) Multicore code is much more difficult to debug than a single thread. Although this has less effect on UEFI, it is still important. A totally asynchronous event can cause code to run on a different core than the one running the code being debugged. This can lead to errant memory modification that is unexplainable in the context of a single thread.
(3) Multiple execution engines are even worse if they errantly modify objects in the others memory space. A clear example (but not the only one) of this is the Management Engine (ME) running at the same time as the host cores.
(4) Lastly, or maybe firstly, object oriented coding practice has drastically (intentionally) made a large percentage of a code base run in many different states of the software (accessed from many places and in many different states). This means that even in BOOT code such as UEFI, much of the actual work is done at a call depth of 10 to 20 levels. Due to code reuse (think of string copy), the actual address of the code running (and maybe doing something bad to the underlying data) does not relate to program state at all. In general, program state will be indicated by code address in one of the top (highest on the call stack) portions of the executing code. Figure 2 is an actual call graph based on a trace of UEFI. Figure 2 illustrates the call depth in modern UEFI.



Cycle:-34646 Total time:1.831 ms Measured time:-981.900 us

#	Function	< 457.682 us >	Incl. Time	Excl. Time
0	CoreDispatcher		1.831 ms	51.538 ns
1	CoreLoadImage		1.459 ms	6.581 ns
2	CoreLoadImageCommon		1.459 ms	166.672 ns
3	CoreInstallProtocolInterfaceNotif		129.921 us	58.122 ns
4	AllocateZeroPool		43.981 us	1.096 ns
5	InternalAllocateZeroPool		43.980 us	9.869 ns
6	InternalAllocatePool		43.863 us	9.868 ns
7	CoreAllocatePool		43.853 us	1.097 ns
8	CoreAllocatePoolEx		43.852 us	8.772 ns
9	CoreInternalAllocatePool		43.829 us	16.449 ns
10	CoreAllocatePoolI		41.003 us	9.874 ns
11	DebugPrint		40.280 us	2.194 ns
12	GetDebugPrintErrorLevel		40.268 us	1.096 ns
13	ReadCmos32		40.266 us	2.193 ns
14	InternalReadCmosN		40.264 us	17.546 ns
15	ReadCmos		9.320 us	9.316 us
16	WriteIoPort		3.289 ns	3.289 ns
17	IoWrite8		0 ns	0 ns

Figure 2: Call Graph of UEFI Trace

Items 2 and 3 above along with exception handling in a given core often produce completely asynchronous events which might either directly or indirectly cause memory corruption. These events are not apparent in static debug and trace must be used to efficiently find them.

The combined effects of these various factors mean that in order to stop a program at or near the symptom of a bug and stop collecting trace at that time, a very large amount of trace will need to be collected (millions to trillions of instructions).

For problems where the root cause was errant memory corruption by a different agent (a very common manifestation in really hard to find bugs) the actual bug will be identified in detailed code examination at some point preceding the symptom based trigger event by huge distances in trace. This means that directly looking through detailed instruction trace, as was done in the 1980s and 1990s is not useful. It might take months to find the fault even once it is captured in trace.

TODAY's Complex Debug Problems need "New Trace Tool Features".

What is needed today is a way to view the captured trace at a higher and more abstract level, related to the way the code was designed, so that the engineer looking at the problem can see what was happening at various spots in trace. This will allow the software engineer to navigate through the captured trace and find the areas he needs too. Trace Hub allows tracing of events in several ways which show the software engineer what the program or thread is doing in different areas in the trace. He can then drill down as needed in any area of instruction trace in order to look at the details. Use of Trace Hub and code call analysis tools provide easy-to-use and fast methods for bouncing between gross system state and code execution detail, even in a huge code base.

Introduction to Trace Hub and its Relation to Intel PT

Intel added the Trace Hub to its processor system in order to provide several trace features that augment the newly added Intel PT. These features include (1) coalescing of trace streams from different sources, with IDs and timestamps, (2) System Trace feature (STM) and (3) common multipath trace transport. In the first generation of Intel chips to offer the trace hub (server and client solutions) the primary advantage is the STM feature set. This allows for code instrumentation (similar to printf, but real time) in several code bases and multiple cores. The post processed trace stream can then be time correlated to the Intel PT trace streams to provide the trace navigating facilities described in this eBook.

The Intel Trace Hub is a piece of hardware (IP within the Intel chips) that is a slave on the MMIO fabric of a given platform or SOC. The vagueness here is necessary because Trace Hub is used in both SOCs as well as the largest server platforms. A write to a memory space that is not routed to one of the actual DDR memory controllers or north-complex IO may be destined for the Trace Hub. In client or server sets of chips today, this means that writes to the Trace Hub are routed down the DMI fabric. In SOCs the nature of the fabric is different. In addition there can be other transports that stimulate trace message generation in the Trace Hub.

In the most basic use, programs running on any core (IA, ME, or…) can write debug messages (think printf or logging) and direct them to Trace Hub instead of to a serial port, console or memory log. This means they are timestamped and correlated with instruction trace. It also means they can be transported in a variety of ways depending on target, silicon and target state.

The address space of the slave interface to the Trace Hub is divided in masters and each master is subdivided into channels. Writes to different addresses within a channel, result in the production of different message types. All messages and timestamp packets are eventually formatted in STPV2 which is a MIPI standard for system trace encoding and transport. The nature of this protocol is intended to be generated with Master/Channel message identification. In this way the message identification is implicit instead of needing to be handled by the software under debug. This mechanism provides for a more code efficient (low insertion loss) method of instrumenting code. This means instrumentation inserted in the code has less effect on the codes size and speed of execution.

In client and server chip sets the important sources for software debug are:

(A) Direct program writes from the host cores
(B) Direct program writes from the ME
(C) AET (Architectural Event Trace)

In SOCs, in addition to the above, Intel PT may be routed through the Trace Hub. A simple view of the Trace Hub is shown in Figure 3.

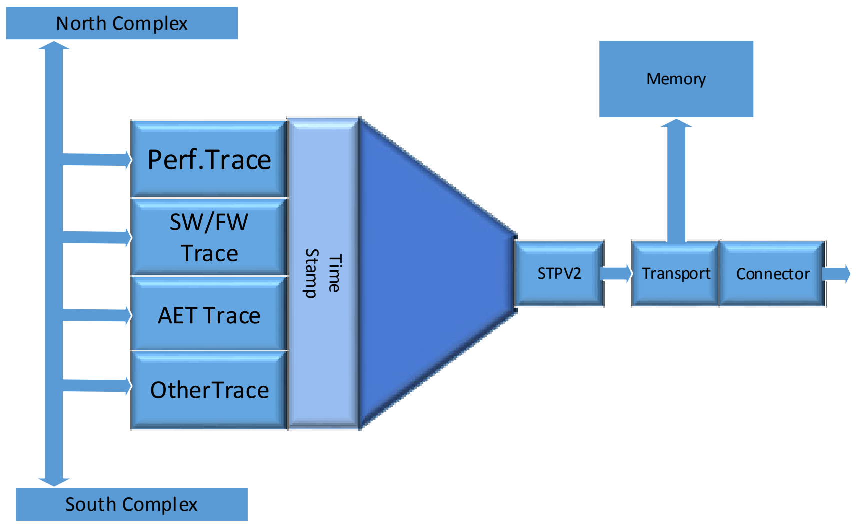


Figure 3: Simple View of Trace Hub

In Figure 3, above, there is a time stamp unit shown. This unit can place a time stamp on each individual trace message. There is a timestamp alignment mechanism that allows Trace Hub time stamp to be directly correlated to each Intel PT (Instruction Trace) stream in the system. Therefore, all views of each of the types of trace can be time aligned with all other views. This means locating an event in a Trace Hub sequence will directly point to a spot in the instruction trace.

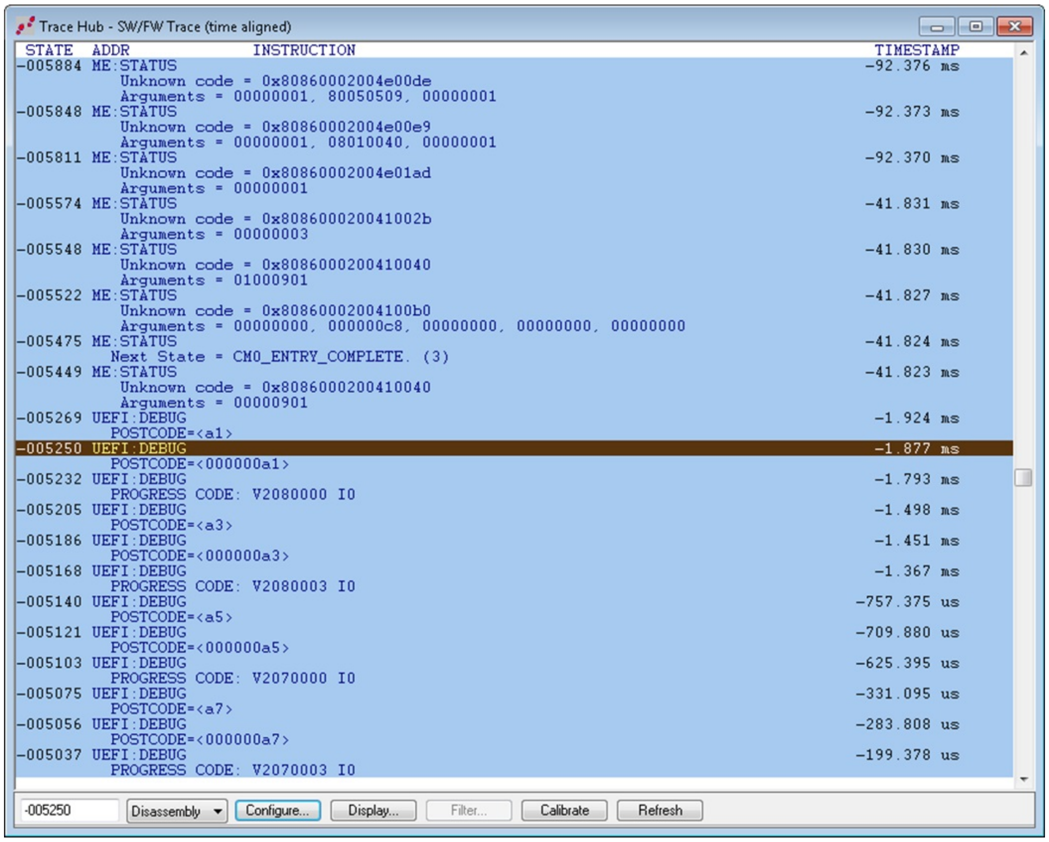
How Trace Hub can shorten the time to find the really hard bugs.

Isolating a bug usually begins by observing a symptom (blue screen?) and then attempting to set a breakpoint at the point in the code that is generating the symptom. (This might be as simple as searching the code base for the string that is printed at the point of the symptom. A breakpoint can then be set on the code that outputs that string). For simple bugs this is almost the end of the process as it is likely that statically inspecting the code will quickly reveal the bug. Harder to find bugs almost always show up at this point as a piece of code operating on bad data. The executing code is not making an error; it is just processing data that is not correct.

With execution (instruction) trace, it is then possible to see what function called the function processing the wrong data. In a large body of code like UEFI, this is still likely to be a properly operating piece of code that is, again operating on bad data. This story, and the time it takes to wade through trace, can go on and on. A much more effective way to view the trace at this point is to look at a less fine grained resolution view of it. A call analysis tools might be a good solution. This can show the software engineer what major functions are running and which have just run. It will be obvious what called the offending code errors and passed down bad data.

For really complex situations, STM trace with good instrumentation in the code base can make this quick and easy. There needs to be instrumentation in the code that outputs messages when major functions of the code base are started or important nodes within these functions are reached. It turns out that the UEFI code base already has many such lines of instrumentation which currently are passed to the serial port or to the console (depending on where in the BOOT process the message is generated. In UEFI (EDK2 generated) many of these messages are generated with a "DEBUG" macro which uses syntax much like printf, including a printf string. These STM messages can easily be redirected to the Trace Hub. With this small change to the UEFI code base, there is now the two level trace system described here. In the future, these messages can be augmented to take even better advantage of STM with Processor Trace.

A source level software debugger, like SourcePoint™ from ASSET InterTech, should have trace list displays which clearly display the formatted STM messages, with time stamps, and with the ability to directly synchronize the cursor in Intel Processor Trace to the STM message trace listing. In this way moving from the macro view to the micro view is trivial. This is illustrated in Figure 4 below.



```
STATE   ADDR                INSTRUCTION                                              TIMESTAMP
-005884 ME:STATUS                                                                   -92.376 ms
           Unknown code = 0x80860002004e00de
           Arguments = 00000001, 80050509, 00000001
-005848 ME:STATUS                                                                   -92.373 ms
           Unknown code = 0x80860002004e00e9
           Arguments = 00000001, 08010040, 00000001
-005811 ME:STATUS                                                                   -92.370 ms
           Unknown code = 0x80860002004e01ad
           Arguments = 00000001
-005574 ME:STATUS                                                                   -41.831 ms
           Unknown code = 0x808600020041002b
           Arguments = 00000003
-005548 ME:STATUS                                                                   -41.830 ms
           Unknown code = 0x8086000200410040
           Arguments = 01000901
-005522 ME:STATUS                                                                   -41.827 ms
           Unknown code = 0x80860002004100b0
           Arguments = 00000000, 000000c8, 00000000, 00000000, 00000000
-005475 ME:STATUS                                                                   -41.824 ms
          Next State = CM0_ENTRY_COMPLETE. (3)
-005449 ME:STATUS                                                                   -41.823 ms
           Unknown code = 0x8086000200410040
           Arguments = 00000901
-005269 UEFI:DEBUG                                                                   -1.924 ms
          POSTCODE=<a1>
-005250 UEFI:DEBUG                                                                   -1.877 ms
          POSTCODE=<000000a1>
-005232 UEFI:DEBUG                                                                   -1.793 ms
          PROGRESS CODE: V2080000 I0
-005205 UEFI:DEBUG                                                                   -1.498 ms
          POSTCODE=<a3>
-005186 UEFI:DEBUG                                                                   -1.451 ms
          POSTCODE=<000000a3>
-005168 UEFI:DEBUG                                                                   -1.367 ms
          PROGRESS CODE: V2080003 I0
-005140 UEFI:DEBUG                                                                 -757.375 us
          POSTCODE=<a5>
-005121 UEFI:DEBUG                                                                 -709.880 us
          POSTCODE=<000000a5>
-005103 UEFI:DEBUG                                                                 -625.395 us
          PROGRESS CODE: V2070000 I0
-005075 UEFI:DEBUG                                                                 -331.095 us
          POSTCODE=<a7>
-005056 UEFI:DEBUG                                                                 -283.808 us
          POSTCODE=<000000a7>
-005037 UEFI:DEBUG                                                                 -199.378 us
          PROGRESS CODE: V2070003 I0
```

Figure 4: STM trace of UEFI and ME

In addition to using STM to provide a guide to what functions are running, it can also be used to trace the data a program is operating on. Intel PT, much like many other trace mechanisms on high performance application processors, does not trace the values of data being operated on. It only traces which instructions were executed. All decisions in code are made on data values (A==B). Many times to truly understand why a program decision went wrong it is necessary to know the value of a program variable at some specific point in that program. It is not practical to trace all data all the time. This would consume two much trace transport bandwidth. STM allows inserting very small code snippets that dump particular values just at the time it is desired to test the value. These data probes are inserted temporarily when working through a problem. Again they are timestamped so that correlating with Instruction Trace is easy. This process in general is referred to as data tracing.

In summary, the use of Trace Hub described here provides navigation, drill down, and data value probe capabilities. These features are exactly what is needed to make it easy and more efficient to work with large buffers of Instruction Trace. Again, it is Instruction Trace that will actually reveal the specific software malfunction.

What it takes to Use the Trace Hub

There are several things involved in setting up an environment to use Intel's Trace Hub. Some of these are within the tool set used, some are within the target software being debugged and some may involve hardware configuration. Exactly how each item is accomplished is dependent on the version of Intel chips you are using as well as the debug tools you are using. The various items required for 6th generation Intel chips include:

A. Target:
   (1) Provide an area in RAM to store trace buffers. In the UEFI world this is accomplished by allocating a reserved, non-cacheable area for all trace buffers. This is described in detail in the BIOS writers specification provided by Intel.
   (2) Build in STM messages as needed into the code bases that are to be debugged (instrument your code). For console/serial port messages from UEFI, this is a simple modification to a couple of files. ME messages are built into the firmware distributed by Intel.

B. Target, via debug tool:
   (1) Set up registers that enable the Trace Hub as well as point to the storage buffer. In SourcePoint this is all done via settings in the SourcePoint GUI interface.

C. Via Tool at experiment time:
   (1) Establish an end point for the experiment. This is usually a breakpoint (trigger) that stops the processors from executing.
   (2) Run the experiment
   (3) Select the display desired in the tool and the tool should automatically perform the trace buffer post processing, using the trace buffer contents, the object files associated with the source files and possibly the target static information. This last item is dependent on tool set up. The post processing of STM information will also require a

metadata file that provides target/program information required for the post processing of the raw STPV2 data.

Once this is all set up, each successive experiment will cause the tools to capture and prepare all of this automatically so that the displays are available at the click of a button.

STM via Trace Hub Overcomes Performance Issues of Printf Techniques

UEFI, like other embedded code bases, is riddled with printf style debug statements. In the EDK2-based version of UEFI these statements are actually macros which call debug print routines with several possible destinations. These macros, such as “DEBUG”, use a syntax and process which includes, as an argument, a printf style text formatting string. The processing code to turn the arguments into a readable string can run thousands of instructions in the target code being debugged. This execution time added to the backpressure caused by synchronous call to drivers for slow transports such as RS232 serial IO can cause the execution speed of the boot code to increase from tens of seconds to several minutes. Not only is this console logging process time consuming for the software engineer, it often changes the nature of the code running program enough to stop the bug from occurring.

The speed of the Trace Hub as a transport, in addition to its other advantages, removes the effect of the transport induced backpressure. This improvement combined with modern tools that move string formatting to the tools and to display time, almost completely remove the timing effects of enabling all of the debug messaging.

Taking these advantages even farther, ASSET InterTech’s SourcePoint offers a proprietary version of this tool based string formatting that makes it transparent to the software engineer, so that he adds messages, just like he did before yet gains all the advantages of ASSET’s “Tool Hosted Printf”.

Summary

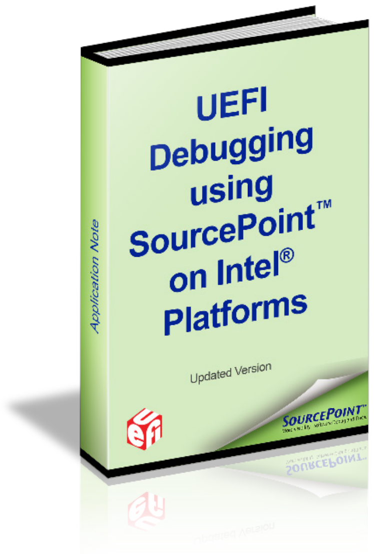
Intel's Trace Hub when combined with Intel Processor Trace, and used with modern trace based debug tools, can greatly reduce the schedule time used to find the hardest code bugs. This can often reduce UEFI development time on a new platform by several months. These techniques and tool features are new to the Intel processor-based firmware development world (mostly, UEFI and ROM base UEFI drivers). ASSET InterTech's Source Point is a powerful example of a tool whose trace features really surface the strengths and potential of these newly added debug mechanisms in Intel processors. STM through the Trace Hub is the navigation tool that makes Intel instruction trace really be the answer to locating complex software system bugs.

Conclusions

Software engineers and their managers should evaluate the newly added trace features available in the latest Intel Processors and SOCs. These features combined with strong remote-hosted software debuggers, such as ASSET InterTech's SourcePoint can have a positive impact on development schedule development time for UEFI code bases (or any other firmware running on these processors)

Learn More

Spending too much time trying to navigate the growing UEFI code base or debugging UEFI code after you inserted your module? Check out this eBook.



[Register Today!]