Everything You Always Wanted to Know About Embedded Trace

Preprint · May 2021

DOI: 10.36227/techrxiv.14595960.v1

Thomas B. Preußer
ETH Zurich
62 PUBLICATIONS **525** CITATIONS
SEE PROFILE

Smitha Gautham
Virginia Commonwealth University
28 PUBLICATIONS **29** CITATIONS
SEE PROFILE

Carl R. Elks
Virginia Commonwealth University
91 PUBLICATIONS **403** CITATIONS
SEE PROFILE

Alexander Weiss
Accemic Technologies GmbH
43 PUBLICATIONS **512** CITATIONS
SEE PROFILE

Everything You Always Wanted to Know About Embedded Trace

Thomas B. Preußer
Accemic Technologies

Smitha Gautham
Virginia Commonwealth University

Abhi D. Rajagopala
Virginia Commonwealth University

Carl Elks
Virginia Commonwealth University

Alexander Weiss
Accemic Technologies

***Abstract*—Decades of advances in computer architecture, software-intensive applications and system integration have created significant challenges for embedded systems designers and test engineers. Intrusive software instrumentation and breakpoint-based debugging are often viewed as the primary options for observing operational system internals. This narrow sight creates complicated test flows and convoluted debugging procedures. Modern embedded computing systems offer Embedded Trace as the technological answer to the encountered observability conundrum. Although an integral part of virtually all modern processors, it is frequently overlooked. Its technical foundations are little known to application engineers, test engineers, and project managers. This article explains Embedded Trace as an essential technology in the testing and debugging toolbox. It highlights its capabilities, limitations and opportunities.**

Introduction

As embedded systems become ubiquitous, we increasingly rely on their functionality to perform tasks that were once laborious, costly, tedious, or even unattainable. Advanced embedded system technology is one of the key driving forces behind the rapid growth of Cyber-Physical System (CPS) applications. Applications in robotics, autonomous systems, telemedicine, and avionics are examples of *software-intensive systems* taking on increasingly complex software functionality. For example, a 2018 mid-priced automobile has in the aggregate about 100M lines of code distributed over dozens of embedded processors. This rising trend in software-intensive systems in turn requires advanced hardware platforms, such as Systems-on-a Chip (SoC) or multi-core heterogeneous processors. The confluence of advanced embedded computing hardware and software-intensive systems creates challenges for testing and verification activities, especially for systems where high levels of reliability, safety, and security are required. As noted before [1], advanced embedded processing applications can breed convoluted software anomalies including heisenbugs, which evade debugging due to changing temporal hardware/software interactions under inspection. The ability to detect, debug and isolate such faults efficiently and effectively is essential for a cost-effective system verification.

A key prerequisite to detect and debug such software anomalies is the *observability* of execution flows, access patterns, and dataflows of a programmed system at time intervals of interest. In addition, a desirable property is *non-intrusiveness*, i.e. the avoidance of a probe effect by interfering with the system operation. Most often, we observe program behavior by software instrumentation including breakpoints and `printf`-style debugging. This has known limitations, namely, being overly intrusive to program execution. Especially in complex embedded systems with multi-threaded programs and real-time requirements, software instrumentation can result in timing and synchronization issues that alter program behavior [2].

A promising technology to address the observability issue is *Embedded Trace* (ET). ET is best thought of as a set of real-time streams of software execution data emitted by a processor non-intrusively. ET opens a window to the system behavior for monitoring a software in execution. This can reveal memory access patterns, the execution control and data flow, protocol sequences, etc. Although it is an integral part of almost all modern processors, ET is often seen as “just another port” on a board. The basic technology of ET and how it works is virtually unknown to application engineers, test engineers, and project managers. Their fundamental understanding of ET is very much needed at this time as it can be vital for verifying and debugging software, especially in critical systems. We illustrate a few motivating examples.

Flight Software – Civil Aviation certification authorities mandate to ”test what you fly, and fly what you test”. This ensures that safety claims are based on actual flight software and avoids the danger of activating dormant heisenbugs by the removal of test code. However, it also implies that any test instrumentation must remain part of critical flight software. It makes all instrumentation overhead permanent, often at the cost of system performance. Additionally, it mandates that any execution data collection in operation must undergo the whole test and certification process. ET, on the other hand, enables verifiable software testing and online execution data collection from lean release code without instrumentation.

Automotive Electronic Systems – Dozens of vendor-specific Electronic Control Units (ECUs) are integrated in modern automobiles to realize the full driving experience. These ECUs and their integrated software components are often characterized as black boxes with their internal functionality hidden due to IP proprietary concerns. If there are any timing integration problems (which happens), the need to observe the execution behavior of an ECU arises in order to debug the problem. ET enables the observation of the executing ECU software without requiring access to its source code.

This article provides insights into the technology of ET for embedded systems application engineers, and for the testing and verification community. It highlights the capabilities and limitations of ET by detailing what information it delivers, what is needed for its useful interpretation, and what opportunities it creates in simplifying and streamlining test and debug procedures. Understanding the fundamentals of ET and its application to testing enables embedded cyber-physical systems that are more reliable, safer, and more secure.

Traditional Software Instrumentation

An obvious and widely established observation technique is software instrumentation. The application software is modified to produce the desired information, such as control flow indicators, via standard output channels. Concrete means range from manual `printf` debugging,

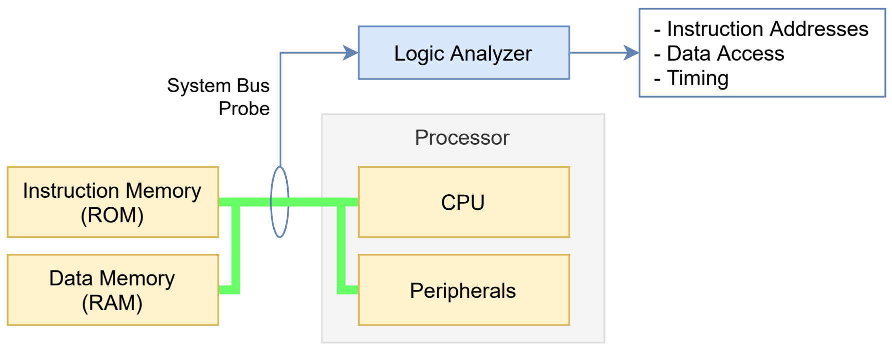


Figure 1. The Olden Days – All CPU Activity Is Visible on the External Bus

over automated source code instrumentation all the way to instrumenting binary translation.

Unfortunately, all of these approaches impact the timing behavior and the memory footprint of an application massively. This implies serious disadvantages in the domain of embedded systems. The changed timing behavior compromises the validity of functional tests. Integration and system tests over timing-critical control loops cannot use this technique at all. Concurrent programs may suffer from the introduction of phantom synchronization by the arbitration of shared output channels. Safety-critical software already deployed in the field would have to repeat the complete test and certification process for any change in its instrumentation. Less critical application domains sometimes choose to mitigate the observability challenge by demonstrating test coverage and functional correctness in two separate test runs, one with decelerated instrumented code and the other with stripped release code. This repetition is costly, especially when using expensive test equipment, such as HIL test benches. The confidence in this approach is fading with growing system complexity as it increases the likelyhood of heisenbugs. In spite of its drawbacks, software instrumentation is still widely used – often due to the lack of a handy alternative.

Embedded Trace

Motivation

In the olden days, processors were slow (some 10 MHz) and only had external program memory. Eavesdropping on the memory bus, as shown in **Figure 1**, was sufficient to observe the memory accesses by the CPU. Following the program execution along the fetched instructions was, thus, rather trivial.

CPUs have become faster, and their architectures have been changing dramatically:

- Instructions and data are cached near the CPU, inside the same chip. Individual memory accesses are no longer visible externally.
- Particularly, embedded devices integrate more and more memory (RAM or Flash) internally for faster access times and lower system cost. Also, accesses to such memory are no longer observable externally.
- To meet the demand for more and more integrated computing power and to reduce energy consumption, several CPU cores are integrated into one processor. Whatever external effects remain observable, they now also pose the challenge of attributing them to the correct concurrent control flow among the ones scheduled across truly parallel cores.

All these advances in processor architecture and system integration mandated ET as a radically new approach to observing what is happening inside the processor.

ET is the integration of functional units that make the activity of CPUs observable. It needs to address a significant bandwidth problem as monitoring a single CPU comprehensively requires information about the executed instructions and affected CPU registers. For a 32-bit CPU, a naïve encoding would produce trace data of roughly 70 bits for each instruction, including its address as well as register and flag updates. This translates to a bandwidth demand of 70 Gbit/s when running at 1 GHz. This is far beyond an economically reasonable solution. Clearly, the trace data stream

Table 1. Most Relevant Trace Message Categories

Control-Flow Messages	Timing Messages
Synchronization	Timestamps
Branching	Cycle Counts
Exceptions	Clock Speed

Data-Flow Messages	Other Messages
Memory Addresses	Trace Buffer Overflows
Data Values	Lightweight Instrumentation

must be compressed.

Protocol Overview

Actual trace implementations such as Arm® CoreSight™ Program Flow Trace™ (PFT) [3], Arm® CoreSight™ Embedded Trace Macrocell™ (ETM) [4], Intel® Processor Trace (Intel® PT) [5, § 35], and Nexus 5001 Forum™ [6] compress the transmitted control flow trace vigorously. They inject relevant operational information into their output to facilitate a highly efficient, context-aware compression. System observability can be further boosted by tracing the data flow. However, the implied trace data is much harder to compress. This results in significantly higher bandwidth requirements and, hence, more costly trace interfaces. Therefore, most implementations refrain from implementing this capability.

We start the illustration of the trace compression techniques with an overview over the most relevant trace messages as categorized by **Table 1**.

Control-Flow Messages The continuous output of the program counter alone would consume significant trace bandwidth. This is overcome (a) by assuming that the executed application is known to the observer and (b) by exploiting the default sequential execution of instructions. This allows to use the following strategy for trace compression:

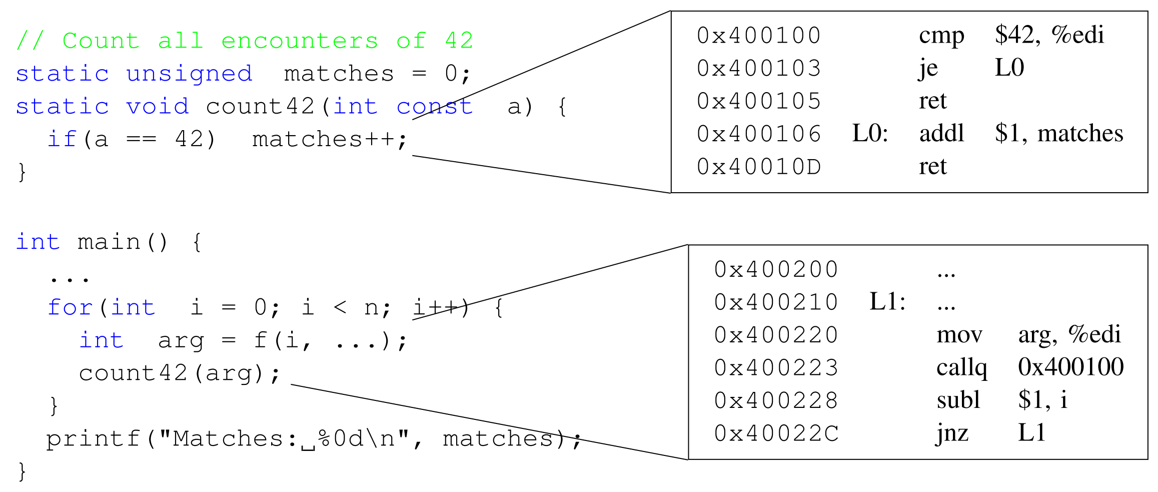
Synchronization Messages
are generated in greater intervals, e.g., every few thousand messages. Only they establish a complete application context and a concrete program counter to identify the reference point for the further trace data interpretation. The execution of the sequential code following this point is implied.

Branch Messages
communicate actual control flow decisions. A single bit indicates whether a conditional branch has been taken or not. Branches not taken imply the sequential continuation of the execution. They do not require any further trace data. Neither do taken direct branches as their fixed target can be inferred from the executing application binary. Only taken indirect branches trigger the generation of an alternate larger message that enables the observer to reconstruct the dynamically computed branch target address. The handling of unconditional direct branches is specific to the ET architecture. Some choose to establish posts in the control flow and produce taken execution indicators just as they do for executed conditional direct branches (e. g. Arm® PFT). Others pass over this clearly implied control flow without any representation in the emitted trace (e. g. Intel PT).

Exception Messages
are generated for externally induced control flow diversions such as by interrupts. They typically provide a hint on the nature of the exception, identify where the implied control flow was interrupted and contain all information necessary to resume the control flow reconstruction, either including or skipping the interjected handler depending on the trace configuration.

Note the vigorous reduction of the control flow trace to the resolution of dynamic control flow decisions. Its encoding is the other ingredient for pushing the average trace bandwidth demand below a single bit of trace data per executed instruction. The decision indicators generated for conditional branches are aggregated into messages with very small protocol overhead. Up to five (ARM® CoreSight™) or six (Intel® PT) of them are transmitted within a single byte. Address transmissions, as after the executions of indirect branches, refer back to the preceding communicated instruction pointer so as to only encode differences. This typically avoids the transmission of complete 32- or even 64-bit addresses.

Timing Messages Depending on the ET architecture, different message types conveying timing information may be available. In addition to wall-clock messages, cycle counts are of special importance. They indicate how many CPU



```
// Count all encounters of 42
static unsigned  matches = 0;
static void count42(int const  a) {
  if(a == 42)  matches++;
}

int main() {
  ...
  for(int  i = 0; i < n; i++) {
    int  arg = f(i, ...);
    count42(arg);
  }
  printf("Matches:␣%0d\n", matches);
}
```

```
0x400100         cmp    $42, %edi
0x400103         je     L0
0x400105         ret
0x400106   L0:   addl   $1, matches
0x40010D         ret
```

```
0x400200         ...
0x400210   L1:   ...
0x400220         mov    arg, %edi
0x400223         callq  0x400100
0x400228         subl   $1, i
0x40022C         jnz    L1
```

Resolved Return			Compressed Return	
PSB			PSB	
FUP	`0x400220`	– Synchronization –	FUP	`0x400220`
⋮			⋮	
PSBEND			PSBEND	
TNT	`N`	– Conditional branch `@0x400103` not taken –	TNT	`NTT`
TIP	`0x__0228`	– Return –		
TNT	`T`	– Conditional branch `@0x40022C` taken (loop) –		
⋮			⋮	

Figure 2. Illustrating Execution Trace Example

clock cycles have elapsed since the last timing update. Observers can typically choose to receive cycle-count messages with each branch message, in programmable time intervals or not at all. This way, the significant trace bandwidth consumption by high-frequency cycle-count messages can be balanced against other desired trace quality properties.

Data-Flow Messages Data trace is difficult to compress. Depending on the trace architecture, the address of a data access, the transmitted value and the type of access may be communicated. In a 32-bit system, each access can result in a trace message of more than 60 bits in length. Due to its high bandwidth requirements, data trace is not available in all implementations and otherwise regularly limited. For instance, it may be constrained to designated address regions or to producing partial information such as the addresses of write accesses only.

Other Messages There are a number of other trace message types. They establish the trace context as by allowing the OS to communicate context switches and convey trace-specific information as for signaling internal trace buffer overflows. Also, messages to transport programmatically injected user data for very lightweight instrumentation are part of many trace protocols.

Illustrating Trace Example

For an illustration of the vigorous compression of the control flow trace, consider the example of **Figure 2**. It comprises a simple iteration over some input for counting all occurrences of the value `42`. Assume that the compiler has generated the annotated `x86_64` assembly for the code of interest. Two possible Intel® PT trace observations are tabulated.

Every now and then, a synchronization block is emitted. It is opened by a `PSB` message whose encoding can be uniquely detected even in an unsynchronized trace stream. It is, thus, a point, at which a decoder can always tap in safely. From there on, a decoder has to slice the stream into messages on the basis of a variable-length prefix code. The synchronization block itself is terminated by a PSBEND message. In between, the state is established that is needed for the context-aware compression of the subsequent actual trace. This ensures that not only the decoding of messages but also the interpretation of their data is completely synchronized. Typical constituents of this state are a time basis (TSC, CBR), a process identifier (PIP), and the current program counter (FUP). Assume that the latter positions our observed execution just at preparing arg as the argument to count42().

The following actual trace only communicates dynamic control-flow decisions that cannot be inferred from the assembly. Particularly, the control transfer into function count42 is simply implied. The first event identified in the trace is the decision triggered by the comparison with 42 in this function. Receiving a *not taken* indication in our example, we can infer that the encountered value was apparently not a 42.

For the sake of continuing our reconstruction of the executed control flow, we observe that the next non-sequential control flow transition is prompted by the function return. Two options exist for its trace representation. The trace source may always choose the resolved encoding. Treating the return just like an indirect branch, it communicates the return address as an explicit re-entry target. This communication is, indeed, compressed as it conveys only a patch against the most recently transmitted instruction pointer, which, in our example, was established by the FUP message. The trace then continues with indicating that the conditional branch implementing the loop was taken.

Observe that the transmission of the instruction pointer is costly even when considering that it is regularly just a small patch against the last established value. In the case of function return addresses, a shadow call stack maintained by the decoder would typically be able to provide the needed value as well. Such a widely supported optional return compression allows the trace source to turn the transmission of the program counter into a single-bit *taken* indicator. To ensure that the trace can be interpreted correctly, this very effective compression is predicated on a few conditions: (a) the actual return address must match the address that was pushed by the corresponding call; (b) the return was not separated from its corresponding call by a synchronization block; and (c) the return address never sunk below an architecture-defined stack depth during the execution of the enclosed call hierarchy. This guarantees that a decoder, even if it has just synchronized on the trace, has seen all information needed to resolve the compressed return. Finally, note in the trace example that, as a secondary effect, the sequence of *taken* and *not-taken* indicators can be aggregated into a single, one-byte message for an improved payload utilization.

Embedded Trace Evolution

As system observability was being challenged by advancing integration and faster clock speeds, monitoring capabilities had to become an aspect of the system design itself. While bond-out chips are able to expose internal bus activity to in-circuit emulators, this approach does not scale with the growing system complexity. The design-for-test philosophy eliminated the need and cost for designated bond-out chips by integrating boundary scan chains, as introduced by the popular Joint Test Action Group (JTAG) standard [7]. They enable valuable operational insight at a significantly silicon reduced cost for debugging purposes on a slow timescale. However, the continuous monitoring of a system running under real-time conditions stays far out of reach for this technique.

In contrast to Intel®, Arm® with its strong background in embedded applications embraced the idea of ET early. The messaging protocols of their ETM illustrate the steps of detaching from classic signal probing. ETMv1 uses two physical output channels. The 3-bit *pipeline status* (PIPESTAT) reports the operation of the execution stage of the instruction pipeline on a cycle-by-cycle basis. Whereas this status channel closely resembles a signal probe, the trace packet port (TRACEPKT) communicates accompanying address information as a paced data stream. Internal FIFO buffers flatten bursts for gentler bandwidth requirements. The alignment of the trace data with the pipeline status must be inferred from the latter, which communicates the presence and type of trace packets. Some status bits communicated in the delay slots after each executed branch are repurposed to convey the current slack between these two information flows. ETMv2 emancipates the trace packet stream by introducing packet headers for an inline differentiation of packet types. ETMv3, finally, abolishes the pipeline status altogether and merges all trace communication into a single *trace data* stream. Designated *P-headers* aggregate the execution status of consecutive instructions efficiently.

More recently introduced trace protocols, in particular Arm® PFT and Intel® PT, focus on the concise communication of the program control flow. While the ETMv1-v3 protocols trace the execution of every single instruction, these protocols restrict their communication to control flow diversions that are not already implied. They eschew integral data tracing capabilities. This strong focus makes these ET implementations scale better and more economically with the performance of modern processor systems. They are, nonetheless, trade-offs that limit observability. Some mitigation may be offered through light-weight user instrumentation. Both the Arm® Instrumentation Trace Module (ITM) and the `PTWRITE` mechanism of Intel® PT enable an application to inject user data into the emitted trace stream at run time using very cheap instructions, which particularly avoid competing over software-implemented logging facilities. However, these mechanisms are not supported by all trace-enabled processors.

Also, the evolution of the ETM protocol family continues. ETMv4 has dropped the strict tracing of every single instruction in favor of ideas adopted from PFT. It, nonetheless, takes a more holistic approach by including both data trace and the tracing of the execution of conditional non-branch instructions. The availability of these features is implementation-dependent, their activation is optional. Another remarkable feature of the ETMv4 protocol is its capability to communicate the successful completion or the cancellation of a speculative code execution.

Further proposals for the trace data reduction come from the scientific community. Milenković et al. [8] propose to target repetitions, as they are found in the trace of an iterating loop, by classic compression techniques. While this promises a significant reduction of the average trace bandwidth, it also demands a significant investment in on-chip buffer capacity both for flattening remaining trace peaks and for providing the trace history to facilitate the compression. No commercial implementation has yet committed to this trade-off. They rather remain limited to variations of delta compression. The compression of the instruction pointer is the ubiquitous example shared across all trace protocols. Hochberger and Weiss [9] propose a completely different approach to monitoring a processor. They synchronize an external emulation of the CPU core (digital twin) using runtime information. This much more easily accessible replica enables the complete system observation and analysis. It exposes internals, such as stack pointers or register values, which typically remain hidden when using traditional trace approaches. The required bandwidth here no longer depends on the trace protocol, but on the effort required for synchronization. However, the emulated twin eventually faces similar scaling bounds as a bond-out chip.

Trace Data Processing

Figure 3 illustrates the three main options for processing trace data. Option 1 is the trace data capture in system memory. This solution is often used in desktop environments. However, it has a significant impact on system behavior as it competes with the monitored application over shared system resources. Avoiding this behavioral feedback, trace data can also be captured by external tools via a designated trace interface. Feeding traditional trace tools establishes Option 2. They are essentially large memory buffers collecting the trace data for later offline processing. Both of these buffering approaches share two critical disadvantages. Firstly, the observation time is strictly bounded by the buffer size. Depending on the trace bandwidth, it may allow trace snapshots of a few milliseconds or, at most, seconds. In such severely constrained timeframes, the analysis of long-running integration and system tests, statistically significant measurements of worst-case execution times, or searches for complex non-deterministic errors are only possible to a very limited extent. Secondly, the later offline processing of recorded trace data implies a long latency between observation and perception. This is not only an inconvenience for the engineers involved but also precludes any innovative exploitation of the system observability for prompt reactions.

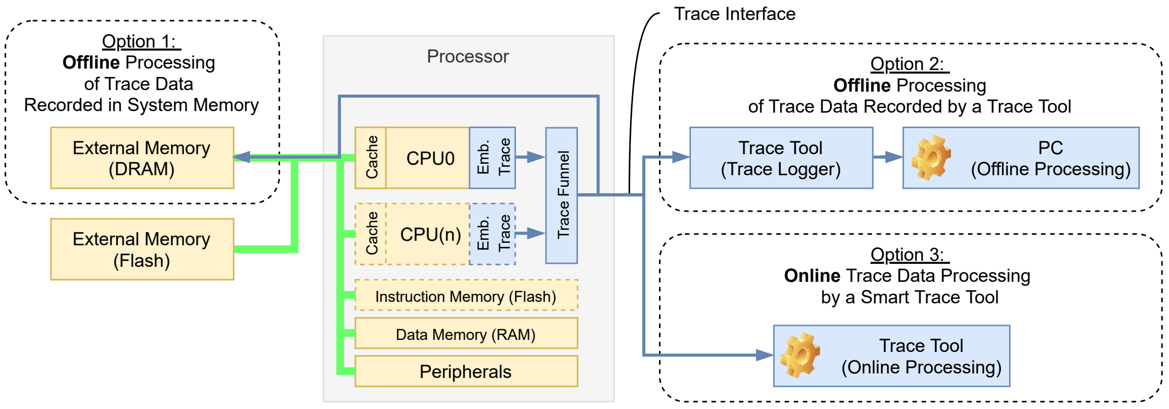


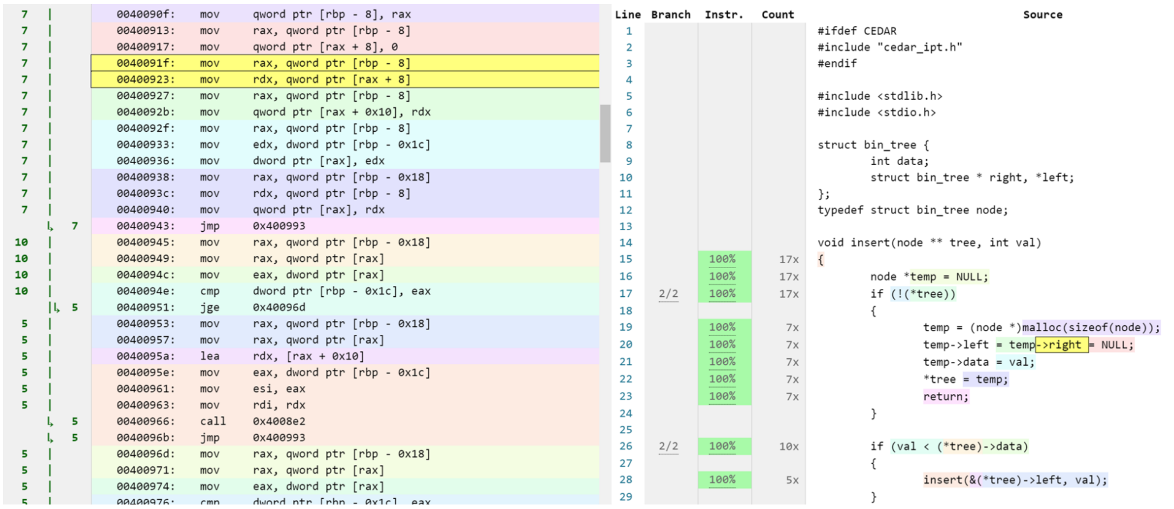
Figure 3. Trace Data Buffering and Processing

Recent novel trace processing solutions [10] feature the online analysis of processor execution trace as shown by Option 3. The large buffer memory for decoupling the offline downstream trace processing is no longer required. The challenge for this approach is that every processing step must be able to match the bandwidth of the trace source. This particularly includes the trace decompression and the control flow reconstruction. This demanding computation must often cope with the execution traces from multiple fast CPUs that are running at nominal clock speeds above 1 GHz. A particular challenge is the management of the contextual information extracted from application binaries for the interpretation of their execution traces. The online processing demands its prompt availability upon a context switch. This leaves no time for an on-demand generation. Nonetheless, the online analysis approach is the right trade-off for debugging and monitoring an embedded CPS performing a known set of tasks. It removes all limitations in terms of observation time span and latency. Making a continuous on-the-fly control flow reconstruction available enables various backend tasks to derive and analyze streams of elaborated control flow events. Possible use cases include (a) the recording of branch information for a test coverage analysis, (b) the validation of observed event sequences against a formal temporal logic specification, (c) the measurement of basic block execution times for worst case execution time (WCET) estimations or (d) the tracking of critical reaction times (WCRT).

Mapping the Object Code Observation

An important fact to keep in mind is that ET reports the program flow followed by the CPU. This traverses the control flow graph of the executed binary rather than the one of the source code known to the programmer. They are separated by a compiler. Particularly, transformations by optimizing compiler passes may complicate the correspondence between both representations. Common code is hoisted out of branches, invariants are hoisted out of loops. Loop controls may migrate from head to foot or vice versa. Even, whole control flow edges may disappear, for example, (a) by the inference of higher-level arithmetic instructions as for computing the minimum or maximum, or (b) by combining multiple branching conditions arithmetically before performing a conclusive jump. Likewise, control flow edges may be introduced as the compiler is adding code to handle corner cases correctly. Thus, extra branches are introduced regularly for handling `NaN` values in conditions performing floating-point comparisons.

Compilers can assist relating object code to corresponding source code by emitting debug information. It is typically provided in the DWARF format and conveys plenty of information including, for example, the association of source code with object code address ranges, and the local mapping of source code variables to registers and memory locations. As illustrated in **Figure 4**, this DWARF debug information enables the visualization of a traced program execution in terms of the source code. The left-hand side of the figure shows the actual trace observation annotating



```
 7  |      0040090f:  mov   qword ptr [rbp - 8], rax
 7  |      00400913:  mov   rax, qword ptr [rbp - 8]
 7  |      00400917:  mov   qword ptr [rax + 8], 0
 7  |      0040091f:  mov   rax, qword ptr [rbp - 8]
 7  |      00400923:  mov   rdx, qword ptr [rax + 8]
 7  |      00400927:  mov   rax, qword ptr [rbp - 8]
 7  |      0040092b:  mov   qword ptr [rax + 0x10], rdx
 7  |      0040092f:  mov   rax, qword ptr [rbp - 8]
 7  |      00400933:  mov   edx, dword ptr [rbp - 0x1c]
 7  |      00400936:  mov   dword ptr [rax], edx
 7  |      00400938:  mov   rax, qword ptr [rbp - 0x18]
 7  |      0040093c:  mov   rdx, qword ptr [rbp - 8]
 7  |      00400940:  mov   qword ptr [rax], rdx
    ↳   7 00400943:  jmp   0x400993
10  |      00400945:  mov   rax, qword ptr [rbp - 0x18]
10  |      00400949:  mov   rax, qword ptr [rax]
10  |      0040094c:  mov   eax, dword ptr [rax]
10  |      0040094e:  cmp   dword ptr [rbp - 0x1c], eax
    |↳  5 00400951:  jge   0x40096d
 5  |      00400953:  mov   rax, qword ptr [rbp - 0x18]
 5  |      00400957:  mov   rax, qword ptr [rax]
 5  |      0040095a:  lea   rdx, [rax + 0x10]
 5  |      0040095e:  mov   eax, dword ptr [rbp - 0x1c]
 5  |      00400961:  mov   esi, eax
 5  |      00400963:  mov   rdi, rdx
    ↳   5 00400966:  call  0x4008e2
    ↳   5 0040096b:  jmp   0x400993
 5  |      0040096d:  mov   rax, qword ptr [rbp - 0x18]
 5  |      00400971:  mov   rax, qword ptr [rax]
 5  |      00400974:  mov   eax, dword ptr [rax]
 5  |      00400976:  cmp   dword ptr [rbp - 0x1c], eax
```

```
Line Branch Instr. Count                   Source
  1                       #ifdef CEDAR
  2                       #include "cedar_ipt.h"
  3                       #endif
  4
  5                       #include <stdlib.h>
  6                       #include <stdio.h>
  7
  8                       struct bin_tree {
  9                               int data;
 10                               struct bin_tree * right, *left;
 11                       };
 12                       typedef struct bin_tree node;
 13
 14                       void insert(node ** tree, int val)
 15          100%    17x  {
 16          100%    17x          node *temp = NULL;
 17   2/2    100%    17x          if (!(*tree))
 18                               {
 19          100%     7x                  temp = (node *)malloc(sizeof(node));
 20          100%     7x                  temp->left = temp->right = NULL;
 21          100%     7x                  temp->data = val;
 22          100%     7x                  *tree = temp;
 23          100%     7x                  return;
 24                               }
 25
 26   2/2    100%    10x          if (val < (*tree)->data)
 27                               {
 28          100%     5x                  insert(&(*tree)->left, val);
 29                               }
```

Figure 4. Object Code to Source Code Mapping

each instruction with its execution count. The right-hand side transfers this information back into the realm of the programmer who can, thus, understand hot execution paths or coverage issues within structural tests.

Conclusions

With the advent of advanced embedded system technology and its use in safety-critical applications, design-time and runtime verification are becoming important concerns to enhance the dependability of these complex systems. We discuss how ET has become an enabling technology for the verification of CPSs in real time. Modern ET technology addresses the age-old problem of the intrusiveness of fine-grain program observability based on instrumentation. We illustrate the challenges in using ET, namely information processing at high data rates, and present an effective solution to address this issue. We plan to explore these features and to investigate further the utilization of ET to achieve a Dependable DevOps (Development and Operation) continuum where ET can be used during both development and operation. The DevOps approach aims to carry forward the assurance of safety and security from design time to runtime in a systematic way.

Acknowledgment

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the KMU Innovative project CoCoSI (Project 01IS19044). Other parts of this work were supported by the Virginia Commonwealth Cyber Initiative USA, an investment in the advancement of cyber R&D, innovation, and workforce development. For more information, visit www.cyberinitiative.org.

■ REFERENCES

1. A. Weiss, S. Gautham, A. V. Jayakumar, C. R. Elks, D. R. Kuhn, R. N. Kacker, and T. B. Preusser, "Understanding and fixing complex faults in embedded cyberphysical systems," *Computer*, vol. 54, no. 1, pp. 49–60, 2021.
2. A. Goodloe, "Challenges in high-assurance runtime verification," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 446–460.
3. *CoreSight™ Program Flow Trace™*, ARM, 2011.
4. *Embedded Trace Macrocell ETMv1.0 to ETMv3.5*, ARM, 2011.
5. "Intel® 64 and IA-32 architectures software developer's manual," vol. 3C, Apr 2021.
6. *The Nexus 5001 Forum - Standard for a Global Embedded Processor Debug Interface*, IEEE-ISTO, Jun 2012.
7. "IEEE standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture," *IEEE Std 1149.7-2009*, 2010.
8. A. Milenković, V. Uzelac, A. Milenković, and M. Burtscher, "Caches and predictors for real-time, unobtrusive, and cost-effective program tracing in embedded systems," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 992–1005, Jul 2011.

9. C. Hochberger and A. Weiss, “Acquiring an exhaustive, continuous and real-time trace from SoCs,” in *International IEEE Conference on Computer Design*, 2008, pp. 356–362.
10. Accemic Technologies. (2020) Cedartools® – look inside your processor. [Online]. Available: https://accemic.com/cedartools/

Thomas B. Preußer heads the research team at Accemic Technologies, 01217 Dresden, Germany. His research interests include massively parallel compute acceleration and data processing in heterogeneous FPGA-based systems as well as computer arithmetic. Preußer received his Ph.D. (Dr.-Ing.) in computer engineering from TU Dresden. He is a member of the ACM. Contact him at tpreusser@accemic.com.

Smitha Gautham is a postdoctoral scholar in the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, Virginia, 23284, USA. Her research interests include the design and assessment of heterogeneous runtime verification architectures, runtime safety and security monitors for cyber physical systems, and model-based design assurance and verification. Gautham received her Ph.D. in electrical engineering from Virginia Commonwealth University. Contact her at gauthamsm@vcu.edu.

Abhi D. Rajagopala is a postdoctoral scholar in the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, Virginia, 23284, USA. His research interests include high-performance and reconfigurable system design, CPU and hardware accelerator architectures, and hardware realization of safety-critical systems. Rajagopala received his Ph.D. in electrical engineering from the University of North Carolina. He is a member of IEEE. Contact him at rajagopalaad@vcu.edu.

Carl R. Elks is an associate professor in the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, Virginia, 23284, USA. His research interests include the analysis, design, and assessment of dependable embedded cyber-physical systems of the type found in critical infrastructure. Elks received his Ph.D. in electrical engineering from the University of Virginia in 2005. He is a member of IEEE. Contact him at crelks@vcu.edu.

Alexander Weiss is the co-founder and CEO of Accemic Technologies, 83088 Kiefersfelden, Germany. His research interests include cyber-physical systems runtime analysis, software verification, functional safety, and cyberattack detection. Weiss received his Ph.D. (Dr.-Ing.) in computer science from TU Dresden. Contact him at aweiss@accemic.com.