RESEARCH

Hardware-Assisted Instruction Profiling and Latency Detection

Suchakrapani Datt Sharma*† and Michel R Dagenais

*Correspondence: suchakrapani.sharma@polymtl.ca
Department of Computer and Software Engineering, École Polytechnique de Montréal, Édouard-Montpetit, H3T 1J4 Montréal, Québec, Canada
Full list of author information is available at the end of the article
†Equal contributor

Abstract

Analysis of time constrained and mission critical systems is a challenging task. Debugging and profiling tools can alter the execution flow or timing, can induce *heisenbugs* and are thus marginally useful. In such scenarios, tracing tools can be of crucial importance. Software tracing, however advanced it may be, depends on consuming precious computing resources. In real-time embedded systems, a hardware-tracing and debugging infrastructure can be highly beneficial. In this paper, we analyze state-of-the-art hardware tracing support, as provided in modern 6th generation Intel processors, and propose a new technique which uses the processor hardware support for tracing without any code instrumentation or tracepoints. We demonstrate the utility of our approach with contributions in three areas - syscall latency profiling, instruction profiling and software-tracer impact detection. We present improvements with our hardware assisted approach, as compared to traditional software only tracing and profiling, in terms of its performance and the granularity of data gathered with this approach. The performance impact on the target system - measured as time overhead - is on average 2-3%, with the worst case being 22%. We also define a way to measure and quantify the time resolution provided by hardware tracers for trace events, and observe the effect of fine-tuning hardware tracing for optimum utilization. As compared to other in-kernel tracers, we observed that hardware based tracing has a much reduced overhead, while achieving greater precision. Moreover, the other tracing techniques are ineffective in certain tracing scenarios.

Keywords: Performance Analysis; Profiling; Tracing; Real-Time Systems; Debugging; hardware tracing

1 Introduction

Real-time embedded systems are becoming increasingly complex to debug and diagnose. One of the main factors is the increasing complexity and real-time constraints which limit the use of traditional debugging approaches in such scenarios. Shorter task deadlines mean that the faithful reproduction of code execution can be very challenging. It has been estimated that developers spend around 50% to 75% of their time debugging applications at a considerable monetary cost [1]. In many scenarios, *heisenbugs* [2] become near impossible to detect on embedded systems. Long-running systems can have bugs that display actual consequences much later than expected, either due to tasks being scheduled out or hardware interrupts causing delays. Important parameters that need to be analyzed while doing a root cause analysis for a problem include the identification of costly instructions during execution, the detection of failures in embedded communication protocols, and the analysis of instruction profiles that give an accurate representation of which instructions consume the most CPU time. Such latent issues can only be recorded faithfully using tracing techniques. Along with accurate profiling, tracing provides a much needed respite to developers for performance analysis in such scenarios.

We focus in this work on two important common issues in real-time systems: the efficient detection/tracking of hardware latency and the accurate profiling of syscalls and instructions, with an ability to detect program control flow more accurately than with current software approaches. We discuss our new analysis approach, which utilizes conditional hardware tracing in conjunction with traditional software tracing to accurately profile latency causes. The trace can be decoded offline to retrieve the accurate control flow, data even at instruction granularity, without any external influence on the control flow. As software tracing can induce changes in the control flow, with our system we can further detect the cause of latency induced by the software tracers themselves, on the software under observation, to nanosecond range accuracy. We observed that our technique can gather more information with low overhead (1%) for synthetic workloads, as compared to pure software based in-kernel tracing approaches (63%), for observing time-stamped control-flow.

Pure software profiling and tracing tools consume the already constrained and much needed resources on embedded devices. Over the years, hardware tracing has emerged as a powerful technique for tracing, as it gives a detailed and accurate view of the system with almost zero overhead. As with most technologies, formalizing and standardizing is still an ongoing process in this domain. However, the IEEE Nexus 5001 standard [3] defines 4 classes of tracing and debugging approaches for embedded systems. Class 1 deals with basic debugging operations such as setting breakpoints, stepping instructions and analysing registers - often directly on target devices connected to hosts through a JTAG port. In addition to this, Class 2 supports capturing and transporting program control-flow traces externally to host devices. Class 3 adds data-flow traces support, in addition to control-flow tracing, and Class 4 allows emulated memory and I/O access through external ports. Hardware tracing modules for recent microprocessors (Class 2-Class 4) can either utilize (1) on-chip buffers for tracing, recording trace data from individual CPUs on the SoC, and send it for internal processing or storage, or (2) off-chip trace buffers that allow trace data to flow from on-chip internal buffers to external devices, with specialized industry standard JTAG ports, and to development host machines, through high performance hardware trace probes [4, 5]. These hardware trace probes contain dedicated trace buffers (as large as 4 GB) and can handle high speed trace data. As we observed in our performance tests (section 4), the former approach can incur overhead in the range of 0.83% to 22.9%, mainly due to strain on memory accesses. We noted that trace streams can generate data in the range of hundreds to thousands of MB/sec (depending on trace filters, trace packets and packet generation frequency). Thus, there is a tradeoff in choosing either an external analysis device or on-chip buffer recording. The former gives a better control, (less dependency on external hardware which is curcial for on-site debugging), but incurs a small overhead for the memory subsystems on the target device. The later provides a very low overhead system but requires external devices and special software (often proprietary) on development hosts. The generated trace data is compressed for later decoding with specialized debug/trace tools [6, 7] that run on host machines, as illustrated in Figure 1.

Device memory is limited, thus there are multiple ways to save tracing data using either of the two approaches discussed above. Therefore, to achieve maximum performance, recent research deals with compressing the trace output during the decoding phase to save transfer bandwidth. [1]. Earlier, part of the focus was on the unification of the traces, which is beneficial for Class 3 devices [8]. This provides a very detailed picture of the execution at almost no overhead on the target system.

In this paper, we mainly focus on on-chip local recorded traces, pertaining to Class 2 devices, owing to their low external hardware dependency and high availability in commonly used architectures such as Intel x86-64. With our proposed approach, using hardware-trace assistance, we were able to trace and profile short sections of code. This would ensure that precious I/O bandwidth is saved while maintaining sufficient granularity. We used Intel's new Processor Trace (PT) features and were able to gather accurate instruction profiling data such as syscall latency and instruction counts for interesting events such as abnormal latency. In profile mode, we can also selectively isolate sections of code in the operating system, for instance idling the CPU, taking spinlocks or executing FPU bound instructions, to further fine-tune the systems under study.

The remainder of the paper is organized as follows. Section 2 gives a general overview of program-flow tracing, its requirements, and limitations of the current sampling systems to handle these. It also introduces the concept of hardware tracing to overcome such limitations. We discuss state-of-the-art techniques used in software tracing and finally concentrate on our research scope. In section 3, we introduce our hardware and hardware-assisted software tracing based architecture. We then explain how we used our approach to isolate and find syscall latencies caused by software tracers in systems. We also explain how time and instruction profiling at instruction level can be achieved, based on our new algorithm implemented in the trace decoding phase. In section 4, we start with a detailed experiment, measuring overhead and trace size, for Intel's Processor Trace hardware tracing infrastructure. We have also proposed a new metric to define the granularity of the temporal and spatial resolution in a trace. We then show how the hardware based trace profiler can help visualize anomalies through histograms.

2 Background

In an ideal environment, developers would want to know as much as possible about the flow of their programs. There are three important artifacts that can be gathered during a program execution - flow of instructions during program, their classification, and deduction of the program flow with timing information. Static analysis of binaries, to understand how the program runs, allows the developers to visually analyze how the compiler generates instructions, estimate how the instructions may execute, and can be used further for code coverage [9, 10]. Such information is also vital for debuggers to generate and aid in the breakpoint debugging approach. Recently, the focus on pure static code analysis tools has been mostly in the security domain, for analysing malicious or injected code in binaries [11, 12] or optimizing compilers based on the analysis of generated code [13]. However, the actual execution profiles can differ from what static tools can anticipate, due to the complexities of newer embedded computer architectures in terms of pipelines, instruction prefetching, branch prediction and unforeseen runtime behaviour such as hardware interrups. Therefore, to understand the effect of individual instructions or function blocks, the instructions executed can be profiled at runtime. The use of counting instructions for blocks of code, at program execution time, has been proposed and explored in-depth before [14]. Therefore, the instruction sequence and profile can be recorded and then replayed later on. However, some of these earlier approaches dealt with inserting instrumentation code, to profile instructions and other interesting events. Sampling based techniques, developed earlier such as DCPI [15, 16], have also been discussed extensively before, where authors demonstrated the use of hardware counters provided by the processor for profiling instructions. Merten et al. [17] have earlier proposed the use of a Branch Trace Buffer (BTB) and their hardware table extension for profiling branches. Custom hardware-based path profiling has been discussed by Vaswani et al. [18], where they observe that low overhead with hardware-assisted path profiling can be achieved. Recent advances, especially in the Linux kernel, discuss how profiling tools like Perf can be used to generate execution profiles, based on data collected from special hardware counters, hardware blocks that record branches or pure software controlled sampling [19, 20].

2.1 Program Flow Tracing

Recording instruction flow or branches in a program can provide information about how a program actually executes in comparison to how the expected execution. The comparison of an anomalous program flow trace with that of a previous one can let the developer know what was the effect of changes on the system. It can also be used to track regressions during new feature additions. At lower levels, such as instructions flow, bugs that occur in long running real-time systems can now be detected with more confidence, as the complete execution details are available. With recent hardware support from modern processors, this has become easier than ever. We discuss details about such hardware support further in section 2.2. Larus et al discussed quite early about using code instrumentation to inject tracing code in function blocks, or control-flow edges to track instructions or deduce the frequency of their execution [21, 14]. They observed overhead of 0.2% to 5%, without taking into consideration the effect of the extra overhead of disk writes (which they observed as 24-57% in those days). Other more powerful tools, which effectively perform execution profiling or control-flow tracing, can be built using similar binary modifying frameworks such as Valgrind [22]. Even though this framework is more data-flow tracing oriented [23], some very insightful control-flow tools have been developed, such as Callgrind and Kcachegrind [24]. Program-flow tracing can either encompass a very low level all-instruction trace generation scheme, or a more lightweight branch-only control-flow trace scheme.

Instruction Tracing Tracing each and every instruction to deduce the program flow can be quite expensive if instrumentation is required. Hence, architectures such as ARM and PowerPC provide hardware support for such mechanisms in the form of

NSTrace (PowerPC), EmbeddedICE, Embedded Trace Macrocell (ETM), Program Trace Macrocell (PTM) (now part of ARM CoreSight) and MIPS PDTrace [25, 26]. The basic idea is to snoop the bus activity at a very low-level, record such data and then reconstruct the flow offline from the bus data. External hardware is usually connected as bus data *sink* and special software can then use architecture level simulators to decode the data. The benefit of a complete instruction flow trace is that there is highly detailed information about each and every instruction for accurate profiles, in-depth view of memory access patterns and, with the support of time-stamped data, a very accurate overall tracer as well. However, the amount of data generated is too high if external devices are not used to sink the data. Indeed, memory buses are usually kept busy with their normal load, and an attempt to store the tracing data locally incurs bus saturation and additional overhead. An approach to reduce such bandwidth and yet keep at least the program-flow information correct is to use branch only traces.

Branch Tracing The issue of memory related overhead for hardware program/data flow traces has been observed earlier as well [21, 14]. Even though hardware can generate per-instruction trace data at zero execution overhead, such an additional data flow may impact the memory subsystem. Hence, just choosing the instructions that cause a program to change its flow greatly reduces the impact. Such control-flow instructions (like direct/indirect jumps, calls, exceptions etc) can indeed be enough to reconstruct the program flow. Dedicated hardware blocks in the Intel architecture, such as Last Branch Record (LBR), Branch Trace Store (BTS) [27], and more recently Intel Processor Trace (PT) choose to only record branches in the currently executing code on the CPU cores. By following the branches, it is quite easy to generate the instruction flow with the help of additional offline binary disassembly. For example, for each branch instruction encountered in the flow, a record for the branch taken/not-taken and its target can be recorded externally. This is then matched with the debug information from the binary to reconstruct how the program was flowing. We detail and discuss the branch tracing approach, as well as instruction tracing, in section 2.2, where we show a state-of-the-art branch tracing approach using Intel PT as an example.

2.2 Hardware Tracing

As discussed previously, the complete instruction and branch tracing is suported by dedicated hardware in modern multi-core processors. However, hardware tracing has deep roots in the embedded systems world. It is expensive in terms of the high I/O bandwidth required by the tracing hardware involved. Faced with limited resources, from early on, embedded developers have tried to extend their current hardware capabilities by resorting to *hardware tricks* such as utilizing external hardware trace buffers and recording execution based on"action codes" from the trace buffers. A similar rudimentary approach has been discussed before by Ball [28] where an Intel 80188 based hardware is connected to an external logic analyzer, through a 74ACT138 decoder, in such a configuration that data written to unused memory or I/O locations triggers the logic analyzer to store the "action codes" that can record and replay the execution flow.

These codes can be certain points in the program, effectively creating an external trace buffer. This, in essence, formulates a preliminary approach of using external devices to perform hardware tracing. Similar techniques are still used in modern embedded processors, which provide external hardware recorders and tracing devices access to the processor data and address buses. ARM's early implementation of EmbeddedICE (In-circuit Emulator) was an example of this approach. Eventually, processor chip vendors formally introduced dedicated and more advanced hardware tracing modules such as CoreSight, Intel BTS and Intel PT. In a typical setup, such as shown in Figure 1, trace data generated from the trace hardware on the chip can be funneled to either the internal buffer for storage, or observed externally through an external hardware buffer/interface to the host development environment, for more visibility. In both the cases, the underlying techniques are same but performance varies according to the need of the user and the hardware implementation itself.

2.2.1 Tracing Primitives

Since an important part of our research deals with program flow tracing, we discuss how hardware tracing blocks can be used to implement it. The basic idea is to record the control-flow instructions along with some timing information (if needed) during the execution of the program. Different architectures have different approaches for deciding on the optimum buffer size, trace compression techniques, and additional meta-data such as timing information, target and source instruction pointers etc. We explain such techniques along with an overview of the tracing process in this section. A program flow trace can be broadly broken down into three individual elements.

Trace Configuration Most of the hardware trace modules can be fine-tuned by writing data to certain control registers, such as Model Specific Registers (MSR) in Intel or the Coresight ETM/ETB configuration registers for ARM. For example, writing specific bits in MSRs can control how big a trace buffer will be or how fine-grained or accurate the timing data will be generated. An optimum configuration leads to better trace output - the effect of which is discussed in this paper.

Trace Packets A hardware trace enabled execution generates all the hardware trace data in a compressed form for eventual decoding. This can consist of different distinguishable elements called trace packets. For example, in the context of Intel PT, these hardware trace packets can contain information such as paging (changed CR3 value), time stamps, core-to-bus clock ratio, taken-not-taken (tracking conditional branch directions), record target IP of branch, exceptions, interrupts, source IP for asynchronous events (exceptions, interrupts). The amount or type of packets enabled and their frequency of occurrence directly affect the trace size. In the control-flow trace context, the most important packets that are typically common to different architectural specifications of trace hardware are:

- **Taken-Not-Taken**: For each conditional direct branch instruction encountered (such as jump on zero, jump on equal), the trace hardware can decode if that specific branch was taken or not. This is illustrated with Intel PT's trace output as an example in figure 2. We can observe that Intel PT efficiently utilizes 1 bit per branch instruction to encode it as a taken or not taken branch.The earlier implementations such as Intel BTS used 24 bits per branch, which caused an overhead between 20% to 100% as the CPU enters the special debug mode, causing a 20 to 30 times slowdown [29, 30].

- **Target IP**: Indirect unconditional branches (such as register indirect jumps) depend on register or memory contents, they require more bits to encode the destination (Instruction Pointer) IP of the indirect branch. This can also be the case when the processor encounters interrupts, exceptions or far branches. Some implementations such as Intel PT provide other packets for updating control flow, such as Flow Update Packet (FUP), which provide source IP for asynchronous events such as interrupts and exceptions. In other scenarios, the binary analysis can usually be used to deduce the source IP.
- **Timing**: Apart from deducing the program-flow, packets can be timed as well. However, time-stamping each and every instruction can be expensive in terms of trace size as well as extra overhead incurred. ARM CoreSight provides support for accurate time-stamp per instruction. However, the usecase is mainly aimed at usage of an external high speed buffer and interface hardware through a JTAG port. For on-device tracing such as Intel PT, the packet size can be kept small by controlling the frequency of time-stamps being generated and the type of time-stamps. For example, a timing packet can either be the lower 7 bytes of the TSC value as an infrequently recorded TSC packet or can be just a more frequent 8-bit Mini Timestamp Counter (MTC) packet occurring in between two TSC packets. MTC packets record incremental updates of CoreCrystalClockValue and can be used to increase the timing precision with fewer bits utilized. Trace timing in some implementations can further be improved by a cycle accurate mode, in which the hardware keeps a record of cycle counts between normal packets.

In the next section we discuss how we can leverage hardware tracing techniques and utilize it for efficient and more accurate profiling and tracing.

3 Trace Methodology

In order to get useful instruction profiling and tracing data for use-cases such as accurate detection of interrupt latency, we propose a framework that utilizes an hardware-assisted software tracing approach. The major focus of our work is on post-mortem analysis of embedded systems. Hence, the underlying technologies used aim at recording raw trace or program-flow data at runtime, and eventually perform an offline merge and analysis, to get in depth information about abnormal latency causes, or generate instruction execution profiles. The data generated in hardware tracing can reach a range of hundreds of MB per second. Various approaches have been taken to reduce this overhead. Apart from careful configuration of the trace hardware, various methods such as varying the length of TNT packets (short/long), IP compression, indirect transfer return compression [31] are employed to control precisely the trace size, with he aim of reducing memory bus bandwidth usage. Previous work often focused on trace compression and even better development of tracing blocks itself [1]. In contrast, we chose to leverage the latest state-of-the-art hardware such as Intel PT and carefully isolate interesting sections of the executed code to generate short hardware traces. These short traces can be eventually tied to the corresponding software trace data to generate a more in-depth view of the system at low cost. This can also be used to generate instruction execution profiles in those code sections for detecting and pinpointing anomalous sections of the program, right down to the executed instruction. This gives a unique and better approach as compared to other techniques of sample based profiling (such as Perf) or simulation/translation based profiling (such as Valgrind) mainly due to the fact that there is no information loss, as the inferences are based on the real instruction flow in a program, and the overhead of simulated program execution is completely removed. Choosing only specific sections of code, to trace and profile with hardware, also means that we do not require external trace hardware and can rely on internal trace buffers for post-mortem analysis. In this section, we first show the design of our framework itself and demonstrate how we can use Intel PT hardware-assisted software tracing. We also explain our three main contributions, detailing how we could profile interrupts/syscall latency and evaluate the impact of software tracers themselves. We start with some background on Intel PT, and then explain the architecture of our technique.

3.1 Intel PT

Intel's MSR based Last Branch Record (LBR) and the BTS approach for branch tracing have been widely explored before [29, 30]. Eventually, the popularity of Intel in the embedded domain and the benefits of the hardware tracing approach led to more advancements in the branch-tracing framework in the form of Intel PT. Branch trace data with PT can now be efficiently encoded and eventually decoded offline. The basic idea, as shown in Figure 2, is to save the branching information during program execution, encode and save it. Later on, the trace data along with run-time information such a process maps, debug-info and binary disassembly, we can fill in the gaps between the branches and form a complete execution flow of the application. During the decoding of the compressed recorded branch data, whenever a conditional or indirect branch is encountered, the recorded trace is browsed through to find the branch target. This can be merged with debug symbols and static analysis of the binary code to get the intermediary instructions executed between the branches. Therefore, with Intel PT's approach, we do not need to exclusively store each and every instruction executed but just track branches - allowing on-device debugging and less complex implementation of hardware and debugging software. Apart from that, with the simple MSR based configuration of the hardware, we have the ability to set hardware trace start and stop filters based on an IP range to allow a more concise and efficient record of trace at runtime. To illustrate how PT works, listing 1 shows a raw PT trace recorded for a short section of some hypothetical execution.

Listing 1 Harware trace packets generated by PT during a typical tracing session. The highlighted packets are timing related

```
PSB                         }
PAD                         }
TSC 0x45e897b4d39ba         }
PAD                         }     (i)
TMA CTC 0x6457 FC 0x1c      }
PAD                         }
CBR 0x27                    }
PSBEND                      }
MTC 0x8b
MTC 0x8c
...
TNT TTTTNN (6)              ⟶         (ii)
TNT T (1)
TIP 0x9c6d3ce0              ⟶         (iii)
TNT TTN (3)
TIP 0x3cf0
TNT T (1)
MTC 0x2e
TNT NNTNTN (6)
TNT NTNTNN (6)
TNT NTNNTT (6)
```

The trace prologue *(i)* starts with a PSB synchronization packet followed by timings packets such as time stamp counter value (TSC), Timing alignment packet (TMA) and Core-to-bus ratio (CBR) packet. It ends with the PSBEND packet. Subsequent timing is maintained by configuration dependent mini-timestamp counter (MTC) packets and TSC packets to keep track of time. Conditional branches can be identified by TNT packets *(ii)* with subsequent branches taken encoded as a T (1) and not-taken as N (0). The indirect branches are indicated by Target IP (TIP) packets *(iii)* which indicate that control flow always changed. With analysis of the binary, the PT trace decoder in userspace can generate a program flow graph by associating the TNT and TIP packets with the disassembeld binary addresses and instructions between the two control-flow instructions. The decoder has been open sourced by Intel for a rapid adoption in other tools such as Perf [32]. We incorporated PT in our framework, owing to its low overhead and versatility as presented later in section 4.2 where we discuss its performance and overhead.

3.2 Architecture

For the hardware part of our design, we developed a framework based on the PT library, for decoding the hardware trace, and a reference PT driver implementation provided by Intel to enable, disable and fine tune trace generation [33, 34]. An overview of our hardware-assisted trace/profile system is shown in figure 3. The Control Block is a collection of scripts to control the PT hardware in CPUs through the simple-pt module. The control scripts can be used for configuring the PT hardware for filtering, time-stamp resolution and frequency control. It can enable/disable traces manually for a given time period. We use the control scripts for generating instruction and latency profiling data. Any PT data is generated and sent to a ringbuffer, the content of which is dumped to the disk for offline analysis. Along with the trace data, runtime information such process maps, CPU information and features are saved as *sideband* data. This is essential for trace reconstruction. The Trace Decoder uses the Intel's processor trace decoder library to reconstruct the control flow based on runtime information, debug information from binary, and per-CPU PT data. This data is converted to our intermediate format that is consumed by the Instruction Profiler and Latency Profiler modules that can generate visualizations.

We now elaborate on our three contributions that cover instruction/syscall latency profiling and the performance impact of software tracing systems on embedded and real-time production systems.

3.3 Delta Profiling

As seen in figure 3, the raw PT data from the processor can be reconstructed to generate a program control flow. It is, however, important to analyze this huge information in a meaningful manner. Therefore, we present an algorithm to sieve through the data and generate instruction execution profiles based on patterns of occurrence of instructions in the control flow. These profiles can be used to represent the histograms based on a time-delta or an instruction count delta. This can work for single instructions to observe histograms during a simple execution, as well by just counting the occurrence of a set of instructions. This approach is significantly different from sample based profiling, as it is based on true instruction flow and can pinpoint errors at finer granularity in short executions. As interrupts are quite significant in real-time embedded systems, we choose to profile instructions that are responsible for disabling and enabling interrupts in the Linux kernel. Thus, we generate two histograms that represent time-delta and instruction count delta of intervals between interrupt enabling and disabling instructions. We observed that interrupt disabling and enabling in Linux on a x86 machine is not just dependent on two instructions, `sti` and `cli` respectively, but also on a pattern of instructions that use pushf and `popf` to push and pop the entire EFLAGS register, thus clearing or setting the interrupt flag in the process. Thus, to effectively profile the interrupt cycle, we identified the instruction patterns during decoding and grouped and identified them as `superSTI` and `superCLI` instructions. For incoming coded PT streams, we devised an algorithm shown in listing Algorithm 1 that is able to generate these profiles. For example, when we apply this during the trace decode time, we can obtain the time taken between two consecutive interrupts enable and disable in the execution, or the number of instructions executed between them. These target super-instructions pairs (S_t), which are actually a pseudo-marker in the instruction stream based on pattern matching, can be given as input to the profiler. Based on the mode, it can either begin instruction counting or timestamp generation and stores it in a database. We then iterate over the database and generate the required visualizations.

We can extend this technique of identifying patterns in the code to record more interesting scenarios. For example, in the Linux kernel, CPU idling through the `cpu_relax()` function generates a series of repeating `nop` instructions. Similarly, the *crypto* subsystem in the Linux kernel aggressively uses the less-recommended FPU. We were able to successfully identify such patterns in the system based purely on PT, without any active software tracer.

Algorithm 1: Hardware Delta Profiling

```
Data: ① {D : $t_p ... t_q$}, where D is the coded stream for time $T_{q-p}$ ② Target Instruction Set I,
      which is either individual instruction pair $I_t$ or super-instruction pair $S_t$ and ③ Profile
      Mode: $T\Delta$, $IC\Delta$
Result: Time-Δ histogram, Instruction Count-Δ histogram, Instruction Count histogram of I
begin
    $iC \leftarrow 0$
    $resetFlags()$
    for $i \in D$ do
        $x = decodeInstruction(i)$
        $incrementCount()$
        if $x = (I_t$ or $S_t)$ and $Mode = T\Delta$ then
            if $set(F)$ then
                $unset(F)$
                $ts_{n+m} \longleftarrow t_x$
                $\Delta t = ts_{n+m} - ts_n$ $addToDatabase(DB, \Delta t)$
            else
                $ts_n \longleftarrow t_x$
                $set(F)$
        if $x = (I_t$ or $S_t)$ and $Mode = IC\Delta$ then
            if $set(F)$ then
                $unset(F)$
                $IC\Delta = getDelta()$
                $resetCounter(iC)$
                $addToDatabase(DB, IC\Delta)$
            else
                $startCounter(iC)$
                $set(F)$
        $count(iC)$
    $generateHistogram(DB)$
```

We present an implementation of the algorithm in section 4.3 where we elaborate more on our delta profiling experiment.

3.4 Syscall Latency Profiling

Syscalls affect the time accuracy of systems, especially in critical sections, as they form a major chunk of code executed from userspace. For example, filesystem syscalls such as `read()`, `open()`, `close()` etc. constitute 28.75% of code in critical sections of Firefox. [35]. Profiling syscall counts for a given execution is easy and can be performed with simple profilers such as Perf, or even through static or dynamic code analysis techniques based on `ptrace()`. However, to understand, the extra time incurred in the syscalls, we can get help from software tracers. As the software tracers themselves affect the execution of syscalls, an accurate understanding can only be achieved by an *external observer* which does not affect the execution flow. In such scenarios, hardware tracing is a perfect candidate for such an observer. We used PT and devised a way to visualize syscall stacks in our proposed technique, after decoding, to compare them between multiple executions. This gave us a deep and accurate understanding of any extra time incurred in syscalls, right down to individual instructions. As an example, Figure 4(a) illustrates the effect of an external tracer (LTTng) on the `mmap()` syscall with the help of a callstack. The callstack shown is for a short section of code from the library mmap call to the start of the syscall in the kernel. We see in the figure that the highlighted call-path reached the `lttng_event_write()` function from the second ring of the `entry_SYSCALL_64()` function in the kernel. The layers represent calls in a callstack, with the call depth going from the innermost to the outermost layers. The path to the `lttng_event_write()` function took 9.3% of all instructions in the recorded callstack. As the code sections are short, and the data represented is hierarchical in nature, it is easy to visualize them on sunburst call-graphs [36, 37] for a clear visual comparison.

We can observe such a short callstack from another execution, in Figure 4(b), where LTTng tracing is disabled, and we notice the absence of extra calls which were added as layers and peaks in Figure 4(a). The metrics in the sunburst graphs are calculated based on the number of instructions executed along a particular call path. The visualization is interactive and its implementation is based on the D3 javascript library.

3.5 Software Tracer Impact

With the PT profile, we can observe how much extra time and instructions any external software tracer added to the normal execution of the syscall. This can also be used as a basis for analyzing the overhead of known tracers on the test system itself. For example, we observed that the extra time taken in the syscall is due to the different paths the syscall has when tracing is enabled, as compared to when tracing is disabled. A lot of code in kernel is untraceable such as C macros and blacklisted functions built with the `__attribute__((no_instrument_function))` attribute that don't allow tracers to trace them. This is usually a mandatory precaution taken in the kernel to avoid tracers going in sections of code that would cause deadlocks. However, PT allowed us to get much finer details than other pure software tracers, as it acts as an external observer and can even record calls to those functions. We monitored how LTTng changes the flow of 7 consecutive syscalls in a short section of traced code. As an example, for `mmap()` calls, we observed that with software tracing and recording enabled, a total of 917 additional instructions were added to the normal flow of the syscall, which took an extra 173 ns. For short tracing sections, an overhead of 579 ns was observed on average for `open()` syscalls, with 1366 extra instructions. We observed that the overhead also varies according to the trace payload for syscalls, as LTTng specific functions in the kernel modules copy and commit the data to trace events. Therefore, with the hardware-trace assisted tool, we can get instruction and time accurate overhead of the tracer's impact on the system itself. Such detailed information about a software tracer's impact is not possible to obtain by conventional software tracers themselves. PT based hardware tracing allows to study the flow through those unreachable sections of code (assembly, non-traceable functions in the kernel) along with a higher granularity. In section 4.2 we further compare the overhead of Linux kernel's Ftrace tracer with PT.

4 Experimentation and Results

4.1 Test Setup

The test machine has an Intel Skylake i5-6600K processor which supports Processor Trace and runs a patched Linux Kernel version 4.4-rc4 on a Fedora 23 operating system. To get minimum jitter in our tests, we disabled CPU auto-scaling and fixed the operating frequency to 3.9GHz. The system has 16 GB main memory and a 500 MB solid state drive.

4.2 PT Performance Analysis

The most important requirement for a performance analysis framework is that it should have minimum impact on the test system itself. Therefore, before deciding on the trace hardware for our framework, we tested PT's performance. The impact of PT on the system has not been thoroughly characterized before. Therefore, as a first part of our contribution, we developed a series of benchmarks to test how much overhead the tracing activity itself causes. We measured four aspects - the execution overhead in terms of extra time, the trace bandwidth, and the trace size and temporal resolution with varying time accuracy. We also compared PT's overhead with that of current default software tracers in Linux kernel.

4.2.1 Execution Overhead

Similar to such synthetic tests done for measuring the Julia Language performance [38] against C and Javascript, we added more indirect branch intensive tests, such as TailFact which causes multiple tail-calls for factorial computation and Fibonacci to illustrate conditional branches. We also tested an un-optimized and optimized Canny edge detector to check the effect of jump optimizations in image processing tasks.

Listing 2 Code for Omega test to test TNT overhead

```
mov $42, %r8
mov $42, %r9
cmp %r8, %r9
je <nop> ; TNT packet
```

As conditional branches constitute most of the branch instructions, to get a precise measurement for a conditional branch, we tested a million runs of an empty loop (Epsilon) against a loop containing an un-optimized conditional branch (Omega), as shown in listing 2. Therefore, the TailFact test gives us the upper limit of overhead for indirect branch instructions while the Omega test gives us the upper limit for conditional branches.

Table 1 Execution overhead and trace bandwidth of Intel PT under various workloads. The TailFact and Omega tests define the two upper limits

Benchmark	*Bandwidth (MBps)*	*Overhead*	
		C (%)	V8 (%)
TailFact	2200	**22.91**	-
ParseInt	1420	9.65	10.36
Fib	1315	5.86	5.80
RandMatStat	340	2.58	**20.00**
CannyNoOptimize	303	2.55	-
PiSum	339	2.47	6.20
CannyOptimize	294	2.34	-
Sort	497	1.05	6.06
RandMatMul	186	**0.83**	**11.08**
Omega	205	**11.78 (8.68)**	-
Epsilon	217	3.10 (0.0)	-

Observations Our test results have been summarized in table 1. We can see that excessive TIP packets generated due to tail-calls from the recursive factorial algorithm cause the maximum overhead of 22.9%, while the optimized random matrix multiplication (RandMatMul) overhead is 0.83%. The optimization in the C version of RandMatMul is evident as it aggressively used vector instructions (Intel AVX and SSE) from the BLAS library [39] during a DGEMM, thus generating very few TIP or TNT packets, as compared to the unoptimized loop based multiplication in Javascript, which through the V8's JIT got translated to conditional branches. This explains the difference, as seen in the table, where the Overhead for V8 is 11.08%. Same is the case with RandMatStat, which also generated more TIP packets thus pushing the overhead to 20%. In order to observe the TNT packet overhead, the Omega test generated pure TNT packets. As this includes one million extra conditional jump overhead from the test loop for both Epsilon and Omega, we can normalize the overhead and observe it to be 8.68%.

4.2.2 Trace Bandwidth

The direct co-relation between the trace size, packet frequency and hence the bandwidth of trace is quite evident. We record the trace data generated per time unit and quantify the trace bandwidth for our micro-benchmarks in table 1.

Observations We see that the trace bandwidth is quite high for workoads with high frequency TIP packets. Larger TIP packets increase the bandwidth and cause a considerable load on the memory bus. Overall, for moderate workloads, the median bandwidth lies between 200-400 MBps.

4.2.3 Trace Size

Apart from the inherent character of the applications (more or less branches) that affect the trace size and timing overhead, Intel PT provides various other mechanisms to fine tune the trace size. The basic premise is that the generation and frequency of other packets such as timing and cycle count information can be configured before tracing begins. To test the effect of varying such helper packets, we ran the PiSum micro-benchmark from our overhead experiments, which mimics a more common workload with userspace-only hardware trace mode. We first started with varying the generation of synchronization packets called PSB, while the cycle accurate mode (CYC packets) and the MTC packets were disabled. The PSB frequency can be controlled by varying how many bytes are to be generated between subsequent packets. Thus, for a higher number of bytes, those packets will be less frequent. As PSB packets are accompanied with a TSC packet, this also means that the granularity of timing varies. The same was repeated with MTC packets while the PSB packet generation was kept constant and the CYC mode was disabled. Similar tests were done with CYC packets, where PSB packet generation was kept constant and MTC packet generation was disabled. Figures 5, 6 and 7 show the effect of varying the frequency of packets on the generated trace data size.

Observations We observe in Figures 5, 6 and 7 that, as expected, when the time period (indicated by number of cycles or number of bytes in-between) for CYC, MTC or PSB packets is increased, the trace size decreases. However, it moves towards saturation, as opposed to a linear decrease. The reason we observed is that, for a given trace trace duration, there is a fixed amount of packets that are always generated from the branches in the test application. This sets a minimum threshold. Furthermore, in Figures 5, the trace size did not increase further for time periods $< 2^6$ cycles, because the maximum number of CYC packets that can be generated for our synthetic workload was reached at 2^6 cycles. The trace data size can however further increase for other workloads with higher frequency of CYC packets, when kernel tracing is enabled. In our tests, we found that the lower and upper bounds of trace data size, based on lowest and highest possible frequencies of all packets combined, are respectively 819 KB and 3829 KB.

4.2.4 Temporal Resolution

In addition to the effect of different packet frequencies on the trace data size, it is also important to observe how much timing granularity we loose or gain. This can help the user decide what would be the trade-off between size, timing granularity of the trace and the tracing overhead, to better judge the tracer's impact on the target software. We therefore define a Temporal Resolution Factor (TRF). For a given known section of code, with equidistant branches,

$$TRF = \frac{N_f}{max(P) - min(P)} \times (p - min(P))$$

where,

$$p = \left(\frac{\Delta I_c Sort[n-1]}{2}\right) \times \left(\frac{\Delta T Sort[n-1]}{2}\right)$$

and P is the set of all p. Here, ΔT and ΔI_c are the time and instruction delta between subsequent decoded branches and n is the length of the total decoded branches. The formula calculates the median of the sorted datasets of ΔT and ΔI_c and normalizes them with a factor of N_f for an accurate representation. The value of n, however, varies according to the frequency of packets. Hence, with the packets we estimate the averages based on the maximum we can obtain. This experiment was coupled with the trace size experiment above so that, for the same executions, we could observe our temporal resolution as a function of the trace size as well. The results are presented in Figures 5, 6 and 7 with TRF on the second Y axis. TRF varies between 0-100. Lower TRF values represent better resolutions.

Observations It is interesting to see a clear trend that the data size is inversely proportional to TRF. Hence, the larger the trace size, the better is the temporal resolution in the trace. Another important observation is the sudden increase in resolution when CYC packets are introduced. We observed that the highest resolution we obtained for our tests was 14 ns and 78 instructions between two consecutive events in the trace ($TRF = 4.0 \times 10^{-9}$). This compares to the lowest resolution of 910.5 μs and 2.5 million instructions ($TRF = 100$) between two events with no CYC and far apart PSB packets. The reason for such a huge variation is that the introduction of the cycle accurate mode generates CYC packets before all CYC eligible packets (such as TNT, TIP). The CYC packets contain the number of core clock cycles since the last CYC packet received, which is further used to improve the time resolution. With a large number of CYC-eligible packets being generated in quick succession, the trace size as well as the resolution increases drastically, as compared to MTC and PSB packets. For CYC observations in Figure 5, the resolution for $< 2^6$ cycles is not shown, as it covers the whole execution in our workload for which we are interested. Thus, we always get a constant maximal number of CYC packets and the TRF value saturates. Therefore, we only included valid cycles $> 2^6$. This can however vary for real life usecases such as kernel tracing where the branches are not equally spaced and the code section is not linear. However, the TRF values obtained can be a sufficient indicator of upper and lower bounds of resolution.

Table 2 Comparison of Intel PT and Ftrace overheads for synthetic loads

Benchmark	*Baseline*	*With PT*	*With Ftrace*	*Overhead*	
				PT (%)	Ftrace (%)
File I/O (MBps)	32.32	31.99	19.76	1.00	**63.56**
Memory (MBps)	5358.25	5355.59	4698.52	0.00	14.04
CPU (s)	19.001	19.007	19.121	0.00	0.6

4.2.5 Ftrace and PT Analysis

We also compared the hardware control flow tracing with the closest current software solutions in the Linux kernel. An obvious contender in kernel control-flow tracing for our Intel PT based tool is Ftrace [40]. Both can be used to get the execution flow of the kernel for a given workload - with our approach providing a much more detailed view than Ftrace. We therefore used the Sysbench synthetic benchmarks to gauge the overhead of both approaches for disk I/O and memory and CPU intensive tests. We configured Ftrace in a mode as close as possible to Intel PT by outputting raw binary information to a trace buffer. We also set the per-CPU trace buffer size to 4MB. Our findings are presented in table 2. We can see that, as compared to PT, FTrace was 63% slower for a Sysbench File I/O workload with random reads and writes. This generates numerous kernel events for which Ftrace had to obtain time-stamps at each function entry. In the PT case, the time-stamps are an inherent part of the trace data generated in parallel through trace hardware. This explains the huge difference in overhead. A similar difference is observed in the memory benchmark as well. In the case of the CPU benchmark, the work was userspace bound and hence the trace generated was smaller - thus a non-statistically significant overhead in PT and 0.6% overhead in Ftrace.

4.3 Delta Profiling Instructions

In these sets of experiments we show how our Algorithm 1 is able to generate histograms for instruction and time delta for `superSTI` and `superCLI` instructions groups. In order to quantify how much time is spent in a short section where interrupts are disabled, we created a synthetic test module in the kernel that disables and enables interrupts as our input. We control the module using `ioctl()` to cycle the interrupts. We pin the userspace conterpart of our test module on CPU0 and analyze the hardware trace for CPU0. Our algorithm is implemented during the trace decoding phase to generate the instruction delta and time delta in an intermediate format which we then plot as histograms. This helps pinpoint how many instructions were executed in the interval between two consecutive interrupt disable and enable. Looking at figure 8, we can see that most of the interrupts disabled intervals executed around 90-100 instructions. For the same execution, we observe in figure 9 that most of the interrupts disabled intervals have a duration in the range 40-80 ns. We can then look for the interrupts disabled intervals of abnormally high duration which are at the far right in the histogram. Delta profiling of actual instruction flows therefore allows for an overview of the interrupt cycling in the kernel for a particular short running task. As discussed in section 5, to get more in depth analysis, we can take snapshots when abnormal latencies are encountered, to get a more in depth view upon identifying them from the histogram profiles.

5 Conclusion and Future Work

New techniques used in hardware tracing are now empowering developers in the domain of software performance debugging and analysis. We observe that hardware assisted tracing and profiling provides a very fine granularity and accuracy in terms of control flow and time. Our system utilizes hardware-assistance from Intel Processor Trace (PT). We observed that Intel PT, with a minimal overhead in the range of 2-5%, was able to provide a highly detailed view of control flow. We also present a detailed analysis of trace size and temporal resolution provided by PT, while fine tuning its configuration. With the help of our framework, we were able to generate detailed targeted callstacks for syscalls and observe differences between multiple executions. We also demonstrated a way to trace the software tracers themselves and show how similar kernel control-flow tracers such as Ftrace cause overheads as high as 63%, while PT was able to generate similar yet more detailed results with 1% overhead. Hardware tracing also allowed us to gather traces from parts of the kernel that are invisible to traditional software tracers. PT assisted cycle-accurate profiling was able to provide resolution as high as 14 ns, with timed events only 78 instructions apart. However, our analysis of Intel PT and the information released by Intel suggests that many additional features such as using PT in Virtual machine tracing are left yet to be explored. An extremely low overhead and low impact hardware-assisted tracing technique such as Intel PT has the potential to disrupt the tracing domain.

Tracing Tools Integration

A very important task from a research and implementation perspective is the ease and availability of hardware tracing. Its true potential can be achieved with low overhead software tracers like LTTng that can integrate PT for generating small trace snapshots at *hot* events. We already cover one such contribution in our paper in section 3.4, but a simple to use accessible API for Intel PT can be developed so that the tracing tools can leverage its full potential. New in-kernel tracing frameworks such as eBPF can now provide an easy way to insert tracing oriented bytecode in the kernel for performance analysis. Dynamic tracing with eBPF can benefit from hardware tracing by conditionally generating snapshots from the hardware tracing APIs.

Latency Snapshots

We hinted how, in addition to syscall latency profiles, more in-depth analysis on other non-deterministic latencies can be done. An interesting observation relevant for realtime systems is also a in depth analysis of IRQ latency. Newer tracepoints in the kernel introduced by tracers such as LTTng [41] allow recording software trace events when IRQ a latency beyond a certain threshold is reached. We can refine the idea with recording hardware trace snapshots in such conditions, thus obtaining more detailed control-flow and timing information. This hardware-assisted software tracing can add significant value to current tracing tools.

Virtual Machine Trace and Analysis

Intel PT can also be used to detect VM state transitions, in host or guest only hardware tracing, for more in depth analysis of VMs without any software tracing support. We observed [31], that PT can also generate VMCS Packets and set the NonRoot (NR) bit in PIP packets when hardware tracing is enabled in VM context. The extra information in hardware traces allows the decoder to identify VM entry and exit events and load specific binaries for rebuilding control-flow across VMs and host. Thus, with no software intrusion and a low overhead, we can get accurate VM traces, compare and quantify their performance.

List of Abbreviations

PT : Processor Trace
BTB : Branch Trace Buffer
ICE : In-circuit Emulator
ETM : Embedded Trace Macrocell
PTM : Program Trace Macrocell
LBR : Last Branch Record
BTS : Branch Trace Store
MSR : Model Specific Registers
FUP : Flow Update Packet
TSC : Time-stamp Counter
MTC : Mini Time-stamp Counter
PSB : Packet Stream Boundary
TNT : Taken-Not-Taken
TMA : TSC/MTC Alignment
CBR : Core-to-Bus Ratio
TIP : Target Instruction Pointer
TRF : Temporal Resolution Factor
LTTng : Linux Trace Toolkit next generation
BLAS : Basic Linear Algebra Subprograms
AVX : Advanced Vector Extensions
SSE : Streaming SIMD Extensions
CYC : Cycle Count
VMCS : Virtual Machine Control Structure
NR : Non Root
IRQ : Interrupt Request
eBPF : extended Berkeley Packet Filter
VM : Virtual Machine

Competing interests

The authors declare that they have no competing interests.

Author's contributions

SDS reviewed the state-of-the-art in the field, defined objectives of this research and analyzed the hardware tracing tools, techniques and their impact on modern systems. He did research on hardware trace decoder, delta profiling algorithm, syscall latency visualizations and software tracer impact. He also implemented these analyses and devised all the experiments presented in this paper that characterize performance of Intel Processor Trace. MRD initiated and supervised this research, led and approved its scientific contribution, provided general input, reviewed the article, and issued a final approval for submission.

Acknowledgements
We are grateful to Ericsson, EfficiOS, NSERC and Prompt for funding, Intel developers for making Processor Trace accessible, Francis Giraldeau for technical guidance and Naser Ezzati-Jivan for his comments.

References

1. AK Tewar, AR Myers, A Milenković, mcftraptor: Toward unobtrusive on-the-fly control-flow tracing in multicores. J. Syst. Archit. **61**(10), 601–614 (2015). doi:10.1016/j.sysarc.2015.07.005
2. T Ball, S Burckhardt, J Halleux, M Musuvathi, S Qadeer, Deconstructing concurrency heisenbugs. in Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference On, pp. 403–404 (2009). doi:10.1109/ICSE-COMPANION.2009.5071033
3. The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface v3.01. `http://nexus5001.org/`. Accessed 9 Mar 2016 (2012)
4. ARM, DSTREAM High-Performance Debug and Trace Unit. `http://ds.arm.com/ds-5/debug/dstream/`. Accessed 9 Mar 2016
5. Green Hills Software SuperTraceProbe. `http://www.ghs.com/products/supertraceprobe.html`. Accessed 9 Mar 2016
6. ARM, ARM DS-5 Development Studio. `http://ds.arm.com/ds-5/`. Accessed 9 Mar 2016
7. Green Hills Software TimeMachine Debugging Suite. `http://www.ghs.com/products/timemachine.html`. Accessed 9 Mar 2016
8. S Tallam, R Gupta, Unified control flow and data dependence traces. ACM Trans. Archit. Code Optim. **4**(3) (2007). doi:10.1145/1275937.1275943
9. C Boogerd, L Moonen, On the use of data flow analysis in static profiling. in Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference On, pp. 79–88 (2008). doi:10.1109/SCAM.2008.18
10. BA Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, DWR Marsh, Industrial perspective on static analysis. Software Engineering Journal **10**(2), 69–75 (1995)
11. B Livshits, Improving software security with precise static and runtime analysis. Dissertation, Stanford University, Stanford, CA, USA (2006)
12. K Goseva-Popstojanova, A Perhinschi, On the capability of static code analysis to detect security vulnerabilities. Inf. Softw. Technol. **68**(C), 18–33 (2015). doi:10.1016/j.infsof.2015.08.002
13. J Lee, A Shrivastava, A compiler optimization to reduce soft errors in register files. SIGPLAN Not. **44**(7), 41–49 (2009). doi:10.1145/1543136.1542459
14. T Ball, JR Larus, Optimally profiling and tracing programs. ACM Trans. Program. Lang. Syst. **16**(4), 1319–1360 (1994). doi:10.1145/183432.183527
15. JM Anderson, LM Berc, J Dean, S Ghemawat, MR Henzinger, SqTA Leung, RL Sites, MT Vandevoorde, CA Waldspurger, WE Weihl, Continuous profiling: Where have all the cycles gone? ACM Trans. Comput. Syst. **15**(4), 357–390 (1997). doi:10.1145/265924.265925
16. J Dean, JE Hicks, CA Waldspurger, WE Weihl, G Chrysos, Profileme: hardware support for instruction-level profiling on out-of-order processors. in Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium On, pp. 292–302 (1997). doi:10.1109/MICRO.1997.645821
17. MC Merten, AR Trick, EM Nystrom, RD Barnes, WqmW Hmu, A hardware mechanism for dynamic extraction and relayout of program hot spots. SIGARCH Comput. Archit. News **28**(2), 59–70 (2000). doi:10.1145/342001.339655
18. K Vaswani, MJ Thazhuthaveetil, YN Srikant, A programmable hardware path profiler. in Proceedings of the International Symposium on Code Generation and Optimization. CGO '05, pp. 217–228. IEEE Computer Society, Washington, DC, USA (2005). doi:10.1109/CGO.2005.3. http://dx.doi.org/10.1109/CGO.2005.3
19. A Nowak, A Yasin, A Mendelson, W Zwaenepoel, Establishing a base of trust with performance counters for enterprise workloads. in 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 541–548. USENIX Association, Santa Clara, CA (2015)
20. G Bitzes, A Nowak, The overhead of profiling using pmu hardware counters. Technical report, CERN, openlab (2014)
21. JR Larus, Efficient program tracing. Computer **26**(5), 52–61 (1993). doi:10.1109/2.211900
22. N Nethercote, J Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007). doi:10.1145/1273442.1250746
23. N Nethercote, A Mycroft, Redux: A dynamic dataflow tracer. Electronic Notes in Theoretical Computer Science **89**(2), 149–170 (2003). doi:10.1016/S1571-0661(04)81047-8
24. J Weidendorfer, Sequential performance analysis with callgrind and kcachegrind. in Tools for High Performance Computing, pp. 93–113. Springer, Berlin, Heidelberg (2008)
25. PA Sandon, YqC Liao, TE Cook, DM Schultz, P Martin-de-Nicolas, Nstrace: A bus-driven instruction trace tool for powerpc microprocessors. IBM J. Res. Dev. **41**(3), 331–344 (1997). doi:10.1147/rd.413.0331
26. B Vermeulen, Functional debug techniques for embedded systems. IEEE Design Test of Computers **25**(3), 208–215 (2008). doi:10.1109/MDT.2008.66
27. A Vasudevan, N Qu, A Perrig, Xtrec: Secure real-time execution trace recording on commodity platforms. in System Sciences (HICSS), 2011 44th Hawaii International Conference On, pp. 1–10 (2011). doi:10.1109/HICSS.2011.500
28. ST Ball, Adding diagnostic helps to the target system. in Debugging Embedded Microprocessor Systems, pp. 41–45. Elsevier, USA (1998)
29. C Pedersen, J Acampora, Intel code execution trace resources. Intel Technology Journal **16**(1), 130–136 (2012)
30. ML Soffa, KR Walcott, J Mars, Exploiting hardware advances for software testing and debugging (nier track). in Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, pp. 888–891. ACM, New York, NY, USA (2011). doi:10.1145/1985793.1985935. http://doi.acm.org/10.1145/1985793.1985935
31. Intel, Intel processor trace. in Intel 64 and IA-32 Architectures Software Developer's Manual, pp. 3578–3644.

Intel Press, Denver, CO (2015). Accessed 9 Mar 2016.
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
32. A Kleen, Adding Processor Trace Support to Linux. `http://lwn.net/Articles/648154/`. Accessed 9 Mar 2016 (2015)
33. A Kleen, Simple PT. `https://github.com/andikleen/simple-pt`. Accessed 9 Mar 2016
34. Intel, Intel Processor Trace Decoder Library. `https://github.com/01org/processor-trace`. Accessed 9 Mar 2016 (2015)
35. L Baugh, C Zilles, An analysis of i/o and syscalls in critical sections and their implications for transactional memory. in Performance Analysis of Systems and Software, 2008. ISPASS 2008. IEEE International Symposium On, pp. 54–62 (2008). doi:10.1109/ISPASS.2008.4510738
36. P Moret, W Binder, A Villazón, D Ansaloni, A Heydarnoori, Visualizing and exploring profiles with calling context ring charts. Softw. Pract. Exper. **40**(9), 825–847 (2010). doi:10.1002/spe.v40:9
37. A Adamoli, M Hauswirth, Trevis: A context tree visualization &#38; analysis framework and its use for classifying performance failure reports. in Proceedings of the 5th International Symposium on Software Visualization. SOFTVIS '10, pp. 73–82. ACM, New York, NY, USA (2010). doi:10.1145/1879211.1879224. http://doi.acm.org/10.1145/1879211.1879224
38. Julia Language Benchmarks. `http://julialang.org/benchmarks/`. Accessed 9 Mar 2016
39. OpenBLAS, OpenBLAS, an Optimized BLAS Library. `http://www.openblas.net`. Accessed 9 Mar 2016
40. Ftrace - Function Tracer. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`. Accessed 9 Mar 2016
41. M Desnoyers, Emit Tracepoint in Preempt and IRQs Off Tracer.
`http://lists.lttng.org/pipermail/lttng-dev/2015-October/025151.html`. Accessed 9 Mar 2016

Figure 1 Hardware tracing overview

Figure 2 An odd-even test generates corresponding Taken-Not-Taken Packets

Figure 3 The architecture of our proposed hardware-assisted trace-profile framework. Simple PT [33] is used for trace hardware control

Figure 4 The effect of an external software tracer on the `mmap()` syscall, obtained from a near zero overhead hardware trace, is visible in (a) as extra layers as compared to (b), which took a shorter path and a shallower callstack. The rings represent the callstack and are drawn based on instruction count per call

Figure 5 Trace size and resolution while varying *valid* CPU cycles between two subsequent CYC packets. Lower TRF value is better

Figure 6 Trace size and resolution while varying *valid* CPU cycles between two subsequent MTC packets. Lower TRF value is better

Figure 7 Trace size and resolution while varying *valid* bytes of data between two subsequent PSB packets

Figure 8 Histogram of Instruction count delta for `superSTI` and `superCLI` instructions generated using Delta Profiling algorithm

Figure 9 Histogram of Instruction count delta for `superSTI` and `SuperCLI` instructions generated using Delta Profiling algorithm