Hardware-Assisted Software Event Tracing

Adrien Verge
Department of Computer and Software Engineering
Ecole Polytechnique Montreal
Montreal, Canada
adrien.verge@polymtl.ca

Naser Ezzati-Jivan
Department of Computer and Software Engineering
Ecole Polytechnique Montreal
Montreal, Canada
n.ezzati@polymtl.ca

Michel R. Dagenais
Department of Computer and Software Engineering
Ecole Polytechnique Montreal
Montreal, Canada
michel.dagenais@polymtl.ca

ABSTRACT

Event tracing is a reliable and low-intrusiveness method to debug and optimize systems and processes. Low overhead is particularly important in embedded systems where resources and energy consumption is critical. The most advanced tracing infrastructures achieve a very low footprint on the traced software, bringing each tracepoint overhead to less than a microsecond. To reduce this still non-negligible impact, the use of dedicated hardware resources is promising. In this paper, we propose complementary methods for tracing, that rely on hardware modules to assist software tracing. We designed solutions to take advantage of CoreSight STM, CoreSight ETM and Intel BTS, which are present on most newer ARM-based systems-on-chip and Intel x86 processors. Our results show that the time overhead for tracing can be reduced by up to 10 times when assisted by hardware, as compared to software tracing with LTTng, a high-performance tracer for Linux. We also propose a modification to the Perf tool to speed BTS execution tracing up to 65%.

Keywords

ARM CoreSight, debugging, dedicated hardware, event tracing, Intel BTS, LTTng

1. INTRODUCTION

Tracing is a monitoring method to record the runtime behavior of a program, for debugging, optimization or performance measurement purposes. Traces are generated during execution. They contain series of timestamped events, which may be used to understand and model a process execution. Traces can be analyzed live or later, locally or on a remote host. Unlike classical debugging, tracing is focused on low-intrusiveness, allowing to study a process with a minimal alteration of its execution. Although lightweight, software tracing solutions have non-zero side-effects because of extra executed code, cache perturbation and other alterations to the execution path [1].

To facilitate program development, hardware manufacturers such as Intel and ARM embed dedicated debugging circuits in their newer processors. For instance, Intel BTS and ARM CoreSight ETM provide program tracing (recording the sequence of executed instructions), whereas ARM CoreSight STM is designed to timestamp instructions that write to a specific area. Although having different purposes, hardware debugging circuits provide dedicated resources (comparators, buffers) that can be used to trigger events and store data with almost zero overhead. By reconfiguring these resources, we want to take advantage of hardware circuits to make tracing more lightweight.

The objective is to measure the benefits of using specialized hardware components in the Linux Trace Toolkit next-generation (LTTng) [2], a reference infrastructure for kernel and user-space tracing for Linux. We designed solutions to retrieve trace information, similar to LTTng software traces, from hardware blocks on ARM (with CoreSight ETM, STM, ETB [3]) and Intel (with BTS [4]). We developed tracing tools that configure these hardware circuits and retrieve data of interest to produce traces. We measured the time overhead of such devices versus non-traced executions, in order to compare to LTTng-UST (LTTng User Space Tracing) pure-software tracing.

We detail the test environment and the hardware debugging components that we tested in section 2. The methodology and results are presented in section 3 for using hardware-assisted software tracing, and in section 4 for using execution path tracing hardware. We describe related work in section 5. We conclude and discuss future work in section 6.

2. TEST ENVIRONMENTS

Our experiments were run on three specific platforms. However, the hardware modules used for tracing are found in many other processors and systems-on-chip.

The platforms used were:

- an OMAP3530 system-on-chip with a ARM Cortex-A8 CPU and 512 MiB of LPDDR, integrated onto a Beagleboard-xM development card;
- an OMAP4430 system-on-chip with a dual-core ARM Cortex-A9 and 1 GiB of LPDDR2, integrated onto a Pandaboard development card;

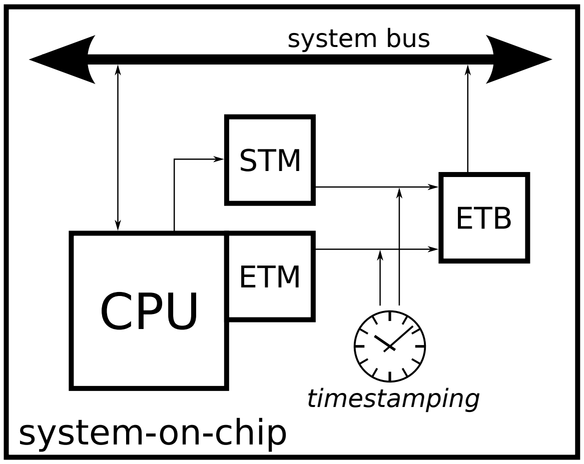


Figure 1: Overview of the CoreSight components used

- a desktop computer with an Intel Core i7-3770 processor (4 physical cores at 3,4 GHz) and 6 GiB of DDR3 RAM.

The first two platforms integrate ARM CoreSight debugging components, whereas the third one embeds Intel Branch Trace Store registers. These devices are presented in the next subsections.

2.1 ARM CoreSight

ARM CoreSight [3] is a set of hardware blocks that provide trace and debug functionalities for complex systems-on-chip. There is a variety of trace sources and collectors, allowing traces of different types to be timestamped and multiplexed in the same output.

The only CoreSight components that we used are ETM, STM and ETB. Their interaction is summarized in Figure 1.

2.1.1 CoreSight ETM

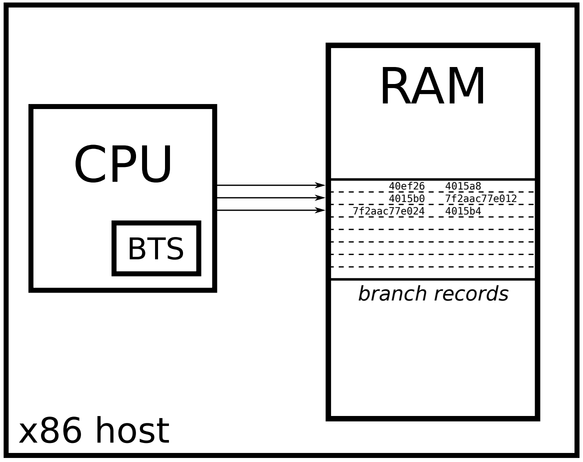
The Embedded Trace Macrocell (ETM) is an instruction and data tracer. It enables reconstruction of program execution by recording jumps. When activated, ETM watches the core's internal buses to detect branches with low interference with execution. ETM timestamps branches with cycle-level precision, and supports output filtering and compressing.

2.1.2 CoreSight STM

The System Trace Macrocell (STM) records and timestamps software events [5]. It allows real-time instrumentation of software by providing a memory area where software writes are converted to hardware messages. These messages are automatically timestamped and assigned to the requested channel. The area is divided into several channels, which allows multiple programs to be debugged at the same time.

The version we used is the TI STM, present on our development board. It is an earlier version of STM, not designed by ARM but with very close capabilities and the same output format [6]. In the rest of this article, we use the acronym "STM" indifferently for both versions.

Figure 2: Overview of Intel Branch Trace Store



2.1.3 CoreSight ETB

The Embedded Trace Buffer (ETB) stores traces from different sources in a single place. It collects streams output from the ETM and the STM, and allows deferring the retrieval of traces, acting as a buffer.

2.2 Intel

2.2.1 BTS

The Branch Trace Store (BTS) [4] system stores every branch taken during execution to a user-defined area in memory. The BTS registers are part of Intel's Model-Specific Registers (MSR), embedded in newer processors made by Intel since the Pentium 4 [7].

BTS allows to define a zone in main memory where the CPU will automatically store entries when encountering a branch. Any deviation from the execution flow (that is, not executing the next instruction) is saved in a 24-byte entry. BTS can throw an interrupt when the buffer overflows a given threshold. The user is responsible for draining this buffer to save tracing data. It can also be configured as a circular buffer, if only a backtrace is needed.

Embedded systems specialists estimate the BTS overhead between 20% and 100% [8], partly because the CPU enters a specialized debug mode associated with a 25 to 30 times slowdown [9, 4].

The BTS design is summarized in Figure 2.

2.3 Measuring the overhead of tracing

The experiments presented here aim at lowering the tracing overhead, thus minimizing the impact on the traced processes execution. Different measurable side effects can estimate this impact, for example the total elapsed execution time, extra system calls, accesses to memory and page faults, as well as cache memory usage or energy consumption. Among these, we selected the execution time as reference, since it is the major criteria for developers in most cases. Moreover, most other effects (such as page faults) also impact the execution time.

In the rest of this article, the term "overhead" will refer to the modification to traced process normal execution. To estimate the "overhead", we will measure the execution time difference associated with tracing. Depending on the experiment, this is obtained via the `time` command or with the `gettimeofday` system call. To reduce variability and influence of unrelated events, each test was run several times and over long execution times (between one and hundreds of seconds). In the case of two programs running in parallel (a trace producer and a trace consumer), only the traced program is timed. We assume that with long execution times, both the producer and the consumer enter a steady state, so that the producer's execution time is relevant by itself.

3. USING SYSTEM TRACING HARDWARE

The main functionality of tracing systems is to provide tracepoints. When the execution of a program reaches a tracepoint, an event is recorded along with customizable information such as timestamp or process ID.

Modern tracers like LTTng, focused on performance, achieve a good efficiency with low-overhead and precise tracepoints. However, pure-software methods still induce a slowdown, mainly due to synchronization and timestamp computation (which may involve a system call in the worst case).

3.1 System Trace Macrocell

3.1.1 Design

The STM module offers system tracing capability: writing to a specific zone in memory generates timestamped messages, and stores them in a dedicated buffer. This led us to design a tracing solution that takes advantage of these hardware circuits. Instead of using the `tracepoint` function provided by the LTTng-UST library, our prototype directly writes to the STM memory area. Similarly, the trace consumer does not read from shared memory, but retrieves data from the ETB when needed.

The software part of recording tracepoints is lightened: timestamping is automatically done via a hardware clock, as is transportation to the ETB for further fetching. Synchronization between trace producers is guaranteed, as long as each process writes to its own STM channel. There are hundreds of possible channels (the exact number depends on the hardware implementation), so many programs can be traced at the same time. Channel information is included in the generated output, to enable message decoding and demultiplexing after retrieval from the ETB.

Our implementation results in two programs: a trace producer, i.e. a traced program in which events occur; and a trace consumer, i.e. the program that will fetch raw hardware data from the ETB, decode and store it. The ETB has a fixed size, for this reason, the producer and the consumer must communicate to avoid overflow. They share a common page in memory to communicate and synchronize with semaphores. In particular, they both update a counter that represents buffer usage. The system's scheduler is meant to keep a balance between trace production and trace consumption. However, if the ETB becomes full, the producer stops and yields CPU time to the consumer to empty it.

3.1.2 Results

We measured the performance difference between this approach and the original LTTng-UST in two configurations, meant to be representative of either the worst case, or a more realistic situation. In the first case, the tracing overhead is maximum: the benchmark process runs a loop that does only tracing. In the second case, some arithmetic computation is executed in each iteration, to simulate a real program behavior. In both cases, tracing at each iteration is done either by calling LTTng-UST's `tracepoint`, or by interacting with our library to use CoreSight STM. This is then compared to the execution with no tracing at all. In the same manner, the trace consumer is either LTTng's session daemon, or our custom program to retrieve data from the ETB, decode it and store it to disk. It is important to note that these results show intentionally high tracing overhead, because we want worst-case results. For real-life programs, overhead is much lower, typically a few percent [2].

The benchmarks were run on a Pandaboard and showed that LTTng-UST's tracing time can be significantly reduced when using CoreSight STM and ETB. Here, we use a simple tracepoint type with a 24-bit integer payload. We could use any size of trace data as payload e.g., 16 bits which gives a reduced overhead (Figure 4, but in our experiments we choose 24 bits as a common size for better comparison.

Figure 3 presents the average execution time of iterations when looping 10^7 times, with our test program recording a tracepoint at each iteration. We show results for the untraced program (a simple loop iterating a volatile variable), the same program traced with LTTng-UST, and then traced using STM and ETB. In both case, the LTTng tracing delay is reduced by approximately 91% when using hardware-assisted tracing.

We then measured tracing performance with respect to tracepoint types, i.e., the tracepoints payload length. We ran the same programs with a fixed number of iterations (10^6) but with payload varying from 3 to 100 bytes. The results are shown in Figure 4. Please note that the numbers shown in this figure are the times needed only for tracing the system and no analysis time is included on that. Using the STM and ETB hardware modules is efficient for small messages, but does not scale well for payloads bigger than 60 bytes. STM + ETB has reduced performance for larger payloads, because the time needed to process a tracepoint is roughly proportional to the payload size for this tracepoint. STM encodes the payload data, and the kernel needs to process and decode this data when retrieving it from the ETB. Thus, for an empty payload, there is nothing to encode/decode: it is very fast; and for a large payload, it takes longer time.

Whereas, with LTTng, every tracepoint results in storing data in RAM (in a cache i.e., a ring buffer), which is very fast. The part that takes most time with LTTng is retrieving the clock time for every tracepoint, because it includes a syscall and a context switch, which STM does instantaneously using hardware. So with LTTng, having a small or large payload doesn't change much, because the part that takes time is independent of that payload size. We couldn't use the same cache (ring buffer in RAM) in STM + ETB as well, because STM and ETB are dedicated hardware modules, independent from RAM.

Figure 3: Average execution time of programs traced with LTTng-UST, with hardware (STM + ETB), and not traced. We present a program looping, first with only a tracepoint in each iteration, and then the same program performing real calculation in each iteration.

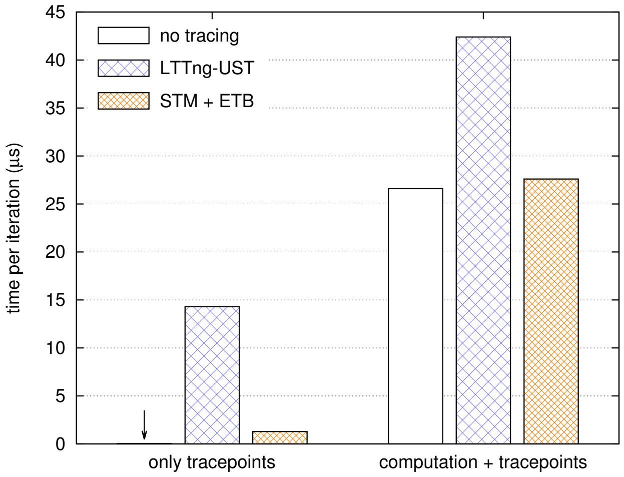
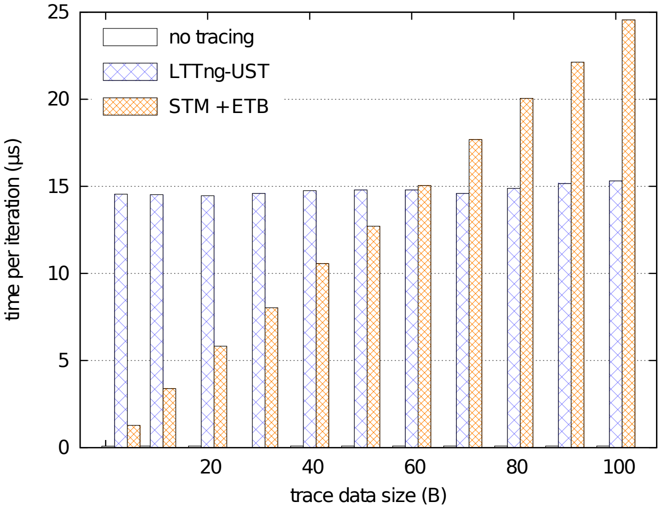


Figure 4: Average execution time of programs traced with LTTng-UST, with hardware (STM + ETB), and not traced. We present tracepoints with different payload size.



These results also highlight the time taken by LTTng-UST to synchronize with the trace consumer with memory barriers, but also its ability to take advantage of the cache when trace payloads get longer, as mentioned earlier.

Since most tracepoints typically used with LTTng do not exceed a few bytes, using CoreSight STM and ETB is a significant improvement to the low-intrusiveness of LTTng. Moreover, the timestamping provided by CoreSight is cycle-precise, which is not the case with LTTng on every platform.

4. USING EXECUTION PATH TRACING HARDWARE

Some hardware debugging components provide the ability to save the execution path, i.e. the sequence of addresses for executed instructions. When timestamped, this information is sufficient to trace the complete execution of a program, or a section thereof, and constitutes a superset of the program location information conveyed by tracepoints. Nonetheless, tracing infrastructures like LTTng have the ability to associate a payload, and even a context, with the tracepoints (for instance, the thread ID or the number of encountered page faults), which is not available with hardware tracing components. However, if this context information is not needed, and only the sequence of reached tracepoints is of interest, software tracing can benefit from hardware assistance.

4.1 Embedded Trace Macrocell

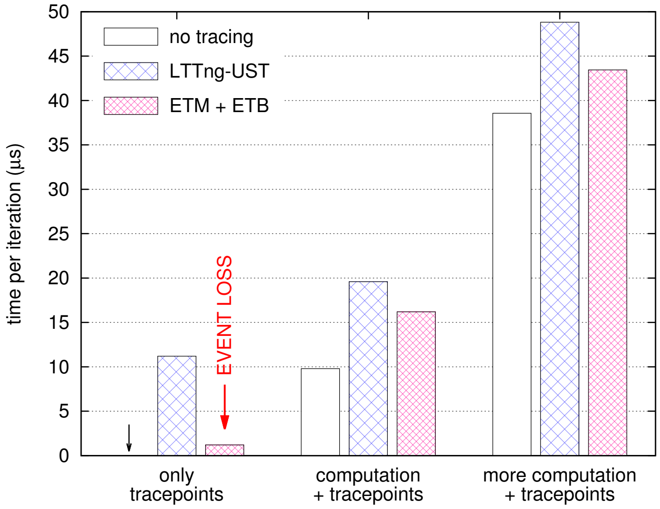
4.1.1 Design

The ETM hardware module performs execution path saving in a highly compressed format, and without interfering with the traced program execution. Moreover, it can be enabled and disabled by triggers, including address matching and context ID matching. This allows tracing a specific process, in a specific address range. Moreover, the traced program does not need to be recompiled to be traced: only the virtual addresses of its symbols must be known.

We chose to use ETM to trace the program when it enters the portion of code corresponding to the tracepoint. For instance, if we want the call to function `foo` to be traced, we set up ETM to start when the program is at the address of `foo`, and stop at the same address + 4. ETM is also configured to trigger only when the interesting process is running. Due to hardware limitations, this design only allows one tracepoint, with no payload. If several events need to be traced, a full execution trace or a mix of pure-software and hardware tracepoints can be used.

We implemented this design with two programs: a trace producer, which represents the traced program; and a trace consumer, whose role is to regularly drain the ETB and save its contents to disk for further decoding. The trace producer enables the ETM by communicating with the kernel, via an entry in `sysfs`. We patched the Linux kernel to enable full configuration of the ETM, especially the address range selection and context ID tracking [10]. This patch has been sent to the Linux Kernel Mailing List. Once the ETM is activated, it is configured to trigger at the address of a simple function that does an arithmetic computation. The test program, after activating the ETM, enters a loop calling this

Figure 5: Average execution time of programs traced with LTTng-UST, with hardware (ETM + ETB), and not traced. We present programs that do a loop where each iteration performs either only tracing, or tracing plus computation (with more time spent on computation in the third program).



function at each iteration. It is thus equivalent to the other test cases studied.

4.1.2 Results

This implementation is compared to two others: the first one is the same program but untraced; the second one has hardware tracing replaced by a call to LTTng-UST's `tracepoint` in each iteration (and the hardware trace consumer replaced by LTTng's session daemon). We evaluated the tracing time overhead by running these on a Beagleboard. Figure 5 presents the execution time of our three benchmark programs in three different configurations: recording only tracepoints, performing some extra computation in each iteration, and performing even more computation. In this experminet, tracepoint type size is zero (no ETM payload and empty LTTng tracepoint). Once again, it is important to note that these results show the overhead in the worst-cases, because our benchmark programs do almost nothing but recording tracepoints.

This is also true for ”tracepoints + computation” scenarios, because even for the bars on the right (”more computation + tracepoints”), the program is a loop that produces tracepoints at an insanely high rate. It does something like a hundred useless instructions between each tracepoint (whereas a normal program would generate tracepoints much less frequently).

In other words, a ”normal” program in a general use-case on ARM would not generate that much tracepoints per second. In this study, we pushed the tracing tools (both ETM and LTTng) to their limits (around 100k to 1M events / second) to highlight their differences.

First, it is noticed that, for very high tracing frequencies, events are lost by the ETM. However, this situation is infrequent: it happens when events occur more often than 10^5 times per second.

ETM drops packets when the buffer is full. This means when ETM is configured to generate trace packets too frequently, and the kernel routine to empty the buffer is not called frequently enough, trace packets are dropped.

Rigorously, we cannot trust the ETM + ETB method because of packet loss. But this only happen in extreme cases (like the infinite loop that does nothing but triggering tracepoints). Moreover, event loss wouldn't happen if the kernel interrupted the process when the ETB buffer is full.

Apart from cases with lost events, results show that using ETM and ETB reduces the time overhead from 30% to 50% when compared to LTTng-UST. This demonstrates the performance benefits of using dedicated hardware to trace a given location in a program, in addition to the fact that there is no need to recompile the traced program. However, this solution only provides the time (although cycle-precise) when the program reaches a specific point, and no other information. Also, its most significant drawback is that it only allows tracing a few points in the program (because the number of available address triggers in ETM remains very limited). Configuring the ETM to trace the whole program is possible, but every branch would be recorded and a significant tracing overhead would be incurred.

4.2 Branch Trace Store

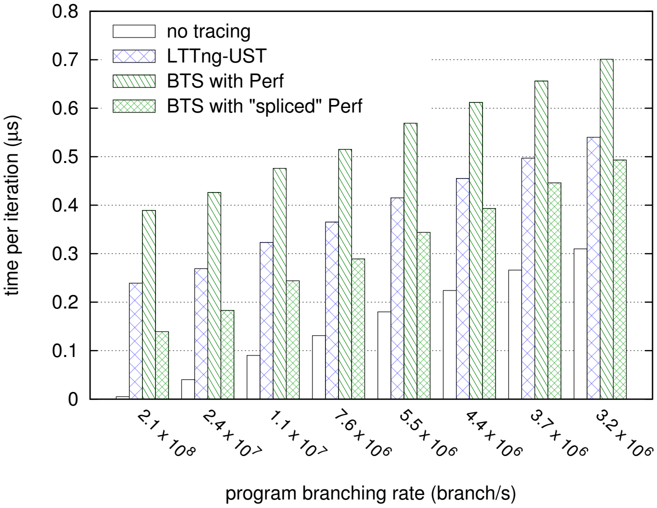
4.2.1 Design

The Branch Trace Store (BTS) registers, included in most newer x86 processors from Intel, allow saving every branch taken by the CPU to an area in main memory. Each branch is stored as a couple (origin address, destination address). The area starting address and size is user-definable, but cannot be configured to limit tracing to a specific process or address range. BTS does not timestamp records either. For these reasons, software has to spend more time for tracing control and synchronization: enable or disable it on context switches (to trace only one process), separately collect timestamps, and drain the buffer to save data to disk.

With BTS, every branch of the program is stored. The advantage of this behavior is that it provides much more information than the use of sparse tracepoints. Its drawback is a significant perturbation of the traced process execution, due to frequent memory accesses (to store branches) and related cache usage.

We wanted to know if the use of BTS, although known to have a significant performance impact, could do better than pure-software tracing, which is also very intrusive when used to trace the complete execution path. For this purpose, we designed test programs meant to be traced either with LTTng, or using BTS. In order to have comparable results, we needed to devise programs that trigger BTS branching records only where wanted, that is at the place where we would insert tracepoints. For this purpose, our benchmark programs are just loops performing arithmetic computations (to simulate real software activity): at the end of each it-

Figure 6: Time overhead of tracing when using LTTng-UST or Intel BTS hardware module, for programs with events traced at various frequencies. BTS results are presented for the original Perf implementation, and with our "splice" modification to avoid a multiple copies.



eration, the looping branch is recorded. The LTTng-traced version of these programs contains a `tracepoint` at the end of each iteration. Finally, in order to measure the effect of tracepoints sparsity, we varied the length of the loop computation. This changes the relative frequency of branches in the program.

4.2.2 *Re-implementing Perf*

We modified the Linux kernel to handle BTS more efficiently [11]. The default behavior (illustrated in Figure 7) was to set up one buffer per CPU, and to copy its whole content to a larger place in RAM every time a buffer full interrupt or a context switch occurs. Then, the user-space tracing daemon would copy the contents of this intermediate buffer to disk. The trace file being opened without the `O_SYNC` flag, writing to disk is not synchronized and data is possibly copied once more to a temporary buffer.

To avoid the time-wasting multiple copies in RAM (especially the one performed during the handling of an interrupt), we re-implemented this Linux kernel section to use ring-buffers. BTS is then configured to store entries in a sub-buffer; when it is full, BTS is reconfigured to use the next sub-buffer in the ring-buffer. This way, there is no urgent need to copy the BTS data to make room: this data can be saved to disk later by a kernel task. The size of the sub-buffers is the same as in Perf (64 KiB), their number was chosen to allocate a bit less resources than Perf does in its default implementation, that is 16 kernel-space pages plus 128 user-space pages per core (576 KiB per core for original Perf, 512 KiB in our implementation). Because the trace data does not transit through user-space before being stored, we use the term "splice" for our design. It is illustrated in Figure 8.

4.2.3 *Results*

We measured the overhead induced by tracing with LTTng-UST, with BTS using the regular Perf interface, and with BTS using our modified kernel interface ("splice"). The results for programs with various branching rates are shown in Figure 6. To give the reader an idea of the branching rates presented here, we measured as a reference the rates for common programs: `md5sum`: 7.4×10^6; `sha256sum`: 3.7×10^7; `gcc`: 6.70×10^8. These values, given in branches per second, were measured on the desktop machine described in section 2.

First, this experiment once more shows that the tracing overhead highly depends on the tracing frequency. Bars on the left, in Figure 6, show the overhead for programs that do nothing but recording tracepoints. Secondly, we compare the different tracing solutions overhead to LTTng, our reference tracer. The experiment reveals that using BTS through the regular Perf interface incurs between 30% and 60% time overhead, when compared to tracing with LTTng. This highlights the heavy bus usage and the double copy done by the Branch Trace Store system in its original Linux implementation. However, when using our "splice" design (in a kernel patched to use ring-buffers and avoid a double copy), BTS tracing performs much better than the original. It even overtakes LTTng with an overhead reduced by 10% to 45%.

These results highlight the limits of Perf's handling of BTS: multiple copies inducing a heavy bus usage that competes with the traced program's execution. Our re-implementation use ring-buffers to avoid multiple copies, resulting in a performance enhancement that makes BTS lighter than LTTng. Still, BTS remains costly for a hardware mechanism, due to a large trace volume (24 bytes per branch). The reader should note that our benchmark is valid for programs that record tracepoints at each branch, which is not often the case with higher level event tracing.

5. DISCUSSION AND RELATED WORK

This section presents related work in the two main related areas: software tracing tools and tracing systems using hardware components.

5.1 Software tracing

A wide range of solutions have been developed to trace programs with pure-software [12]. They either use interrupt-based methods or instrumentation (code modification). We present here the main software tracers for Linux.

5.1.1 *Ptrace*

The ptrace [13] system call is an old Unix functionality enabling a process to control another one, including reading its state and intercepting its system calls. It is used in debuggers and programs with moderate requirements on performance, such as gdb and strace. By giving access to a process internals, ptrace provides powerful control, albeit at a significant performance cost [13]. This is in large part because each communication between the tracer and the traced process needs at least two context switches. Because of ptrace's intrusiveness, newer tools have been designed with lower impact on the traced process execution.

Figure 7: Operation of Perf and BTS in the original implementation

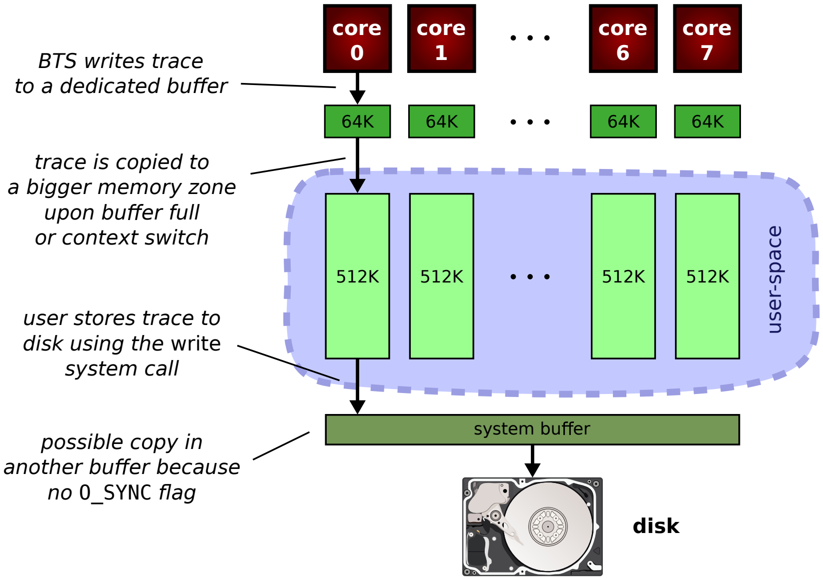
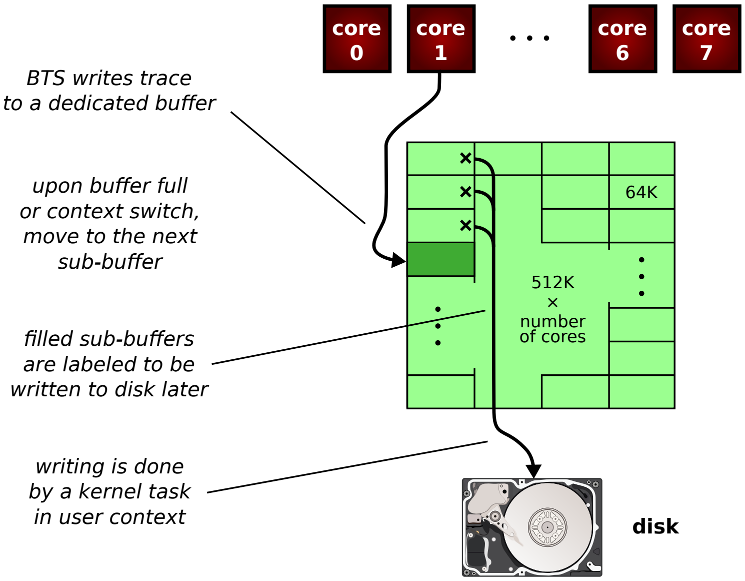

core 0
core 1
. . .
core 6
core 7
BTS writes trace to a dedicated buffer
64K
64K
. . .
64K
64K
trace is copied to a bigger memory zone upon buffer full or context switch
512K
512K
. . .
512K
512K
user-space
user stores trace to disk using the write system call
system buffer
possible copy in another buffer because no 0_SYNC flag
disk

Figure 8: Operation of Perf and BTS in our new "splice" implementation


core 0
core 1
. . .
core 6
core 7
BTS writes trace to a dedicated buffer
upon buffer full or context switch, move to the next sub-buffer
64K
512K × number of cores
filled sub-buffers are labeled to be written to disk later
writing is done by a kernel task in user context
disk

5.1.2 Ftrace

Ftrace [14] is a tool to monitor the system's behavior by instrumenting the kernel. It is meant to debug and profile kernel-level problems by tracing events such as system calls, interrupt handlers or scheduling functions. As Ftrace was developed to trace the kernel's internals, it does not support user-space program tracing. Moreover, it synchronizes on multi-core systems by spinlocking with interrupts disabled, which has a non-negligible impact on performance [15].

5.1.3 SystemTap

SystemTap [16] is an infrastructure that provides functional and performance debugging for Linux. It uses dynamic probes (Kprobes [17]) to hook on specific points of execution in the kernel, and more recently Uprobes [18] to instrument functions in user-space. Custom instrumentation can be defined via a scripting language, that once compiled into a kernel module, outputs trace information in text format. SystemTap is limited by its output format, which is not efficient for tracing programs with highly frequent events or very large traces.

5.1.4 LTTng

LTTng [2] is a tracing infrastructure focused on performance and output format flexibility. It achieves efficiency by using scalable and lockless methods such as read-copy-update (RCU) [19] and allocating per-CPU data structures. Hence, tracepoints are fast and the impact on cache is minimized.

LTTng records events to a ring buffer associated to the CPU executing the tracepoint. There is also a consumer daemon to share memory (i.e., ring buffers) with user space applications or kernel tracing modules to collect trace events and send them to disk or other destinations over the network.

LTTng's output trace complies with the Common Trace Format [20], a flexible and lightweight format supporting arbitrary event types and compression. LTTng supports high-performance kernel-space and user-space tracing (both sharing the same clock source), and all traces can be displayed in a graphical analyzer such as TMF [21], for better interpretation. For these reasons, we chose LTTng-UST (user-space tracing version of LTTng) as a reference to conduct our experiments. The results described in the rest of this paper refer to LTTng version 2.3.

5.2 Hardware-assisted tracing

Some tools take advantage of dedicated hardware capabilities in order to debug and measure program performance. We present here software solutions that use hardware components related to tracing.

5.2.1 Perf

Perf [22] is a program profiler for Linux. It was not initially meant for tracing, yet it can trace the sequence of instructions executed by a process on new Intel platforms. To achieve this, Perf uses Intel's BTS hardware registers, which are detailed in section 2. BTS control is done in the kernel through the Perf application binary interface, hence custom tracing programs can re-use this ABI to take advantage of these hardware capabilities.

Since recently, Perf also provides support for Last Branch Record (LBR) [23] registers on Intel processors. This feature enables automatic recording of the last taken branches. Depending on the CPU version, LBR stores from 4 to 16 records [23], which is useful for call stack debugging, but too limited for event-based tracing.

5.2.2 Linux

The Linux kernel provides basic CoreSight ETM support for OMAP3 chips. CoreSight ETM is a hardware facility to trace the sequence of executed instructions. It is detailed in section 2. Controling ETM is achieved via an entry in the sysfs virtual filesystem, but only a limited subset of the options offered by ETM is available. For instance, it is neither possible to change the address range to trace, nor can a specific context ID be followed (which however, ETM is capable of). Patches (including ours) were proposed to add more functionality and support.

5.3 Other tracing hardware

Other hardware facilities exist on different architectures for program tracing. We present here two common architectures offering interesting features.

5.3.1 Freescale

Freescale processors offer debug facilities compliant with the Nexus standard [24], in particular an on-chip trace buffer that can capture real-time bus information [25]. It can be configured to either store data for specific memory accesses, or changes in the execution flow, which allows program tracing. The triggering system is similar to CoreSight ETM, allowing to start and stop trace upon combinations of conditions, such as address range matching and opcode type. Another feature called Data Acquisition enables instrumented software to write data directly in the debugging channel. Also, trace buffers have enough capacity (typically 16 KiB) and bandwidth to trace events from all cores without loss [26].

The generated trace data can then be fetched by another machine via a JTAG interface connector [27], stored in a dedicated buffer or sent to main memory. Although trace can be collected by proprietary software, there is no public documentation available. A direct comparison was therefore not possible and it was not included in our study.

5.3.2 Intel

Intel has designed a new execution tracing solution called Processor Trace (PT) [28]. It enhances and unites previous tracing implementations [4, 23] in an optimized extension that is capable of timestamping, filtering, and tracking specific processes (via the CR3 register). The dedicated hardware facilities also include caching buffers to store small traces without accessing main memory [28], thus avoiding the BTS bus usage drawbacks.

Processor Trace is promising but not available on silicon at this time. Yet, Intel provides open-source libraries for generating and decoding traces [28]. Support in Linux is planned through the Perf ABI [29].

5.4 Existing use of CoreSight and BTS

The hardware facilities that we used in our study have existed for a few years [3, 4]. They are already taken advantage of in existing tools, for various purposes.

5.4.1 Debugging

CoreSight provide debugging features such as hardware and software breakpoints that can be used to stop a program or execute step by step. These mechanisms are designed for multi-core processors, hence they are able to send stop signals to other processors and act on a whole system. These features are used by development environments [30, 31], and software-hardware co-debug platforms have been proposed [32, 33].

5.4.2 Security

Tracing infrastructures are also used for security. Scherer and Horvath propose a watchdog solution to monitor the system on ARM-based platforms. They use CoreSight debug and trace hardware blocks to make and record measurements inside the system and estimate its healthiness [34]. On Intel x86 platforms, Branch Trace Store (BTS) can be used to detect security breaches. By monitoring addresses of executed instructions to detect deviations, Yuan et al. propose system security enhancements. They rely on the BTS debugging registers to detect abnormal control flows and identify unexpected code execution [35].

5.4.3 Programming

Another use of execution tracing is to speed up development processes by exposing execution information alongside the source code. Tralfamadore [36] is a system that displays execution trace analysis in a source code browser. By using BTS, it helps developers to track the control flow and write their programs more logically.

6. CONCLUSION AND FUTURE WORK

We have presented solutions to take advantage of various hardware debugging modules when doing software tracing, and compared their overhead to LTTng, a performance oriented tracer for Linux. The hardware modules studied belonged to two categories: software writes timestampers and program tracers.

We used STM from the former category, to instrument programs without needing to synchronize and compute a timestamp when recording a tracepoint. We showed a 10 times decrease in time overhead when tracing a user-space program on a Pandaboard. We then studied program tracers, i.e. hardware modules to record the flow of executed instructions. These do not provide the same information as tracers do but can be suited for tracing, depending on one's needs. On ARM platforms, ETM offers good efficiency because of its trace compression and triggering capabilities: tracing a program takes 30% to 50% less time when hardware-assisted. On Intel x86, the BTS registers are not adapted to event recording, mainly because they lack automatic enabling/disabling with address matching and trace compression. However, they can be used for this purpose by tracing every branch. Using BTS for recording user-space tracepoints on an Intel Core i7-3770 is 30% to 60% slower than using LTTng with the standard implementation, whereas our kernel modification with ring-buffers performs up to 45% faster.

The significant performance increase offered by some of our solutions, especially STM, should lead to an integration into LTTng in the near future. We also plan to extend this work to other interesting hardware features, including Freescale processors (which provides a program tracing functionality), and Intel Processor Trace (PT) [28], a new hardware tracing infrastructure that Intel will embed in its future processors. Intel PT is promising because of its triggering and filtering capabilities, similar to CoreSight ETM.

7. REFERENCES

[1] M. Desnoyers, *Low-impact operating system tracing*. PhD thesis, Ecole Polytechnique de Montreal, 2009.

[2] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006, pp. 209–224, Citeseer, 2006.

[3] ARM, "Coresight components technical reference manual," tech. rep., ARM, 2006.

[4] I. Corp., "Branch trace store (bts), intel® 64 and ia-32 architectures software developerś manual," 2013.

[5] R. Mijat, "Better trace for better software," *ARM Ltd*, 2010.

[6] M. Alliance, "Specification for system trace protocol (stp)." `http://mipi.org/specifications/debug#STP`, 2013. [Online; accessed 19-July-2016].

[7] I. Corp., "Table b-2. ia-32 architectural msrs, intel® 64 and ia-32 architectures software developerś manual," 2011.

[8] C. Pedersen and J. Acampora, "Intel code execution trace resources," *Intel R Technology Journal*, vol. 16, pp. 130–136, 2012.

[9] M. L. Soffa, K. R. Walcott, and J. Mars, "Exploiting hardware advances for software testing and debugging (nier track)," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 888–891, ACM, 2011.

[10] A. Verge, "Arm coresight: Etm: Fix a vmalloc/vfree failure and enhance tracing control." `https://github.com/adrienverge/linux/tree/patch_etm_v3`, 2014. [Online; accessed 19-July-2016].

[11] A. Verge, "perf: Intel bts: Add "splice" output mode." `https://github.com/adrienverge/linux/tree/patch_perf_bts_splice`, 2013. [Online; accessed 19-July-2016].

[12] N. Ezzati-Jivan and M. R. Dagenais, "Multi-scale navigation of large trace data: A survey," *Wiely Concurrency and Computation: Practice and Experience*, 2016.

[13] P. Padala, "Playing with ptrace, part i," *Linux Journal*, vol. 2002, no. 103, p. 5, 2002.

[14] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*, pp. 47–54, Citeseer, 2009.

[15] M. Desnoyers and M. R. Dagenais, "Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer," 2009.

[16] F. C. Eigler and R. Hat, "Problem solving with

systemtap," in *Proc. of the Ottawa Linux Symposium*, pp. 261–268, Citeseer, 2006.

[17] S. Goswami, "An introduction to kprobes, 2005." `http://lwn.net/Articles/132196`, 2015. [Online; accessed 19-July-2016].

[18] J. Keniston and S. Dronamraju, "Uprobes: User-space probes," *Linux Foundation Collaboration Summit*, 2010.

[19] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *AUUG Conference Proceedings*, p. 175, AUUG, Inc., 2001.

[20] M. Desnoyers, "Common trace format (ctf) specification." `http://git.efficios.com`, 2011. [Online; accessed 19-July-2016].

[21] Ericsson, "Open source application for viewing and analyzing traces." `http://www.tracecompass.org`, 2016. [Online; accessed 19-July-2016].

[22] J. Edge, "Perfcounters added to the mainline," *LWN, July*, vol. 10, p. 50, 2009.

[23] I. Corp., "Lbr stack, intel® 64 and ia-32 architectures software developerś manual," 2013.

[24] H. OḰeeffe, "The nexus 5001 forum standard providing the gateway to the embedded systems of the future." `http//www.ieeeisto.org`, 2004. [Online; accessed 19-July-2016].

[25] Freescale, "Application note an4420: Linux kernel program tracing using nexus." `http://www.freescale.com`, 2011. [Online; accessed 19-July-2016].

[26] Freescale, "In *eref 2.0: a programmerś reference manual for freescale power architectureÂő processors*. chapter 9, 787–803." `http://www.freescale.com`, 2011. [Online; accessed 19-July-2016].

[27] C. M. Maunder and R. E. Tulloss, *IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary Scan Architecture*. IEEE Computer Society Press Los Alamitos/Washington, DC, 2010.

[28] J. Reinders, "Intel processor tracing." `http://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2013. [Online; accessed 19-July-2016].

[29] A. Shishkin, "perf: Add support for intel processor trace.." `http://lwn.net/Articles/576551/`, 2013. [Online; accessed 19-July-2016].

[30] T. Instruments, "Code composer studio ide." `http://www.ti.com/tool/ccstudio`, 2011. [Online; accessed 19-July-2016].

[31] ARM, "Ds-5 development studio." `http://ds.arm.com`, 2014. [Online; accessed 19-July-2016].

[32] K.-J. Lee, A. Su, L.-F. Chen, J.-W. Jhou, J. Kuo, and M. Liu, "A software/hardware co-debug platform for multi-core systems," in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pp. 259–262, IEEE, 2011.

[33] J. K. A. P. Su, K.-J. Lee, J. Huang, G.-A. Jian, C.-A. Chien, J.-I. Guo, and C.-H. Chen, "Multi-core software/hardware co-debug platform with arm coresightâĎć, on-chip test architecture and axi/ahb bus monitor," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1–6, IEEE, 2011.

[34] B. Scherer and G. Horvath, "Trace and debug port based watchdog processor," in *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, pp. 488–491, IEEE, 2012.

[35] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as pmu deviation: detecting and identifying security attacks using performance counters," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, p. 6, ACM, 2011.

[36] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Tralfamadore: unifying source code and execution experience," in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 199–204, ACM, 2009.