Suchakrapani Datt Sharma, Michel Dagenais. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY : 1– Mon. Year

Enhanced Userspace and In-Kernel Trace Filtering for Quasi-Realtime Systems

Suchakrapani Datt Sharma[1*], *Student Member, IEEE* and Michel Dagenais[2], *Senior Member, IEEE*

Department of Computer and Software Engineering, École Polytechnique de Montréal, Montréal H3T 1J4, Canada

E-mail: [1]suchakrapani.sharma@polymtl.ca; [2]michel.dagenais@polymtl.ca

Received September 16, 2015

Abstract Tracing can be defined as a very fast system-wide fined grained logging mechanism, useful to detect performance issues in software. Trace tools like LTTng have a very low impact on the traced software as compared to traditional debuggers. However, for long runs, in resource constrained and high throughput environments, such as embedded network switching nodes and production servers, the collective tracing impact on the target software adds up considerably. The overhead is not just in terms of execution time but also in terms of the huge amount of data to be stored, processed and analyzed offline. This paper presents a novel way of dealing with such huge trace data generation by introducing a Just-In-Time (JIT) filter based tracing system, for sieving through the flood of high frequency events, and recording only those that are relevant, when a specific condition is met. With a tiny filtering cost, the user can filter out most events and focus only on the events of interest. We show that in certain scenarios, the JIT compiled filters prove to be 3 times more effective than similar interpreted filters. We also show that, with increasing number of filter predicates and context variables, the benefits of JIT compilation increase with some JIT*ed* filters being even 3x faster then their interpreted counterparts. We further present a new architecture, using our filtering system, which can enable *co-operative tracing* between kernel and userspace by sharing data efficiently. We compared the data access performance on our shared memory system and found an almost 100x improvement over traditional data sharing for co-operative tracing. We also demonstrate an illustrative use case in which this shared memory can be used while tracking syscall latency.

Keywords tracing, debugging, filtering, operating systems, interpreters

1 Introduction

With the traditional debugging approach, it becomes quite difficult to gather very low level as well as time accurate details about the system's behavior in quasi real-time or soft real-time systems. Sampling based profiling tools are also of limited use in such cases. Therefore, a fast logging mechanism, called tracing, is employed. Tracing can be divided according to the functional aspect (static or dynamic) or by its intended use (kernel or userspace tracing – also known as tracing domains).

*Corresponding Author

This work was supported by research grants from Ericsson, EfficiOS, Natural Sciences and Engineering Research Council of Canada (NSERC) and Prompt Québec

Tracing usually involves adding special *tracepoints* in the code. A tracepoint looks like a simple function call, which can be inserted anywhere in the code (in the case of userspace applications) or be provided as part of the standard kernel tracing infrastructure (tracepoint 'hooks' in the Linux kernel). Each tracepoint hit is usually associated with an event. For instance, the events in Linux kernel are very low level and occur frequently. Some examples are syscall entry/exit, scheduling calls, etc. For userspace applications, these can be any function call entry in the program. This indeed is a very efficient way to follow a program execution, rather than traditional debugging, specially in scenarios where the effect of pausing, waiting for user interaction and collecting data, can alter the behavior of a normal execution and yield incorrect results. Sometimes, the error cannot be reproduced in normal scenarios, due to the presence of time dependent errors in programs, which do not arise systematically or even frequently (for example, a *heisenbug*) [1]. For such cases, low overhead, low disturbance, tracing tools are invaluable.

Tracing involves storing the associated data in a special buffer whenever an event occurs. For a detailed execution trace of a very fast system, with high frequency trace events, this data is huge and contains precise time-stamps of the tracepoints hit, along with any optional event-specific information (value of variables, registers, etc). All this information can be stored in a specific format for later retrieval and analysis. In many cases, the trace data contains a lot of uninteresting, redundant, information during normal execution and needlessly consumes a lot of storage space. There can be situations where the target system is resource constrained, such as an embedded network controller, where a huge number of trace events can be generated at very high speed for hundreds of days in a row [2, 3]. It would be very inefficient to store all the traced data and try to retrieve it for offline analysis. In such situations, *trace filters* can be used to discard unwanted tracepoints and record only those specific ones that are of interest. The trace filters are comprised of multiple *filter predicates* which essentially are the conditions to be checked. The predicates are joined together with boolean operators and form a boolean expression that returns either a TRUE or FALSE. More about this will be discussed in later sections.

Most tools employ some form of filtering. However, as shown in Figure 1, the filtering scheme used in most state-of-the-art tools is the same – (1) define the filter predicates in a high level statement form, (2) create a predicate tree, and possibly a more efficient bytecode representation, (3) when the tracepoint is hit, walk the associated predicate tree while evaluating the conditions, or interpret the associated bytecode and evaluate the filter outcome. Another approach, as in Figure 1(c) is to (Just-In-Time) JIT compile the filter bytecode to native code and execute it on the machine. This yields a significant performance improvement as compared to interpreting the bytecode directly.

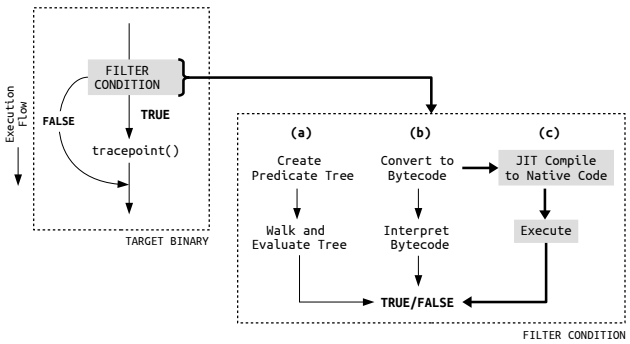

Fig. 1: Overview of filtering in trace and debug context. The bold path (c) is the approach which yields minimum overhead

Some interesting prototyping results were reported by Alexei Starovoitov. An implementation of JIT*ed* Berkeley Packet Filter (BPF) bytecode to

kernel tracing demonstrated an improvement from 32ns to 4ns per call (best case scenario) [4]. Along the same way, we improved tracing performance using JIT*ed* filters by enhancing the state-or-the-art tracing architecture. It has proven to be more robust than the current filtered tracing techniques and has lead to reduced trace storage size, and hence efficient diagnosis of problems. The benefits of having a very fine control over tracing have always been important from the developers perspective, but the filter computation overhead has always been a hindrance. In this paper, we present a new tracing scheme which tries to minimize this overhead and hence allows a more flexible use of the JIT*ing* technique for other purposes such as conditional tracing.

We also introduce the concept of *co-operative tracing* where, through an efficient sharing mechanism, kernel tracing can be guided from userspace or vice versa. Important data, such as performance counters or kernel-aggregated values, can be shared between the kernel and userspace filters, to achieve assisted tracing. For example, in certain scenarios, where the kernel may be monitoring syscall latency, the userspace process may provide hints about the expected *normal* latency, allowing for the dynamic adjustment of the maximum latency threshold, used in the kernel latency monitoring tracing filters.

The remainder of the paper is organized as follows : We start with a discussion on the basic building blocks of tracing. The techniques such as static and dynamic code instrumentation, on which the Linux tracing infrastructure is built, is explained. We also discuss its relevance in our context of trace filtering and our new scheme for kernel-userspace co-operative tracing. We then explain some commonly used filtering techniques used in scenarios like in-kernel network packet filtering and see how different tools like DTrace, LTTng and SystemTap approach filtering of traces. We move on to explain the design of bytecode interpreters relevant for filter design. Subsequently, JIT compilation techniques and their use in tracing are discussed. We introduce our proposed method and architecture for a JIT based optimized trace filtering framework, its design and its benefits. Its performance against current interpreted filtering techniques is evaluated and presented. We move on to propose our co-operative tracing system as an extension to the JIT based trace filtering system, to achieve high speed kernel-userspace tracing on resource constrained soft-realtime systems. We discuss how the current bytecode based filtering systems evolve to a generic system, and the efficient data sharing mechanism that we propose which yields close to 100x improvements over current data sharing mechanisms. We also expose how this architecture can be used independently, not just for conditional trace filtering, but for taking certain *actions* (like record a trace, aggregate data, share data with userspace etc.) based on conditions being met or not. We see how this architecture differs from approaches taken by tools such as DTrace and SystemTap. Finally, the results from the performance benchmarks, inferences drawn from the results, and the directions for future work are presented.

2 Literature Review

Most of the previous relevant work on filtering focused on network packets [5] and not on tracing. McCanne et al. proposed quite early a bytecode based virtual machine for in-kernel BSD network packet filtering, called as Berkeley Packet Filter (BPF). This interpreted technique delivered a performance of up to 20 times faster than the original tree based designs such as those of the CMU/Stanford Packet Filter (CSPF).

In Pathfinder, Bailey et al. [6] proposed a new way of specifying filters declaratively. This reduces the number of times a pattern has to be evaluated. Engler et al. presented DPF [7] where they showed an improvement of about 13x to 26x as compared to Pathfinder's implementation, due to the use of native compilation techniques, while keeping the same declarative language format of filter description.

BPF in its original format was further improved by Begel et al. in BPF+ [8], where they performed compiler optimizations to eliminate redundant predicates during filter generation. They also eventually implemented an elementary JIT compiler for BPF to improve its performance further - similar to what DPF had done with the Pathfinder's implementation.

Wu et al. recently proposed Swift [9], a new and complete packet filtering system based on a CISC ISA, and a BPF compatible API to simplify the code specification. They showed up to 3x the performance of corresponding in-kernel BPF implementations, mainly due to the aggressive use of SIMD instructions provided by i386 and x86_64.

Very recently, BPF was improved and evolved into an *extended* BPF (eBPF) implementation in the Linux kernel [4] with enhancements to register management, bytecode generation and optimizations using a modern compiler infrastructure. Its architecture was slightly modified to provide a more versatile in-kernel tiny virtual machine. This provided some of the foundation work for the userspace trace filtering and kernel-userspace tracing improvements we propose in this paper.

DTrace [10], originally developed for Solaris, is a purely script driven tool which consists of a new language (D language) for defining trace scripts. The trace scripts get compiled into an intermediate format (DIF) and are subsequently executed in DTrace's own in-kernel virtual machine. It is now possible to insert probes in userspace applications but this simply generates an interrupt, and the probe handler still executes in kernel space. For sharing data between probe executions, DTrace supports global variables, thread-level variables and aggregations. Aggregations can use per-cpu buckets and can thus be incremented with low overhead, without locking, at high frequency. The actual aggregation, with heavier locking, is only needed when extracting the aggregated value, typically at the end. Thread-level storage also avoids locking. Reentrency could be an issue if DTrace allowed the same thread-level variable to be accessed from normal and from interrupt context. Global variables in DTrace are not lock protected, and concurrent access can lead to corruption. Thus, although a very elaborate, popular and convenient scripting system for tracing and monitoring, DTrace suffers from several limitations. All scripts execute from kernel space and the only userspace to kernel interaction is tracepoints in applications generating costly traps. Furthermore, DTrace suffers from scalability problems and offers limited support for sharing global variables.

SystemTap has been developed along similar lines, to gather trace data dynamically. However, for kernel tracing, SystemTap generates C code to be compiled as a kernel module and loaded dynamically. This differs from the BPF and DTrace approach of executing bytecode within the kernel [11]. While this approach, in theory, offers the best performance with native code, it suffers from the requirements of needing a full compilation environment for the target kernel at runtime. SystemTap scripts can define and use global variables. Those are automatically read or write locked when accessed from the scripts, in case the scripts could be executing concurrently in probe handlers. This severely limits the scalability in scenarios requir-

ing data sharing. Furthermore, while probes can now be hooked to userspace code, this generates an interrupt and the corresponding scripts execute in kernel space, just like DTrace. There is thus no provision for scripts executing in userspace and sharing data with the kernel. We discuss this further in sections 3.4 and 5, where comparisons with the newer eBPF approach are made.

3 Background

Most tracing tools are built on underlying mechanisms which deliver different performance under various scenarios. In terms of performance, the most important factor across all tools is the reduced overhead. As each tracepoint execution incurs some time, this added time can potentially slow down the normal execution of the software and yield different results. The goal is therefore to have negligible overhead, insuring that the behavior is the same, with and without tracing. We now discuss some basic concepts and relevant techniques that many state-of-the-art tracing tools employ.

3.1 Static Instrumentation

Instrumentation in computing is the process of adding a certain code in any given application, with the inserted code snippet performing tasks related to diagnosing errors, profiling activities or gathering traces. The piece of code is intended to run fast and create a minimum overhead. In may cases, this code can be added *statically*, where it is added before compilation – for example, as a small function call at the trace target function entry and exit. When compiled with this instrumentation, each call to the trace target function entry and exit will lead to the instrumentation being run. This *static instrumentation* can also be done at compile-time where the code can be inserted by the compiler backend. An example is the `-finstrument-functions` switch for GCC, which inserts automatically profiling function calls for every function entry and exit. There are also other switches like `-fno-stack-limit` that generates code to ensure that the stack does not grow beyond a certain value [12]. Most compilers provide ways to define such instrumentation code. The Linux kernel provides manually inserted static trace points using the `TRACE_EVENT` macro [13]. It exposes trace hooks on which other kernel tracing systems can be built upon.

3.2 Dynamic Instrumentation

The other type of instrumentation is *dynamic instrumentation*, sometimes also called Dynamic Binary Instrumentation (DBI). Traditional static techniques insert code at compile time, and this inserted code is persistent. Whenever the specific function is called, the instrumentation code also runs and incurs some overhead – even when the developer does not necessarily want the instrumentation code to run. It also limits the instrumentation to only software for which the code is available for recompilation. To overcome these limitations, instrumentation techniques have been developed to add code to already compiled binaries. Instrumentation can either be performed on the binary residing on disk or by attaching to running processes, as when attaching a debugger to a running process.

Indeed, debuggers like GDB often rely on different forms of dynamic instrumentation. For the runtime instrumentation techniques, the backbone of all the approaches is the ability to halt the process, modify its memory, execute code and rewrite/restore registers. For on-disk dynamic instrumentation, the ability to load the binary, parse, disassemble and patch it with instrumentation code is required. Dynamic instrumentation

tools can be built using the TRAP based approach, the trampoline approach or a more elaborate JIT technique.

TRAP Instrumentation This technique is used in tools such as older Kprobes [14, 15] and GDB's normal tracepoints [16, 17]. The target instruction in the target function is replaced with an exception causing instruction (such as `int 3` on i386 architecture), and the exception handler then executes the instrumentation code. Traditional debuggers use this approach to implement breakpoints, with the help of debugging calls (such as `ptrace()` on Linux) to insert the TRAP and read the status and content of the debugged process.

JIT Instrumentation Other tools like Valgrind [18] use JIT based techniques. The binary is first disassembled and converted into an intermediate representation (IR). The IR is then instrumented with analysis code (such as the memory analysis code of `memcheck`). It is then recompiled back to the machine code and this instrumented code is stored in a code-cache. This can then be executed on Valgrind's synthetic CPU. It is much like interpreting native instrumented instructions in a process virtual machine [19]. With this scheme, the tool (Valgrind) has a very good control over the target executable. However, being very costly, this is not appropriate for the tracing domain.

Jump-pad Instrumentation Most of the good instrumentation frameworks such as Dyninst, and tools like GDB (for fast tracepoints) [20], however employ the much faster trampoline approach. Dyninst usually replaces the complete target function with a patched version in which a jump is placed at the specified *instrumentation point* in the target. This jump transfers the execution to a *trampoline* which executes the displaced instructions from the target function. Then, the stack is adjusted and the CPU registers are saved on stack. Finally, a call to the *snippet* (instrumentation code) is made. Upon return, the stack is rearranged, the original register state is restored, and the execution continues. As the execution stream is not disturbed, and the snippet execution only incurs some jumps and a few more instructions instead of a costly trap, this process is one of the fastest instrumentation approaches.

3.3 Applications in Tracing

Static tracing for LTTng, in kernel and userspace, is implemented using the static instrumentation techniques where a `tracepoint()` call may be placed anywhere in a function, and with supporting macros can generate very fast and accurate tracing data [21]. During compilation, this call gets expanded to an actual tracing function, according to the tracing context. This is the most optimum tracing mode. The Linux kernel's own tracing infrastructure, `ftrace`, provides static as well as dynamic tracing, depending on how it is used. Other tracing tools like SystemTap provide dynamic tracing through the use of Kprobes, Jprobes and Uprobes [22]. SystemTap also uses Dyninst for userspace tracing to gain some performance as well. The Kprobe approach has been used extensively to insert instrumentation code in the non-blacklisted kernel functions. These have traditionally been TRAP based, but jump-pad based probes have also been made available recently. Dynamic tracing with LTTng is based on the kernel's Kprobe technique.

Irrespective of what technology they are built upon, activated tracepoints may generate a lot of data. This motivates the work on filters and how they can be used to filter out a large fraction of uninteresting trace data.

3.4 Filters

Filtering is widely used in computing – from filter queries supplied to SQL databases to providing sand-boxed secure execution environments by filtering out syscalls [23]. The basic idea of a filter F is to find a small subset S from an large input set L, given a certain criteria of selection of the subset. The criteria being that application of filter F to the each element i of L returns TRUE.

$$S = \{i \in L : F(i)\}$$

Where F(x) can be defined as a boolean function whose outcome depends on the *filter predicates* $P_1, P_2..P_n$ These predicates are the heart of the filter itself and are joined with boolean operands. In our tracing context, a filter function F with an expression E and operators ($\star$) can be defined as follows,

For every $i \in L$, let

$$F(i) = \begin{cases} TRUE & \text{if } E = \{P_1 \star P_2 \star ..P_n\} \\ & \text{is } TRUE \text{ for } i \\ FALSE & \text{otherwise} \end{cases}$$

In operating systems and software applications, the need for filtering is most prominent in network packets. A lot of network traffic on the system causes packets of various protocols, sizes, having different source and destination, to pass through the device and the kernel. A user may wish to select very specific packets from the incoming stream, for example those which are either of type ARP or TCP/IP, originating from location SRC and having size less than 1KB. An expression E for a filter can then be built as follows,

$$(\underbrace{(type = ARP)}_{P_1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P_2})\mathbf{AND}$$
$$\underbrace{(origin = SRC)}_{P_3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P_4}$$

Continuing with this example, a predicate tree can be built from this expression. The concept of building trees and evaluating them for boolean outcomes has been used before in filters like the NIT [5] in SunOS and Linux kernel's internal network packet filter. In their earlier stages, these filters had a predicate tree walker which walked the nodes, evaluating them, and eventually reaching a final decision.

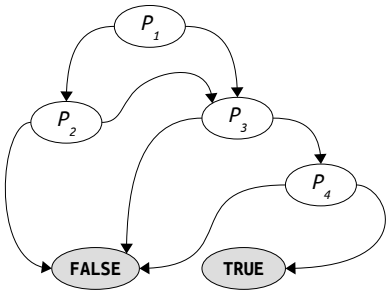

Fig. 2**:** The filter CFG representation. Right edge TRUE and left edge FALSE

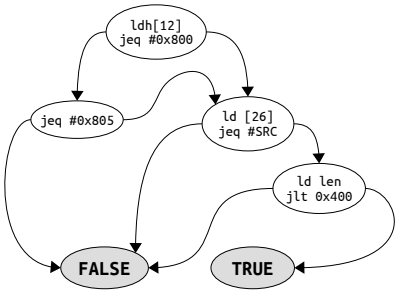

Fig. 3**:** The filter represented as *classic* BPF bytecode CFG

From the seemingly infinite number of packets being transferred from the device, the predicate tree formation and walking algorithm requires a considerable amount of computation to evaluate each packet. To overcome this, an initial version of the Berkeley Packet Filter (BPF) introduced a bytecode interpretation based filtering [5, 24]. A new control flow graph (CFG) representation of the example above is shown in Figure 2. The nodes of the CFG were converted to bytecode as shown in Figure 3 and interpreted by a small in-kernel

register based BPF interpreter. At the time of its introduction in BSD, BPF gave an improvement of 20 times over earlier techniques. This was also evident in recent patches to the Linux kernel where, in certain scenarios, BPF based filtering brought down the filtering costs from 139ns to 32ns [4]. We now discuss ways to improve this further by techniques such as JIT compiling (JIT*ing*) the bytecode.

Filter Performance Optimizations The maximum time consumed in VM execution is actually the cost of instruction dispatch [25, 26]. The computation can be equivalent to a few machine instructions but the dispatch mechanism usually takes a maximum of 10 to 12 machine instructions and involves a time consuming indirect branch. The dispatch mechanisms are typically either of switch or threaded type. To give a short overview, a switch dispatch may contain a large `switch-case` statement where, for each opcode of the VM, there would be one case statement to fetch and evaluate the opcode – as discussed in Figure 1 (b). Then, the next instruction is fetched and evaluated, as shown in the code snippet in Listing 1.

Listing 1: Interpreter dispatch for add

```
while(1) {
  switch (instr) {
  /* add */
  case ADD:
     regs[r1] = regs[r2] + regs[r3];
     break;
     ..
```

The upgraded BPF+ implementation [8] incorporated many tiny data-flow optimizations such as removing redundant predicates from the CFG during the BPF bytecode generation phase, the identification of potential lookup tables and the optimization of register usage etc. The authors also did an early JIT implementation and converted the bytecode to native code with a simple register assignment scheme. They obtained a speedup of up to 6.6x between unoptimized BPF code and JITed native code in certain scenarios with a varying number of predicates. As shown in Listing 2, a simple snippet from the x86 BPF JIT implementation, from a recent Linux kernel version, shows a similar ADD instruction conversion.

Listing 2: JITing the 'addition' bytecode

```
switch (filter[i].code) {
case BPF_S_ALU_ADD_X:
   /* add %ebx,%eax */
   EMIT2(0x01, 0xd8);
   break;
case BPF_S_ALU_ADD_K:
   if (!K)
      break;
   if (is_imm8(K))
      /* add imm8,%eax */
      EMIT3(0x83, 0xc0, K);
   else
      /* add imm32,%eax */
      EMIT1_off32(0x05, K);
   break;
   ..
```

For every bytecode instruction passed to the `switch`, instead of interpreting and dispatching the equivalent operation, this minimal JIT compiler emits the x86 opcodes and stores them into a code cache upon running for the first time. For the subsequent filter runs, the code is executed natively from the code cache and bypasses the instruction dispatch mechanism (Figure 1 (c)). This considerably reduces the overhead, as stated before. The main performance gain by JIT*ing* filter bytecode is achieved when the events occur at a high frequency, and run long enough, such as in 'always on' systems. We have used a similar principle for our filtering architecture. Along with micro-optimizations to the BPF system and the usage of the fastest tracing approach, we have proposed

a very fast trace filtering system, as described in subsequent sections.

3.4.1 Trace Filtering

The need for filtering in tracing tools has been addressed before in tools such as DTrace and LTTng. Not tracing and storing uninteresting events becomes a priority when event frequency is high. Filters are applied in the execution path of each tracing event. To reduce the overhead, many systems defer the trace filtering at analysis time. Trace viewing and analysis frameworks such as TraceCompass are optimized for performing complex analysis [27]. Cantrill et al. have discussed the importance of runtime filtering earlier [10]. With a better filtering infrastructure, it is possible to filter out traces at runtime as well. We now discuss some trace filtering approaches the have been used before, and then move on to explain our filtering design in the next section.

Speculative Tracing DTrace provides a filtered tracing mechanism called speculative tracing. The basic idea is to record the trace data tentatively in a separate speculation buffer, and later decide whether to *commit* data to the main tracing buffers or *discard* it based on checking the data with `speculate()` function. An example is shown in [10] and [28] where the authors describe how a filtered trace of all functions entries is only committed if a particular syscall such as `ioctl()` returns a failure. While it is seen as a runtime filtering approach, the speculation involves writing the data to the buffer and possibly copying it to the principal buffer. The DTrace filter execution architecture itself consists of custom bytecode generation and interpretation using a small in-kernel DTrace virtual machine. This predicate condition interpretation, coupled with the data copies, makes the overhead of this approach comparable to that of tools using bytecode interpretation.

LTTng Trace Filtering LTTng works similarly. The expression are converted to bytecode and then interpreted. We take an example of LTTng User Space Tracer (UST) where a filter is set on an event. As shown in figure 4, when the client encounters a filter expression for a specific userspace event to be enabled, it first parses it using a custom lexer-parser and then converts it into a syntax tree.

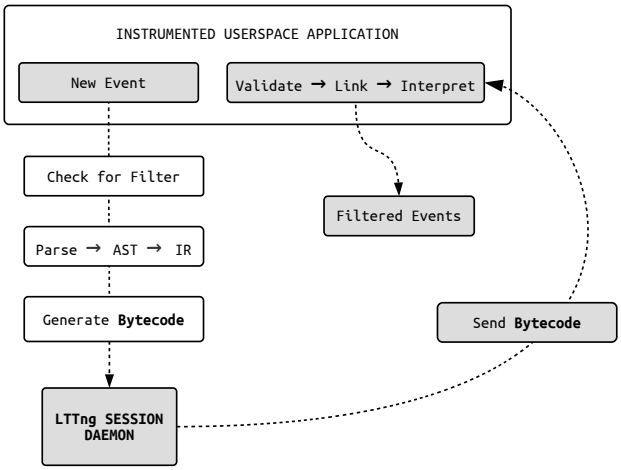

Fig. 4**:** The *client* is responsible for conversion of a filter expression to the bytecode, which though `lttng-sessiond` is sent to the instrumented userspace application for validation, linking and an eventual interpretation per event

The nodes of the syntax tree are visited and classified. Then, the intermediate representation (IR) is generated and a small verification is done on IR. Currently, there is no support for binary arithmetic operations, as the trace filtering needs were very limited. Only logic and comparisons operations are provided. Also, except for logical operators, nesting of other operators is not allowed. The IR is checked to ensure that no wildcard is used in-between string literals and that only valid operators are used. Then, the bytecode is generated by traversing the tree in post-order.

The generated bytecode and data is saved to the context and transmitted to the session daemon, `lttng-sessiond`, which sends the bytecode to the userspace application targeted for event for filtering. There, the bytecode execution process starts. First, the bytecode is linked to the target event to create a *bytecode runtime*. Next, a range overflow check for different instruction classes is done and the bytecode is validated for illegal instructions. Finally, the bytecode is sent to LTTng's own filtering virtual machine.

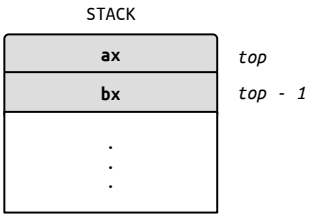

Fig. 5**:** The LTTng interpreter is a stack based with two registers (`ax` and `bx`) aliased to top of stack

LTTng's interpreter is a hybrid stack / register based virtual machine. As seen in figure 5, it is a stack based VM consisting of two registers `ax` and `bx` aliased to the top of stack. This makes operations easy, just like on register based machines, as the push and pops are reduced. At the same time, this gives more flexibility just as for stack machines. The interpreter is a threaded, instruction dispatch based interpreter [29, 30], but can be used as a normal dispatch in scenarios where compiler support is not available.

Listing 3: LTTng's machine interpreting a bytecode

```
OP(FILTER_OP_NE_S64):
{
   int res;
   res = (estack_bx_v != estack_ax_v);
   estack_pop(stack, top, ax, bx);
   estack_ax_v = res;
   next_pc += sizeof(struct binary_op);
   PO;
}
```

Listing 3 shows how the 'signed not-equal-to' operator is interpreted. Here, the operation is performed directly using the macros `estack_bx_v` and `estack_bx_a` which point to the two values to be tested on the execution stack. The LTTng interpreter is quite efficient in relation to the limited scope it has (simple filter execution). However, as we have observed, both by analysis of the source code and through performance numbers discussed in section 6.4, further optimization is possible with the use of JIT*ing*, better optimizations in the bytecode compiler and adding more features (arithmetic operations) to make it more flexible. The overhead is within the range of those tools using bytecode interpretation.

4 Filtered Tracing Architecture

We propose a novel userspace trace filtering architecture, with an improved overall tracing performance, as compared to available tracing tools. We choose LTTng and eBPF as the main drivers for this tracing architecture. We now describe the underlying framework on which our filtered trace architecture is based, and present a justification behind that choice.

4.1 Base Framework

eBPF The idea to convert BPF bytecode to native code, as discussed in Section 3.4, has been exploited recently again by Starovoitov for an improved BPF implementation in Linux kernel. The earlier implementations, also called *classic* BPF in the Linux kernel, consisted of two 32 bit registers – A and K. The conditional branch had two jump targets JT (jump if true) and JF (jump if false). There were 32 bit memory slots for filter data. As the main goal of BPF was packet filtering, there are dedicated 'extensions' where the developer can

load and store data from packets directly. Keeping mind the good performance and the simplicity of BPF, efforts have been ongoing to make it more generic and modern. The newer version called, *extended* BPF (or eBPF) has many improvements. The instruction set has been changed, and was designed with emphasis on the importance of JIT and underlying architectures on which it is run. eBPF now has 10 internal registers and one frame pointer. The calling convention is similar to current architectures, like ARM64 and x86_64, avoiding extra copies in calls [31]. With this calling convention, the eBPF registers also map one to one to the x86_64 and other hardware registers. This simplifies the JIT compiler implementation as well. As the main target of eBPF is a generic kernel interpretation framework, it sports a robust verifier and has a concept of 'BPF maps', an abstract data type to share data between the kernel and userspace. There are various helper functions as well, and a dedicated `bpf()` syscall has been proposed to update and access the maps that the BPF programs keep on updating. However, for tracing purposes in userspace, eBPF needs to be optimized for filtering, so that filtering operations can directly occur in userspace. Our adaptation aims to achieve that. Apart from filtering, our extensions can provide co-operative conditional tracing from userspace.

LTTng The Linux Trace Toolkit next generation (LTTng) is a very fast and extremely low overhead tracing tool developed at DORSAL[1]. With a non-activated tracepoint inserted in the code, it gives near zero impact on the overall execution of the target application. This distinguishes LTTng from the other tools, making it an excellent choice for real time applications. Its tracing technique implements a fast wait-free read-copy-update (RCU) buffer for storing data from tracepoint execution [32]. Its efficiency and scalability was demonstrated in various performance comparisons. LTTng-UST is the userspace tracing counterpart of LTTng. The major factor for such an increase in performance is the use of a lock-less ring buffer in LTTng-UST, as it efficiently manages multiple readers trying to access the same resource simultaneously [21]. However, LTTng-UST still lacks in areas such as providing an improved dynamic tracing mechanism and an efficient filtering mechanism for userspace tracing. Our contributions also lead to a JIT compiler based bytecode for LTTng, in addition to its interpreted filter bytecode, provided by default in userspace. It also provides a base to add an initial support for JIT based kernel filtering as well.

Coupled with eBPF's efficient JIT*ed* filtering technique, LTTng-UST's fast tracing performance can lead to a improved overall performance as compared to interpreted approaches used by DTrace and LTTng's default interpreter. The design of our new eBPF based JIT compiler and interpreter framework, for userspace and kernel trace tracing, is influenced by the network filtering approach for which eBPF was originally designed. Our filtering scheme, however, deviates from this network-centric approach. It aims to provide improved performance specifically for userspace tracepoint filtering, and for combined kernel and userspace tracing. The reach of eBPF usage has also been extended by allowing LLVM/GCC based backends to generate very efficient BPF bytecode from a restricted C interface, while maintaining similar performance.

[1] http://dorsal.polymtl.ca/en

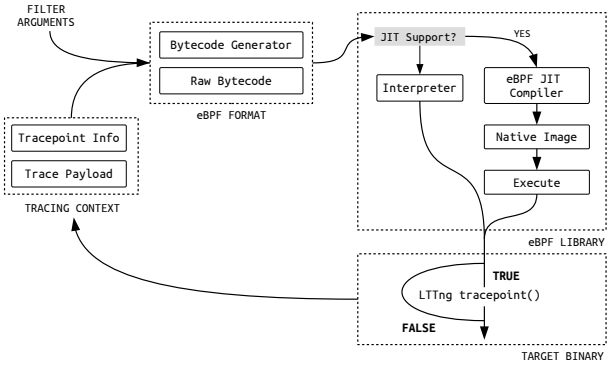

Fig. 6**:** The architecture of our proposed eBPF based trace filtering system

The system architecture is shown in Figure 6. The filter arguments are declared by the user either manually, in eBPF bytecode, or can be generated by the LLVM based backend which converts those simple 'C'-like expressions in eBPF bytecode. The filter also needs information about the trace payload and the tracepoint context, this can be obtained from the target binary in which the filter is run. It can then be fed to our userspace implementation of the eBPF library. The library either checks for the JIT support on the architecture on which it is run, or can be configured to always JIT the bytecode. The JIT*ed* code is saved to a code cache and the filter is run around the `tracepoint()` call. As a fallback, the bytecode could be interpreted if the JIT compilation fails. We now explain in details the various steps taken during filtering, in this proposed architecture.

Bytecode Preparation As stated before, there are 2 ways to provide bytecode to the interpreter (and for later JIT*ing*). In the first mode, the user specifies the filter by hard-coding the eBPF opcode macros such as `BPF_LD_IMM64(BPF_REG_0, 1), BPF_EXIT_INSN()` etc. in the target program, or manually assembling bytecodes for an eBPF program and loading it as shown in Listing 4. This is useful only when the filter is either small of the developer is proficient enough to write BPF assembly manually. The other option is to specify the filter in C and let the recently developed LLVM's eBPF backend generate the eBPF bytecode binary. The compiler converts the eBPF filter, specified in a restrictive C format, to a binary with a `.text` section containing the executable filter eBPF bytecodes. We implemented a small method to extract the opcodes from the section and pass it on to the interpreter or the JIT compiler library. This approach is beneficial because there is opportunity for the developer to use optimization routines from the LLVM tools.

Listing 4: eBPF program for a sample filter

```
    ldd r1, (0)r1
    mov r2, 42
    jeq r1, r2 goto TRUE
    mov r0, 0
    ret
TRUE:
    mov r0, 1
    ret
```

We now discuss some characteristics of the bytecode itself. As mentioned earlier in Section 4.1 the newer bytecode of eBPF is closer to native architectures like the x86. Using a similar format leads to a more uniform and portable design. The register layout is shown in Table 1, derived from the filter documentation in the kernel [31].

Table 1: Register mapping for eBPF-x86

eBPF	*x86*	*Purpose*
R0	`rax`	Return value from function / exit value from eBPF
R1	`rdi`	First argument
R2	`rsi`	Second argument
R3	`rdx`	Third argument
R4	`rcx`	Fourth argument
R5	`r8`	Fifth argument
R6	`rbx`	Callee saved
R7	`r13`	Callee saved
R8	`r14`	Callee saved
R9	`r15`	Callee saved
R10	`rbp`	Frame pointer

The R0 register in eBPF is where the exit value from eBPF programs is stored. Upon eBPF program return, R0 is set 1 for TRUE and 0 for FALSE, just as shown in Listing 4. Register R1 is where the filter context is loaded. For example, the context is often made available to the target program through a structure, filled with arguments on which filtering is to be performed. These arguments can be the payload fields from the LTTng tracepoint or, for more complex scenarios, these values can be obtained at runtime (LTTng's context such as PID/TID). A pointer to this structure can be passed in register R1, which is then accessed as filter context by the eBPF program. In addition, tracing filters regularly need to compare strings, since several tracepoint payload fields are formatted as strings (e.g., the filename in the `open()` syscall). We thus implemented a `bpf_strcmp()` function which can be called from within the eBPF code. Such helper functions make eBPF filter programs more flexible. As discussed later in section 5.3, we used these helper functions to further extend the filtering system.

Native Code Compilation The main feature of the system, and the leading reason for improved performance, is the JITing of the bytecode. The JIT compiler behavior is illustrated in Listing 2. The JIT compilation process for this library is a simple one-to-one JIT, for each instruction (or group of eBPF instructions) there is a direct translation to native code instructions. The compiler backend is non-optimizing. Indeed, the LLVM `clang` compiler frontend performs most of the interesting optimizations, before sending the intermediate representation to the bytecode generation backend. The native code then follows closely the generated bytecode. For illustrative purposes, in Listing 5, we explain the machine code compilation for Listing 4 on an x86-64 system.

Listing 5: JITed eBPF program for sample filter

```
0   push %rbp                          }
1   mov %rsp, %rbp                     }  (1)
4   sub $0x228, %rsp                   }
b   mov %rbx, -0x228(%rbp)             }
12  mov %r13, -0x220(%rbp)             }  (2)
19  mov %r14, -0x218(%rbp)             }
20  mov %r15, -0x210(%rbp)             }
27  xor %rax, %rax                     }  (3)
29  xor %r13, %r13                     }
2c  mov (%rdi), %rdi                   }
30  mov $x2a, %rsi                     }  (4)
3a  cmp %rsi, %rdi                     }
3d  jz  0x4b                           }
3f  mov $0x0, %rax                     }  (5)
49  jmp 0x55                           }
4b  mov $0x1, %rax                     }
55  mov -0x228(%rbp), %rbx             }
5c  mov -0x220(%rbp), %r13             }
63  mov -0x218(%rbp), %r14             }  (6)
6a  mov -0x210(%rbp), %r15             }
71  leave                              }
72  ret                                }
```

The compiler first *emits* some standard instructions to build the function preamble (1). Some variables are allocated on the stack as well

for later use, and the values of callee saved registers are saved (2). This is a standard preparation for a JITed filter binary. eBPF's R0 and R7 registers, used as the old A and K registers, are cleared (3). The filter context value supplied in R1 (`rdi`) is loaded and compared with a predefined value (4). Based on this comparison, 0 or 1 is loaded in R0 (`rax`) and a jump to the exit routine is taken (5). A standard set of bytecodes is also emitted for the exit, the callee saved registers are restored and the filter function is exited (6).

Deviating from the Linux kernel's eBPF approach, our eBPF library is lighter, and the JIT compiler faster, by not including support for special instructions that perform direct computations in the kernel on network packet data structures. Since we do not need the BPF map data structures, the compiler and interpreter now being in the userspace, these were removed as well. Instead, to extend the filtering library to a generic assisted-tracing library, we propose our own shared-memory based communication system between the kernel and userspace eBPFs, as detailed in section 5.3. For filtering, the native code also supports calls to new helper routines for tracing specific string comparison functions, such as `bpf_strcmp`. The architecture is kept flexible, so that other helper functions can be added as desired.

We tested the performance of the filter, and the filtered tracing architecture, in relation to various factors such as filter execution speed, and compared it with the performance of LTTng's userspace trace filtering system based on bytecodes. For practical reasons, because of the limitations of the C pre-processor for defining variable length argument lists, LTTng's bytecode filter currently limits the number of filter predicates to 10, which limited us for our test cases. However, the design of eBPF based filters has no such restrictions for similar tests. For now, the number of instructions that can be executed with eBPF is kept at 4096, with support for tail-calls so that multiple filters can be chained as desired. For a similar filter predicate type, we could filter on 50 predicates with our design, as compared to 10 with LTTng's current interpreted filter, in the tests that we performed. The design of the experiments and our findings are elaborated in Sections 6 and 6.4.

5 Improved Tracing Infrastructure

In the previous sections, we discussed how eBPF and LTTng can be used to develop a new and efficient filtered tracing architecture. We now explore the use of eBPF to provide a new way of performing dynamic tracing in the kernel. We eventually propose and present a new co-operative kernel-userspace tracing system, which supports dynamically defining conditional tracing, and a more efficient data sharing mechanism. We now briefly describe similar approaches taken by other tools.

5.1 Dynamic Tracing

Some of the most interesting developments in the tracing infrastructure has been the ability to dynamically insert tracing probes and take actions when those dynamic probes are hit. The dynamic tracing tools at kernel level are available with different granularity. One of the approaches that has proven to be very flexible is defining a scripted tracing language, which is dynamically compiled at runtime to some IR or bytecode, and then intended to be interpreted in-kernel. Based on the instructions, certain 'functions' or 'actions' can be executed to gather data into buffers, to be read later from the userspace. Some famous examples are ProbeVue [33] and DTrace [28]. They provided scripting languages like D and Vue which would

be compiled to an intermediate format. For example, in the case of DTrace, the D program input (such as the one shown in Listing 6), though the `dtrace` command or a userspace application, would go through the same process of lex-parse to generate the parse tree, and then be compiled to a D Intermediate Format (DIF). A visual survey of the DTrace code reveals that the DTrace compiler offers very limited optimizations (integer constant folding and peephole optimization) as compared to the enhanced optimizations performed in LLVM for eBPF bytecode.

Listing 6: Sample D script [28]

```
syscall::write:entry
/execname == "foo" && uid == 1001/
{
   self->me = 1;
}
```

This DIF would be compiled by the assembler to the DIF Object (DIFO) as shown in Listing 7.

Listing 7: The D Script in Listing 6 compiled to a DIFO [28]

```
OFF OPCODE   INSTRUCTION
00: 25000001 setx DT_INTEGER[0], %r1 ! 0x1
01: 2d050001 stts %r1, DT_VAR(1280) !
                  DT_VAR(1280) = "me"
02: 23000001 ret %r1
```

This is then coupled with data tables (strings and variables) to form the DTrace Object Format (DOF) – which is the actual bytecode interpreted by the in-kernel DTrace VM. The VM is a RISC machine with a fixed register set. The instruction length is fixed to 4 bytes. To retrieve values from the kernel, DTrace provides a driver that communicates with a userspace library (which can be used with other DTrace consumers like `lockstat` and `intrstat`). This is one of the most comprehensive dynamic tracing infrastructure available. However, it requires a custom VM in-kernel, and the interpretation cost can be high for long running or badly written scripts. Another approach was that of SystemTap, where the SystemTap scripts would be translated to pure C language and then compiled as kernel modules. These could then be loaded at runtime in the kernel to provide tracing support. It eliminates the need for an in-kernel VM, but the cost of tracepoint executions and data accesses has been high as compared to other dynamic tools [34].

5.2 Data Sharing

Apart from the cost of the tracepoint execution, the cost of collecting and aggregating data is an important consideration as well. Most tools employ a *producer-consumer* design where the trace events can send data (producer) to a buffer (either in kernel or userspace), and the filled buffers become available (in userspace) for analysis, storage or display (consumer). Neira-Ayuso et al. have discussed various kernel-userspace data sharing mechanisms before [35]. For very large data bandwidth, the best strategy is to minimize the number of context switches or syscalls. In DTrace, the `libdtrace` library is responsible for consuming data retrieved from the buffers at probe execution. DTrace provides per-CPU buffers in the kernel that are filled with relevant data. Based on the `ioctl()` arguments in the library, an action is taken on the buffer, such as copying data to the relevant userspace buffer. LTTng provides very efficient shared memory per-CPU ring buffers for one-way sharing of data from userspace to kernel. With support for Kprobes as well, it is an efficient dynamic tracing system for the kernel. However, there is no specific tracing script support in LTTng, for more advanced analysis or aggregation, as can be done with tools like DTrace. In its current form, eBPF allows two-way sharing of data (in BPF maps) from userspace to kernel, based on

the `bpf` syscall. Refer figure 7.

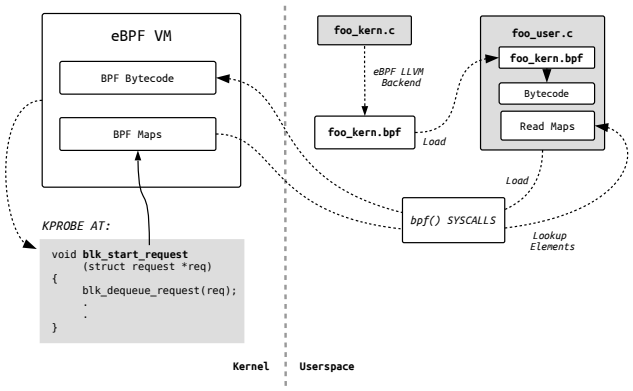

Fig. 7: An eBPF program in its current form, with the kernel part (`foo_kern.c`) and a userspace part (`foo_user.c`). The userspace part uses the `bpf()` syscall to load bytecode in the eBPF kernel VM, as well as reading and updating data in *BPF maps*

Aggregated or filtered values, stored in hash-tables or array-maps, can also be accessed and updated directly from within an eBPF program bytecode, using the `BPF_CALL` instruction and helper functions, since the program is already in kernel context. Even though eBPF is efficient and flexible, as it can be dynamically compiled and be used to aggregate data, it would benefit from a more efficient way to transfer data. eBPF itself is not a complete tracer but an infrastructure upon which tracers can be built. The main benefit of eBPF is that it generates dynamically compiled, JIT*ed* code for tracing. With a more efficient data sharing system, used cooperatively with the LTTng tracing system, it can provide an overall benefit, in terms of speed as well as flexibility, to scripted tracing. The resulting system provides a better tracing infrastructure than offered by currently available tools. We now discuss our co-operative tracing approach based on eBPF and LTTng.

5.3 KeBPF and UeBPF Interactions

Now that we have a system to dynamically execute JIT compiled code in our programs, we can think beyond just filtering, and make decisions and take 'actions' based on aggregated values, in kernel as well as userspace. On the kernel side, this effort is presently ongoing in the form of small eBPF scripts that can aggregate data and share it with userspace [36]. Refer to Figure 7. The Kernel eBPF (KeBPF) machine provides access to the shared values in the form of array-maps or hash tables using a syscall. The user can decide to perform aggregations on the values in hash tables in kernel and concurrently read them from userspace. With some effort, eBPF programs can also be used to take decisions based on remembered state (e.g., aggregated values). Our implementation of Userspace eBPF (UeBPF) as a library opens new possibilities to collaboratively trace and share data from userspace to kernel and vice-versa.

5.4 Illustrative Use Case

We show the importance of interaction between KeBPF and UeBPF programs using an example. For diagnosing system performance, it can be beneficial for the user to track the latency of syscalls issued by a particular userspace process. For that, we developed a custom module where the userspace process registers itself using some `ioctl()` and then probe the `sys_enter` and `sys_exit` trace events along with the time-stamps for each syscall. We could then compute, for each syscall, how much time the syscall was taking and thus track the particular syscall latency. We can keep track of all syscalls and set a threshold to decide when to record an event or not, based on the syscall latency threshold. If the elapsed time for a syscall is more than the threshold, the event can be recorded, or otherwise be discarded. However,

the latency threshold should not be the same for each syscall. It can vary from syscall to syscall and can vary based on the complexity of the request and the underlying hardware speed. We can therefore add specific *hooks* in the userspace application which can specify expected thresholds to the kernel. These hooks then can be set from within eBPF programs so that the user can dynamically change the threshold values even at function granularity.

On the kernel side, the kernel can share data with the userspace application to *assist* it in tracing, based on conditions such as checking if CPUs have been switched, if we are in a blocking state while waiting for a device etc. All such *process states* can be shared and the process can conditionally decide to trace or not based on these conditions. This requires a fast data sharing mechanism between the KeBPF and UeBPF programs, for minimum overhead. We therefore implemented a `mmap` based shared memory, between kernel and userspace, so that KeBPF and UeBPF programs can share data directly. Other approaches, such a Perf based events and LTTng's data sharing, use fast shared memory as well, however only in the context of tracing data, flowing from the producer to the consumer. Also, as discussed before, SystemTap and DTrace are limited in how variables can be shared between different probes executed in kernel mode, and offer no way of executing code in userspace and thus for communicating with such code. DTrace's buffers are accessible from the userspace `libdtrace` library, but this involve copies from kernel buffers.

Our sharing is between two VMs (KeBPF and UeBPF). Therefore, there is a direct access to take decisions on tracing from both sides, right at the bytecode level. In that context, a shared memory access enables very efficient communications, and useful usage scenarios. Coming back to the example, as shown in Figure 8, we have a process with PID 42 that registers with our syscall latency tracker module.

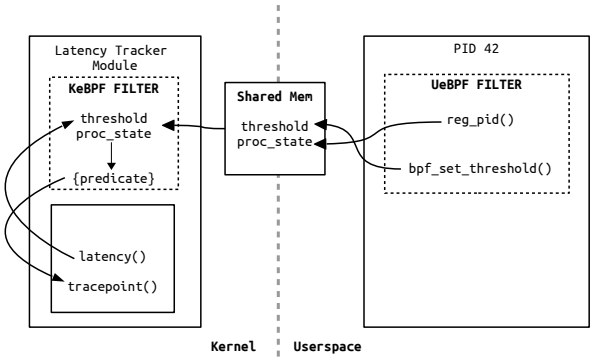

Fig. 8**:** The KeBPF-UeBPF shared memory implementation showing syscall latency thresholds being set dynamically from within a UeBPF filter program

The process contains a UeBPF filter attached to certain function level *hooks* in the application (as discussed before) which calls our implemented eBPF helper function `bpf_set_threshold()`. This helper function, when called from within the UeBPF filter, writes the updated threshold for the given process/syscall in a shared memory mapped location, shared with KeBPF. This way, the thresholds can be dynamically adjusted, and kernel tracing output can be controlled. In addition to this, the userspace can continually fill the `proc_state` structure with current *process state* so as to control other parts of kernel/userspace tracing. The alternative and default way, as of now, is to use the `bpf()` syscall and create/update BPF maps from userspace. Each `bpf()` syscall, however, incurs more cost for the same operation, as compared to a direct read or write in our shared memory. In addition, in the default eBPF maps, each value requires an explicit copy in the kernel from the userspace, which is avoided in our shared memory approach.

In our tracing approach, kernel and userspace scripts are each executed in their respective con-

text. This enables very fast userspace tracing, avoiding context switches or traps at each userspace tracepoint. This is the reason behind the unrivaled performance of the LTTng userspace library. The excellent communication performance between userspace and kernel space, enabled by the shared memory implementation, then opens up a lot of possibilities, as will be shown in the experiments, because of this high-performance architecture. There are, however, precautions required for using this shared memory channel. From the security point of view, the kernel scripts should treat appropriately these userspace supplied values. Furthermore, appropriate synchronization mechanisms must be used, depending on the access protocol.

For single-threaded synchronous access, no synchronization is required. For instance, a userspace script, executed from a single-threaded process, may specify a threshold just before the application issues a system call. Upon finishing the system call, while the application is still blocked, a kernel script would check if the threshold was exceeded, in which case it could write a stack dump to the trace. In this scenario, no synchronization is needed.

There are several cases where thread-level storage can also avoid synchronization issues. Thread-level storage can easily be built using arrays indexed by the thread id, or using similar mechanisms. One common scenario is aggregating counts (e.g., number of bytes read, number of packets received). This could lead to severe scalability problems if a single global variable protected by a lock was used. Instead, one variable per thread (or per core) is typically used and no synchronization is required. The variables can then be read and aggregated at the end, once the scripts are deactivated, not being concurrently updated any more. Alternatively, the variables can be read, even while they are being incremented, as accesses to aligned, word-size, variables are atomic.

For shared, concurrently accessed, global variables, the situation is more problematic. For instance, tracers in probe handlers either avoid any locking, like LTTng with atomic lockless operations, or only allow probes in specific contexts where locking is possible. For example, SystemTap limits the context where probes can be inserted, avoiding NMI interrupts for instance, and automatically protects accesses to global variables with locks. Our implementation currently does not impose any particular access scheme or locking protocol. The userspace RCU algorithms would be applicable to a mixed kernel and userspace environment, but the current URCU library implementation would need to be extended to communicate with a kernel counterpart [32, 37].

Our proposed approach allows a direct link between two VMs, one in userspace and one in kernel, to aggregate data, share data with zero copy overhead, and set filtered tracing and conditional actions for each other. Results and inferences from our performance tests on our shared memory implementation, for co-operative KeBPF-UeBPF tracing, are presented in the next section.

6 Experimentation

In order to demonstrate the effectiveness of the proposed architecture and algorithms, we divide the experimentations into two sets. The first set focuses on the pure performance of native code filters, and their performance when tracing is enabled with varying parameters. The second set evaluates how our shared memory implementation performs as compared to the `bpf()` syscall based approach, used by the default in-kernel eBPF implementation.

6.1 Test Environment

All tests were done on a machine running Fedora 20 with default 64-bit kernel 3.15 and the eBPF patched kernel 3.17-rc7 for kernel eBPF tests. We used LTTng v2.6 on our workstation running an Intel i7-3770 featuring 4 cores, with hyper-threading disabled, and 16 GB of memory, for tracing and observing its interpreter performance.

6.2 Filter Experiment Set

There are multiple factors on which the trace filters performance can be measured. The most important is overhead, which can be defined as the extra time or effort required to complete a task when an external factor acts upon a control experimental setup. In terms of tracing/filtering, the time taken due to the addition of tracing and filtering can be compared to a baseline value (normal execution time of the target process). This extra time is the overhead and is the primary measure of the impact caused by any proposed addition to the tracing system. To evaluate the performance, we designed a synthetic benchmark with *operator chaining*. As shown in Figure 9, The filter predicates $(P_1, P_2..P_N)$ are simple string comparisons connected with a boolean operator $(\star)$, which is usually an AND/OR. The important time measurements for us are the time required to build and setup the filter (t_K), the time to evaluate the filter (t_e) and, upon evaluation, the time taken to execute the tracepoint code (t_t). Thus, the total time relevant for our observations is,

$$T = t_K + t_e + t_t \tag{1}$$

To evaluate t_e, we took AND/OR operator chained predicates, doing string comparisons, and observed them for a varying number of events, under a biased condition (the filter always returned TRUE). Refer Figure 10. This measured the performance of eBPF JITed vs interpreted and hardcoded filters, so that we could understand better how the native compiled filters in userspace were functioning. We then devised another comprehensive test to compare AND/OR chained predicates, doing string comparisons with varying *depth-of-evaluation (DoE)*, for a 100 million events run. Varying the DoE meant that the filter was evaluating to FALSE, and skipping the remaining predicates, after P_X predicates. This is the same as having a filter length equal to the position of the P_X predicate, and the filter evaluating to TRUE. We varied the DoE from $P = 5$ to $P = 40$ with steps of 5. Refer to Figure 11. In the following test, we included the tracepoint time factor t_t as well. For this, we used similar tests and varied the events and the number of predicates, but the filter was kept biased as TRUE, so that the tracepoint was called and we could measure t_t. This gave us the total $(t_e + t_t)$, needed to fully characterize our system. We have neglected t_K. The intended use case is high performance trace filters, with high frequency events observed over long durations. The preparation time for filters is thus amortized over a large number of executions, and is negligible under such conditions.

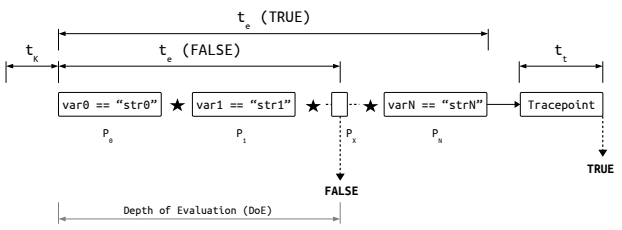

Fig. 9**:** Our trace filter test design

In this same experiment, we compared the time taken by similar LTTng-UST's interpreted filters with our eBPF interpreted and JITed approach. However, we limited the number of predicates to only 9 variables, due to LTTng currently

not allowing more than 9 distinct string variables as trace payload. Refer to Figure 12 for results.

6.3 Shared Memory Experiment Set

For this experiment set, we created a synthetic benchmark to evaluate the performance of our KeBPF-UeBPF shared memory implementation and see how it compares with the default syscall based sharing system used in KeBPF. We started off by creating an eBPF program which populates an eBPF array-map with 1000 integers with random values. We then lookup these values from userspace using the `bpf()` syscall with arguments `BPF_MAP_LOOKUP_ELEM` and measured the time for multiple runs. We compare this with the time taken to read the same values updated in Kernel eBPF using our shared memory, and then read from userspace using a simple `read()`, or a helper function in UeBPF which calls `read()`. The default `bpf()` implementation took 218ns/read whereas our shared memory implementation took 2.2ns/read, which is explained by the fact that the shared memory can be directly accessed.

6.4 Results and Inferences

In the first test case for filter optimizations, also presented in Figure 10, we observe that for 100 million events and a 50 predicates filter, the interpreted eBPF filter is 4.3x slower than the hard-coded filter (considered as a lower bound reference). The native compiled eBPF filter, however, is only 1.4x slower than the hard-coded reference. Even though the JIT compiled filter performance is expected to be similar to that of the actual hard-coded filter, since both are executing native machine code, we can see that this small overhead is due to extra instructions being executed for each filter, as seen in Listing 5. The overall filter performance is consistent for 1M and 10M events, indicating that there would be a consistent filter overhead reduction in the 3x range, when using native compiled eBPF filters, as compared to similar interpreted filters, for tracing scenarios with long predicates and high event frequency.

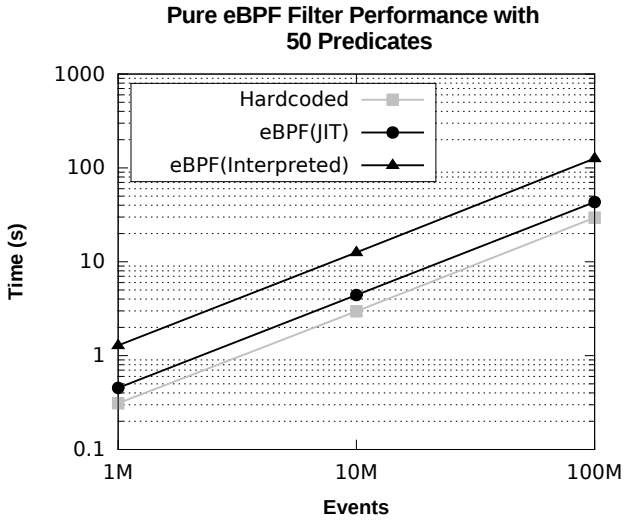

Fig. 10**:** Pure eBPF filter performance with 50 predicate TRUE biased AND chain

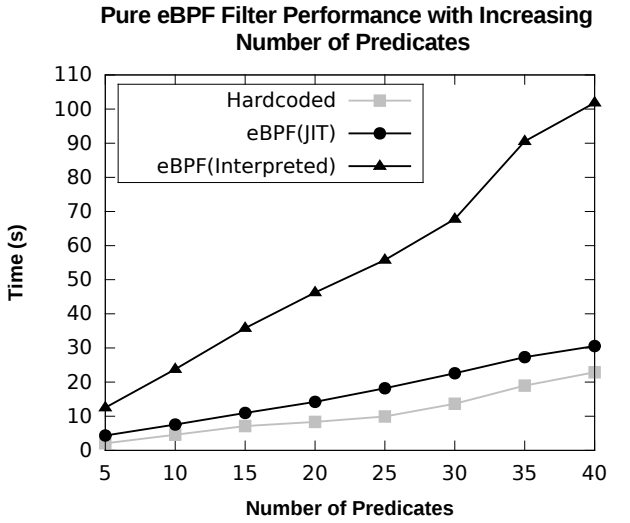

Fig. 11**:** Pure eBPF filter performance with 100M events and a TRUE biased AND chain

In the second test case, as shown in Figure 11, we observe that with a constant 100 million events and an increasing number of predicates, the

benefit of natively compiled eBPF trace filters increases marginally. The performance of a 10 predicate JIT*ed* eBPF filter was 3.1x better than a similar interpreted filter. This increased to 3.2x for 20 predicates, and a little over 3.3x for 40 predicates. This shows that even for filters with unusually long predicate chains, the performance was consistent with that of natively compiled filters.

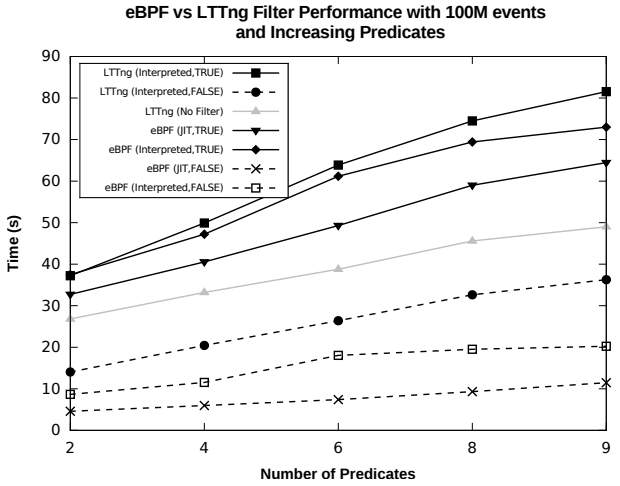

Fig. 12**:** eBPF vs LTTng's filter performance with increasing number of TRUE/FALSE biased AND chain predicates

In the third test scenario, we compared LTTng's interpreted filter performance with that of eBPF's JITed filter performance by observing the biased FALSE cases in Figure 12. For 9 predicates, eBPF's JITed filter was 3.1x faster than a similar LTTng's interpreted filter. We further observed that eBPF's interpreted filter itself was 1.8x faster than LTTng's interpreted filter, pointing to a better register-based eBPF interpreter. We then proceeded to see how these filters fared with LTTng tracing enabled. In that case, if the filter evaluated to TRUE, the tracepoint was recorded and the observed time included the tracepoint time. We choose to compare our observations with the LTTng (No Filter) mode as the reference line, where no filter was set and all tracepoints were recorded. For a filter of 9 AND chained string comparisons, biased to TRUE, the interpreted LTTng had an overhead of 325ns/event, as compared to JIT*ed* eBPF filter's 154ns/event, when LTTng (No Filter) was taken as reference. Our JIT*ed* approach was thus 2.1x faster. This demonstrates the fact that, with the small cost of JIT*ed* filtering with our library (154ns/event in this case), the user can implement filtering at little cost to potentially save a lot of resources by cutting down on unnecessary events that can easily be filtered.

In the case of KeBPF-UeBPF shared memory implementation, we got an overall improvement of 99x over the default implementation. The time taken by 1000 reads of an integer array-map is shown in Table 2.

Table 2: Time taken for 1000 reads of an integer array-map

	$Time(ns)$	$StdDev$
Baseline	2120	210
eBPF-shm	2247	984
eBPF-syscall	218203	801

Our *eBPF-shm* shared memory is close to the baseline values taken using simple read calls. *eBPF-syscall*, in Table 2, shows the time taken to read using the `bpf()` syscall. Going through the eBPF code in the kernel shows that a similar process, of using this syscall to update a map value in the kernel, would incur a syscall time as well as the time involved in copying the value to the kernel space. In our shared memory system, however, there would be no extra copy involved, and KeBPF and UeBPF can share data directly at a high speed, as observed in our test.

7 Conclusion and Future Work

This paper presents two contributions to tracing techniques in userspace. First, we improved the trace filtering mechanism by using Just-In-Time (JIT) compilation to convert trace filter bytecode to native machine code. We used the Linux kernel's eBPF based bytecode technique, and improved it for tracing in userspace context. We targeted LTTng-UST as the tracer, due to its low overhead, and observed that our native filtering approach surpasses the filtering performance of similar high performance state-of-the-art tools. We show that, with our technique, we can filter traces in record time to have smaller traces and provide more efficient tracing, in long running high frequency in production tracing scenarios, such as embedded soft-realtime systems and networked nodes.

As a second contribution, we developed a shared memory system between the default Kernel eBPF and our Userspace eBPF, by extending the eBPF system at both levels. This enables sharing data at greater speeds and using it to do *assisted tracing* from kernel to userspace and vice-versa. We demonstrated this using a basic syscall latency tracing example, where the thresholds could be dynamically adjusted at function level granularity using *hooks* in the userspace application, right from within UeBPF. KeBPF could then access it to make decisions on recording or discarding syscall events. The interaction between a kernel VM and a userspace VM is significant as it allows a direct interaction between decision making sections. Along with the benefit of zero-copy overhead, it provides flexibility for performing conditional actions on kernel-userspace shared data - such as performance counter values, process states (off-CPU state, wait threshold, syscall latency threshold, resources thresholds etc.).

There are some limitations in our current approach, however, which will motivate some of our future work. We have observed that very specific and long filter predicate usecases in trace filtering can have a negative effect. The overall trace becomes small, and important events which give a context to the tracing scenario get missed out. To overcome this, we can use profile-guided tracing where the generated bytecode can perform some non intrusive profiling on the tracing, get some feedback and also record traces which are relevant to the filter scenario - even if it does not statisfy the filter condition. Triggers for system-wide (kernel+userspace) tracing can be defined and, when enabled, in addition to the intended filtered tracepoint, would also record a system-wide trace for some predefined or dynamically defined duration. We can also utilize LLVM's compiler infrastructure to support some high level meta language to define tracing specific scripts, and move towards traditional script based filtering when required, while keeping all the benefits of low overhead and speed provided by LTTng. The UeBPF library also could benefit from more explicit support for data sharing through multiple threads. In some specific usecases, it may also be worthwhile to investigate hardware based trace filtering, where an eBPF machine would be implemented not just as a JIT compiler but as specialized hardware. Our current userspace eBPF implementation could also be extended to provide support for program flow tracing, such as with Intel Processor Trace (PT) [38], where eBPF programs from userspace could conditionally trigger a PT.

Acknowledgment We would also like to thank Ericsson, EfficiOS, NSERC and Prompt for funding, LTTng project authors for their technical guidance and Francis Giraldeau for his comments and code review.

References

[1] Ball T, Burckhardt S, de Halleux J, Musuvathi M, Qadeer S. Deconstructing concurrency heisenbugs. In: *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. 2009; pp. 403–404.

[2] Bligh M, Desnoyers M, Schultz R. Linux Kernel Debugging on Google-sized clusters. In: *Proc. of 2007 Ottawa Linux Symposium*. Kernel.org. 2007; p. 290.

[3] Ezzati Jivan N. Multi-Level Trace Abstraction, Linking and Display. Ph.D. thesis, École Polytechnique de Montréal. 2014.

[4] Starovoitov, Alexei. Tracing: accelerate tracing filters with BPF [LWN.net]. `http://lwn.net/Articles/598545/`.

[5] McCanne S, Jacobson V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93. Berkeley, CA, USA: USENIX Association. 1993; pp. 2–2.
URL `http://dl.acm.org/citation.cfm?id=1267303.1267305`

[6] Bailey ML, Gopal B, Pagels MA, Peterson LL, Sarkar P. PathFinder: A Pattern-Based Packet Classifier. In: *Proceedings of the First Symposium on Operating Systems Design and Implementation*. Citeseerx. 1994; pp. 115–123.

[7] Engler DR, Kaashoek MF. DPF: Fast, flexible message demultiplexing using dynamic code generation. In: *ACM SIGCOMM Computer Communication Review*, vol. 26. ACM. 1996; pp. 53–59.

[8] Begel A, McCanne S, Graham SL. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In: *ACM SIGCOMM Computer Communication Review*, vol. 29. ACM. 1999; pp. 123–134.

[9] Wu Z, Xie M, Wang H. Design and Implementation of a Fast Dynamic Packet Filter. *Networking, IEEE/ACM Transactions on*. 2011; 19(5):1405–1419.

[10] Cantrill BM, Shapiro MW, Leventhal AH. Dynamic Instrumentation of Production Systems. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04. Berkeley, CA, USA: USENIX Association. 2004; pp. 2–2.
URL `http://dl.acm.org/citation.cfm?id=1247415.1247417`

[11] Jacob B, Larson P, Leitao BH, da Silva SAMM. *SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems*. IBM Redpaper. 2009.

[12] GNU GCC Documentation. `https://gcc.gnu.org/onlinedocs/gcc-4.3.4/gcc/Code-Gen-Options.html`.

[13] Rostedt S. Using the TRACE_EVENT() macro (Part 1) [LWN.net]. `http://lwn.net/Articles/379903/`.

[14] Keniston J, Panchamukhi PS, Hiramatsu M. Kernel Probes (Kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[15] Hiramatsu M. The Enhancement of Kernel Probing - Kprobes Jump Optimization. In: *LinuxCon*. 2010; .

[16] Debugging with GDB: In-Process Agent. `https://sourceware.org/gdb/current/`

onlinedocs/gdb/In_002dProcess-Agent.html.

[17] Brown A, Wilson G. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. The Architecture of Open Source Applications. CreativeCommons. 2011.
URL http://books.google.ca/books?id=pgI1AwAAQBAJ

[18] Nethercote N, Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.* 2007; 42(6):89–100.
URL http://doi.acm.org/10.1145/1273442.1250746

[19] Valgrind Manual. http://valgrind.org/docs/manual/manual.html.

[20] Buck B, Hollingsworth JK. An API for Runtime Code Patching. *Int J High Perform Comput Appl.* 2000;14(4):317–329.
URL http://dx.doi.org/10.1177/109434200001400404

[21] Goulet D. Unified Kernel/User-Space Efficient Linux Tracing Architecture. Master's thesis, École Polytechnique de Montréal. 2012.
URL http://publications.polymtl.ca/842/

[22] Prasad V, Cohen W, Eigler FC, Hunt M, Keniston J, Chen J. Locating system problems using dynamic instrumentation. In: *Proc. of 2005 Ottawa Linux Symposium*. Citeseer. 2005; p. 494.

[23] Kim T, Zeldovich N. Practical and Effective Sandboxing for Non-root Users. In: *USENIX Annual Technical Conference*. 2013; pp. 139–144.

[24] Corbet J. BPF: the universal in-kernel virtual machine [LWN.net]. http://lwn.net/Articles/599755/.

[25] Shi Y, Casey K, Ertl MA, Gregg D. Virtual Machine Showdown: Stack Versus Registers. *ACM Trans Archit Code Optim.* 2008; 4(4):2:1–2:36.
URL http://doi.acm.org/10.1145/1328195.1328197

[26] Davis B, Beatty A, Casey K, Gregg D, Waldron J. The Case for Virtual Register Machines. In: *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03. New York, NY, USA: ACM. 2003; pp. 41–49.
URL http://doi.acm.org/10.1145/858570.858575

[27] Gebai M, Giraldeau F, Dagenais M. Fine-grained preemption analysis for latency investigation across virtual machines. *Journal of Cloud Computing.* 2014;3(1):23.
URL http://dx.doi.org/10.1186/s13677-014-0023-3

[28] McDougall R, Mauro J, Gregg B. *Solaris Performance and Tools(c) Dtrace and Mdb Techniques for Solaris 10 and Opensolaris.* Prentice Hall. 2006.

[29] Ertl MA, Gregg D. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In: *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01. London, UK, UK: Springer-Verlag. 2001; pp. 403–412.
URL http://dl.acm.org/citation.cfm?id=646666.699857

[30] Gagnon EM, Hendren LJ. SableVM: A Research Framework for the Efficient Execution

of Java Bytecode. In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01. Berkeley, CA, USA: USENIX Association. 2001; pp. 3–3.
URL `http://dl.acm.org/citation.cfm?id=1267847.1267850`

[31] Schulist J, Borkmann D, Starovoitov A. Linux Socket Filtering aka Berkeley Packet Filter (BPF). `https://www.kernel.org/doc/Documentation/networking/filter.txt`.

[32] Desnoyers M, McKenney PE, Stern AS, Dagenais MR, Walpole J. User-Level Implementations of Read-Copy Update. *IEEE Trans Parallel Distrib Syst*. 2012;23(2):375–382.
URL `http://dx.doi.org/10.1109/TPDS.2011.159`

[33] Lascu O, Bodily S, Harvala M, Singh AK, Song D, Berg FVD. *IBM AIX Continuous Availability Features*. IBM Redpaper. 2008.

[34] Beamonte R, Dagenais MR. Linux Low-Latency Tracing for Multicore Hard Real-Time Systems. *Advances in Computer Engineering*. 2015;2015:Article ID 261094.

[35] Neira-Ayuso P, Gasca RM, Lefevre L. Communicating Between the Kernel and User-space in Linux Using Netlink Sockets. *Softw Pract Exper*. 2010;40(9):797–810.
URL `http://dx.doi.org/10.1002/spe.v40:9`

[36] Gregg B. eBPF: One Small Step. `http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html`.

[37] McKenney PE. Is Parallel Programming Hard, And, If So, What Can You Do About It? *Linux Technology Center, IBM Beaverton*. 2015;.
URL `https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html`

[38] Reinders J. Processor Tracing. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`.