UNIVERSITÉ DE MONTRÉAL

LOW-IMPACT SYSTEM PERFORMANCE ANALYSIS USING HARDWARE ASSISTED TRACING TECHNIQUES

SUCHAKRAPANI DATT SHARMA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
FÉVRIER 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

LOW-IMPACT SYSTEM PERFORMANCE ANALYSIS USING HARDWARE ASSISTED TRACING TECHNIQUES

présentée par : SHARMA Suchakrapani Datt
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. QUINTERO Alejandro, Doctorat, président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BOYER François-Raymond, Ph. D., membre
M. GAGNON Étienne M., Ph. D., membre externe

DEDICATION

"Somewhere, something incredible is waiting to be known"

- Carl Sagan

Dedicated to the unknown cosmos.

ACKNOWLEDGEMENTS

I would first like to express my sincere gratitude to my research adviser and supervisor, Prof. Michel Dagenais, for giving me this opportunity to work under his guidance and helping me grow intellectually and personally. His interest in my work and passion in the field of research helped me steer through the largest of hurdles.

I would also like to thank Francis Giraldeau who has been a professional companion and a dear friend for guiding me through the ups and downs of the PhD life. He welcomed me to this beautiful land of Québec and advised me from time to time with his vast experience and knowledge of computers. He has been a personal inspiration to me. I am also grateful to Genevieve Bastien for her support during my research. Her advice has been a key factor in development of technologies resulting from this work. A thanks is also due to Tanushri Chakravorty for being around for yet another graduation journey. Her moral support has been invaluable.

I would like to thank Mathieu Desnoyers and Julien Desfossez for their work on tracers in Linux and giving me an opportunity to discuss Linux internals with them. I am also thankful to EfficiOS, Ericsson, the Natural Sciences and Engineering Research Council of Canada (NSERC) and prompt Québec for providing funds to carry out this research.

I express my gratitude to all past and present *inhabitants* of the DORSAL Lab with whom I periodically engaged in intellectual exchanges – in particular, Simon Marchi, François Rajotte, Raphaël Beamonte, Hani Nemati, Mohamad Gebai, Houssem Daoud, Naser-Ezzati Jivan, Matthew Khouzam and Thomas Bertauld who have helped diversify my knowledge. Many thanks to *Dream Theater*, *Indian Ocean* and *Indus Creed* as well, for their inspiring music throughout these years.

I would also like to thank my uncle Ganesh Dutt Sharma, for sharing with me his love for physics and my father for allowing excursions to his research lab. They have been instrumental in imbibing a scientific conscience and the culture of *reason* in me.

Finally and most importantly, I would like to thank my mother for making me who I am as a human being. There are no words in any language that can express her importance in my research.

RÉSUMÉ

Les applications modernes sont difficiles à diagnostiquer avec les outils de débogage et de profilage traditionnels. Dans les système de production, la première priorité est de minimiser la perturbation sur l'application cible. Les outils de traçage sont très appropriés pour l'étude des performances de tels systèmes car les événements sont enregistrés et l'analyse se fait a posteriori. Une des principales exigences des systèmes de traçage est le faible surcoût. L'activation d'un nombre réduit d'événements aide à respecter cette exigence, mais au prix de la diminution de la granularité de la trace.

Dans cette thèse, nous présentons notre travail de recherche qui traite du problème de la granularité limitée des traces en maintenant un faible surcoût sur les applications cibles. Nous présentons de nouvelles techniques et algorithmes qui abordent le problème en se basant d'abord sur une approche de filtrage logiciel et de traçage coopératif, puis en explorant des mécanismes plus avancés de traçage matériel. Nous avons proposé une approche efficace de traçage conditionnel dans l'espace noyau et utilisateur qui se base sur des mécanismes de filtrages compilés en code natif. Afin d'atteindre l'objectif d'avoir une trace détaillée du système, nous expliquons que les processeurs modernes contiennent des blocs de traçage matériel qui n'ont pas encore été entièrement exploités dans le domaine du traçage. Nous caractérisons leur performance et nous analysons les paquets de traces, leur relation avec l'exécution du logiciel, et les possibilités de les utiliser pour une trace détaillée. Nous proposons des techniques à faible surcoût, assistées par le matériel, rendant possible une analyse détaillée permettant la détection des latences d'interruption et des appels systèmes. Nous présentons aussi une nouvelle technique qui se base sur les paquets de trace à bas niveau du processeur pour analyser efficacement les processus et les ressources utilisées dans une machine virtuelle. De plus, nous avons identifié et solutionné des problèmes reliés au traçage matériel en utilisant l'assistance logicielle du système d'exploitation, ouvrant ainsi la voie à des recherches plus approfondies sur les approches coopératives de traçage matériel-logiciel. Comme nos techniques sont axées sur les exigences du traçage à haute vitesse dans les systèmes embarqués et de production traitant des transactions à haute fréquence, nous avons constaté que nos progrès dans le domaine du traçage matériel-logiciel se sont avérés très utiles pour détecter la contention des ressources et la latence dans les systèmes.

ABSTRACT

Modern applications are becoming hard to diagnose using traditional debugging and profiling tools. For production systems the first priority is to have minimal disturbance on the target application. To analyze performance of such systems, tracing tools are imperative where events can be logged and analyzed post-execution. One of the key requirements for tracing solutions however, is low overhead. A generic solution can be to select only a few events to trace, but at the cost of trace granularity. In this thesis, we present our research work that deals with the problem of lack of high granularity in traces while maintaining a low-overhead on target applications. We present our new techniques and algorithms that approach the problem initially from a software filtering and co-operative tracing approach, and then explore more advanced hardware tracing mechanisms that can be used. We have proposed an efficient kernel and userspace conditional tracing approach, with an enhanced native compiled filtering mechanism. Continuing towards our goal to have a detailed trace of a system, we further discuss how modern processors contain new hardware tracing blocks that have not yet been fully explored and exploited in the tracing domain. We characterize their performance and analyze the trace packets, their relation with software executions and opportunities to utilize them for a detailed trace. We therefore propose low-overhead hardware assisted techniques that allow a fine grained instruction based interrupt and system call latency detection mechanism. We also present a new algorithm that shows how such low-level trace packets coming directly from the processor, can be effectively utilized to analyze even the processes or resources consumed inside a VM. We have also identified and improved upon issues related to hardware tracing itself using software assistance from operating systems thus laying out ground for further research in hardware-software co-operative tracing approaches. As our techniques are focused towards requirements of high speed tracing in embedded or production systems, catering high frequency transactions, we have found that our advancements in the hardware-software domain have proved to be invaluable in detecting resource contention and latency in systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ARM	Advanced RISC Machines
ART	Always Running Timer
AVX	Advanced Vector Extensions
BLAS	Basic Linear Algebra Subprograms
BPF	Berkeley Packet Filter
BSD	Berkeley Software Distribution
BTS	Branch Trace Store
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CSPF	CMU/Stanford Packet Filter
CYC	Cycle Count
DBI	Dynamic Binary Instrumentation
DBT	Dynamic Binary Translation
DCPI	Digital Continuous Profiling Infrastructure
DGEMM	Double-Precision General Matrix Multiply
ETB	Embedded Trace Buffer
ETM	Embedded Trace Macrocell
FPU	Floating Point Unit
GCC	GNU Compiler Collection
HAVAna	Hardware Assisted VM Analysis
I/O	Input/Output
ICE	In-Circuit Emulation
IRQ	Interrupt Request
ISA	Instruction Set Architecture
JIT	Just-In-Time
JTAG	Joint Test Action Group
JVMDI	Java Virtual Machine Debugger Interface
LBR	Last Branch Record
LLVM	Low Level Virtual Machine
LTTng	Linux Trace Toolkit Next Generation
MIPS	Microprocessor without Interlocked Pipeline Stages

MTC	Mini-Timestamp Counter
NMI	Non-Maskable Interrupt
NR	Non-Root
PIP	Paging Information Packet
PSB	Packet Stream Boundary
PT	Processor Trace
PTM	Program Trace Macrocell
SIMD	Single Instruction Multiple Data
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
SoC	System-on-Chip
TIP	Target Instruction Pointer
TMA	Timing Alignment
TNT	Taken-Not-Taken
TRF	Temporal Resolution Factor
TSC	Timestamp Counter
VM	Virtual Machine
VM(s)	Virtual Machine(s)
VMA	Virtual Memory Area
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Monitor
pCPU	Physical CPU
vCPU	Virtual CPU

CHAPTER 1 INTRODUCTION

System analysis tools are indispensable to developers. The advent of newer many-core systems have made it hard for current debuggers and tracers to keep up in terms of features and performance. To get a detailed view of system internals, a more in-depth analysis is required which usually involves probing millions of locations in application code. In software with large code-bases, such as the Linux kernel, thousands of statically instrumented tracepoints exist. Analyzing such software requires thousands of such tracepoints to be enabled at runtime, which eventually generates a huge amount of trace data that needs to be analyzed. For resource constrained systems, it would be inefficient to store such a huge amount of trace data and retrieve it for offline analysis. In our work, we first investigated a software-only approach where we focused on efficient filtering of trace data at runtime to weed out unnecessary tracepoints and select only the interesting ones. We also proposed a kernel-userspace co-operative tracing approach where the trace filters would co-operatively work on conditions that can be changed dynamically.

Furthermore, we observed that modern debugging and tracing tools are not utilizing the full support of hardware provided by the current generation processors which have in-built tracing blocks. These hardware tracing techniques could allow a more detailed way of analyzing data by further reducing the runtime overhead. We investigated this line of thought and came up with innovative hardware-assisted tracing techniques that can be combined with software tracing tools to provide low-overhead and more granular analysis of modern systems.

1.1 Definitions and Basic Concepts

1.1.1 Instrumentation

In the domain of computer program analysis, we define the term *instrumentation* as the addition of extra code in the analysis program either before or after compilation that enables the observer to gather metrics about the program as it eventually executes. These metrics can be as simple as a histogram of the number of times a function got called or complex memory and thread sanitation routines. Most of the advanced tools used for profiling, tracing or debugging that we eventually discuss are built upon such instrumentation primitives. In this thesis, we classify instrumentation as either static, dynamic or hybrid in nature.

Static Instrumentation

In many cases, before compilation begins, the analysis code can be added as part of the target program source. For example, in the case of a simple function profiler, a function that increments a small counter can be added at every entry of all functions in the target program. This is usually the case for large software such as the Linux kernel which use statically inserted *tracepoints* in all major functions. Various tools hook onto their tracepoints in the kernel to gather data for further analysis. We will elaborate on this in subsequent sections. Such instrumentation is inexpensive but has a drawback of always being in the critical execution path during execution. Another approach to static instrumentation is when the compiler itself instruments paths in the code to execute analysis code [1]. In any case, static instrumentation requires the source code knowledge or compiler assistance, to introduce analysis code in the target application.

Modern compilers such as GCC and LLVM Clang allow special symbols or callback-function hooks to be inserted at entries and exits of all functions in the source code at compile time. At runtime, an instrumentation agent can latch onto those functions and execute its own analysis code. As an example, the Linux kernel can be compiled with the gcc `-pg` switch which inserts a `mcount` symbol at each function entry and exit. This can be latched onto by the Ftrace tracer at runtime to provide dynamic tracing capabilities. Similarly, for any userspace program, GCC offers the `-finstrument-functions` switch, which provides compile time instrumentation of calls to `__cyg_profile_func_enter()` and `__cyg_profile_func_exit()` functions. These can then be interposed at runtime and be used to run analysis code. The analysis infrastructure and tools such as LTTng, Ftrace, SystemTap and eBPF heavily utilize static instrumentation which is part of the Linux kernel infrastructure.

Dynamic Instrumentation

Analysis code can also be inserted in binaries post compilation – either on application binaries residing on the disk or more importantly while the target application is executing. This is known as Dynamic Binary Instrumentation (DBI) and is of special interest to us as it allows the insertion of code while the process is executing. DBI eliminates the need of having a prior knowledge of program source code while providing the benefit of enabling and disabling the analysis code We classify most of the dynamic instrumentation techniques used in building frameworks and tools as follows :

TRAP Based : This technique is used in tools such as older Kprobes [2, 3] and GDB's normal tracepoints [4, 5]. During execution, when the TRAP is encountered, the OS halts

the program and its state is saved. It sends a SIGTRAP signal to the process which can handle the exception. Usually in the handler, the original instructions are executed, out of line and then the intended instrumentation code is executed. Upon return from the handler, the original instruction is restored and the execution continues. On x86, `0xCC` is reserved for such an interrupt. 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.

Jump-pad Based : Most of the good instrumentation frameworks such as Dyninst, PIN and tools like GDB (for fast tracepoints) [6], 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.

JIT Translation : Other tools like Valgrind [7] use Just-in-Time (JIT) code translation 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 [8]. 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.

1.1.2 Tracing

Tracing can be defined as a very fast system-wide fined grained logging mechanism. 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. Sampling based profiling tools are also only moderately useful in such cases. Therefore, the fast logging mechanism called tracing is employed. Tracing tools and frameworks are built upon the base instrumentation techniques discussed previously. 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). Static tracing requires source code modification and recompilation of the target binary/kernel, whereas in dynamic tracing one can insert a tracepoint directly into a running process/kernel, or in a binary residing on the disk.

Tracepoint

Tracing usually involves adding a special *tracepoint* in the code. This tracepoint can look like a simple function call, which can be inserted anywhere in the code (in case of userspace applications) or be provided as part of the standard kernel tracing infrastructure.

Event

Each tracepoint hit is usually associated with an event. When a tracepoint is hit, a callback function is called which allows trace data to be collected, aggregated or stored in a buffer. The events are very low level and occur more frequently. Some examples are syscall entry/exit, scheduling calls, etc. For userspace applications, these can be some important functions in the target application such as a request processing thread of a server or an insert operation in a database.

Trace Buffer

In general, tracing involves storing associated data in a special buffer, whenever an event occurs. For a detailed execution trace of a very fast system, this data is obviously 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.

1.1.3 Program Flow

Another important way to debug and profile programs is by analyzing the paths a program took while executing. This can help in understanding if the call stack for a given function was the one that we expected, or be used to profile function executions and observe latency between any two points in the program. Information about possible program paths can be obtained by static or dynamic analysis of source code, or more advanced hardware-assisted techniques. While static analysis of a program through its source code can yield a detailed Control Flow Graph (CFG), and eventually a way to visualize the program flow, it is only at runtime that the information about reachable CFG paths can be known. As an example, the

Ftrace tracer in the Linux kernel uses the software based hybrid instrumentation techniques discussed above to obtain the program flow at function granularity. The program flow can be obtained at a very high granularity through either analyzing every instruction executed by the processor or through offline reconstruction of instruction flow by recording only branches and change-of-flow instructions in the program.

Hardware Tracing

A major part of the work presented in this thesis revolves around special hardware tracing blocks that are part of modern processor chips. These on-chip tracing blocks generate high frequency trace packets at instruction or branch granularity in a program which allows the analysis tool to reconstruct program and/or data flow by looking at the instruction flow and timings. This allows a more fine-grained analysis of applications. As an example, Intel provides the Branch Trace Store (BTS), Last branch Record (LBR) and Processor Trace (PT) hardware blocks which allow recording each branch status and its target. Such trace hardware is implemented on silicon with enable/disable control from control registers of the processor. This is the lowest level of tracing, which allows analysis of instructions as they execute. The same hardware can also be configured to generate timestamp packets to analyze latency issues very closely in real-time systems. As there is no tracepoint being called in the software and trace recording happens in parallel, the overhead of this hardware-assisted approach on the target application is minimal. We elaborate more on this in the next chapter.

1.2 Problem and Scope

Up until recently, code analysis tools have always been simpler in terms of design and functionality owing to the fact that test applications were monolithic and processing hardware was simple. For the more complex modern, multi-core and distributed systems of today, even though debuggers can give a detailed view of the target process at any given time, they require pausing the process and running a manual or conditional automated analysis. This alters the time-correctness of applications. Tracing, as discussed before, has proven to be an important technique in such cases. It is robust and useful for long-running systems in production for which halting the process for analysis is not an option. For a detailed view of such systems however, as expected, enabling millions of trace events generates a lot of data and hence causes unwanted overhead in the system. This forces the analyst to disable multiple tracepoints, which then impedes the understanding of proper application context post-execution. Even though modern tracers allow preliminary filtering of events, the conditions on which they check are restricted to only the kernel context. Moreover, tracing tools

and frameworks do not leverage advancements in processor hardware over the years, due to their complexity and lack of empirical data on their performance, even when these hardware tracing frameworks could address the problem of detailed trace data obtained with low overhead.

Research Questions The state-of-the art analysis techniques have not considered such limitations and lack any advancement that allow the reduction of overhead and maintaining reproducible and useful traces. In view of the issues discussed above, we go a bit further and define three important research questions that have not yet been addressed in-depth in this domain :

- Is it possible to develop a new, modern tracing technique that allows a more detailed view of the system but at a lower overhead and with a larger trace context ?
- Can we further enhance the information gathered from the control flow traces and reduce the overhead on production systems using modern hardware techniques coupled with software assistance ?
- Will the newly proposed algorithms and techniques, be robust, have low overhead and cause minimum perturbations to the target system ?

1.3 Research Objectives

For obtaining a highly detailed view of the system but at a low cost to the system itself, the conditional aspect of tracing has not been investigated in-depth before – especially considering the cost of filtering high-frequency events with a software approach. Conditional tracing can be used where filters can be applied over tracepoints, which allow the tracepoint to fire and actually record data only when a particular condition is verified. The conditions need to have a context of the operating system as well as the target process and would require some sort of expression language that could be evaluated at runtime at a high speed. Apart from that, modern processor chips now contain hardware tracing blocks that have not been sufficiently utilized to obtain detailed traces. Therefore, the lack of very accurate trace information is still a major issue for production systems. For this research, the general objective has been to solve the problem of low-overhead yet detailed traces by exploring pure software as well as hardware-assisted techniques recently becoming available.

Going further, we identified specific objectives of our research as follows :

- To propose a new tracing technique that introduces the concept of software based conditional tracing which can yield a holistic yet detailed view of the target applications.

— To propose algorithms that utilize hardware tracing support in modern processors for a detailed analysis of target applications and virtual machines with a low-impact on the observed system.
— To develop new algorithms and techniques that optimize the state-of-the-art hardware trace analysis approach for improving accuracy of trace data gathered for an overall improved tracing framework.

1.4 Contributions

In-line with the research objectives stated above, this thesis presents the following original contributions in the field of systems and program analysis :

— A fast conditional tracing architecture which incorporates JIT compiled filters and an improved kernel VM - user VM direct data sharing mechanism for low overhead co-operative tracing.
— An instruction and time delta profiling algorithm which utilizes on-chip hardware tracing blocks for a low-overhead program flow and latency analysis. This technique also allows us to profile software tracing overhead with cycle-accurate resolution.
— A hardware-assisted virtual machine analysis algorithm which utilizes trace packets from on-chip hardware to detect resource contention in virtual CPUs, and processes running inside the VM with near-zero overhead and in a non-intrusive manner.
— A robust, low-overhead, kernel-assisted hardware trace reconstruction algorithm which allows hardware traces to be reconstructed in partial or incorrect decoding scenarios when runtime compiled code is executed on the hardware.

1.5 Outline

This thesis is organized as follows. Chapter 2 is dedicated to the review of state-of-the-art literature and associated software frameworks, the techniques they use and our inference from them. Chapter 3 outlines our research methodology and explains the process of generating research leads, identifying problems, actionable items, specific milestones and eventual outcomes in terms of research papers and a succinct description of their content. It presents an overall view of the body of this research. We then move towards the core body of the research which has been presented as four articles.

We first present a new and improved trace filtering and conditional tracing architecture as the article titled “Enhanced Userspace and In-Kernel Trace Filtering for Production Systems”. We propose our new kernel and userspace co-operative tracing approach as well as a fast

tracing architecture based on eBPF and LTTng. This article appears in the November 2016 issue of *Journal of Computer Science and Technology* (Springer) and constitutes Chapter 4 of this thesis.

The second article discusses new hardware trace based high resolution and low overhead filtering techniques and a new algorithm to generate instruction and time delta profiles for accurate profiling of interrupt and syscall latency. This article is titled "Hardware-Assisted Instruction Profiling and Latency Detection" and appears in the August 2016 issue of *Journal of Engineering* (IET) and is presented in Chapter 5 of the thesis.

Our third article presents a novel non-intrusive algorithm to trace VMs, with very low overhead hardware-assisted tracing, through analysis of raw hardware trace packets. This work has been presented in the International Workshop on Cloud Computing Systems, Networks, and Applications at IEEE Globecom conference and appears in *Proceedings of Globecom Workshops* (IEEE). This is reproduced as Chapter 6 in this thesis.

The fourth article in Chapter 7 presents a new algorithm and an in-kernel implementation which tackles the issue of faulty hardware trace reconstruction when runtime code such as JIT compiled sections are being executed in a target process. This work has been submitted to *ACM Transactions on Computer Systems*

Finally, in Chapter 9, we present a summary and discussion of our research contributions, their impact on the tracing ecosystem and provide recommendations for future work that may be carried out in this field.

CHAPTER 2 LITERATURE REVIEW

In this chapter we attempt to classify and review techniques used within popular trace, profile and debugging tools - both software and hardware based. This has helped us gain an understanding of the benefits and deficiencies in state-of-the-art methods and algorithms.

2.1 Tracing and Instrumentation

As discussed in Chapter 1, static and dynamic instrumentation techniques form the core of the tracing, debugging and profiling tools and frameworks available. We first begin with a case-study of a small experimental dynamic tracing tool called ***DyTrace***, which we developed while analyzing tracers in-depth. This gives the reader a deeper understanding of what goes on behind most popular trace frameworks from a very low-level perspective while discussing the state-of-the-art techniques used in developing them. We shall then move towards the description of other dynamic and static instrumentation techniques and details of other modern trace frameworks and related recent advancements.

2.1.1 CASE STUDY : DyTrace

The goal of this initial effort was to build a minimal tracing tool that allows insertion of static tracing code in a binary dynamically. For this, we used a binary instrumentation framework called Dyninst and a special POSIX syscall called `ptrace()`.

ptrace()

Linux has a special system call called `ptrace()` which "provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing" [9]. A target child process on which `ptrace` has to be used goes through the following states :

1. Child process is stopped upon receiving a signal (like SIGSTOP)
2. Parent receives child status from `wait()` (it is safe to start ptrace)
3. Ptrace operations are performed on the child
4. Parent signals child again to continue execution

A typical `ptrace()` call may be written as,

```
long ret = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
```

where the first argument is a special ptrace code, the second is the PID of the target child process to control, and the third and fourth arguments are the memory addresses for modification and data structures to be written to the process respectively. As an example in the above code line, the `PTRACE_ATTACH` code is used when a task (T) wishes to control the child (C) with PID as `pid`. It makes C a child of the tracing task T. Similarly, in the following listing,

```
long ret = ptrace(PTRACE_POKEDATA, pid, addr, new_val);
```

The call writes `new_val` to address `addr` in the address space of the child task. Such actions are useful for debuggers where variables can be changed upon pausing the process to observe its behavior. There are other ptrace codes such as `PTRACE_SETREGS`, `PTRACE_GETREGS`, `PTRACE_PEEKTEXT` etc. which perform similar related tasks. The modifications done by `ptrace` itself are at the binary level and hence have a direct effect on the execution. Consider an `execl` call,

```
execl("/bin/ls", "ls", NULL);
```

According to the calling conventions for the calls on a x86 machine, first the syscall number is loaded in `%eax` and then the subsequent arguments are loaded in `%ebx`, `%ecx`, `%edx`, `%esi` and `%edi`. After setting these registers, the soft interrupt 0x80 is called as `int $0x80` which signals the kernel to go ahead with the system call [10]. However, before executing the syscall, the kernel checks if the process is being traced or not. So, in the traditional fork-exec model, if the child has a `ptrace()` call before an `execve()` call, the kernel will hand over the control to the parent before `execve()`. At the syscall, the kernel saves the value of the `eax` register which has the syscall number. The parent then continues to call ptrace with actions such as modifying/reading the memory or registers. This is usually followed by a `PTRACE_CONT` code which continues the child execution and the syscall but now with modified register/memory values. Thus, it can be seen that `ptrace` can act as a building block for any dynamic instrumentation technique and an instrumentation primitive. The subsequent sections discuss some tools built using such primitives and their performance.

Dyninst

Dyninst presents a very simple and powerful API for dynamic instrumentation [6]. There are two main essential terminologies involved in performing instrumentation, *snippets* and *points*. Points are simply the locations in a program where instrumentation can be inserted. They can be function entry, function exit etc. Snippets are the abstraction for the code that can be inserted at the points. Snippets are not directly written in assembly, but instead Dyninst uses an intermediate representation via an abstract syntax tree. If the complexity of the snippet increases, it can be built separately using a C-like syntax based scripting support in DynC.

Dyninst introduces the concept of *mutator* which is the program that is supposed to modify the target (*mutatee*). This mutatee can either be a running application or a binary residing on disk. The process attaching or creating a new target process allows the mutator to control the execution of the mutatee. This can be achieved by either `processCreate()` or `processAttach()` which returns a `BPatch` object. This functionality is achieved internally using the `ptrace()` call discussed before. The mutator then gets the *program image* using the object, which is a static representation of the mutatee. Using the program image, the mutator can identify all possible instrumentation points in the mutatee. The next step is creating a snippet for insertion at the identified point. The mutator can then create a snippet, to be inserted into the mutatee. Building small snippets can be trivial. For example, small snippets can be defined using the `BPatch_arithExpr` and `BPatch_varExp` types. The snippet is compiled into machine language and copied into the application's address space.

Jump-Pad For executing the built snippets, the concept of jump-pad or trampolines is used. The normal execution flow is modified by *jumping* to another memory location, executing the snippet code there and then returning, as illustrated in figure 2.1.

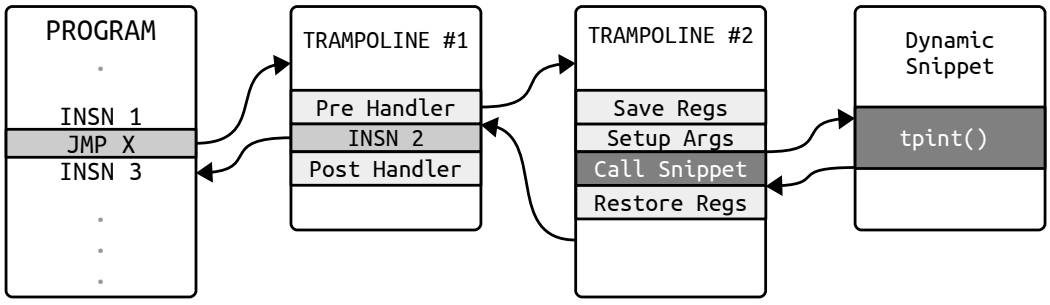

Figure 2.1 Trampoline approach used by Dyninst

The instruction at the instrumentation point is replaced by a jump to a base trampoline.

The base trampoline then jumps to a mini-trampoline that starts by saving any registers that will be modified. Next, the instrumentation is executed. The mini-trampoline then restores the necessary registers, and jumps back to the base trampoline. Finally, the base trampoline executes the replaced instruction and jumps back to the instruction after the instrumentation point.

Instruction Instrumentation While building the experimental DyTrace tool, particularly at the instrumentation phase, the changes made at the assembly level to the executing process's executable memory were observed. Figure 2.2 shows how Dyninst uses the trampoline approach at instruction level to insert our own static tracing functions dynamically.

We see a test function where our tool intends to insert our static tracing function `tpint()` just before its exit. This `tpint()` function simply saves the integer value provided to it as an argument to a trace file - thereby *tracing* the value dynamically as the program executes and demonstrating a small trace tool.

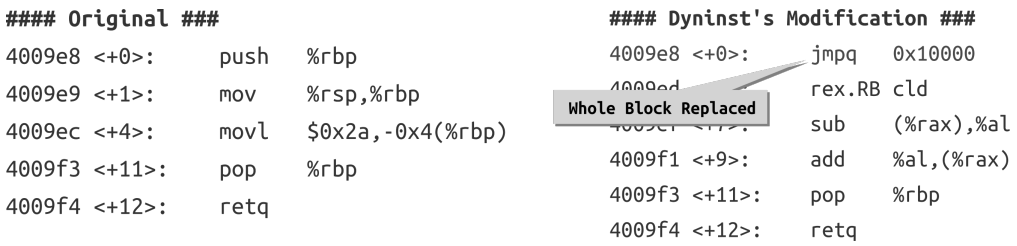

Figure 2.2 Original code and jumps inserted for trampolines

When instrumentation at function exit was done for Dyninst, it replaced the whole function block and patched a jump on entry with address to its trampoline. Dyninst executes the instructions out-of-line and then returns from the function. Figure 2.3 shows what goes inside the trampoline. First, the remaining function is executed out-of-line. The basic idea for instrumentation is to prepare the stack first, and save the current state by pushing all the registers. Then, it continues to execute the snippet (containing the pre-built tracepoints from the library), restores the stack state, pops all the registers and finally restores the original stack pointer and returns. In our small experiment with DynTrace, as shown in the figure, a variable with the value 43 (trace *payload*) was recorded dynamically, using a static tracing function `tpint()` by instrumenting it at end of the target function.

Irrespective of whether the available tracing frameworks are in userspace or in the kernel, the same basic static and dynamic tracing techniques are used. As an example, the jump-pad technique is used by GDB in its fast userspace tracepoints as well as by Kprobes to provide

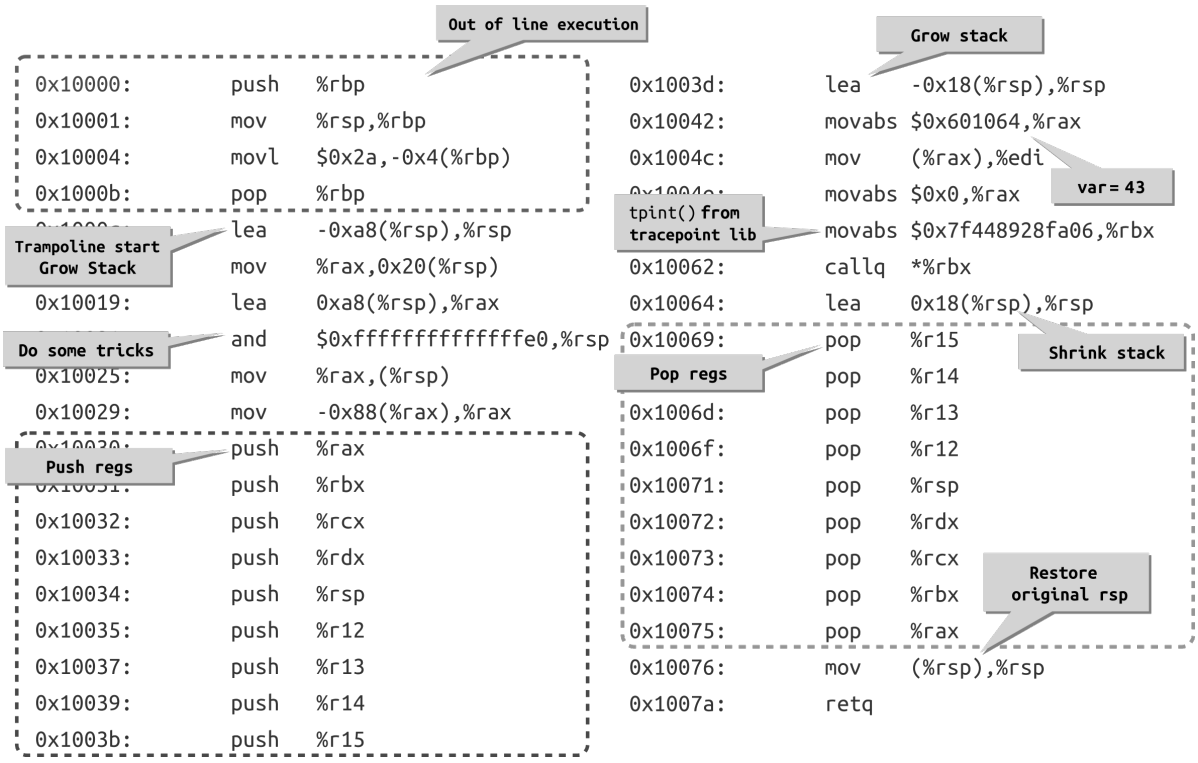

Figure 2.3 Inside a Dyninst trampoline

an efficient dynamic tracing framework in the kernel. Before moving further, we discuss some common intermediate techniques used in kernels on which high level trace frameworks can hook onto.

2.1.2 Kprobes

With the basic mechanism in place to modify instructions, developers have provided a support infrastructure in the Linux kernel in the form of Kprobes, to instrument almost any kernel function on-the-fly and gather debugging and performance information without any disruption [3]. Tracing tools can build upon the Kprobe interface by providing kernel modules. The module's `init` function registers one or more probes, and the `exit` function un-registers them. A registration function such as `register_kprobe()` specifies where the probe is to be inserted and what handler is to be called when the probe is hit. Upon a probe registration, Kprobes makes a copy of the probed instruction and replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., `int 3` on i386 and x86_64). When a CPU hits the breakpoint instruction, a trap occurs, the CPU's registers are saved, and control passes to Kprobes. Kprobes executes the `pre_handler` associated with the Kprobe, passing the addresses of the Kprobe struct and the saved registers to the handler . A newer *fast*

kprobes implementation which removed the slower TRAP based dependency on `int 3` was proposed and implemented recently, which aims to improve the performance further [2]. This is based on the jump-pad instrumentation approach discussed before. Most tracers such as LTTng, SystemTap or Perf, that wish to provide dynamic tracing capabilities in the Linux kernel, use Kprobes and get the trace payload data through their own mechanisms.

2.1.3 TRACE_EVENT

The majority of static tracepoints in the Linux kernel are provided using the `TRACE_EVENT()` macro [11, 12]. It is one of the most common ways to connect a tracer to the tracepoints in the kernel, mainly due to the fact that it allows the developers to just use the macro to add tracepoints and acts as an abstraction for the trace or profile tools. The same tracepoint can then be used by all the popular Linux tracers - Perf, LTTng, SystemTap or Ftrace. This macro allows the kernel developers to basically define a callback function hook that would be called upon a tracepoint hit. It also allows arguments that should match the function prototype and a structure that can contain the tracepoint data. Tools such as LTTng or SystemTap can then define their own tracepoint wrappers as part of statically or dynamically generated kernel modules that match those defined in the `TRACE_EVENT` macro statically in the kernel code. This allows a standard location of tracepoints strategically placed in the Linux kernel source code but a facility for other tracers to hook onto them and use their own mechanisms of data transfer or computation.

2.1.4 Tracing Infrastructure Hierarchy

Static and dynamic tracing functionality in the userspace and Linux Kernel is provided by standard static and dynamic instrumentation techniques discussed earlier. There are a huge number of tracers that have been developed over the years, with varying capabilities and support. They can either support the kernel domain or userspace domain, or both. Tracers can sometimes hook to the standard interfaces described above or use their own low level static and dynamic tracing mechanisms. While tracers have been described elsewhere in detail, there has been no attempt to classify them based on the underlying infrastructure. Therefore, before going further to describe some state-of-the-art tracing tools, we classify them. This should help the reader to better understand as we describe some common trace frameworks. Figure 2.4 shows some common trace frameworks and the underlying mechanisms they use. The red color signifies kernel support and the black color signifies userspace support. The same figure also indicates the dynamic or static capabilities by direction of the arrows - either towards static instrumentation or dynamic instrumentation. As an example, we can

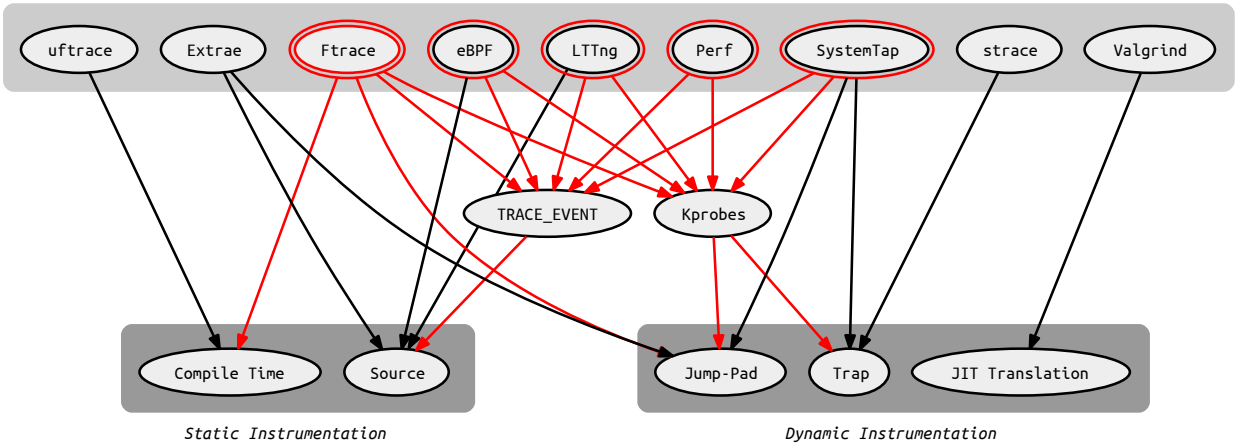

Figure 2.4 Dependency graph of tracing tools and frameworks showing kernel (■) & user-space (■) tracers

see that Ftrace, the kernel tracer allows tracing with the help of kprobes based events that in turn use either an optimized jump-pad based approach or a trap based technique. The same tracer also allows dynamically activated static tracepoints, which use compile-time static instrumentation approach. We now discuss some of these tools in detail.

Ftrace

The Linux kernel's native trace support is provided by its function tracer called `ftrace` [13]. It is a kernel-exclusive tracer which allows recording a trace of all eligible kernel functions along with accurate timestamps. It can then provide a function call graph representation or other analysis outputs such as interrupt-off or scheduling latency [14]. Ftrace simultaneously allows two mechanisms for tracing with trace control and data recording provided using the kernel's Debugfs pseudo-filesystem [15]. There are multiple trace targets available that can be used, but most of them either use the event tracer or a custom hooking mechanism.

Event Tracing Event tracing in ftrace is provided using the kernel's static tracing hooks exposed as `TRACE_EVENT()` macro. This is similar to how Perf or LTTng use them. The tracepoint handler function of Ftrace allows the data to be collected in its own buffer, thus providing uniform information, as obtained from other tracers. An important feature of the event tracing mechanism of Ftrace is the ability to use filters. These filters allow a basic degree of limited conditional tracing. However, the filtering is done after the trace data has been recorded in the buffer. Another important limitation of filters in Ftrace is the inability of

Ftrace to access register values and variables at tracepoint locations. However, each tracepoint contains high resolution timing information which is valuable in diagnosing latency issues.

Dynamic Tracing Even though the name suggests the dynamic nature of tracing, this mechanism is essentially a dynamically activated static instrumentation technique. At kernel compile time, when the `CONFIG_DYNAMIC_FTRACE` option is used, the buildsystem uses GCC's compile time instrumentation option which inserts the `mcount` symbol at function entry. This allows Ftrace to directly hook onto most of the functions in the kernel directly and execute tracing or filtering code. Tim Bird has discussed this earlier in relation to function latency calculation in the Linux kernel [16] where these capabilities of Ftrace have been utilized. Apart from that, Ftrace also allows true Kprobe based dynamic tracing, which is itself based on jump-pad or trap based dynamic instrumentation. This allows Ftrace to dynamically probe not just entries of functions but also offsets within the function and functions-exits using the Kretprobes. This, however, carries all the limitations of Kprobes, such as restrictions on certain kernel functions. Overall, Ftrace is one of the most stable kernel tracing tools in use and is actively maintained by the kernel community.

2.1.5 SystemTap

SystemTap [17] allows the dynamic insertion of tracepoints as well as collecting traces for tracepoints defined using `TRACE_EVENT`. The trace collected can be displayed on the console while it is generated, or can be saved to a file. The flight recorder mode can dump the trace in memory to a file for it to be analyzed later. Instrumentation and trace code for SystemTap is written as scripts. The language is similar to C in terms of syntax and supports all ANSI C operations. The supported data types are only integers and strings but the same SystemTap script can be used to declare multiple instrumentation points. The declaration of an instrumentation point is composed of two parts. The first part is used to identify the event that you want to associate with the instrumentation point. The second part is the code to execute when the tracepoint is encountered. SystemTap also supports conditional tracing. To set a condition, the *if* style syntax is used :

```
probe kernel.function("vfs_read")
{
  dev_nr = $file->f_path->dentry->d_inode->i_sb->s_dev
  if (dev_nr >= 3)
    printf ("%x\n", dev_nr)
}
```

In the above script, the instrumentation point is inserted at the entry of the function `vfs_read`. The information is extracted from the parameter `file` and is copied into a temporary variable. This condition is then evaluated and the script outputs the value if true.

SystemTap scripts are converted into C code, which is then compiled as a module. This module is then inserted in the kernel and communicates with SystemTap for tracing. There is also a special mode where it is possible to insert C code in scripts mainly to overcome some limitations of the scripting language. Dynamic kernel tracepoints in SystemTap are implemented using Kprobes. SystemTap also takes input from the DWARF debugging information generated during the kernel compilation to determine the addresses of instrumentation points, as well as to resolve references to kernel variables used in scripts. SystemTap then registers the handler passed to Kprobe to retrieve the values of these variables. The dynamic tracepoints can use all the available variables from the instrumentation point addresses.

SystemTap provides a multitude of events that can be associated with instrumentation points. A few examples are listed below :

```
/* Function entry */
probe kernel.function("vfs_read").call

/* Function exit */
probe kernel.function("vfs_write").return

/* specific location in kernel code */
probe kernel.statement("*@fs/read_write.c:42")

/* Specific address in binary */
probe kernel.statement(0xc00424242)
```

SystemTap also allows static tracepoint connections using the `TRACE_EVENT` macro. A static tracepoint can be activated by a script as follows :

```
probe kernel.trace("event_name")
```

The event name is the name given to the static tracepoint during the call to the `TRACE_EVENT` macro. Just as other trace probe types, static tracepoints defined in this way can be conditional too. However, they can only use the variables passed as parameters at the point of instrumentation. This is because the static tracepoint handlers do not receive copies of the

calling stack frame when the event is launched. In addition, SystemTap cannot use the display format specified in the `TRACE_EVENT` using the macro `TP_printk`. To represent the data as defined in the `TRACE_EVENT` macro, the user must redefine how to do it in the script.

Though a very elaborate, feature-rich, and easy to use tool, SystemTap suffers from serious performance issues in terms of data gathering speed and scalability. Some benchmarks which compared SystemTap and LTTng-UST were performed by Julien Desfossez and showed results where in some instances UST was 289 times faster than SystemTap (in flight recorder mode) with the LTTng kernel [18]. This huge difference in performance is due to the use of buffering while collecting trace data for UST, instead of the SystemTap approach to have a system-call transition for each event. Also, a major chunk of work is handled in kernel rather than in user-space by SystemTap [19]. This is an architectural difference which leads to a deteriorated performance. The idea of dynamic compilation, also comes with an interrupt-driven approach and frequent context switches. Apart from that, mechanisms like preparing the code for compilation, the actual compilation to a module and the insertion process into the kernel as a module carries extra overhead, which overshadows the efforts of having an overall better tracing system.

2.1.6 LTTng

The Linux Trace Toolkit next generation is a very fast and extremely low-overhead kernel and userspace tracer. Low overhead, in simple terms, means that even with a *non-activated* tracepoint inserted in the code, it gives near-zero impact on the overall execution of the target application. This makes LTTng a bit different from the other tools and a default choice for real time applications. Its tracing technique implements a fast wait-free read-copy-update (RCU) buffer for storing data from tracepoint execution.

In figure 2.5, it can be seen that the LTTng session daemon acts as a focal point for trace control. Static instrumentation can be defined as tracepoints in the source code of the kernel, as well as in user-space applications with UST. The `TRACE_EVENT` macro can also be used for kernel events. Dynamic instrumentation in kernel is provided by Kprobe, in the same way as for other tools. An instrumented application, which contains the user's desired tracepoints, automatically registers itself to the session daemon, just as its execution starts. This is also the case with the kernel. This is useful for handling simultaneous trace control for multiple trace sessions. Thereafter, the session daemon manages all the tracing activity. The LTTng consumer daemon is responsible for handling the trace data coming from the applications. It exports raw trace data and builds a CTF stream to be written on the disk. The Common Trace Format (CTF) is a compact binary format, which stores all the trace data in a very well

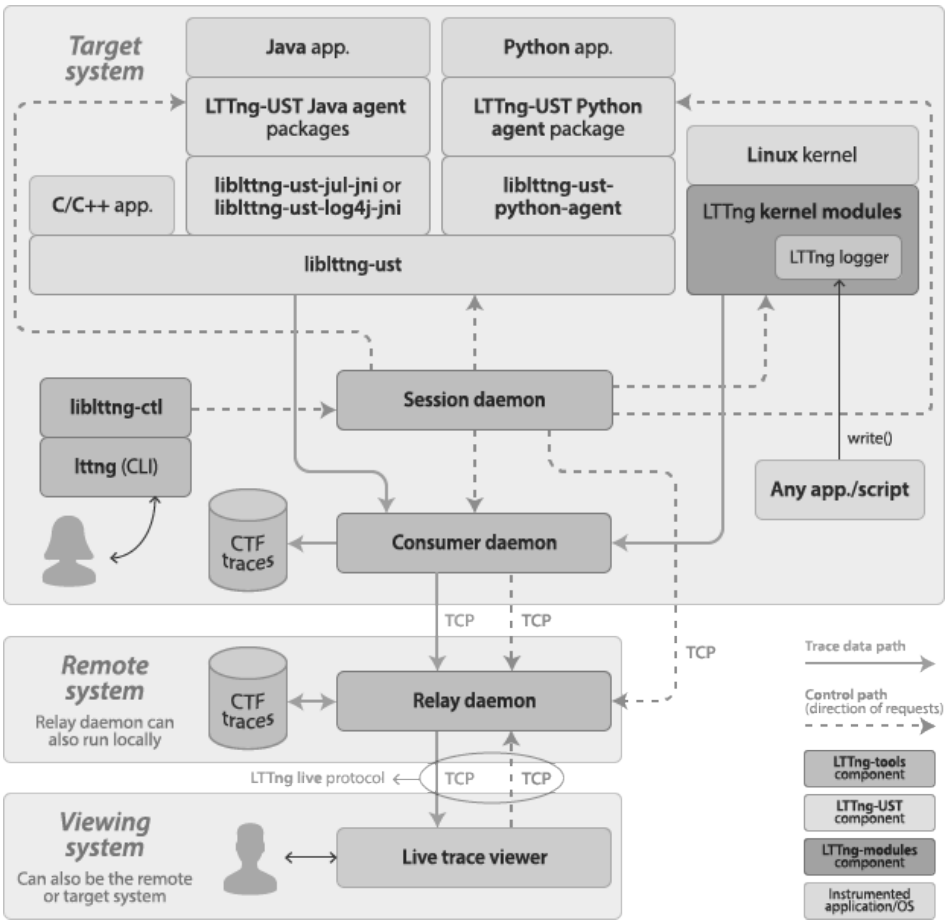

Figure 2.5 LTTng Architecture [1]

structured manner for further analysis by trace viewers and converters, such as Babeltrace (command line), or Trace Compass (graphical). For example, one can view the exact time of each event and the control flow through the various calls in the kernel, graphically in a time-line using Trace Compass.

Performance LTTng is currently the fastest tracer available for userspace tracing. Various performance comparisons with other tracers have revealed this before as well [18] [21]. The major factor for such improvements in performance is mainly due to the use of userspace RCU techniques for having a lock-less ring buffer in LTTng-UST. In figure 2.6, it can be seen how multiple readers trying to access a resource are managed by the RCU technique. The major reasons for such performance benefits are that `rcu_read_lock()` and `rcu_read_unlock()` are very fast [20]. However, LTTng still lacks some good features which some other tools already have such as trace support through scripting, dynamic tracepoint generation and

1. `http://lttng.org/docs/v2.8/`

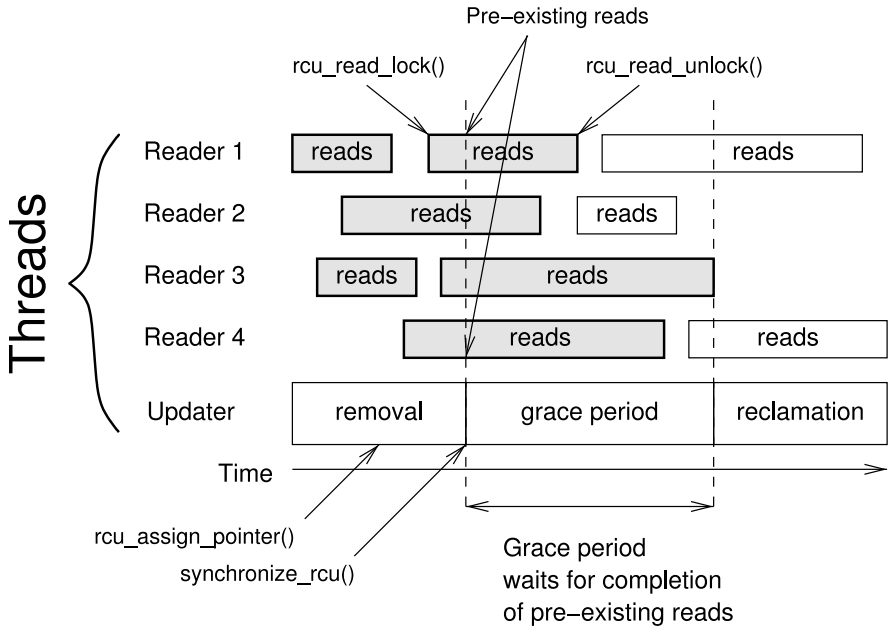

Figure 2.6 Flow of RCU handling resource critical sections [20]

insertion in userspace, and a more elaborate trace filtering framework which supports both userspace and kernel trace filtering. Its current filtering approach is slow and restricts is support for conditional filtering with a kernel-userspace support scheme. We discuss these later in Chapter 4, where we improve upon this and propose a novel scheme.

2.2 Process Virtual Machines

Process Virtual Machines (VMs) are created within a process context and are frequently used in scenarios where code may be dynamically injected in applications such as an OS kernel. Linux as well as Solaris have supported Process VMs such as DTrace and Berkeley Packer Filter (BPF) for tracing and network filtering usecases. This section explores process VMs in depth - focusing on precursors to full fledged process virtual machines, such as the techniques used in their design and bytecode interpreters. Keeping in mind the lightweight implementations required in the context of this research, lightweight and minimal VM implementations are discussed.

2.2.1 Design Strategies

From a low-level design perspective, a virtual machine can be either stack based or register based. After all, a virtual machine is actually just like a real hardware machine implemented

in software, which allows it to execute code for an architecture, other than or the same as the one on which it is being run. To execute some code for that software defined machine, it has to follow the standard procedure to emulate an actual hardware machine :

— Conversion of high level language program to the virtual machine's bytecode (akin to C $\rightarrow$ machine language conversion)
— Setup a Program Counter to keep track of each instruction in the stream
— Fetch the next instruction
— Decode operands based on the ISA
— Execute the instruction and write back data

To execute the instructions, the operands can either be stored on a stack or be considered to be registers for operations to be performed. Therefore, from this perspective, the machine can either be a stack-based or a register-based virtual machine.

Stack based VM Traditionally, a virtual stack has been used for evaluating expressions in VMs. For stack-based architectures locations of operands are implicit, following the stack pointer. In register-based machines, they would have to be specified explicitly in registers and then the operations would have to be performed on them. As an example, in an imaginary language, a code to add two numbers at the top of stack and store the result back on the stack may look something like this :

```
add          ; A+B in accumulator (ACC) now
push ACC     ; ACC pushed back to stack
```

This is of course assuming that the stack pointer is pointing to A and B which are the first two values at the top of stack. Owing to the fact that operands are implicitly defined, it would thus seem that stack-based machined would produce more compact machine code and would be more efficient than its counterpart. Most of the current VMs such as Java VM and .NET CLR are stack-based.

Register based VM Another approach is to store the operands and provide direct addresses from which to get data. These "addresses" are simply the virtual registers. A similar addition operation in a register based VM may look like :

```
add R1, R2, R3   ; A+B stored in R3
```

This assumes that values A and B are stored in registers R1 and R2 already. However, it means that these registers are not implicitly defined and they have to be specified for each

operation. This drastically reduces the number of instructions to be executed but the code size increases somewhat because of the register arguments. This means a trade-off between instruction length and instruction count. Apart from that, register based VMs have been found to be more efficient and quicker than their counterparts. For example, there may be expressions which need to be recalculated each time they appear in the code. This means that a register based VM can optimize this by storing the value in a register for repeated use. Fewer instructions also mean that the number of instruction dispatch needed will be smaller. Usually the instruction dispatch (fetching the instruction from memory) is implemented using a "switch case" and is a hard to predict indirect branch - thus it is expensive [22]. For such reasons, some newer VMs such as Dalvik and Lua VMs are register-based.

A formal study of the performance of stack and register-based VMs was done by Davis, Yunhe et al. [23]. They translated stack-based Java VM code to register-based code and independently studied how it performed. It has been observed that doing so eliminated around 47% of executed VM instructions and the corresponding machine code size gets increased by only 25%. They also observed smaller execution times (32.3% less) for the register-based machines using standard benchmarks. This points to an inclination towards usage of register-based VMs on modern processors. However, real life use cases vary a lot and it is not easy to draw any a priori conclusion favoring any specific approach.

2.2.2 Interpreter Dispatch Mechanisms

The maximum time consumed in VM execution is actually the cost of instruction dispatch [23, 22]. The actual 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 :

Switch Dispatch - The main loop of the interpreter contains a large `switch-case` statement. For each opcode in the virtual machine, there is one `case` statement. An example in listing 2.1 shows a typical function which evaluates the bytecode instruction (`instr`) and then fetches the next instruction from the instruction buffer, after incrementing the virtual program counter (`pc`).

Listing 2.1 Example of a switch dispatch

```
void evaluate() {
  while(1) {
    switch (instr) {
    case add:
       /* add */
       regs[r1] = regs[r2] + regs[r3];
       break;
    case sub:
       /* subtract */
       regs[r1] = regs[r2] - regs[r3];
       break;
    ...
    ...
    }
    instr = instr_buff[pc++];
  }
}
```

This type of dispatch is simple but is somewhat inefficient in some instances. The `break` in the end translates to an unconditional jump to the start of the loop, there is only a single indirect branch for dispatching instructions, and the branch is very unpredictable on branch predictor enabled machines. There is also a range check on opcode to be performed. However, for the JVM, there is already a separate bytecode verifier. Overall, this is an acceptable method for the generic use-cases.

Threaded Dispatch - Another method is to use *threaded code*. This is a technique where the generated code consists only of calls to subroutines. In this dispatching technique, instructions are represented by the address of the routine that implements them and the dispatch consists in fetching that address and branching to the subroutine. A threaded dispatch can be of multiple types - direct, indirect, token dispatch etc. Refer to listing 2.2.

Such an implementation is possible in non ANSI C compilers only. For instance, GNU C provides a facility for goto statements to jump to multiple different locations by usage of labels as first-class values.

Listing 2.2 Example of a switch dispatch

```
void evaluate() {
   Inst dispatch_table = { &&nop, &&add, ... }
   goto dispatch_table[*pc];

add:
   regs[r1] = regs[r2] + regs[r3];
   pc++;
   goto dispatch_table[*pc];

sub:
   regs[r1] = regs[r2] - regs[r3];
   pc++;
   goto dispatch_table[*pc];
...
...
}
```

Translation to machine code generally involves 3-4 machine code instructions for each VM instruction in a direct dispatch, while it takes 9-10 machine code instructions for each VM instruction [24]. Davis et al. have discussed the implementation and efficiency of token threaded dispatch in [22]. The dispatch code is appended to the end of the code for each VM instruction. This allows the dispatch to be scheduled more efficiently and increases the prediction accuracy of indirect branches (45% versus 2%, 20%).

2.2.3 DTrace

Even though a versatile and complete trace aggregation mechanism, we discuss DTrace in this section due to its powerful in-kernel VM approach [25]. This tool was initially developed by Sun Microsystems for its Solaris platform to perform kernel tracing, but was soon ported to MacOS, QNX etc. The concept behind DTrace is similar to other tracing techniques. The user writes instrumentation demands as a C-like D language script. The architecture consists of certain *providers* which are basically kernel modules. These modules perform a particular kind of instrumentation to create *probes* which are then used to gather data. Each provider can publish probes. The DTrace framework can connect to these probes and gather data. A library called `libdtrace` can interface the user level trace consumers to the `dtrace(7d)` kernel driver. It contains within itself the D compiler - this converts the D source to a machine independent form, DIF (D Intermediate Format), for DTrace's own

virtual machine. This is a register based RISC like machine with a small instruction set. Multiple DIF objects constitute a complete program known as a DTrace Object Format (DOF) which contains complete details about the probe, string and variable tables. This DOF is injected into the kernel at instrumentation time [26]. Whenever this probe fires as the system executes, the providers give control to the DTrace framework, which can then carry out tracing directives using the `dtrace_probe()` function. The DTrace approach to tracing, although very comprehensive, is essentially interrupt-driven and adds delays to the overall process. It is possible to insert probes in userspace applications, but this simply generates an interrupt, and the probe handler and the scripts still execute in kernel space. However, Dtrace is still one of the earliest trace techniques that employ in-kernel VMs and formed inspiration for newer techniques such as eBPF based trace aggregation, discussed later in Chapter 4.

2.2.4 Runtime Code Compilation

The previous sections dealt with process VMs and their building blocks, especially, bytecode interpreters. The VMs, for a given scenario, are able to translate the high-level expressions to bytecodes which are then sent to an interpreter and executed. This means, for each expression, that an expression tree is built and translated to bytecode at runtime and then eventually executed. Though a very flexible approach as compared to static program compilation, it is expected to have some startup overhead. In many scenarios like implementing filtering in a web-server using traditional bytecode, this implementation technique will lead to undesired overhead. This, coupled with the time to interpret and execute, has an adverse impact on performance. A way to optimize this model of execution, and reduce the overhead, is to use dynamic compilation techniques such as Just-in-time (JIT) compilation, where instead of directly executing the bytecode in the interpreter, it is first converted to native code for the given architecture and then executed. Though some startup time is still present while using JIT compilation, the speedup obtained in most cases by the use of native code overshadows the other drawbacks by a large factor.

JIT Compilation Overview

JIT compilation has been used in various scenarios and can take up multiple forms - from dynamic translation (architecture to architecture in ISA simulators) to an intermediate interpreter/runtime compiler implementation. In this section, JIT is discussed in the context of its use as an intermediate compiler invoked at runtime and running in parallel to, *or* replacing the interpretation phase - just as in the case of BPF. In [27], an early implementation

of JIT in JDK 1.1 has been discussed in detail. There are two possible execution methods after generating bytecode using `javac` - convert it to native code using JIT and then execute it on the hardware, or use the java interpreter and execute it on the JVM directly. In a traditional non-JIT scenario, at runtime, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. However, the addition of the translation to native code improves the speed to a great extent. Newer Java implementations use HotSpot technology where the code starts getting interpreted at the beginning but, as it detects that certain routines/components are being heavily used, they are dynamically compiled and executed. There are a few major drawbacks and benefits for this dynamic compilation approach, as compared to plain static compilation, which are still relevant in current JIT implementations.

Advantages :

1. One of the major advantages of dynamic compilation is that the generated code is better optimized than statically compiled code [28]. Modern JIT compilers make use of performance counters to collect information about how the program is behaving during runtime, so that dynamic compilation can be better optimized.
2. Better optimized code can be achieved by *inlining* certain methods, program control flow optimization, dead-code removal, loop combining, loop unrolling or architectural optimization during native code generation by chosing optimization strategy suited for the code profile.

Drawbacks :

1. In dynamic compilation, there is always an initial warmup time which induces a certain overhead. This may be small or big depending upon optimizations and the implementation of dynamic compilation.
2. Some optimizations can be non-deterministic in nature and are not suitable for all applications. For example, in real-time and interactive (GUI) applications, it may cause unpredictable behavior [28] if not implemented properly.

To summarize, in the case of plain interpretation of bytecode, execution is slower as compared to bytecode compiled to native code. In the case of static compilation, there will be less opportunities for optimization as compared to JIT compilation. For the limited JIT requirements in our research scenario, a significant speed improvement at a possibly very low initial startup cost can be expected. The subsections below discuss typical JIT compiler elements and subsequently some simple and moderately complex implementations of JIT for our needs.

JIT Compiler Design

A JIT compiler has to be closely coupled to the process VM or the simple bytecode generator in the machine. The modern JIT compiler design may be segregated into four major constituent elements :

VM-JIT Compiler Interface : Assuming that the decision to choose the target section of code for compilation has been reached (based on certain statistics for function use), there has to be an interface from the interpreter which will inform the compiler to begin compilation, specifying certain arguments such as the function name, bytecode etc. Such an interface has the responsibility to bootstrap and invoke the JIT compiler. This interface, along with the actual compiler code, can be considered as the compiler *frontend*. The compiler library can be dynamically loaded, or be made part of the VM itself, and provide methods like `jitCompileInit()` etc. There has to be a provision so that the compiler has access to the required data structures and code in the VM.

Compilation Logic : This can be considered as the actual code of the compiler which will begin compilation. Usually, the main argument it needs is the bytecode data structure required to be translated and it returns the executable native code. For example, GNU LibJIT has a function called `jit_function_compile` which takes as argument, the `jit_function_t` structure [29]. This structure actually contains the IR (Intermediate Representation) of the function, before it is actually compiled. Upon compilation, the actual machine code (for a specific architecture) is generated and stored in the same structure. The generation of code for most simple JIT compilers is a direct switch statement to convert individual IR statements to native code. The compiler can be more sophisticated and attempt to optimize the generated code (*optimizing compiler* or *non-optimizing compiler*). The optimizing type has the ability to selectively compile only commonly used methods, dead-code detection etc. The non-optimizing type usually compiles all the methods, which may be expensive as they consume memory as well as use up valuable time. However, it is simple to implement, and useful for small VM designs (such as filtering machines)

Code & Execution Management : Upon generation of the code, the compiler provides certain functions so that the compiled code can be invoked and executed in the VM's context seamlessly, just as it would have been interpreted. The invocation is pretty straightforward : we can pause the interpretation phase, upon reaching the desired target function, compile the function and then execute it in the same context, for example with some methods like

`jit_function_apply()` in LibJIT [29]. Upon return, we restart from where the pause happened. Before execution, the VM has to figure out how the execution control has to be switched. This can vary for different approaches. For example, in simple compilers, sometimes all the bytecode is pre-compiled in one go, at the start, and then just executed directly by the VM.

Memory Management : This is another major aspect of a JIT compiler. During the compilation and execution phase, there is a need for memory. For example, the compiled code has to be allocated to a specific memory location and has to be freed once used. Managing the stack offsets is yet another task during execution. Apart from that, if it is an implementation where a switch from the VM's interpreter to the compiled code is required repeatedly, the interpreter's PC has to be saved and the native code's jump tables have to be laid out. Also, there is global data, such as dynamically allocated data during code execution, which is supposed to go possibly in the VM memory. Most JIT compilers need to have a well defined strategy for managing memory.

2.2.5 Optimizing Filters

Similar to the filtering approach used by Ftrace [13] in the Linux kernel, DTrace [25] provides a limited filtered tracing mechanism called speculative tracing. It allows the data to be recorded tentatively in a separate speculation buffer, and later decide whether to *commit* data to the main tracing buffers or *discard* it based on the evaluation of a `speculate()` function. As an example in [25] and [26] a filtered trace of all function entries is only committed if a particular syscall such as `ioctl()` returns a failure. The speculation involves writing the data to the buffer, and possibly copying it to the principal buffer only post trace record. This filter predicate condition interpretation, coupled with the data copies, makes the overhead of this approach comparable to that of tools using bytecode interpretation, such as LTTng. Even though it seems to be a runtime filtering approach, the speculation mechanism inherently uses an Ftrace like approach of filtering post-record which provides an opportunity for optimization.

To speed up filtering of network packets being captured at user-level, a preliminary packet filtering mechanism has been introduced to UNIX for quite some time now. The goal was to have some mechanism in the kernel which gives user-level programs access to raw, unprocessed network traffic. The pioneering work done in this area (around 1980) was the CMU/Stanford Packet Filter and its inspired work - the NIT for SunOS, and Snoop in SGI IRIX. However, a major improvement was made when the BSD Berkeley Packet Filter (BPF) was introduced. It claimed performance improvements of around 10 to 150 times faster than Sun's NIT and

1.5 to 20 times faster than CSPF. [30].

BPF was designed as a register based VM and has 2 major components - the network tap subsystem and the packet filtering subsystem. The packet filter is of more interest here as it decides, based on some mechanism, whether the incoming packet is of interest to the listening application and, if yes, how much of the packet it requires. Each of the interested processes has its own user-defined filter implementation. The filters are implemented as a boolean function *true* and *false* for allowing or disallowing the packet to pass. The model for filters is control flow graph based, as it offers a significant performance advantage as compared to the traditional filter models, such as the tree based models.

Bailey et al. [31] proposed a new way of specifying filters declaratively in their Pathfinder tool. The filters could be represented as statements and translated to DAG representation. This mechanism reduces the number of times a pattern has to be evaluated for filtering. Similar to this approach, Engler et al. presented Dynamic Packet Filter (DPF) [32] where they improved upon Pathfinder's approach use of native compilation techniques, while keeping the same declarative language format of filter description. This showed an improvement of about 13x to 26x as compared to Pathfinder's implementation.

A newer approach to improve BPF's performance in the kernel was proposed by Begel et al. in BPF+ [33], where they performed compiler optimizations to eliminate redundant predicates during filter generation. One of their major contributions was to perform bytecode optimizations and propose an elementary JIT compiler for BPF to improve its performance further. They also proposed a bytecode verifying mechanism to ensure safety of executing bytecode in the BPF VM.

Swift is a new packet filtering mechanism proposed by Wu et al. recently [34]. They showed a 3x improvement over in-kernel BPF implementations of that time, mainly due to the aggressive use of SIMD instructions provided by i386 and x86_64. They propose a new VM mechanism based on a CISC like ISA and is compatible with BPF's API to simplify the code specification and generation.

BPF has been further improved recently as an *extended* BPF (eBPF) implementation in the Linux kernel [35] with enhancements to register management, bytecode generation and optimizations using a modern compiler infrastructure. Its architecture has been modified to provide a more versatile in-kernel tiny virtual machine that allows networking, tracing as well as security features to be developed over it. Apart from filtering just network packets, it can now assist in trace filtering or syscall filtering. This provided some of the foundation work for the userspace trace filtering and kernel-userspace tracing improvements we propose in our research.

2.3 Program Flow Tracing

Static analysis of binaries, to understand how the program runs, allows the developers to visually analyze how the compiler generates instructions, estimate how the instructions may execute, and can be used further for code coverage [36, 37]. Such information is also vital for debuggers to generate and aid in the breakpoint debugging approach. Recently, the focus on pure static code analysis tools has been mostly in the security domain, for analysing malicious or injected code in binaries [38, 39] or optimizing compilers based on the analysis of generated code [40]. However, the actual execution profiles can differ from what static tools can anticipate, due to the complexities of newer computer architectures in terms of pipelines, instruction prefetching, branch prediction and unforeseen runtime behaviour such as hardware interrupts. Therefore, to understand the effect of individual instructions or function blocks, the instructions executed can be profiled at runtime. These techniques constitute what we can call as Program Flow Tracing (PFT).

Ball et al. have proposed and explored in-depth, the use of counting instructions for code blocks [41]. The basic idea is to have the ability to record the instruction profile and replay it later on. Some of these earlier approaches, however, dealt with inserting instrumentation code, to profile instructions and other interesting events. As processors advanced, additional hardware counters were added to them which allows recording important events such as cache misses or instructions executed. Access to such hardware is usually provided by querying Model Specific Registers (MSR) of a processor. Hardware counter sampling based techniques have been developed and discussed earlier such as DCPI [42, 43], where authors demonstrated the use of hardware counters provided by the processor for profiling instructions. As a next step for hardware assistance in PFT generation, much more specialized hardware began to be used. Merten et al. [44] have earlier proposed the use of a Branch Trace Buffer (BTB) and their hardware table extension for profiling branches. Vaswani et al. proposed hardware profiling based on their custom-hardware solution [45], where they observed that low overhead with hardware-assisted path profiling can be achieved. Recent advances, especially in the Linux kernel, discuss how profiling tools like Perf can be used to generate execution profiles, based on data collected from special hardware counters, hardware blocks that record branches or pure software controlled sampling [46, 47]. We shall discuss more about hardware PFT techniques in subsequent subsections.

2.3.1 Software PFT Techniques

Larus et al. have discussed techniques to profile functions by observing their runtime frequency of execution using instrumentation to inject tracing code in function blocks, or control-flow edges [48]. They observed overhead of 0.2% to 5%, without taking into consideration the effect of the extra overhead of disk writes (which they observed as 24-57% in those days). This technique of function instrumentation can be eventually used to generate program flow. An eventual addition to static instrumentation technique is dynamic binary translation which allows for finer control of execution through symbolic execution. Nethercode has proposed Valgrind [49] as a JIT translation based scheme for similar software PFT. Even though this framework is more data-flow tracing oriented [50], some very insightful control-flow tools have been developed, such as Callgrind and Kcachegrind [51].

As already discussed earlier, Ftrace [13, 14], also allows a detailed kernel execution generation using its `function_graph` trace target. This allows ftrace to generate a kernel function call-stack for a given duration along with accurate timing details. Froyd et al. [52] have discussed function profiling of applications when the binaries have been optimized at compile time. They propose `csprof`, a tool to unwind a fully optimized binary at arbitary points and suggest necessary modifications to GCC for providing required DWARF2 bytecodes for efficient profiling. We now discuss more advanced techniques that allow much more accurate and high resolution traces with hardware assistance.

2.3.2 Hardware PFT Techniques

For better trace resolution and higher accuracy, recording each and every instruction executed is the key. While this may be a possibility with a software-only approach, with schemes such as those used in Valgrind, the huge overhead renders such analysis useless in real-life scenarios in production systems, and tracing each and every instruction to deduce the program flow can be quite expensive if any kind of software instrumentation is required.

Some hardware tracing modules for recent microprocessors allow the used of on-chip buffers for tracing, recording trace data from individual CPUs on the SoC, and send it for internal processing or storage. In some processors, off-chip trace buffers can also be used that allow trace data to flow from on-chip internal buffers to external devices, with industry standard JTAG ports, and to development host machines, through high performance hardware trace probes [53, 54]. The tools used around such specialized hardware are undocumented and vendor-specific and hence have not been studied further by us.

Recent research deals with compressing the trace output during the decoding phase to save

transfer bandwidth [55], while part of the focus in earlier research was on the unification of such traces as well [56]. This provides a very detailed picture of the execution at almost no overhead on the target system. The ability of hardware to record operations directly from the processor can be either a fine grained all-instruction record scheme or a lightweight branch tracing scheme. Architectures such as ARM and PowerPC provide hardware support for such mechanisms in the form of NSTrace (PowerPC), EmbeddedICE, ARM Coresight and MIPS PDTrace [57, 58]. These hardware blocks allow recording bus activity directly and then reconstruct the flow offline from the recovered data. However, the amount of data generated is too high if external devices are not used to sink the data. In such scenarios, memory buses reach saturation levels as they now try to store data locally in main memory. Such issues of memory related overhead for hardware program/data flow traces have been observed earlier as well [48, 41]. Even though hardware can generate per-instruction trace data at zero execution overhead, such an additional data flow may impact the memory subsystem.

An approach for reducing the impact of hardware tracing is choosing only those instructions that cause a program to change its flow. Instructions like direct/indirect jumps, calls, exceptions etc. are usually enough to reconstruct the program flow with the help of static binaries. The instructions between consecutive branches remain constant and hence can be obtained from the static analysis of binaries. Dedicated hardware blocks in the Intel architecture, such as Last Branch Record (LBR), Branch Trace Store (BTS) [59], and more recently Intel Processor Trace (PT) [60, 61] choose to only record branches in the currently executing code on the CPU cores. Their performance however varies a lot depending upon the technique they use. For example, LBR is an MSR based mechanism and hence can save only the last few branches. BTS allows a larger branch depth. However, it uses 24 bits per branch, which has been observed to cause an overhead between 20% to 100% for varying execution profiles [62, 63]. Intel PT is the latest and the most efficient as it uses 1 bit per branch for direct conditional branches. We elaborate more on an efficient hardware based PFT technique in Chapter 6 and Chapter 7.

2.4 Hardware Trace Reconstruction

Improvements to the generation of dynamic code and Dynamic Binary Translation (DBT) at runtime has been recently discussed by Hawkins et al. [64]. They have proposed a DynamoRIO based approach, with a 7.3 times improvement over its predecessors. Their DBT technique keeps track of JIT compilation happening at runtime by dynamically instrumenting its translation. In addition they follow a static annotation scheme to tag the dynamically generated code for analysis.

A similar technique for code reconstruction has also been proposed by Tikir et al. where they use dynamic recompilation of Java code to lazily insert instrumentation code at runtime [65]. This allows them to gather debugging information from JIT engine when required as the application executes. One of the major drawbacks is that their approach is language specific and deals with additions done to the Java Virtual Machine Debugger Interface (JVMDI) which makes it platform specific as well. Runtime compiled non-JVM code support is not possible in such cases and has not been discussed as well.

A recent patent by Koltachev et al. [66] takes a unique approach to debug JIT applications. They execute the test applications generating native code to be executed under a debug session. The dual-debug session allows control over native as well as interpreted code analysis. Lee et al. [40] have taken a similar approach where they implement runtime interposition with the help of an *agent* that handles all language transitions and allows mixed-mode debugging of code between multiple languages. This also gives them control over generated code but with the trade-off of speed and robustness.

In their research on post-mortem control-flow generation, Ayers et al. [67] point out the difficulty in keeping track of native code generated from Java or other such languages, as there is no standard way to instrument such code. Thus, they provide a customized solution, which uses runtime code instrumentation, at control-flow block granularity, for binary program analysis that records traces for post-analysis. For language agnostic approaches, Intel provides a comprehensive JIT Profiling API to report information about just-in-time generated code that can be used by performance tools. The intended use is by JIT compilers themselves, where they can use the API calls to report execution profiles by sending execution traces to the Intel VTune profiler [68]. The major drawback of this approach is that it requires a very invasive JIT engine re-compilation and a dependency on Intel's API compatibility.

2.5 Summary of Literature Review

We explained the state-of-the-art techniques being used in software-tracing, trace filtering and hardware tracing domain. We realized that the field of software-hardware interaction in tracing is a relatively new highly relevant area, and can provide ample opportunity of research in enhancing trace resolution and overhead.

Our analysis of tracing tools and our experience in building a minimal tracing system, DyTrace exposed us to the latest tools and techniques being used in this domain. We performed a detailed analysis of static and dynamic instrumentation mechanisms and their relevance in modern tracing tools eventually classifying tracing tools by domain and the underlying

techniques they use. We observe that, while tools such as LTTng provide low overhead in tracing with their efficient trace buffering and trace control techniques, other tools such as DTrace allow much better control in trace data being generated. The focus in the tracing domain has also shifted towards lowering the trace overhead, and trace filtering has been an important focus in this regard. Recent work in the Linux kernel with BPF, for network packet and trace filtering has indeed shown this. However, there has been no work done to effectively control the trace data co-operatively, from userspace and kernel as well. This encouraged us to analyze available process VMs and various ways to optimize and utilize them in our approach.

We also realized that there is an obvious lack of detailed analysis of modern trace hardware in processors. While previous generation Intel and ARM processors have been studied in relation to tracing and code profiling using ETM, Intel BTS and LBR, newer and more efficient hardware tracing facilities such as Intel PT were not evaluated in similar studies. There is an obvious interest in the domain of program flow tracing based on our study of the domain, however lack of a formal analysis has proved to be a challenge in its rapid adoption. This was an opportunity to us, as we realized in our analysis that hardware assisted techniques, when used in conjunction with software tracing, could provide a very detailed trace analysis at an ultra-low overhead.

CHAPTER 3 RESEARCH METHODOLOGY

In view of the nature of this research, it may be easy to visualize the progression as two major parallel threads - one dealing with a pure software aspect and the other with an upcoming modern hardware tracing aspect, and an eventual amalgamation of the two leading to a conclusion. We now present the way in which our research progressed over these years and how it resulted in the four papers that are now part of this thesis.

3.1 Research Progression

As discussed briefly in Chapter 1 and as illustrated in figure 3.1, we started off with investigating the software approach to enhance the trace granularity while reducing the overhead. We analyzed the deficiencies in current systems and proposed new architecture for software tracing, marked as [M1a, M1b]. Eventually, our work focused on reducing the overhead further while achieving better granularity in program-flow traces using hardware assistance. This required the development of novel algorithms that support modern hardware tracing analysis and their extension to analyze software latency and Virtual Machines [M2, M3]. We eventually discovered limitations in hardware trace reconstruction owing to our work in [M1b] which led to our concluding contribution in the hardware trace reconstruction domain through a new technique for robust tracing [M4]. This is described briefly in the following sub-sections.

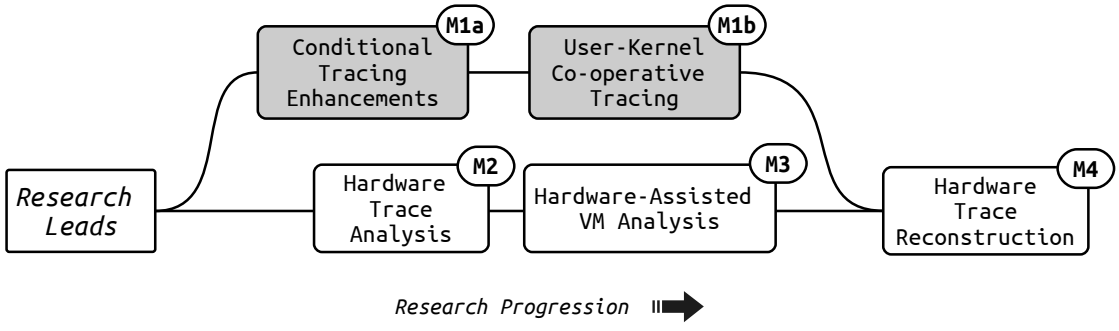

Figure 3.1 research milestones and progression

3.1.1 Research Leads

During our thorough literature review at the beginning of our research we first identified potential areas of contribution based on the state-of-the-art techniques used in tracing.

Trace Context We realized that in order to advance in our research goal of providing more granular tracing with greater context and low overhead, we could employ filtering techniques at runtime with the traces. Most of the runtime filtering approaches used in high speed tracing infrastructures, such as LTTng, were interpreted in nature, and modern techniques such as eBPF had the potential to be adapted for these fast tracers. This is specially needed for userspace events that are generated by processes that run continuously on production systems. While proposing a new filtering tracing infrastructure, it is important to not lose context while the trace size is reduced. We therefore marked it as a potential area for further investigation.

Processor Trace Support We further analyzed advancing hardware tracing support in modern processors, which is a completely new domain of research from the trace analysis perspective. It is yet to be fully exploited and, therefore, was a prime candidate to fulfill our needs of developing a more granular tracing approach. While applying software analysis techniques on VMs, we realized that hardware support for the same was severely lacking and no one had exploited these mechanisms before. There is a potential for analyzing the trace data, providing ways to isolate and generate actionable information from it and explore its advanced capabilities and its possible applications and deficiencies. Thus, this was also marked as a potential area of research.

3.1.2 Problem Definition

While we investigated the research leads, we started formalizing the problem to solve. Based on the literature survey and current tools, we decided that low-overhead, high-granularity tracing needs to be achieved with software as well as upcoming new hardware techniques and therefore defined this as the definitive problem to solve.

3.1.3 Defining Scope

Filtering and hardware tracing is a vast domain in itself. Therefore, our next step was to define the scope of our research and ensure that we fulfill the research objectives while providing valuable proof-of-concepts to demonstrate the performance of the proposed algorithms. For our first research lead, we restricted our research to production systems that generate high-speed trace events and run on commodity Linux distributions. This ensured that we could analyze and augment our experiments as we have high visibility in Linux due to its source being open. Apart from that, cutting edge advancements in our field are primarily focused on Linux and its internals. For the second research lead, even though our techniques can

be extended on any modern processor, we selected Intel's Processor Trace (PT) as the base mechanism for testing and implementation. The 6th generation Skylake processors are mature and support the newest advancements from the silicon technology perspective. It was also the first to support Linux and hence fits well within our criteria of OS selection. Our selection of experimentation framework using open tools ensures that the research scope was wide enough and that our work would be applicable to a wide variety of desktop and embedded platforms. For hardware trace analysis we focused mostly on already compressed trace data and its analysis rather than focusing on hardware implementations or improvements in silicon and architecture. Our research scope for hardware trace analysis was quite vast as well – ranging from instruction flow tracing, call graphs as well as raw trace packet analysis for a detailed view of events. For the reconstruction of traces, we assume that the test applications contain relevant DWARF symbols and that the target system supports in-memory trace record.

3.1.4 Research Body

Once we had defined the problem, scope, objectives and identified potential outcomes for the research, we proceeded on to the research and prototyping part. The main areas of our research that lead to the publications included in this text have been outlined below.

Enhanced Filtered Tracing

The idea of an in-kernel filtering mechanism has been in use before but had been applied to network packet filters. A small in-kernel process VM allowed filter expressions to be tested against incoming packet information such as source or destination IP, size and other packet headers data. One of the most common in-kernel filtering mechanisms is the Berkeley Packet Filter. Recent developments in the Linux kernel extended it further as eBPF which allowed a faster filtering mechanism. In userspace, advanced tracing tools such as LTTng have allowed high speed tracing through its lockless trace buffers, for an overall lower trace overhead, but have fallen short in trace filtering techniques. The first part of our work, corresponding to milestone [M1a], has been presented in Article 1 (Chapter 4) where we propose an enhanced filtered tracing architecture that improves the overall tracing speed by introducing an eBPF based JIT compiled tracing filter mechanism for high-speed userspace tracing.

Co-operative Conditional Tracing

In order to improve upon the conditional tracing aspect, we proposed a new technique to efficiently perform trace filtering. In addition to filters on trace payloads, our proposed ar-

chitecture allowed more advanced conditions to be set. As planned for milestone [M1b], one of the major parts of our research work was a co-operative filtering architecture that used similar bytecode expressions for kernel as well as userspace eBPF machines. This kernel eBPF (KeBPF) and userspace eBPF (UeBPF) could now directly interact with a shared memory architecture that allowed an impact of UeBPF on KeBPF and vice-versa. We demonstrated this in Article 1 with a conditional syscall trace filtering mechanism where syscalls could be recorded by *suggestive* thresholds set by UeBPF. The effect of a runtime compiled native code, from the eBPF bytecode on hardware traces was eventually observed and further analyzed by us in a latter part of our research.

Hardware Trace Analysis

While working on a software oriented approach for reducing the trace overhead, we started working towards milestone [M2]. We analyzed modern hardware tracing support on processors. As there was no prior literature on branch trace based advances in current architectures, we started prototyping and conducting performance overhead analysis of hardware tracing itself. This required the analysis of packets generated by the processor and the packet generation protocols. We realized that branch tracing could be enabled and disabled, with configuration bits set directly in the processor using model-specific registers. However, the added complexity of analyzing the branch trace data is huge.

Latency Analysis We therefore proposed a new algorithm for analyzing such data and ways to utilize it for syscall and interrupt latency computation. We selected Intel PT as a base for gathering accurate instruction profiling data and applied the algorithms developed to get latency profiles in such cases. With our new algorithm, for syscall latency and tracer overhead, hardware traces for syscalls could now be taken and their kernel call stacks observed. This enabled an accurate instruction by instruction view of each call, with a nanosecond resolution and precision. Our new algorithm worked on an instruction grouping and specialization technique that we developed to analyze instruction traces directly from hardware trace reconstruction. This work has been presented in Article 2, (Chapter 5 of this thesis).

Tracing Virtual Machines

Further analysis of hardware trace data showed a sequence of packets that enabled us to identify the time spent in processing by individual virtual CPUs in a VM. We developed a novel algorithm called *HAVAna*, that analyzed the hardware trace packets and allowed us to generate interactive resources as well as process trace views using Trace Compass, with a

nanosecond range resolution. Our algorithm worked with raw hardware trace packets, from processors that contained paging information packets to help us identify individual processes inside the VMs without any agent running inside the VM. No tracing was enabled on the host except the direct dump of hardware trace packets, which causes negligible overhead. Our research work for this milestone [M3] has been presented in Article 3 (Chapter 6 of this thesis).

Hardware Trace Reconstruction

Our experience with eBPF in userspace and in kernel, in the first part of the research, lead to a realization that there are issues with hardware trace reconstruction. This was marked as our final milestone [M4]. As branch tracing techniques such as Intel PT rely on reconstructing program flow after the fact, the reconstruction process needs application binaries and runtime information about load addresses. While this may be readily available in most cases, in multiple scenarios, such as JIT translation and static-key instrumentation in the kernel and userspace, the presence of runtime compiled code causes errors in hardware trace reconstruction. We observed this problem multiple times in our analysis during our first research phase with eBPF, and realized that no technique exists that would allow an accurate and non-intrusive way to gather and reconstruct such trace data. We therefore proposed a new technique and an algorithm called *FlowJIT*, that used operating system support to maintain copies of runtime compiled or replaced code and then use it for offline hardware trace reconstruction. We demonstrated FlowJIT with two usecases in Article 4 (Chapter 7 of this thesis).

3.1.5 Experimentation

For each of the research milestones, extensive experimentation was performed to assess the performance and overhead, either in terms of excess time spend, extra instructions incurred or a given research specific metric. We reproduced the test results multiple times and ensured that our observations were statistically significant. Appropriate representation forms such as line graphs, sun-burst graphs or density plots were used to present the results. These have been discussed in detail in subsequent chapters.

CHAPTER 4 ARTICLE 1 : ENHANCED USERSPACE AND IN-KERNEL TRACE FILTERING FOR PRODUCTION SYSTEMS

Authors

Suchakrapani Datt Sharma, *Member, IEEE* and Michel Dagenais, *Senior Member, IEEE*
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {suchakrapani.sharma, michel.dagenais}@polymtl.ca

Published in Journal of Computer Science and Technology (Springer) in November 2016

Reference as S. D. Sharma and M. Dagenais, "Enhanced Userspace and In-Kernel Trace Filtering for Production Systems", *Journal of Computer Science and Technology, Springer*, vol 4, November 2016

4.1 Abstract

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 three 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 compiled filters being even three times faster then their interpreted counterparts. We further present a new architecture, using our filtering system, which can enable co-operative tracing between kernel and process tracing VMs (virtual machines) that share data efficiently. We demonstrate its use through a tracing scenario where the user can dynamically specify syscall latency through the userspace tracing VM whose effect is

reflected in tracing decisions made by the kernel tracing VM. We compared the data access performance on our shared memory system and show an almost 100 times improvement over traditional data sharing for co-operative tracing.

4.2 Introduction

With the traditional debugging approach, it becomes quite difficult to gather very low level as well as time accurate details about the systems' 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).

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) [69]. For such cases, low overhead and 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 can be huge and contains precise time-stamps of the tracepoints hit, along with any optional event-specific information (values 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 [70, 71]. 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 composed 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. We observed that the filtering schemes used in most state-of-the-art tools are the same. This can be seen in Figure 4.1 where we generally (1) define the filter predicates in a high level statement form, (2) create a predicate tree, and possibly a more efficient bytecode representation and (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 4.1(c) is to JIT (Just-In-Time) compile the filter bytecode to native code and execute it on the machine. This yields a significant performance improvement as compared with interpreting the bytecode directly.

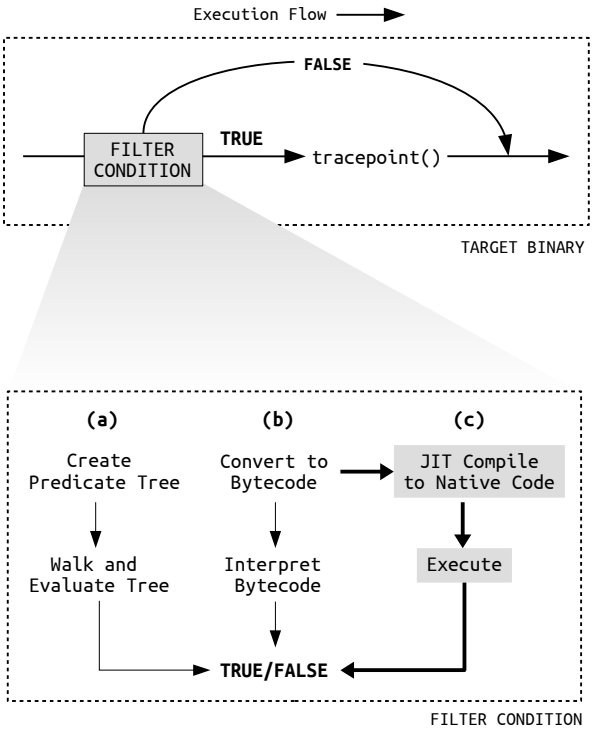

Figure 4.1 Overview of filtering in trace and debug context. The bold path (c) is the approach which yields minimum overhead

Some interesting prototyping results were reported by Alexei Starovoitov. An implementation of JIT compiled Berkeley Packet Filter (BPF) bytecode to kernel tracing demonstrated an improvement from 32 ns to 4 ns per call (best-case scenario) [35]. These improvements are in contrast to the state-of-the-art trace filtering approaches such as those in LTTng-UST

[72], which use bytecode generation-interpretation for evaluating filters (Figure 4.1(b)). We therefore improved tracing performance using JIT compiled filters by enhancing the current tracing architecture as discussed in this paper. It has proven to be more robust than the current filtered tracing techniques and has lead to reduced trace storage size, and hence the 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 technique for 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 the userspace filters, to achieve co-operative tracing.

Our major contributions in this paper are as follows,

— an improved userspace tracing framework which utilizes JIT-based trace filtering, thus reducing the trace footprint and overhead ;
— kernel and userspace co-operative tracing through a kernel VM – user VM direct data sharing mechanism that enables events in userspace to affect kernel tracing and vice versa ;
— a userspace library that provides an implementation of the two above contributions as well as a means of incorporating efficient filtering in generic tools.

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, are 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. Our evaluation of 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, and to achieve high-speed kernel-userspace tracing on resource-constrained soft-realtime systems. We discuss how the current bytecode based filtering systems evolve into 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 recording a trace, aggregating data, sharing data with userspace) based on whether conditions are 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, the inferences drawn from the results, and the directions for future work have been presented.

4.3 Literature Review

Most of the previous relevant work on filtering focused on network packets but not on tracing. McCanne and Jacobson [30] 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) [73].

In Pathfinder, Bailey *et al.* [31] proposed a new way of specifying filters declaratively. This reduces the times a pattern has to be evaluated. Engler and Kaashoek presented DPF [32] where they showed an improvement of about 13x∼26x as compared with 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+ [33], 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 [34], 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 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 [35, 74] 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. Earlier versions of BPF syntax have also been used for packet filtering in Windows.

DTrace [25], 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. Reentrancy 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 being 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 achieved using 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 [75]. 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. They 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 requiring data sharing. Furthermore, while probes can now be hooked to userspace code, they 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 Subsection 4.4.2 and Section 4.6, where comparisons with the newer eBPF approach are made.

4.4 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, ensuring that the behavior is the same, with or without tracing. We now discuss some basic concepts and relevant techniques that many state-of-the-art tracing tools employ.

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 with very little overhead. In many 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. The Linux kernel provides manually inserted static trace points using the `TRACE_EVENT` macro [12]. It exposes trace hooks on which other kernel tracing systems can be built upon. In addition to static code instrumentation, compiled code can also be inserted into binaries on disk, without any source code being available.

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 only to software for which the code is available for recompilation. Instrumentation can either be performed on the binary residing on disk or by attaching to running processes (for example, attaching a debugger to a running process). Dynamic instrumentation tools and frameworks can be built using either (1) TRAP-based approach such as in older Kprobes [3, 76] and GDB's normal tracepoints [5, 4], or (2) a trampoline-based approach such as in Dyninst [6] and PIN [77] or (3) a more elaborate JIT technique as used in tools like Valgrind [8] and PIN.

4.4.1 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 [72]. 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 [17]. 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 trampoline-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.

4.4.2 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 [78]. The basic idea of a filter F is to find a small subset S from a large input set L. The criterion of selecting S is that the application of filter F to each element i of L returns TRUE.

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

Here $F(i)$ 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 the most prominent in network packets. A lot of network traffic on the system causes packets of various protocols, sizes, having different sources and destinations, to pass through the networking device and the kernel. A user would specify its needs in the form of a Boolean expression. There are multiple approaches for evaluating the filter expressions. The concept of building trees and

evaluating them for Boolean outcomes has been used before in filters like the NIT [30] 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, evaluated them, and eventually reached a final binary decision. 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 [30, 79]. The predicates in an expression are expressed as nodes of a control flow graph (CFG). The nodes were converted to bytecode 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 139 ns to 32 ns [35]. We now discuss the ways to improve this further by techniques such as JIT compiling the bytecode.

Filter Performance Optimizations The maximum time consumed in VM execution is actually the cost of instruction dispatch [23, 22]. 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 of either 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 – which is part of the interpretation phase shown in Fig.4.1(b). Then, the next bytecode is fetched and evaluated until the bytecode program is finished.

The upgraded BPF+ implementation [33] 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 and so on. The authors of [33] 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 JIT compiled native code in certain scenarios with a varying number of predicates.

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 (Fig.4.1(c)). This considerably reduces the overhead, as stated before. The main performance

gain by JIT compiling 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 Section 4.5.

4.4.3 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 to analysis time. Trace viewing and analysis frameworks such as TraceCompass are optimized for performing complex analysis [80]. Cantrill et al. have discussed the importance of runtime filtering earlier [25]. With a better filtering infrastructure, it is possible to filter out traces at runtime as well. We now discuss some trace filtering approaches that have been used before, and then move on to explain our filtering design in Section 4.5.1.

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 then 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 [25] and [26] where the authors described 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 expressions 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 Fig.4.2, when the client encounters a filter expression for a specific userspace event to be enabled, the client first parses the expression using a custom lexer-parser and then converts it into a syntax tree.

The nodes of the syntax tree are visited and classified. Then, the intermediate representation

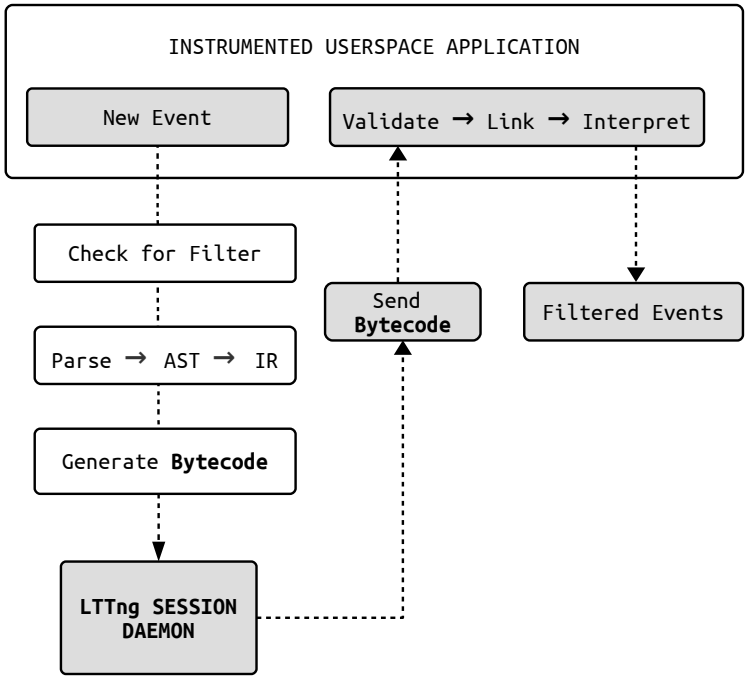

Figure 4.2 The *client* is responsible for conversion of a filter expression to the bytecode, which is sent through `lttng-sessiond` to the instrumented userspace application for validation, linking and an eventual interpretation per event

(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, the 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 filtering. There, the bytecode execution process starts. First, the bytecode is linked to the target event to create a *bytecode runtime*. Second, 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.

LTTng's interpreter is a hybrid stack/register based virtual machine. As seen in Fig.4.3, 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 [24, 81], but can be used as a normal dispatch in scenarios where compiler support is not available.

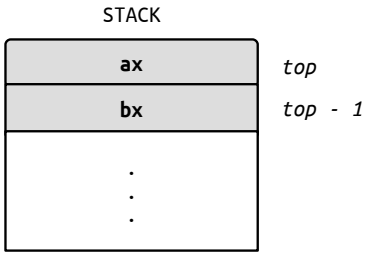

Figure 4.3 The LTTng interpreter is stack based with two registers (`ax` and `bx`) aliased to top of stack

Listing 4.1 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).

Listing 4.1 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;
}
```

However, as we have observed, both by analysis of the source code and through performance numbers discussed in Subsection 4.7.2, further optimization is possible with the use of JIT compilation, 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.

4.5 Filtered Tracing Architecture

We propose a novel userspace trace filtering architecture, with an improved overall tracing performance, as compared with 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.

4.5.1 Base Framework

eBPF The idea to convert BPF bytecode to native code, as discussed in Section 4.4.2, 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 in 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) [35, 74] 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 [82]. 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. 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 the 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) [11] is a very fast and extremely low overhead tracing tool developed at DORSAL[1] [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 [20]. Its efficiency and scalability has been demonstrated in various performance comparisons [11, 83, 18]. 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 [72]. However, LTTng-UST still lacks in areas such as

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

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-based filtering technique, LTTng-UST's fast tracing performance can lead to an improved overall performance as compared with 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.

The system architecture is shown in Fig.4.4. The filter arguments either are declared by the user manually, into 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 the information about the trace payload and the tracepoint context, which 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[2] [2]. The library either checks for the JIT support on the architecture on which it is run, or can be configured to always JIT compile the bytecode. The JIT-compiled 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. Even though our library implementation is directed towards trace filters, it can also be easily used as a basis to build generic filtering tools such as syscall filtering, or database filters in userspace. We now explain in details the various steps taken during filtering, in this proposed architecture.

Bytecode Preparation As stated before, there are two ways to provide bytecode to the interpreter (and for later JIT compilation). 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 4.2. This is useful only when the filter is small or 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

2. [2] `http://step.polymtl.ca/~suchakra/libebpf.tar.gz`

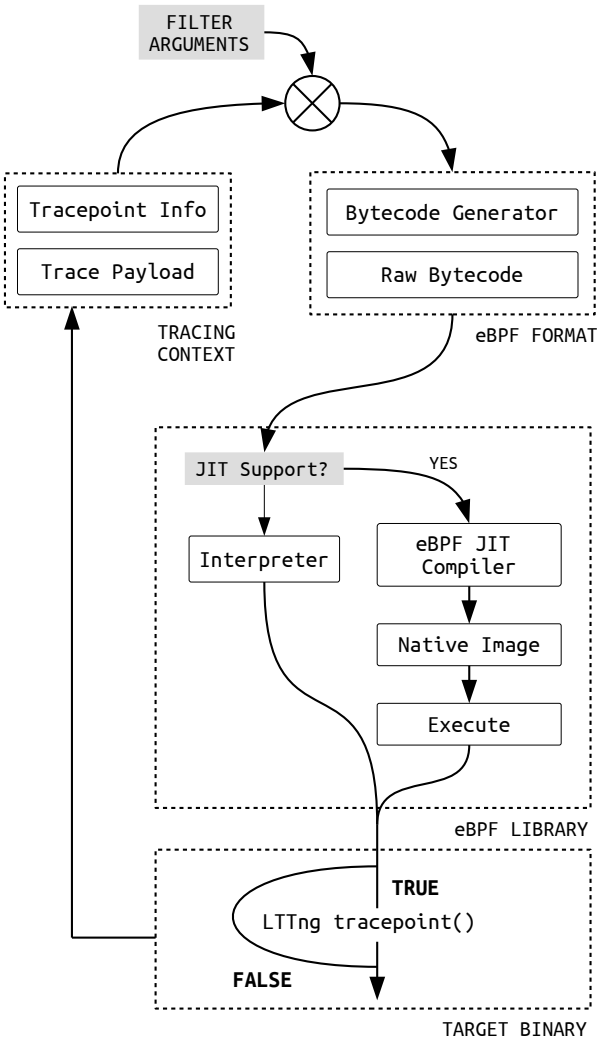

Figure 4.4 The architecture of our proposed eBPF based trace filtering system

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 opcodes on to the interpreter or the JIT compiler library. This approach is beneficial because there is an opportunity for the developer to use the optimization routines from the LLVM tools.

We now discuss some characteristics of the bytecode itself. As mentioned earlier in Subsection 4.5.1 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 4.1, derived from the filter documentation in the Linux kernel [82].

The R0 register in eBPF is where the exit value from eBPF programs is stored. Upon eBPF

Table 4.1 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

program return, R0 is set to 1 for TRUE and 0 for FALSE, just as shown in Listing 4.2. Register R1 is the place 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 therefore 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 Subsection 4.6.3, we used these helper functions to further extend the filtering system.

Listing 4.2 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
```

Native Code Compilation The main feature of the system, and the leading reason for improved performance, is the JIT compilation of the bytecode. The JIT compilation process for this library is a simple one-to-one JIT and 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 4.3, we explain the machine code compilation for Listing 4.2 on an x86-64 system.

Listing 4.3 JIT compiled 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 JIT-compiled 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 excluding the 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 the interpreter now being in the userspace, these have been 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 the userspace eBPFs, as detailed in Subsection 4.6.3. 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 the support for tail-calls so that multiple filters can be chained as desired. In our tests, for a similar filter predicate type, we could filter on 50 predicates with our design, as compared with 10 with LTTng's current interpreted filter, in the tests that we performed. The design of the experiments and our findings are elaborated in Section 4.7.

4.6 Improved Tracing Infrastructure

In Section 4.5, 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.

4.6.1 Dynamic Tracing

Some of the most interesting developments in the tracing infrastructure have been the ability to dynamically insert tracing probes and take actions when these dynamic probes are hit. The dynamic tracing tools at kernel level are available with different granularities. 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 [84] and DTrace [26]. These tools 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 through 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 with the enhanced optimizations performed in LLVM for eBPF bytecode. This DIF would be compiled by the assembler to the DIF object (DIFO). 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 four 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 infrastructures 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 into 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 with other dynamic tools [85].

4.6.2 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. discussed various kernel-userspace data sharing mechanisms before [86].

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 the 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 the two-way sharing of data (in BPF maps) from userspace to kernel, based on the `bpf` syscall (refer to Fig.4.5).

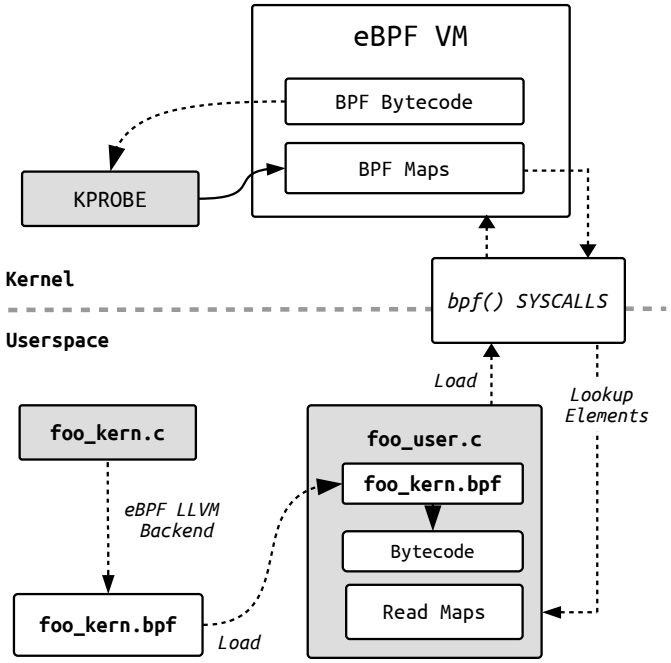

Figure 4.5 An eBPF program in its current form, with the kernel part (`foo_kern.c`) and a userspace part (`foo_user.c`). The userspace part uses the `bpf()` syscall to load bytecode in the eBPF kernel VM, as well as reading and updating data in *BPF maps*

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 has already been 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 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 that offered by currently available tools. We now discuss our co-operative tracing approach based on eBPF and LTTng.

4.6.3 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 [87] (refer to Fig.4.5). 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.

Illustrative Use Case

We show the importance of the 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 probes the `sys_enter` and `sys_exit` trace events along with the time-stamps for each syscall. We can thus 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 is able to 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 then conditionally decide to trace or not. 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 could share data directly. Other approaches, such as Perf-based events and LTTng's data sharing, use fast shared memory as well, but 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, thus 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 Fig.4.6, we have a process with PID 42 that registers with our syscall latency tracker module. 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. The sharing itself is based on a `mmap()` based implementation where KeBPF allocated the required memory during initialization and then mapped it to the userspace address space for direct access. The user can write data using the helper function, and thus can have direct bi-directional access and a zero-copy overhead. Therefore, the thresholds can be dynamically adjusted, and the kernel tracing output can be controlled. In addition, the userspace can continually fill the `proc_state` structure with current *process state* so as to control other parts of kernel/userspace tracing. In contrast, as of now, the state of the art 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 with a direct read or write in our shared memory. We implemented the VM-VM sharing by allocating the page in the kernel and then mapping the memory to grant access from userspace. In addition, in the default eBPF maps, each value requires an explicit copy in the kernel from userspace, which is avoided in our shared memory approach.

In our tracing approach, kernel and userspace scripts are each executed in their respective context. This enables very fast userspace tracing, avoiding context switches or traps at each

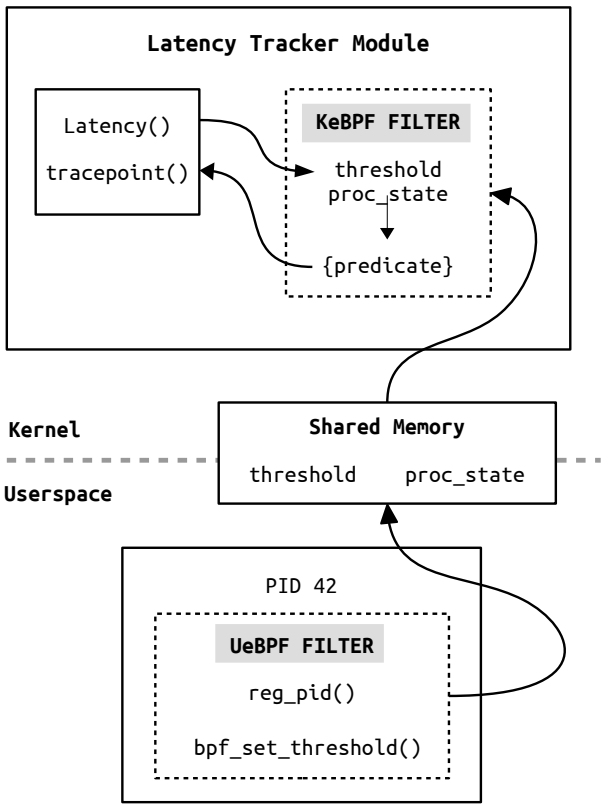

Figure 4.6 The KeBPF-UeBPF shared memory implementation showing syscall latency thresholds being set dynamically from within a UeBPF filter program

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 shown later 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., the number of bytes read, the 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. As an example, the userspace RCU algorithms have proven to provide near zero read-side overhead for concurrency control applications. It accomplishes this by allowing reads to occur concurrently with updates. RCU maintains multiple versions of objects and makes sure they are not discarded until all pre-existing read side critical sections are finished [20, 88]. These 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.

Our proposed approach allows a direct link between two VMs, one in userspace and the other 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.

4.7 Experimentation and Results

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 with the `bpf()` syscall based approach, used by the default in-kernel eBPF implementation.

4.7.1 Test Environment

All tests were done on a machine running Fedora 20 with the 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.

4.7.2 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 with a baseline value (the 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 [3] [3] with *operator chaining*. As shown in Fig.4.7, 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,

$$T = t_K + t_e + t_t \tag{4.1}$$

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 Fig.4.8). This measured the performance of eBPF-JIT 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 run consisting of 100 million events. Varying the DoE meant that the filter was evaluated to be FALSE, and skipped 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 evaluated to be TRUE. We varied the DoE from $P = 5$ to $P = 40$ with steps of 5. Refer to Figure 4.9. 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

3. [3] `http://step.polymtl.ca/~suchakra/libebpf-benchmarks.tar.gz`

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 as the preparation time for filters is amortized over a large number of executions, and is negligible under such conditions. The intended use case is high performance trace filters, with high frequency events observed over long durations.

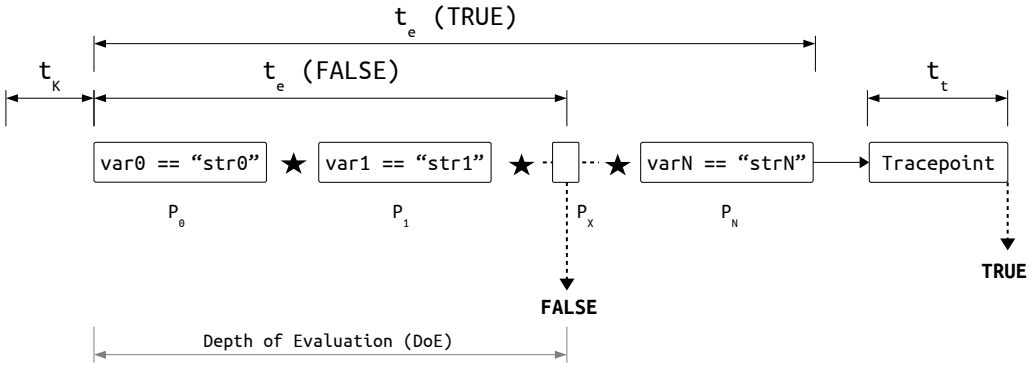

Figure 4.7 Design of the trace filter benchmark. Evaluation time t_e depends on DoE.

In this same experiment, we compared the time taken by similar LTTng-UST's interpreted filters with that by our eBPF interpreted and JIT approach. However, we limited the number of predicates to only nine variables, due to LTTng currently not allowing more than nine distinct string variables as trace payload (refer to Fig.4.10 for results).

Observations In the first test case for filter optimizations, also shown in Fig.4.8, we observed that for 100 million events and a 50 predicates filter, the interpreted eBPF filter was 4.3x slower than the hard-coded filter (considered as a lower-bound reference). The native compiled eBPF filter, however, was 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 were executing native machine code, we could see that this small overhead was due to extra instructions being executed for each filter, as seen in Listing 4.3. The overall filter performance was 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 with similar interpreted filters, for tracing scenarios with long predicates and high event frequency.

In the second test case, as shown in Fig.4.9, we observed that with a constant 100 million events and an increasing number of predicates, the benefit of natively compiled eBPF trace filters increased marginally. The performance of a 10 predicate JIT compiled eBPF filter was 3.1x better than a similar interpreted filter. This increased to 3.2x for 20 predicates, and a

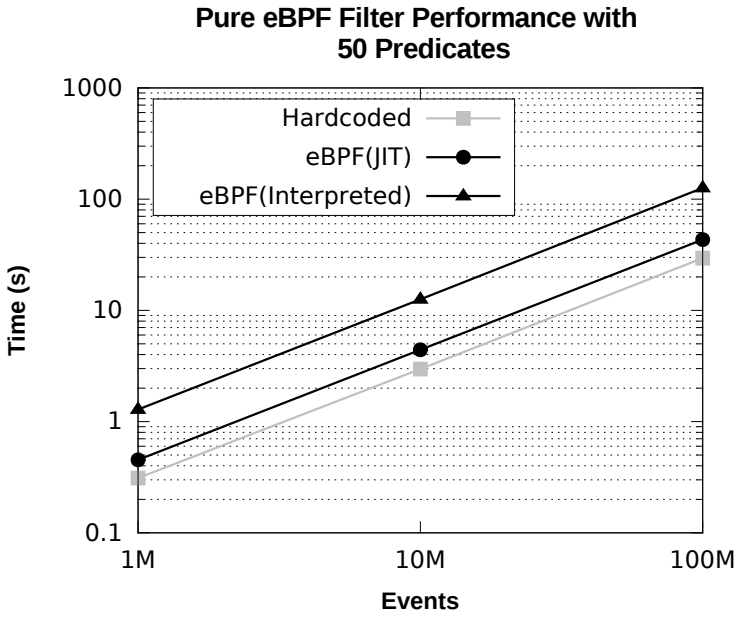

Figure 4.8 Pure eBPF filter performance with a 50 predicate TRUE biased AND chain

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.

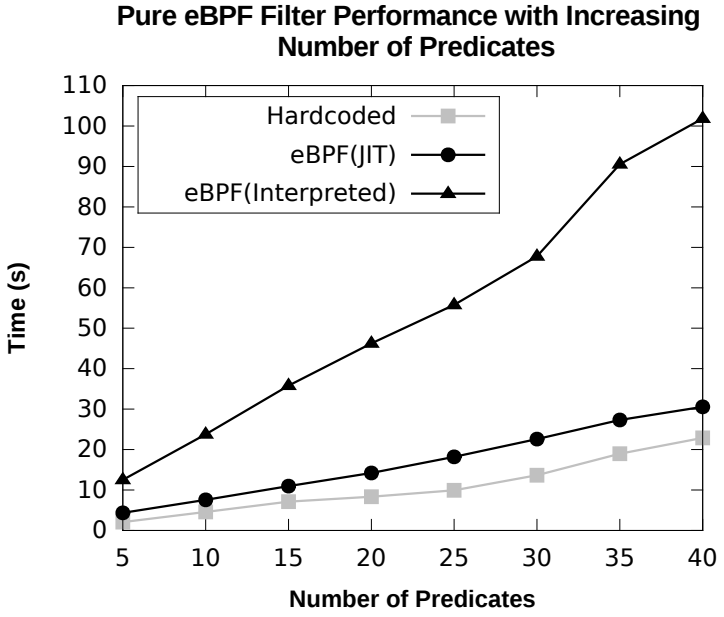

Figure 4.9 Pure eBPF filter performance with 100M events and a TRUE biased AND chain

In the third test scenario, we compared LTTng's interpreted filter performance with eBPF's

JIT compiled filter performance by observing the biased FALSE cases in Fig.4.10.

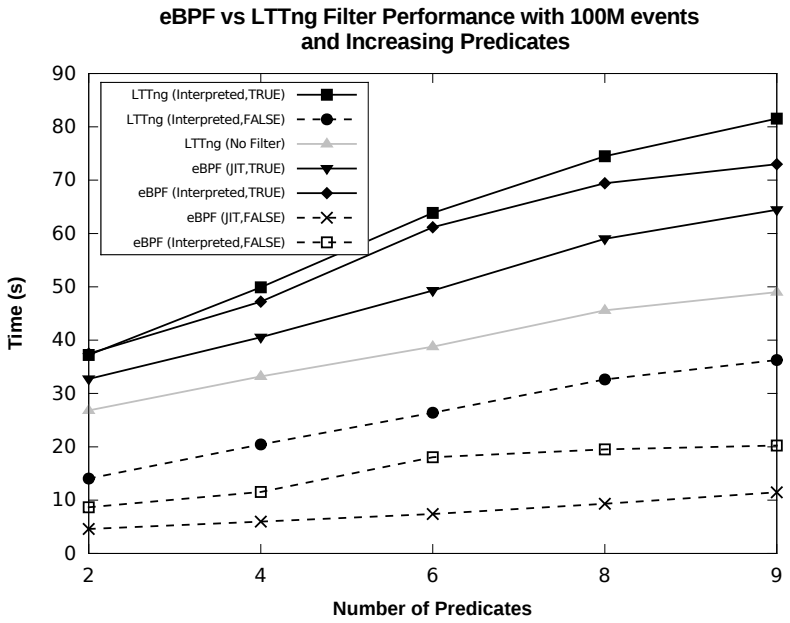

Figure 4.10 eBPF vs LTTng's filter performance with increasing number of TRUE/FALSE biased AND chain predicates

For nine predicates, eBPF's JIT 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 now see how these filters fared with LTTng tracing enabled. In that case, if the filter was evaluated to TRUE, the tracepoint was recorded and the observed time included the tracepoint time. We compared 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 nine AND chained string comparisons, biased to TRUE, the interpreted LTTng had an overhead of 325 ns/event, as compared with eBPF-JIT filter's overhead of 154 ns/event, when LTTng (no filter) was taken as reference. Our JIT based approach was therefore 2.1x faster. This demonstrates the fact that, with the small cost of JIT-based filtering with our library (154 ns/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.

Memory Overhead

Tracing itself may consume a significant amount of memory, some added instructions and data in the program executable, but more importantly, per-CPU structures to buffer events

until they are streamed to disk or the network. For instance, in a typical high throughput tracing setup, LTTng can even be configured to use 64 sub-buffers of 8 MB each. For each tracepoint for which a condition is added with our proposed scheme, a small data structure is needed for the interpreted case to represent the trace actions and predicates. When JIT compilation is used for the condition bytecode, additional memory is required to store the compiled code. Either way, the additional memory consumed is insignificant and much lower at 32 KB for a complex 50 predicates eBPF filter, as compared with the memory requirements of the trace buffers themselves.

4.7.3 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 compared it 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 looked up these values from userspace using the `bpf()` syscall with arguments `BPF_MAP_LOOKUP_ELEM` and measured the time for multiple runs. We compared 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()`.

Observations 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 4.2. The default `bpf()` implementation took 218 ns/read whereas our shared memory implementation took 2.2 ns/read, which can be explained by the fact that the shared memory can be directly accessed, without needing a system call, as detailed in Subsection 4.6.3.

Table 4.2 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 was close to the baseline values taken using simple read calls. eBPF-syscall, in Table 4.2, shows the time taken to read using the `bpf()` syscall. Going through the eBPF code in the kernel we observed 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 was no extra copy involved, and KeBPF and UeBPF could share data directly at a high speed, as observed in our test.

4.8 Conclusion and Future Work

In this paper we presented 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 into 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 showed that, with our technique, we could 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. We devised a rigorous benchmark and found that the performance of the new JIT-based filtered tracing architecture is 3x that of the current interpreted approaches. We provided the implementation in the form of a generic userspace eBPF filtering library that can be used to improve other trace filtering scenarios such as network packets or syscall-based sand-boxing in userspace.

As the 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 co-operative 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 the 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, however, 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 satisfy 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 scrip- based filtering when required, while keeping all the benefits of low overhead and speed provided by LTTng.

The UeBPF library could also benefit from more explicit support for data sharing through multiple threads. We can also port it to non-Linux platforms as a generic library for trace, network packet or syscall filtering. This would expand it from its current Linux specific implementation. 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) [89], where eBPF programs from userspace could conditionally trigger a PT snapshot.

Acknowledgment We would like to thank Alexei Starovoitov for his recent eBPF related work in the Linux Kernel, authors of LTTng project for their technical guidance and Francis Giraldeau for his comments.

CHAPTER 5 ARTICLE 2 : HARDWARE-ASSISTED INSTRUCTION PROFILING AND LATENCY DETECTION

Authors

Suchakrapani Datt Sharma and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {suchakrapani.sharma, michel.dagenais}@polymtl.ca

Published in Journal of Engineering (IET) in August 2016

Reference as S. D. Sharma and M. Dagenais, "Hardware-Assisted Instruction Profiling and Latency Detection", *Journal of Engineering, IET*, November 2016, DOI : 10.1049/joe.2016.0127

5.1 Abstract

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

5.2 Introduction

Modern systems are becoming increasingly complex to debug and diagnose. One of the main factors is the increasing complexity and real-time constraints which limit the use of traditional debugging approaches in such scenarios. Shorter task deadlines mean that the faithful reproduction of code execution can be very challenging. It has been estimated that developers spend around 50% to 75% of their time debugging applications at a considerable monetary cost [55]. In many scenarios, *heisenbugs* [90] become near impossible to detect. Long-running systems can have bugs that display actual consequences much later than expected, either due to tasks being scheduled out or hardware interrupts causing delays. Important parameters that need to be analyzed while doing a root cause analysis for a problem include the identification of costly instructions during execution, the detection of failures in embedded communication protocols, and the analysis of instruction profiles that give an accurate representation of which instructions consume the most CPU time. Such latent issues can only be recorded faithfully using tracing techniques. Along with accurate profiling, tracing provides a much needed respite to developers for performance analysis in such scenarios.

We focus in this work on two important common issues in current systems : the efficient detection/tracking of hardware latency and the accurate profiling of syscalls and instructions, with an ability to detect program control flow more accurately than with current software approaches. We discuss our new analysis approach, which utilizes conditional hardware tracing in conjunction with traditional software tracing to accurately profile latency causes. The trace can be decoded offline to retrieve the accurate program flow even at instruction granularity, without any external influence on the control flow. As software tracing can induce changes in the control flow, with our system we can further detect the cause of latency induced by the software tracers themselves, on the software under observation, to nanosecond range accuracy.

Pure software profiling and tracing tools consume the already constrained and much needed resources on production systems. Over the years, hardware tracing has emerged as a powerful technique for tracing, as it gives a detailed and accurate view of the system with almost zero overhead. However, the IEEE Nexus 5001 standard [91] defines 4 classes of tracing and debugging approaches for embedded systems. Class 1 deals with basic debugging operations such as setting breakpoints, stepping instructions and analysing registers - often directly on target devices connected to hosts through a JTAG port. In addition to this, Class 2 supports capturing and transporting program control-flow traces externally to host devices. Class 3 adds data-flow traces support, in addition to control-flow tracing, and Class 4 allows emulated memory and I/O access through external ports. Hardware tracing modules for

recent microprocessors (Class 2-Class 4) can either utilize (1) on-chip buffers for tracing, recording trace data from individual CPUs on the SoC, and send it for internal processing or storage, or (2) off-chip trace buffers that allow trace data to flow from on-chip internal buffers to external devices, with specialized industry standard JTAG ports, and to development host machines, through high performance hardware trace probes [53, 54]. These hardware trace probes contain dedicated trace buffers (as large as 4 GB) and can handle high speed trace data. As we observed in our performance tests (section 5.5), the former approach can incur overhead in the range of 0.83% to 22.9%, mainly due to strain on memory accesses. We noted that trace streams can generate data in the range of hundreds to thousands of MB/sec (depending on trace filters, trace packets and packet generation frequency). Thus, there is a tradeoff in choosing either an external analysis device or on-chip buffer recording. The former gives a better control, (less dependency on external hardware which is curcial for on-site debugging), but incurs a small overhead for the memory subsystems on the target device. The latter provides a very low overhead system but requires external devices and special software (often proprietary) on development hosts. The generated trace data is compressed for later decoding with specialized debug/trace tools [92, 93] that run on host machines, as illustrated in Figure 5.1.

Device memory is limited, thus there are multiple ways to save tracing data using either of the two approaches discussed above. Therefore, to achieve maximum performance, recent research deals with compressing the trace output during the decoding phase to save transfer bandwidth. [55]. Earlier, part of the focus was on the unification of the traces, which is beneficial for Class 3 devices [56]. This provides a very detailed picture of the execution at almost no overhead on the target system.

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

The remainder of the paper is organized as follows. Section 5.3 gives a general overview of program-flow tracing, its requirements, and limitations of the current sampling systems to

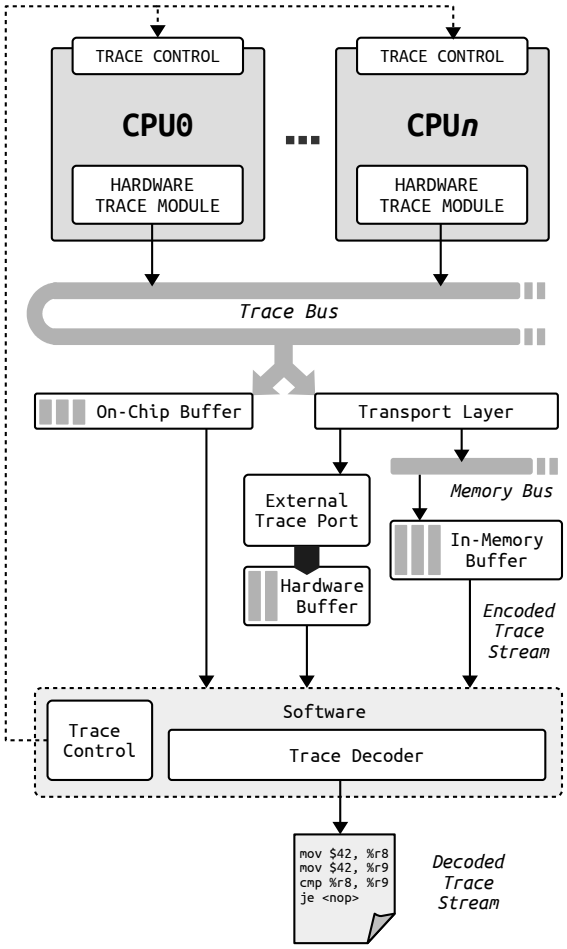

Figure 5.1 Hardware tracing overview

handle these. It also introduces the concept of hardware tracing to overcome such limitations. We discuss state-of-the-art techniques used in software tracing and finally concentrate on our research scope. In section 5.4, we introduce our hardware and hardware-assisted software tracing based architecture. We then elaborate our three contributions that introduce an instruction and time delta profiling technique to identify interrupt latencies, profiling syscall latency in short sections of code and identify causes of latency in software tracers. These contributions utilize our hardware-assisted approach. In section 5.5, as our final contribution, we start with a detailed experiment, measuring overhead and trace size, for Intel's Processor Trace hardware tracing infrastructure as it forms the core of our hardware based approach. We have also proposed a new metric to define the granularity of the temporal and spatial resolution in a trace. We then show how the hardware based trace profiler can help visualize anomalies through histograms.

5.3 Background

In an ideal environment, developers would want to know as much as possible about the flow of their programs. There are three important artifacts that can be gathered during a program execution - flow of instructions during program, their classification, and deduction of the program flow with timing information. Static analysis of binaries, to understand how the program runs, allows the developers to visually analyze how the compiler generates instructions, estimate how the instructions may execute, and can be used further for code coverage [36, 37]. Such information is also vital for debuggers to generate and aid in the breakpoint debugging approach. Recently, the focus on pure static code analysis tools has been mostly in the security domain, for analysing malicious or injected code in binaries [38, 39] or optimizing compilers based on the analysis of generated code [40]. However, the actual execution profiles can differ from what static tools can anticipate, due to the complexities of newer computer architectures in terms of pipelines, instruction prefetching, branch prediction and unforeseen runtime behaviour such as hardware interrups. Therefore, to understand the effect of individual instructions or function blocks, the instructions executed can be profiled at runtime. The use of counting instructions for blocks of code, at program execution time, has been proposed and explored in-depth before [41]. Therefore, the instruction sequence and profile can be recorded and then replayed later on. However, some of these earlier approaches dealt with inserting instrumentation code, to profile instructions and other interesting events. Sampling based techniques, developed earlier such as DCPI [42, 43], have also been discussed extensively before, where authors demonstrated the use of hardware counters provided by the processor for profiling instructions. Merten et al. [44] have earlier proposed the use of a Branch Trace Buffer (BTB) and their hardware table extension for profiling branches. Custom hardware-based path profiling has been discussed by Vaswani et al. [45], where they observe that low overhead with hardware-assisted path profiling can be achieved. Recent advances, especially in the Linux kernel, discuss how profiling tools like Perf can be used to generate execution profiles, based on data collected from special hardware counters, hardware blocks that record branches or pure software controlled sampling [46, 47].

5.3.1 Program Flow Tracing

Recording instruction flow or branches in a program can provide information about how a program actually executes in comparison to its expected execution. The comparison of an anomalous program flow trace with that of a previous one can let the developer know what was the effect of changes on the system. It can also be used to track regressions during new feature additions. At lower levels, such as instructions flow, bugs that occur in long running

real-time systems can now be detected with more confidence, as the complete execution details are available. With recent hardware support from modern processors, this has become easier than ever. We discuss details about such hardware support further in section 5.3.2. Larus et al discussed quite early about using code instrumentation to inject tracing code in function blocks, or control-flow edges to track instructions or deduce the frequency of their execution [48, 41]. They observed overhead of 0.2% to 5%, without taking into consideration the effect of the extra overhead of disk writes (which they observed as 24-57% in those days). Other more powerful tools, which effectively perform execution profiling or control-flow tracing, can be built using similar binary modifying frameworks such as Valgrind [49]. Even though this framework is more data-flow tracing oriented [50], some very insightful control-flow tools have been developed, such as Callgrind and Kcachegrind [51]. Program-flow tracing can either encompass a very low level all-instruction trace generation scheme, or a more lightweight branch-only control-flow trace scheme.

Instruction Tracing Tracing each and every instruction to deduce the program flow can be quite expensive if instrumentation is required. Hence, architectures such as ARM and PowerPC provide hardware support for such mechanisms in the form of NSTrace (PowerPC), EmbeddedICE, Embedded Trace Macrocell (ETM), Program Trace Macrocell (PTM) (now part of ARM CoreSight) and MIPS PDTrace [57, 58]. The basic idea is to snoop the bus activity at a very low-level, record such data and then reconstruct the flow offline from the bus data. External hardware is usually connected as bus data *sink* and special software can then use architecture level simulators to decode the data. The benefit of a complete instruction flow trace is that there is highly detailed information about each and every instruction for accurate profiles, in-depth view of memory access patterns and, with the support of time-stamped data, a very accurate overall tracer as well. However, the amount of data generated is too high if external devices are not used to sink the data. Indeed, memory buses are usually kept busy with their normal load, and an attempt to store the tracing data locally incurs bus saturation and additional overhead. An approach to reduce such bandwidth and yet keep at least the program-flow information correct is to use branch only traces.

Branch Tracing The issue of memory related overhead for hardware program/data flow traces has been observed earlier as well [48, 41]. Even though hardware can generate per-instruction trace data at zero execution overhead, such an additional data flow may impact the memory subsystem. Hence, just choosing the instructions that cause a program to change its flow greatly reduces the impact. Such control-flow instructions (like direct/indirect jumps, calls, exceptions etc) can indeed be enough to reconstruct the program flow. Dedicated hard-

ware blocks in the Intel architecture, such as Last Branch Record (LBR), Branch Trace Store (BTS) [59], and more recently Intel Processor Trace (PT) choose to only record branches in the currently executing code on the CPU cores. By following the branches, it is quite easy to generate the instruction flow with the help of additional offline binary disassembly. For example, for each branch instruction encountered in the flow, a record for the branch taken/not-taken and its target can be recorded externally. This is then matched with the debug information from the binary to reconstruct how the program was flowing. We detail and discuss the branch tracing approach, as well as instruction tracing, in section 5.3.2, where we show a state-of-the-art branch tracing approach using Intel PT as an example.

5.3.2 Hardware Tracing

As discussed previously, the complete instruction and branch tracing is supported by dedicated hardware in modern multi-core processors. They provide external hardware recorders and tracing devices access to the processor data and address buses. ARM's early implementation of EmbeddedICE (In-circuit Emulator) was an example of this approach. Eventually, processor chip vendors formally introduced dedicated and more advanced hardware tracing modules such as CoreSight, Intel BTS and Intel PT. In a typical setup, such as shown in Figure 5.1, trace data generated from the trace hardware on the chip can be funneled to either the internal buffer for storage, or observed externally through an external hardware buffer/interface to the host development environment, for more visibility. In both cases, the underlying techniques are the same but performance varies according to the need of the user and the hardware implementation itself.

Tracing Primitives

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

Trace Configuration Most of the hardware trace modules can be fine-tuned by writing data to certain control registers, such as Model Specific Registers (MSR) in Intel or the

Coresight ETM/ETB configuration registers for ARM. For example, writing specific bits in MSRs can control how big a trace buffer will be or how fine-grained or accurate the timing data will be generated. An optimum configuration leads to better trace output - the effect of which is discussed later in this paper.

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

— **Taken-Not-Taken** : For each conditional direct branch instruction encountered (such as jump on zero, jump on equal), the trace hardware can decode if that specific branch was taken or not. This is illustrated with Intel PT's trace output as an example in Figure 5.2. We can observe that Intel PT efficiently utilizes 1 bit per branch instruction to encode it as a taken or not taken branch.

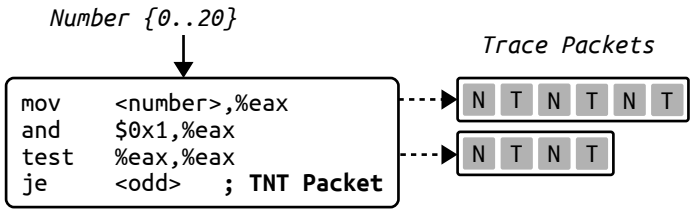

Figure 5.2 An odd-even test generates corresponding Taken-Not-Taken Packets

The earlier implementations such as Intel BTS used 24 bits per branch, which caused an overhead between 20% to 100% as the CPU enters the special debug mode, causing a 20 to 30 times slowdown [62, 63].

— **Target IP** : Indirect unconditional branches (such as register indirect jumps) depend on register or memory contents, they require more bits to encode the destination (Instruction Pointer) IP of the indirect branch. This can also be the case when the processor encounters interrupts, exceptions or far branches. Some implementations such as Intel PT provide other packets for updating control flow, such as Flow Update

Packet (FUP), which provide source IP for asynchronous events such as interrupts and exceptions. In other scenarios, the binary analysis can usually be used to deduce the source IP.

— **Timing** : Apart from deducing the program-flow, packets can be timed as well. However, time-stamping each and every instruction can be expensive in terms of trace size as well as extra overhead incurred. ARM CoreSight provides support for accurate time-stamp per instruction. However, the usecase is mainly aimed at usage of an external high speed buffer and interface hardware through a JTAG port. For on-device tracing such as Intel PT, the packet size can be kept small by controlling the frequency of time-stamps being generated and the type of time-stamps. For example, a timing packet can either be the lower 7 bytes of the TSC value as an infrequently recorded TSC packet or can be just a more frequent 8-bit Mini Timestamp Counter (MTC) packet occurring in between two TSC packets. MTC packets record incremental updates of CoreCrystalClockValue and can be used to increase the timing precision with fewer bits utilized. Trace timing in some implementations can further be improved by a cycle accurate mode, in which the hardware keeps a record of cycle counts between normal packets.

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

5.4 Trace Methodology

In order to get useful instruction profiling and tracing data for use-cases such as accurate detection of interrupt latency, we propose a framework that utilizes a hardware-assisted software tracing approach. The major focus of our work is on post-mortem analysis of production systems. Hence, the underlying technologies used aim at recording raw trace or program-flow data at runtime, and eventually perform an offline merge and analysis, to get in depth information about abnormal latency causes, or generate instruction execution profiles. The data generated in hardware tracing can reach a range of hundreds of MB per second. Various approaches have been taken to reduce this overhead. Apart from careful configuration of the trace hardware, various methods such as varying the length of TNT packets (short/long), IP compression, indirect transfer return compression [61] are employed to control precisely the trace size, with the aim of reducing memory bus bandwidth usage. Previous work often focused on trace compression and even better development of tracing blocks itself [55]. In contrast, we chose to leverage the latest state-of-the-art hardware such as Intel PT and carefully isolate interesting sections of the executed code to generate short hardware traces.

These short traces can be eventually tied to the corresponding software trace data to generate a more in-depth view of the system at low cost. This can also be used to generate instruction execution profiles in those code sections for detecting and pinpointing anomalous sections of the program, right down to the executed instruction. This gives a unique and better approach as compared to other techniques of sample based profiling (such as Perf) or simulation/translation based profiling (such as Valgrind) mainly due to the fact that there is no information loss, as the inferences are based on the real instruction flow in a program, and the overhead of simulated program execution is completely removed. Choosing only specific sections of code, to trace and profile with hardware, also means that we do not require external trace hardware and can rely on internal trace buffers for post-mortem analysis. In this section, we first show the design of our framework itself and demonstrate how we can use Intel PT hardware-assisted software tracing. We also explain our three main contributions, detailing how we could profile interrupts/syscall latency and evaluate the impact of software tracers themselves. We start with some background on Intel PT, and then explain the architecture of our technique.

5.4.1 Intel PT

Intel's MSR based Last Branch Record (LBR) and the BTS approach for branch tracing have been widely explored before [62, 63]. Eventually, the benefits of the hardware tracing approach advanced the branch-tracing framework further, in the form of Intel PT. Branch trace data with PT can now be efficiently encoded and eventually decoded offline. The basic idea, as shown in Figure 5.2, is to save the branching information during program execution, encode and save it. Later on, the trace data along with run-time information such a process maps, debug-info and binary disassembly, we can fill in the gaps between the branches and form a complete execution flow of the application. During the decoding of the compressed recorded branch data, whenever a conditional or indirect branch is encountered, the recorded trace is browsed through to find the branch target. This can be merged with debug symbols and static analysis of the binary code to get the intermediary instructions executed between the branches. Therefore, with Intel PT's approach, we do not need to exclusively store each and every instruction executed but just track branches - allowing on-device debugging and less complex implementation of hardware and debugging software. Apart from that, with the simple MSR based configuration of the hardware, we have the ability to set hardware trace start and stop filters based on an IP range to allow a more concise and efficient record of trace at runtime. The decoder has been open sourced by Intel for a rapid adoption in other tools such as Perf [94]. We incorporated PT in our framework, owing to its low overhead and versatility as presented later in section 5.5.2 where we discuss its performance and overhead.

5.4.2 Architecture

We developed a framework based on the PT library, for decoding the hardware trace, and a reference PT driver implementation provided by Intel to enable, disable and fine tune trace generation [95, 96]. An overview of our hardware-assisted trace/profile system is shown in Figure 7.3. The Control Block is a collection of scripts to control the PT hardware in CPUs through the simple-pt module. The control scripts can be used for configuring the PT hardware for filtering, time-stamp resolution and frequency control. It can enable/disable traces manually for a given time period. This is achieved by setting the `TRACE_EN` bit in the `MSR_IA32_RTIT_CTL` control register that activates or deactivates the trace generation. We use the control scripts for generating raw instruction and latency profiling data. Any PT data

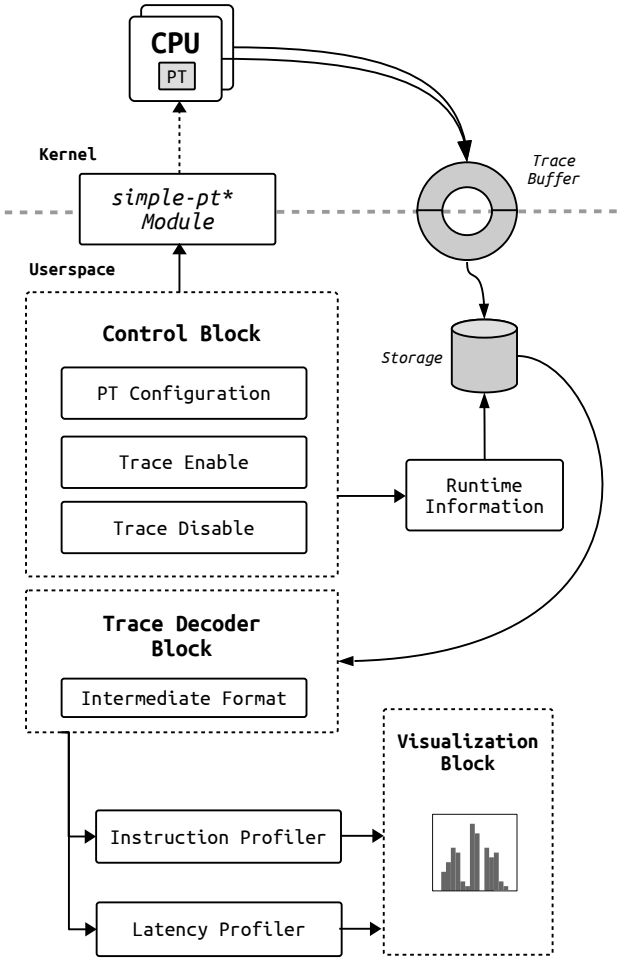

Figure 5.3 The architecture of our proposed hardware-assisted trace-profile framework. Simple PT is used for trace hardware control

generated is sent to a ring-buffer, the content of which is dumped to the disk for offline analysis. Along with the trace data, runtime information such as process maps, CPU information and features are saved as *sideband* data. This is essential for trace reconstruction. The Trace Decoder reconstructs the control flow based on runtime information, debug information from binary, and per-CPU PT data. The decoding process involves reading the processor trace binary byte by byte. Much like assembly opcodes, packets are identified by an arrangement of bytes. We go through individual incoming bytes from the trace buffer to identify the packet and its contents. This is merged with the dissembled binary information to give the packets an execution context. This data is converted to our intermediate format that is consumed by the Instruction Profiler and Latency Profiler modules which can generate visualizations. Our intermediate format consists of a stream of instructions with markers identified by function names. This is converted to visualization data by transforming the markers to a call-stack with instruction or time-delta. More about this is explained in subsequent sections. We now elaborate on our three contributions that cover instruction/syscall latency profiling and the performance impact of software tracing systems on modern production systems.

5.4.3 Delta Profiling

As seen in Figure 7.3, the raw PT data from the processor can be reconstructed to generate a program control flow. It is, however, important to analyze this huge information in a meaningful manner. Therefore, we present an algorithm to sieve through the data and generate instruction execution profiles based on patterns of occurrence of instructions in the control flow.

These profiles can be used to represent the histograms based on a time-delta or an instruction count delta. This can work for single instructions to observe histograms during a simple execution, as well by just counting the occurrence of a set of instructions. This approach is significantly different from sample based profiling, as it is based on true instruction flow and can pinpoint errors at finer granularity in short executions. As interrupts are quite significant in embedded production systems, we choose to profile instructions that are responsible for disabling and enabling interrupts in the Linux kernel. Thus, we generate two histograms that represent time-delta and instruction count delta of intervals between interrupt enabling and disabling instructions. We observed that interrupt disabling and enabling in Linux on a x86 machine is not just dependent on two instructions, `sti` and `cli` respectively, but also on a pattern of instructions that use `pushf` and `popf` to push and pop the entire EFLAGS register, thus clearing or setting the interrupt flag in the process. Thus, to effectively profile the interrupt cycle, we identified the instruction patterns during decoding and grouped and

Algorithm 1 Hardware Delta Profiling

Input : $\{\mathbf{D} : t_p ... t_q\}$, where **D** is the coded stream for time T_{q-p} , Target Instruction Set **I** (either individual instruction pair I_t or super-instruction pair S_t) and ***Mode*** ($T\Delta$ or $IC\Delta$)

Output : Time-Δ histogram, Instruction Count-Δ histogram, Instruction Count histogram of **I**

```
1:  procedure DelaHistogram
2:      iC ← 0
3:      resetFlag(F)
4:      for i ∈ D do
5:          x = decodeInstruction(i)
6:          incrementCount()
7:          if x = (I_t or S_t) then
8:              if Mode = TΔ then
9:                  if isSet(F) then
10:                     unset(F)
11:                     ts_{n+m} ⟵ t_x
12:                     Δt = ts_{n+m} − ts_n
13:                     addToDatabase(DB, Δt)
14:                 else
15:                     ts_n ⟵ t_x
16:                     set(F)
17:                 end if
18:             end if
19:             if Mode = ICΔ then
20:                 if isSet(F) then
21:                     unset(F)
22:                     ICΔ = getDelta()
23:                     resetCounter(iC)
24:                     addToDatabase(DB, ICΔ)
25:                 else
26:                     startCounter(iC)
27:                     set(F)
28:                 end if
29:             end if
30:             count(iC)
31:         end if
32:     end for
33:     generateHistogram(DB)
34: end procedure
```

identified them as `superSTI` and `superCLI` instructions. For incoming coded hardware-trace streams, we devised an algorithm shown in listing Algorithm 1 that is able to generate these

profiles. For example, when we apply this during the trace decode time, we can obtain the time taken between two consecutive interrupts enable and disable in the execution, or the number of instructions executed between them. These target super-instructions pairs (S_t), which are actually a pseudo-marker in the instruction stream based on pattern matching, can be given as input to the profiler. Based on the mode, it can either begin instruction counting or timestamp generation and stores it in a database. We then iterate over the database and generate the required visualizations.

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

We present an implementation of the algorithm in section 5.5.3 where we elaborate more on our delta profiling experiment. Decoded traces from Intel PT were chosen to demonstrate utility of this approach.

5.4.4 Syscall Latency Profiling

Syscalls affect the time accuracy of systems, especially in critical sections, as they form a major chunk of code executed from userspace. For example, filesystem syscalls such as `read()`, `open()`, `close()` etc. constitute 28.75% of code in critical sections of Firefox. [97]. Profiling syscall counts for a given execution is easy and can be performed with simple profilers such as Perf, or even through static or dynamic code analysis techniques based on `ptrace()`. However, to understand, the extra time incurred in the syscalls, we can get help from software tracers. As the software tracers themselves affect the execution of syscalls, an accurate understanding can only be achieved by an *external observer* which does not affect the execution flow. In such scenarios, hardware tracing is a perfect candidate for such an observer. We used hardware traces and devised a way to visualize syscall stacks in our proposed technique, after decoding, to compare them between multiple executions. This gave us a deep and accurate understanding of any extra time incurred in syscalls, right down to individual instructions. We devised a way such that, post-decoding, the trace data was converted to our visualization path format that prepares a raw callstack. Refer Figure 5.4. The function markers in the decoded data are converted to individual execution paths in the hierarchy. Each node in the path represents a function. To each path, we append the instructions that executed for traversing the path up to its tail. Each line in the new data contains a single complete path in the callstack. As an example, Figure 5.5(a) illustrates

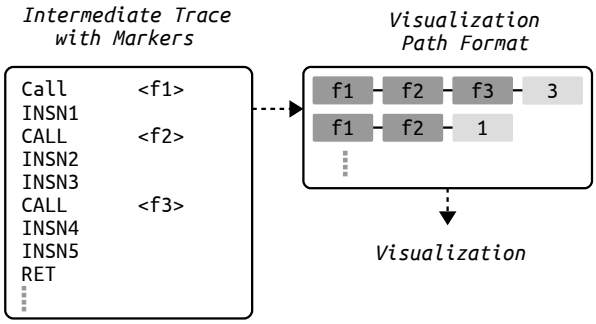

Figure 5.4 A sample trace sequence converted to a section of visualization path format

the effect of an external tracer (LTTng) on the `mmap()` syscall with the help of a callstack. The callstack shown is for a short section of code from the library mmap call to the start of the syscall in the kernel. We see in the figure that the highlighted call-path reached the `lttng_event_write()` function from the second ring of the `entry_SYSCALL_64()` function in the kernel. The layers represent calls in a callstack, with the call depth going from the innermost to the outermost layers. Here, the path to the `lttng_event_write()` function took 9.3% of all instructions in the recorded callstack. As the code sections are short, and the data represented is hierarchical in nature, it is easy to visualize them on sunburst call-graphs [98, 99] for a clear visual comparison.

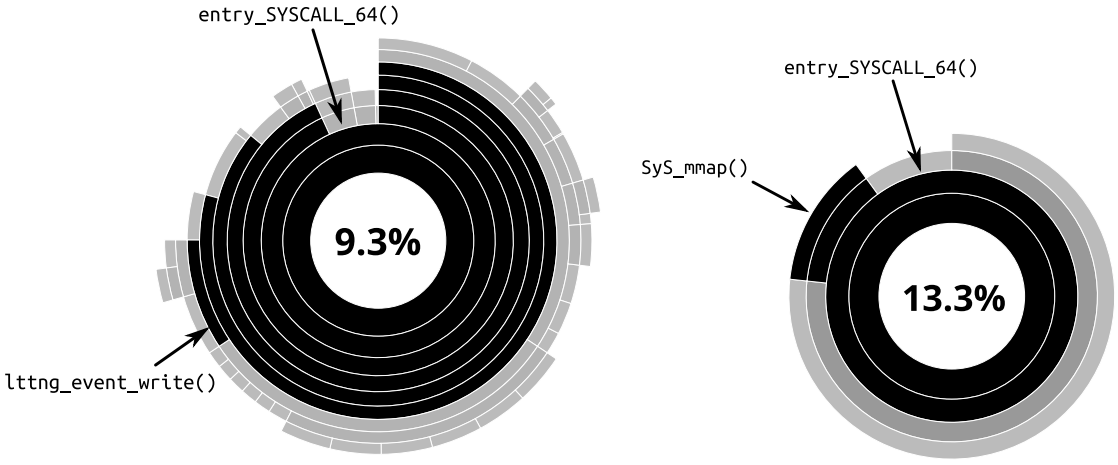

Figure 5.5 The effect of an external software tracer on the `mmap()` syscall, obtained from a near zero overhead hardware trace, is visible in (a) as extra layers as compared to (b), which took a shorter path and a shallower callstack. The rings represent the callstack and are drawn based on instruction count per call

We can observe such a short callstack from another execution, in Figure 5.5(b), where LTTng tracing is disabled, and we notice the absence of extra calls which were added as layers and peaks in Figure 5.5(a). The metrics in the sunburst graphs are calculated based on the number of instructions executed along a particular call path based on the visualization path format. The visualization is interactive and its implementation is based on the D3 javascript library.

5.4.5 Software Tracer Impact

With our hardware assisted profile, we can observe how much extra time and instructions any external software tracer added to the normal execution of the syscall. This can also be used as a basis for analyzing the overhead of known tracers on the test system itself. For example, we observed that the extra time taken in the syscall is due to the different paths the syscall has when tracing is enabled, as compared to when tracing is disabled. A lot of code in the kernel is untraceable such as C macros and blacklisted functions built with the `__attribute__((no_instrument_function))` attribute that don't allow tracers to trace them. This is usually a mandatory precaution taken in the kernel to avoid tracers going in sections of code that would cause deadlocks. However, our PT based approach allowed us to get much finer details than other pure software tracers, as it acts as an *external observer* and can even record calls to those functions. We monitored how LTTng changes the flow of 7 consecutive syscalls in a short section of traced code. As an example, for `mmap()` calls, we observed that with software tracing and recording enabled, a total of 917 additional instructions were added to the normal flow of the syscall, which took an extra 173 ns. For short tracing sections, an overhead of 579 ns was observed on average for `open()` syscalls, with 1366 extra instructions. We observed that the overhead also varies according to the trace payload for syscalls, as LTTng specific functions in the kernel modules copy and commit the data to trace events. Therefore, with the hardware-trace assisted tool, we can get instruction and time accurate overhead of the tracer's impact on the system itself. Such detailed information about a software tracer's impact is not possible to obtain by conventional software tracers themselves. Hardware-assisted profiles allows to study the flow through those unreachable sections of code (assembly, non-traceable functions in the kernel) along with a higher granularity. We tested this with our PT based approach. In section 5.5.2 we further compare the overhead of Linux kernel's Ftrace tracer with Intel PT to see the impact of hardware traces as compared to current function tracing facility.

5.5 Experimentation and Results

5.5.1 Test Setup

The test machine has an Intel Skylake i5-6600K processor which supports Intel Processor Trace and runs a patched Linux Kernel version 4.4-rc4 on a Fedora 23 operating system. To get minimum jitter in our tests, we disabled CPU auto-scaling and fixed the operating frequency to 3.9GHz. This was done to just ensure that synthetic benchmarks were accurate and reproducible. This would not affect the result as the ratio of core crystal clock value to TSC frequency is considered during decoding time to make sure the effect of auto-scaling is accounted for in the time calculations. The system has 16 GB main memory and a 500 GB solid state drive.

5.5.2 PT Performance Analysis

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

Execution Overhead

Similar to such synthetic tests done for measuring the Julia Language performance [100] against C and Javascript, we added more indirect branch intensive tests, such as TailFact which causes multiple tail-calls for factorial computation and Fibonacci to illustrate conditional branches. We also tested an un-optimized and optimized Canny edge detector to check the effect of jump optimizations in image processing tasks. As conditional branches constitute most of the branch instructions, to get a precise measurement for a conditional branch, we tested a million runs of an empty loop (Epsilon) against a loop containing an un-optimized conditional branch (Omega). Therefore, the TailFact test gives us the upper limit of overhead for indirect branch instructions while the Omega test gives us the upper limit for conditional branches.

Table 5.1 Execution overhead and trace bandwidth of Intel PT under various workloads. The TailFact and Omega tests define the two upper limits

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

Observations Our test results have been summarized in table 5.1. We can see that excessive TIP packets generated due to tail-calls from the recursive factorial algorithm cause the maximum overhead of 22.9%, while the optimized random matrix multiplication (RandMatMul) overhead is 0.83%. The optimization in the C version of RandMatMul is evident as it aggressively used vector instructions (Intel AVX and SSE) from the BLAS library [101] during a DGEMM, thus generating very few TIP or TNT packets, as compared to the unoptimized loop based multiplication in Javascript, which through the V8's JIT got translated to conditional branches. This explains the difference, as seen in the table, where the overhead for V8 is 11.08%. Same is the case with RandMatStat, which also generated more TIP packets thus pushing the overhead to 20%. In order to observe the TNT packet overhead, the Omega test generated pure TNT packets. As this includes one million extra conditional jump overhead from the test loop for both Epsilon and Omega, we can normalize the overhead and observe it to be 8.68%.

Trace Bandwidth

The direct correlation between the trace size, packet frequency and hence the trace bandwidth is quite evident. To quantify it, we record the trace data generated per time unit and quantify the trace bandwidth for our micro-benchmarks in table 5.1. To calculate the bandwidth, we record the size of raw trace data generated and the time taken for execution of the individual benchmarks.

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

Trace Size

Apart from the inherent character of the applications (more or less branches) that affect the trace size and timing overhead, Intel PT provides various other mechanisms to fine tune the trace size. The basic premise is that the generation and frequency of other packets such as timing and cycle count information can be configured before tracing begins. To test the effect of varying such helper packets, we ran the PiSum micro-benchmark from our overhead experiments, which mimics a more common workload with userspace-only hardware trace mode. We first started with varying the generation of synchronization packets called PSB, while the cycle accurate mode (CYC packets) and the mini-timestamp count (MTC) packets were disabled. The PSB frequency can be controlled by varying how many bytes are to be generated between subsequent packets. Thus, for a higher number of bytes, those packets will be less frequent. As PSB packets are accompanied with a (Time Stamp Counter) TSC packet, this also means that the granularity of timing varies. The same was repeated with MTC packets while the PSB packet generation was kept constant and the CYC mode was disabled. Similar tests were done with CYC packets, where PSB packet generation was kept constant and MTC packet generation was disabled. Figures 5.6, 5.7 and 5.8 show the effect of varying the frequency of packets on the generated trace data size.

Observations We observe in Figures 5.6, 5.7 and 5.8 that, as expected, when the time period (indicated by number of cycles or number of bytes in-between) for CYC, MTC or PSB packets is increased, the trace size decreases. However, it moves towards saturation, as opposed to a linear decrease. The reason we observed is that, for a given trace duration, there is a fixed number of packets that are always generated from the branches in the test application. This sets a minimum threshold. Furthermore, in Figure 5.6, the trace size did not increase further for time periods $< 2^6$ cycles, because the maximum number of CYC packets that can be generated for our synthetic workload was reached at 2^6 cycles. The trace data size can however further increase for other workloads with higher frequency of CYC packets, when kernel tracing is enabled. In our tests, we found that the lower and upper bounds of trace data size, based on lowest and highest possible frequencies of all packets combined, are respectively 819 KB and 3829 KB.

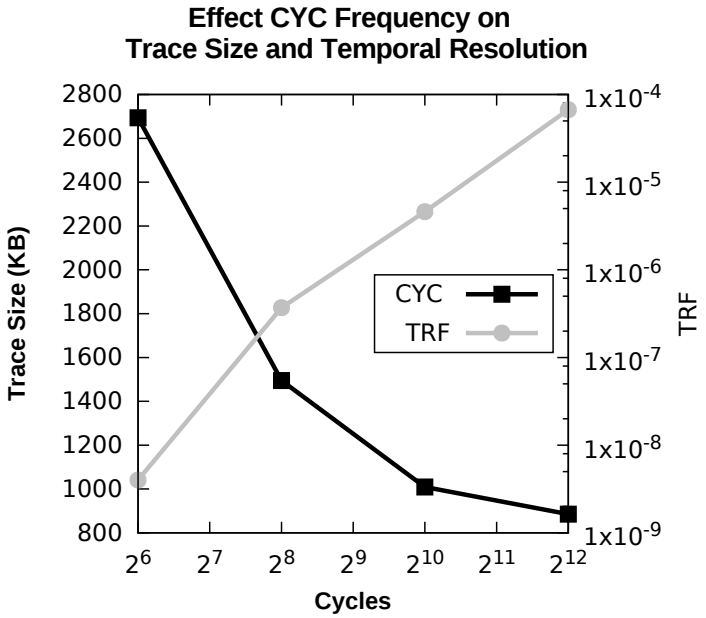

Figure 5.6 Trace size and resolution while varying *valid* CPU cycles between two subsequent CYC packets. Lower TRF value is better

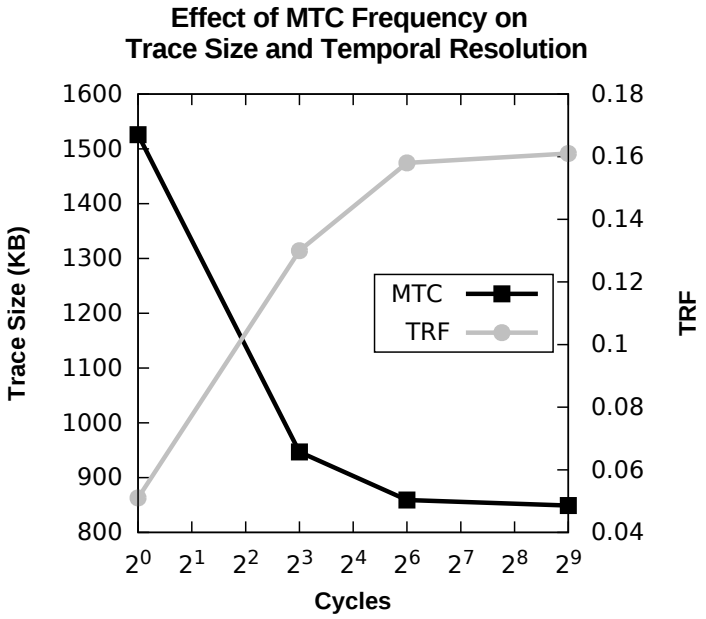

Figure 5.7 Trace size and resolution while varying *valid* CPU cycles between two subsequent MTC packets. Lower TRF value is better

Temporal Resolution

In addition to the effect of different packet frequencies on the trace data size, it is also important to observe how much timing granularity we lose or gain. This can help the user decide

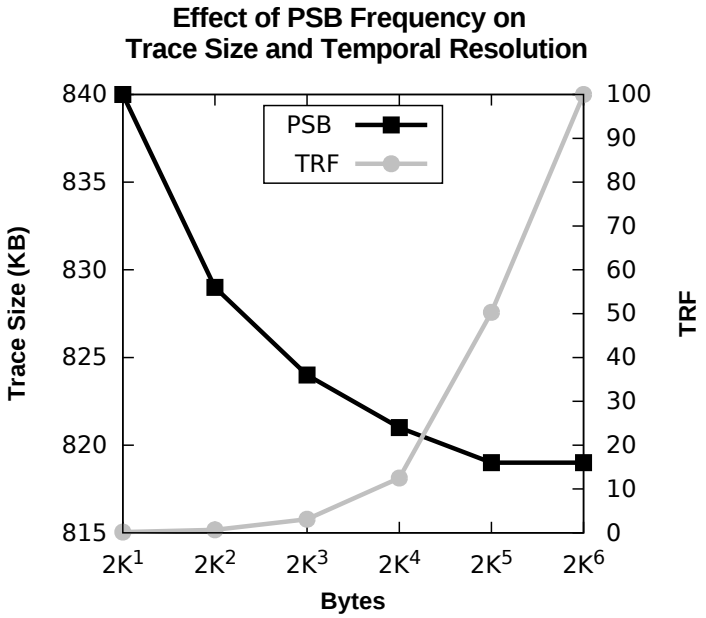

Figure 5.8 Trace size and resolution while varying *valid* bytes of data between two subsequent PSB packets

what would be the trade-off between size, timing granularity of the trace and the tracing overhead, to better judge the tracer's impact on the target software. We therefore define a Temporal Resolution Factor (TRF). For a given known section of code, with equidistant branches,

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

where,

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

and P is the set of all p that represents a median factor in the observed sets. Here, ΔT and ΔI_c are the time and instruction deltas between subsequent decoded branches and n is the length of the total decoded branches. The formula calculates the median of the sorted datasets of ΔT and ΔI_c and normalizes them with a factor of N_f for an accurate representation. The value of n, however, varies according to the frequency of packets. Hence, with the packets we estimate the averages based on the maximum we can obtain. This experiment was coupled with the trace size experiment above so that, for the same executions, we could observe our temporal resolution as a function of the trace size as well. The choice of equidistant branches is intentional for a more controlled and repeatable micro-benchmark. The results are presented in Figures 5.6, 5.7 and 5.8 with TRF on the second Y axis. TRF varies between

0-100. Lower TRF values represent better resolutions.

Observations It is interesting to see a clear trend that the data size is inversely proportional to TRF. Hence, the larger the trace size, the better is the temporal resolution in the trace. Another important observation is the sudden increase in resolution when CYC packets are introduced. We observed that the highest resolution we obtained for our tests was 14 ns and 78 instructions between two consecutive events in the trace ($TRF = 4.0 \times 10^{-9}$). This compares to the lowest resolution of 910.5 μs and 2.5 million instructions ($TRF = 100$) between two events with no CYC and far apart PSB packets. The reason for such a huge variation is that the introduction of the cycle accurate mode generates CYC packets before all CYC eligible packets (such as TNT, TIP). The CYC packets contain the number of core clock cycles since the last CYC packet received, which is further used to improve the time resolution. With a large number of CYC-eligible packets being generated in quick succession, the trace size as well as the resolution increases drastically, as compared to MTC and PSB packets. For CYC observations in Figure 5.6, the resolution for $< 2^6$ cycles is not shown, as it covers the whole execution in our workload for which we are interested. Thus, we always get a constant maximal number of CYC packets and the TRF value saturates. Therefore, we only included valid cycles $> 2^6$. This can however vary for real life usecases such as kernel tracing where the branches are not equally spaced and the code section is not linear. However, the TRF values obtained can be a sufficient indicator of upper and lower bounds of resolution.

Ftrace and PT Analysis

We also compared the hardware control flow tracing with the closest current software solutions in the Linux kernel. An obvious contender in kernel control-flow tracing for our Intel PT based framework is Ftrace [13]. Both can be used to get the execution flow of the kernel for a given workload - with our approach providing a much more detailed view than Ftrace. We therefore used the Sysbench synthetic benchmarks to gauge the overhead of both approaches for disk I/O and memory and CPU intensive tests. We configured Ftrace in a mode as close as possible to Intel PT by outputting raw binary information to a trace buffer. We also set the per-CPU trace buffer size to 4MB. Our findings are presented in table 5.2. We can see that, as compared to PT, FTrace was 63% slower for a Sysbench File I/O workload with random reads and writes. This generates numerous kernel events for which Ftrace had to obtain time-stamps at each function entry. In the PT case, the time-stamps are an inherent part of the trace data generated in parallel through trace hardware. This explains the huge difference in overhead. A similar difference is observed in the memory benchmark as well. In

the case of the CPU benchmark, the work was userspace bound and hence the trace generated was smaller - thus a non-statistically significant overhead in PT and 0.6% overhead in Ftrace.

Table 5.2 Comparison of Intel PT and Ftrace overheads for synthetic loads

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

5.5.3 Delta Profiling Instructions

In these sets of experiments we show how our Algorithm 1 is able to generate histograms for instruction and time delta for `superSTI` and `superCLI` instructions groups. In order to quantify how much time is spent in a short section where interrupts are disabled, we created a synthetic test module in the kernel that disables and enables interrupts as our input. We control the module using `ioctl()` to cycle the interrupts. We pin the userspace counterpart of our test module on CPU0 and analyze the hardware trace for CPU0. Our algorithm is implemented during the trace decoding phase to generate the instruction delta and time delta in an intermediate format which we then plot as histograms. This helps pinpoint how many instructions were executed in the interval between two consecutive interrupt disable and enable. Looking at Figure 5.9, we can see that most of the interrupts disabled intervals executed around 90-100 instructions. For the same execution, we observe in Figure 5.10 that most of the interrupts disabled intervals have a duration in the range 40-80 ns. We can then look for the interrupts disabled intervals of abnormally high duration which are at the far right in the histogram. Delta profiling of actual instruction flows therefore allows for an overview of the interrupt cycling in the kernel for a particular short running task. As discussed in section 5.6, to get more in depth analysis, we can take snapshots when abnormal latencies are encountered, to get a more in depth view upon identifying them from the histogram profiles.

5.6 Conclusion and Future Work

New techniques used in hardware tracing are now empowering developers in the domain of software performance debugging and analysis. We observe that hardware assisted tracing and profiling provides a very fine granularity and accuracy in terms of control flow and time and

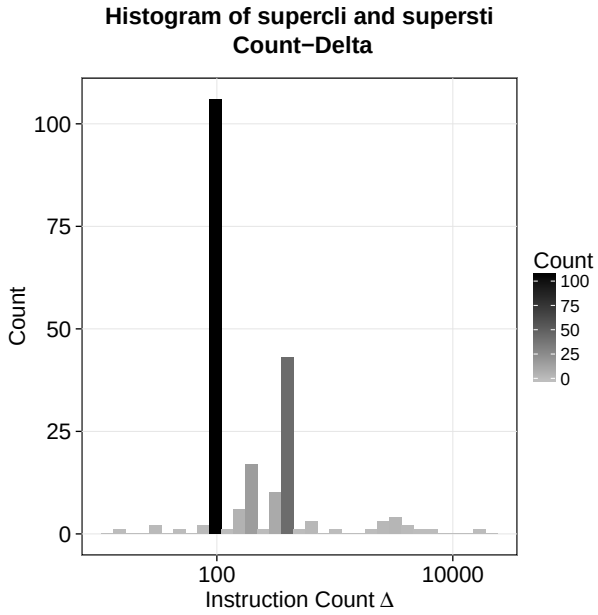

Figure 5.9 Histogram of instruction count delta for `superSTI` and `superCLI` instructions generated using Delta Profiling algorithm

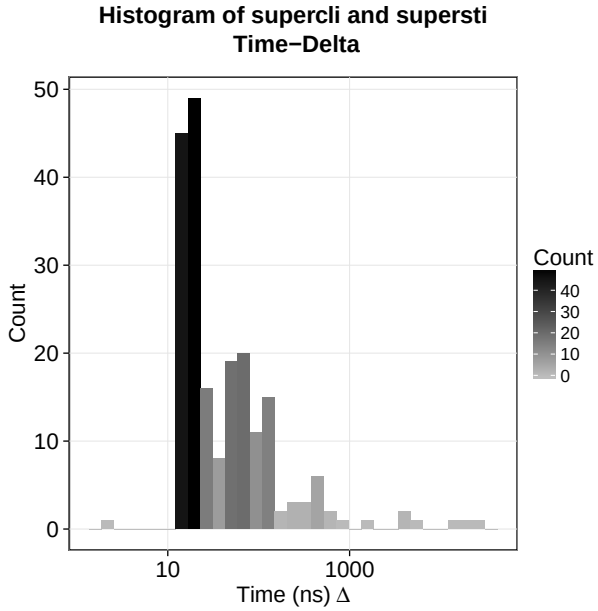

Figure 5.10 Histogram of time delta for `superSTI` and `SuperCLI` instructions generated using Delta Profiling algorithm

present a trace framework that utilizes hardware-assistance from Intel Processor Trace (PT). Our PT based approach, with a minimal overhead in the range of 2-5%, was able to provide a highly detailed view of control flow. We also present a detailed analysis of trace size and

temporal resolution provided by PT, while fine tuning its configuration. With the help of our framework, we were able to generate detailed targeted callstacks for syscalls and observe differences between multiple executions. We also demonstrated a way to trace the software tracers themselves and show how similar kernel control-flow tracers such as Ftrace cause overheads as high as 63%, while PT was able to generate similar yet more detailed results with 1% overhead. Hardware tracing also allowed us to gather traces from parts of the kernel that are invisible to traditional software tracers. PT assisted cycle-accurate profiling was able to provide resolution as high as 14 ns, with timed events only 78 instructions apart. However, our analysis of Intel PT and the information released by Intel suggests that many additional features such as using PT in virtual machine tracing are left yet to be explored.

Latency Snapshots

We hinted how, in addition to syscall latency profiles, more in-depth analysis on other non-deterministic latencies can be done. An interesting observation relevant for realtime systems is also an in-depth analysis of IRQ latency. Newer tracepoints in the kernel introduced by tracers such as LTTng [102] allow recording software trace events when IRQ a latency beyond a certain threshold is reached. We can further refine our idea by recording hardware trace snapshots in such conditions, thus obtaining more detailed control-flow and timing information.

Virtual Machine Trace and Analysis

Our Intel PT based hardware assisted technique can also be used to detect VM state transitions, in host or guest only hardware tracing, for more in depth analysis of VMs without any software tracing support. We observed that PT can also generate VMCS Packets and set the NonRoot (NR) bit in PIP packets when hardware tracing is enabled in VM context [61]. The extra information in hardware traces allows the decoder to identify VM entry and exit events and load specific binaries for rebuilding control-flow across VMs and host. Thus, with no software intrusion and a low overhead, we can get accurate VM traces, compare and quantify their performance.

5.7 Acknowledgement

We are grateful to Ericsson, EfficiOS, NSERC and Prompt for funding and Francis Giraldeau for his technical advice and comments.

CHAPTER 6 ARTICLE 3 : LOW OVERHEAD HARDWARE-ASSISTED VIRTUAL MACHINE ANALYSIS AND PROFILING

Authors

Suchakrapani Datt Sharma, Hani Nemati, Geneviève Bastien and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {suchakrapani.sharma, hani.nemati, geneveive.bastien, michel.dagenais}@polymtl.ca

Published in Proceedings of Globecom Workshops (IEEE) in December 2016

Reference as S. D. Sharma, H. Nemati, G. Bastien and M. Dagenais, 'Low Overhead Hardware-Assisted Virtual Machine Analysis and Profiling", *Proc. Globecom Workshops, IEEE*, December 2016

Abstract

Cloud infrastructure providers need reliable performance analysis tools for their nodes. Moreover, the analysis of Virtual Machines (VMs) is a major requirement in quantifying cloud performance. However, root cause analysis, in case of unexpected crashes or anomalous behavior in VMs, remains a major challenge. Modern tracing tools such as LTTng allow fine grained analysis - albeit at a minimal execution overhead, and being OS dependent. In this paper, we propose *HAVAna*, a hardware-assisted VM analysis algorithm that gathers and analyzes pure hardware trace data, without any dependence on the underlying OS or performance analysis infrastructure. Our approach is totally non-intrusive and does not require any performance statistics, trace or log gathering from the VM. We used the recently introduced Intel PT ISA extensions on modern Intel Skylake processors to demonstrate its efficiency and observed that, in our experimental scenarios, it leads to a tiny overhead of up to 1%, as compared to 3.6-28.7% for similar VM trace analysis done with software-only schemes such as LTTng. Our proposed VM trace analysis algorithm has also been open-sourced for further enhancements and to the benefit of other developers. Furthermore, we developed interactive Resource and Process Control Flow visualization tools to analyze the hardware trace data and present a real-life usecase in the paper that allowed us to see unexpected resource consumption by VMs.

6.1 Introduction

The backbone of modern distributed cloud systems are virtualization technologies that enable VMs to provide the necessary infrastructure. Public and private cloud infrastructure providers allow the users to access a pool of resources based on a Pay-as-Use (PaU) model where numerous automated cloud orchestration tools allow seamless control of bring-up and tear-down of VMs. This flexibility in resource scaling leads to imbalanced workload distribution on the underlying hardware on which the VMs run. Users can also intermittently run demanding applications which may need their VMs to be migrated to difference resource groups. Cloud infrastructure administrators therefore need modern tools for performance analysis of such VMs. However, efficient debugging, troubleshooting and analysis of such massive distributed systems is a still a known challenge [103]. For fine-grained post-mortem root cause analysis of problems occurring on VMs, the administrators may need highly detailed information about the characteristics of VMs on their infrastructure - such as profile of processes running on them, virtual CPU (vCPU) consumption, pattern of scheduling of processes on VMs and their interactions with underlying hypervisors. Most of such information can be gathered by proper configuration tools provided by the host OS kernel. Software-only diagnosis of problems on VMs, however, calls for recording all software events such as occurrences of `vm_entry`, `vm_exit`, `sched_switch`. The added overhead from such events can be mitigated by using tracing tools such as LTTng[104]. However, these tools also alter the execution flow of the VMs and require careful configuration (such as adding additional static tracepoints in QEMU and KVM). In addition, proprietary close-sourced operating systems on specialized hardware may not expose tracing tools or APIs and would be opaque to the administrator. In such scenarios, pure hardware tracing can help in diagnosing abnormal executions.

In this paper, we introduce a novel approach that uses hardware trace support provided in modern processors for VM analysis. Special trace data emitted by the trace hardware on the processor can be collected and analyzed offline to gather in-depth information about execution profiles of VMs and hypervisors on the host. Our approach allows for a near-zero tracing overhead and a new technique to visualize such data.

We demonstrate the uniqueness of our approach through the hardware trace support provided in Intel's Skylake series of processors in the form of Intel Processor Trace (PT)[61]. These trace blocks generate huge amounts of hardware trace data, consisting of mostly branch related packets that can be used to reconstruct the program flow. The trace data also contains certain trace packets such as PIP and VMCS which we record, extract and use with our algorithm to generate synthetic trace events that identify important states of processes in VMs such as entry/exit from hypervisor, to or from the VM, and scheduling events between processes

in VMs. We generate synthetic events from such hardware trace packets that profile vCPU consumption by processes, without any software and operating system (OS) intervention, thus ensuring a low overhead and minimum interference with the VMs. Since this approach is OS independent, it works on any OS platform without any necessary configuration. Through this, we were able to identify processes inside VMs that would cause undesired vCPU load. To the best of our knowledge, there is no pre-existing efficient technique to gather such high level VM analysis from low level hardware trace packets. Our main contributions in this paper are as follows :

— A novel low overhead hardware-assisted approach to extract, group and analyze hardware trace packets gathered from the processor for VM analysis. The VMs, host hypervisor and host OS are oblivious to our tracing and analysis phase. Therefore, there is no need for internal access within VMs, which may not be allowed in most situations due to security reasons.

— A visualization strategy to display these hardware trace events on a time series graph, and identify hard to diagnose issues such as processes contending for resources in VM. Our graphical views show CPU usage inside the VM along with their interaction with the Virtual Machine Monitor (VMM). We also implemented a graphical view for the execution flow of processes inside the VM.

The rest of the paper is organized as follows : Section 6.2 presents related work, comparing the closest approaches to ours. Section 6.3 introduces important processor trace packets for VM analysis and explains the different layers of the architecture that we use in our paper. Here, we also present the algorithm used to retrieve information from processor trace packets. We show a use-case of our analysis in Section 6.4. The added overhead with our approach is compared with existing approaches in Section 6.5. Section 6.6 concludes the paper.

6.2 Related Work

Program flow tracing based instruction counting, and tracking blocks of code, has been discussed earlier [69]. In order to reduce the bandwidth of such tracing, Merten et al. [105] have earlier proposed the use of a Branch Trace Buffer (BTB) and their hardware table extension for profiling branches. Custom hardware-based path profiling has also been discussed by Vaswani et al. [45]. Linux Kernel tools such as Ftrace and Kprobes allow such code instrumentation and program flow deduction. Modern architectures snoop bus activity at very low level inside the processor and allow recording each and every instruction being executed. This, however, generates a huge amount of data. To mitigate this, the new approach is to only record instructions that cause the program flow to change, such as direct/indirect

jumps, calls, exceptions etc. By following these change-of-flow instructions, it is quite easy to generate a complete program flow with the help of additional offline binary disassembly at the decoding phase. Dedicated hardware blocks in the Intel architecture, such as Last Branch Record, Branch Trace Store [59], and more recently Intel Processor Trace (PT) follow this approach. A detailed study of hardware-assisted tracing and profiling with Intel PT has been recently presented in [60].

Hardware-trace based compiler optimization techniques have also been discussed before [106] where results from execution profiles of a software application can be fed back to the compiler to optimize the resulting binary. Jörg et al. in [107] present a data parallel provenance algorithm which uses Intel PT for improving security and dependability in software.

AWS CloudWatch and Openstack Ceilometer are the metering, monitoring and alarming tools for clouds. They provide basic metrics such as physical CPU and number of vCPU used for each VM. The information presented by such tools is not suitable for analyzing VMs. Most existing Linux tools such as `vmstat` gather statistics by reading `procfs` file with significant overhead. Therefore, they are not recommended for implementing a low overhead tool for analyzing and debugging VMs. In [108], the authors proposed a significant multi-layer tracing and analyzing technique to detect anomalies inside VMs. They implemented an execution flow recovery of specific processes by tracing the host and VMs at the same time. In their work, they need internal access to the VMs. Furthermore, their work is limited to the Linux OS since they use LTTng as Linux kernel tracer. PerfCompass[109] uses a VM's kernel trace (from LTTng) and gathers information from *syscall* events to detect anomalies inside the VM. Authors in [110] implemented a vCPU monitoring tool based on *"perf kvm record"*. With their monitoring tool, they are able to gather statistics about CPU usage for processes and the hypervisor. Wang in [111] used Perf to detect over-commitment of pCPUs. From all available CPU metrics, they used LLC which has a direct relationship with pCPUs over-commitment. Our profiling technique uses hardware trace packets that show vCPU usage along with processes execution flow, without any software and OS intervention. Our approach adds less overhead to VMs compared to other techniques and it does not rely on a specific OS or hypervisor. As per our knowledge, no prior work has been done for analyzing VMs at a high level from such low level hardware traces yet. Our work, therefore, is unique and novel in this aspect.

6.3 Hardware Tracing VMs

Hardware trace generation can be configured and controlled by certain configuration registers such as MSRs in Intel or CoreSight ETM/ETB Configuration registers on ARM. In this paper,

we select Intel PT as an experimentation platform for our hardware-asssisted VM analysis approach. Once hardware trace generation is enabled, the tracing blocks from processor cores generate compressed encoded trace packets for eventual decoding. These hardware trace packets can contain information such as paging (changed CR3 value), time stamps, core-to-bus clock ratio, taken-not-taken (tracking conditional branch directions), record the target IP of branches, exceptions and interrupts, and record the source IP for asynchronous events (exceptions, interrupts). Keeping track of all these packets can be quite expensive - especially, if the required analysis is at a high level (such as VMs in our case). Therefore, we isolate only those packets for analysis that are sufficient to reconstruct the flow in the VMs. Some of the important hardware packets and their role in our analysis are as follows :

PIP

The Paging Information Packet (PIP) is generated whenever the `CR3` register value is modified. This includes scenarios such as a task switch, a `MOV CR3` instruction or, more importantly, a VM Entry and VM Exit at the time when VM execution is enabled. This packet allows the decoder to uniquely identity which process was executing on the processor. During VM execution, the packet also contains a Non-Root (NR) bit that can further indicate if the process was executing in a NR context (guest mode) or in the VMM context. Together with other packets generated for `VMXON` instructions, we can generate a detailed view of the VM.

VMCS

This packet is generated at a successful `VMPTRLD` instruction, which indicates interaction between the VMM and the guest OS. The VMCS packet payload consists of the VMCS pointer of the logical processor that will execute the VM guest context. This packet helps us in determining which vCPU was being utilized at what time.

Timing

Timing information for each event can be deduced from 3 more important packets. The first one is Time Stamp Counter (TSC) which gives the lower 7 bytes of the time-stamp counter - the same as the one returned by the `RDTSC` instruction. The Mini Time Counter (MTC) packet contains the 8-bit value derived from the Always Running Timer (ART) on Intel processors. Along with a Timing Alignment (TMA) packet, the MTC and TSC values can be used to estimate the precise timing of each event up to nanosecond precision [61].

For our analysis, we record all these packets and analyze them in post VM execution scenarios.

We also record the physical CPU (pCPU) associated with the relevant packet. We then create synthetic events with all the packets and the relevant context information attached to them.

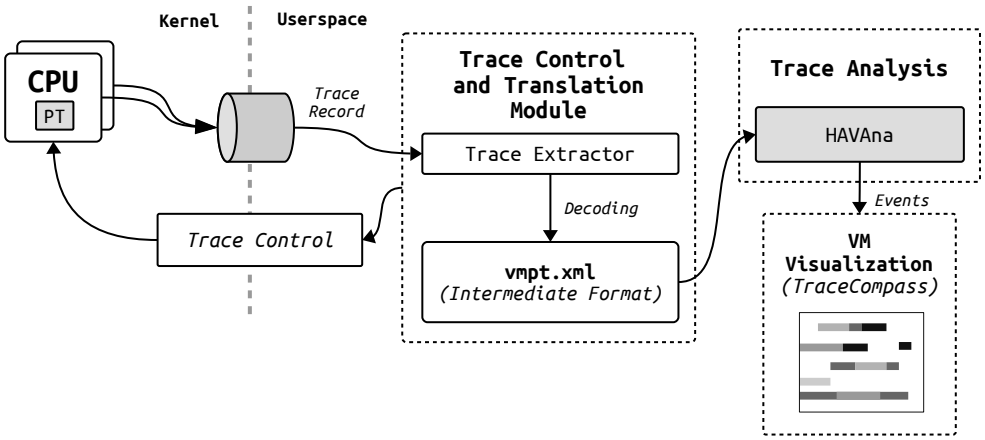

Figure 6.1 System Architecture

6.3.1 System Architecture

As seen in Figure 6.1, the trace control module configures the tracing hardware on the processor. Enabling the tracer generates a huge amount of encoded trace data that is stored on disk, with context information for each pCPU attached to it. The translation module filters and extracts the raw packets for VM analysis. The PIP, VMCS, TSC and MTC packets are decoded from the per-CPU trace stream and converted to an XML derived intermediate format (IF). The synthetic events are identified from the decoded trace and stored in this format. For example, `<event>` tags contain each event with their timestamps along with event specific data. There are two event packets - PIP and VMCS. The main driver for this module is our Hardware-Assisted VM Analysis (HAVAna) Algorithm that is based on the state machine which analyses the packet IF and generates visualizations describing the VM behavior. The XML driven visualizations are consumed by the Trace Compass [80] trace analysis tool for an in-depth interactive view of the VM execution.

6.3.2 HAVAna Algorithm

The main feature of our proposed technique is the state machine that classifies hardware packets and generates synthetic events for visualization. The input to the algorithm is the raw XML event description stored in the IF generated during trace translation. Each event packet from the stream is sent to the state machine shown in Figure 6.2. The occurrence

of a VMCS event packet in the IF, succeeding a PIP packet, marks the process for being scheduled on the vCPU and indicates the beginning of a VM execution at a high level. The process enters the **VMM Mode** (Root Mode). A PIP packet with a new CR3 value and Non-Root (NR) bit (extracted from PIP hardware trace packet) as 1 indicates that a VM process is now being executed. This is marked by the **VM Mode** (Non-Root Mode). Successive transitions of PIP packets with NR bit value indicate the execution switching between VMM and VM mode. Along with the timestamps in all the states gathered from the IF, we can start creating a time series graph that shows the process activity in VM and VMM. By associating vCPUs with VMCS base pointers, we can identify the vCPU consumption as well. The output of the state machine are the synthetic events that are then stored and input to the trace visualization tool. The pseudocode for our algorithm to uncover different states for vCPUs and processes inside VMs is shown in Algorithm 2. It receives events as input and updates the State History Tree [80] as output. For each packet, it checks the name. In case of the packet name is VMCS, it saves the VMCS based address and changes the Status of the related vCPU as VMM (Line 4). When our algorithm receives a PIP packet, it checks the NR field. If the NR field is 1, it first queries the current running vCPU base address and modifies the Status of the related vCPU as VM (Line 8) and then it queries the current running VMM and modifies its Status as *IDLE* (Line 9). It also changes the Status of the current process (identified by the CR3 value) running inside the VM as *VM* (Line 10). If the NR field is zero, and the current status of the vCPU is *VMM*, it modifies the Status of the vCPU, process and VMM as *IDLE*, (Line 13-15). In case the NR field is zero and the current status of the vCPU is *VM*, it sets all the attributes to *VMM*, (Line 18-20).

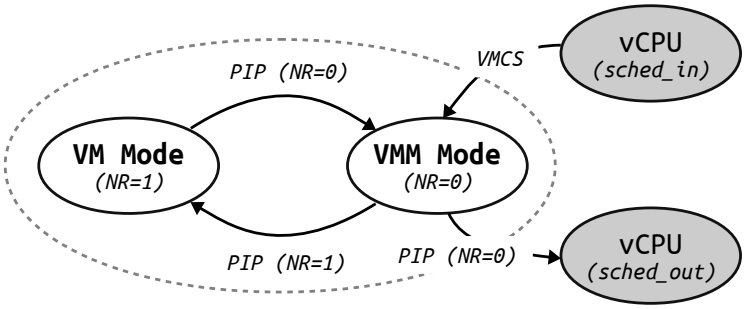

Figure 6.2 HAVAna State Machine

6.3.3 Trace Visualization

For constructing the Synthetic Events (*SE*) for visualization, we follow a XML based scheme similar to the one used by Kouame et al. [112]. In our case, however, we define rules for state

Algorithm 2 HAVAna Algorithm

```
1: procedure HAVAna(Input : Event Packets (P_e[i]) from IF Output : Updated SHT)
2:     SE[i] = parseXML(P_e[i])
3:     if (SE[i].name == VMCS) then
4:         Modify Status attribute of SE[i].base as VMM
5:     else if (SE[i].name == PIP) then
6:         if (SE[i].NR == 1) then
7:             Query Status attribute of current running base
8:             Modify Status attribute as VM
9:             Modify VMM Status as IDLE
10:            Modify Status attribute of SE[i].cr3 as VM
11:        else if (Query Status attribute of SE[i].base == VMM) then
12:            Query Status attribute of current running base
13:            Modify base Status as IDLE
14:            Modify VM Status as IDLE
15:            Modify VMM Status as IDLE
16:        else
17:            Query Status attribute of current running base
18:            Modify base Status as VMM
19:            Modify Status attribute of SE[i].cr3 as VMM
20:            Modify VM Status as IDLE
21:        end if
22:    end if
23: end procedure
```

transitions in XML, as described in Algorithm 2, and input them to the TraceCompass[80] tool, an open source tool for analyzing traces and logs. It provides an extensive and well documented interface to build analysis views and graphs. We have also open sourced [1] our hardware-assisted VM analysis scheme and algorithm. We created two analysis views based on the synthetic events. The first one is the *VM Resource View* that shows the vCPU resource usage by VMMs as well as the processes running on the VM. This can be useful for analyzing transitions between the VMM and VM modes and identify abnormal latencies in either VM, process or VMM mode. The second view is the *VM Control Flow View*, which shows each process on the VM and their flow of execution. We describe these views with a usecase in the following section.

1. `http://step.polymtl.ca/~suchakra/havana.tar.gz`

6.4 Usecases - Resource Contention

To show the efficiency of our approach, we first show our VM Resource View with an example of a 4 threaded application which calculates prime numbers. We configured our test VM with 4 vCPUs pinned to one pCPU, which can represent an ideal low-tier VPS. We ran our test application while recording a hardware trace from Intel PT in a trace buffer. We extracted the trace data, decoded and converted it to the XML IF and applied our HAVAna algorithm. The resulting VM Resource view, as seen in Figure 6.3, shows an execution window of about 3 seconds with the 4 threads executing on the 4 vCPUs while contending for CPU resources. The red bars show the process execution in the VM while the green bars shows the VMM mode execution. As the visualization is interactive, we can zoom the slightly anomalous looking green bar and observe how much extra time was spent in the VMM mode as compared to the VM to VMM switches adjacent to that execution, as shown in the same figure. Usually such behavior is indicative of VM page faults and VM PAUSE states. However further analysis of each extra time requires detailed software trace from the host kernel.

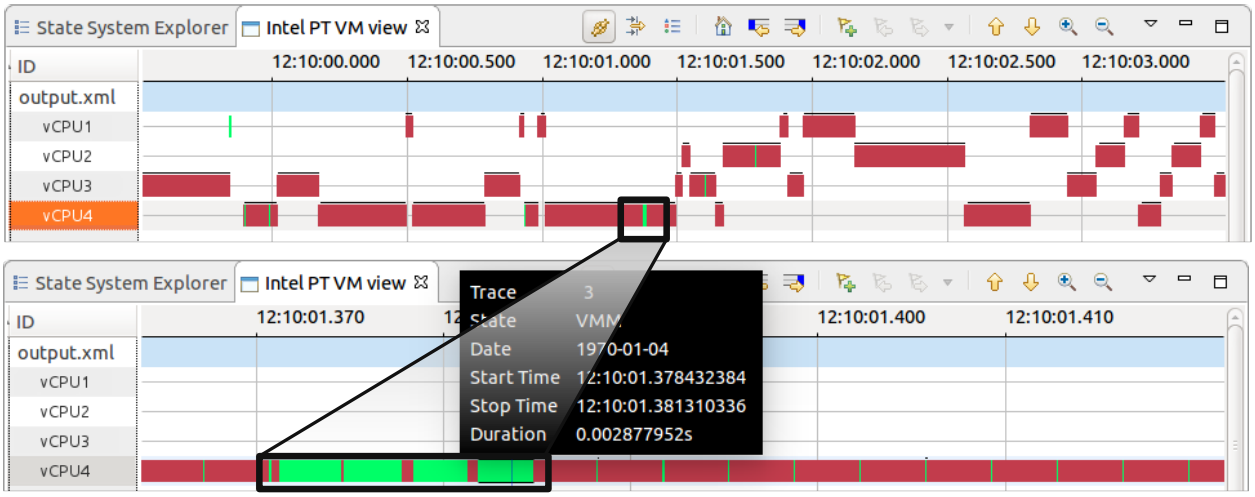

Figure 6.3 Resource View showing 4 vCPUs and their execution distribution on single pCPU along with a zoomed view of the VMM mode

For our VM Control Flow View, we demonstrate a RabbitMQ based message queuing system that performs MD5 hashing. With the same VM configuration as above, we setup 3 worker threads that do the hashing in a round robin fashion and sent 3 jobs simultaneously to them. Each worker process would execute for some duration and the scheduler on the VM then passes the execution to the other worker processes. We can observe such a pattern for the 3 workers in Figure 6.4. Each process intermittently does the job and then relinquishes control

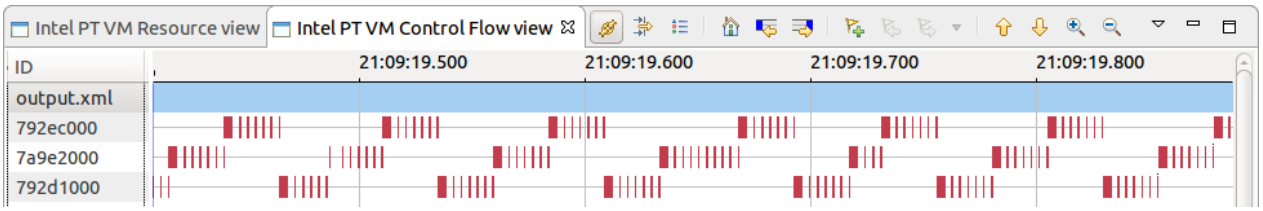

Figure 6.4 Control Flow View showing 3 RabbitMQ worker processes contending for existing pCPU

to the next process in queue and so on. This view can be used to show how the control flow was passed between processes, their relationships with their parents, children and abnormal executions if any. All of these views have been populated with hardware trace data gathered from PT, without any software trace intervention, thus making our approach agnostic of any OS platform or software infrastructure dependency.

6.5 Overhead Analysis Experiments

One of the major benefits of our work is that we avoid interacting with software altogether during the trace recording phase - unlike the current software based tracers such as Ftrace, LTTng or SystemTap, that cause some overhead in the target trace execution while trace recording. To quantify the reduction in trace timing overhead with our approach, we used the `sysbench` benchmark to measure the overhead caused when LTTng kernel tracing was enabled. We compared it to the hardware trace overhead incurred while Intel PT was being used. The test machine was an Intel i5-6600K processor clocked at 3.5GHz with 16GB of main memory. We ran our tests and benchmarks on a vanilla Linux kernel v4.5. We used KVM as kernel hypervisor and QEMU as its userspace counterpart. Our results have been summarized in Table 6.1.

Table 6.1 PT based VM trace and LTTng trace overhead

Benchmark	*Execution Overhead*(%)	
	PT	LTTng
File I/O	1.00	28.72
Memory	0.00	3.67
CPU	0.00	3.72

We observed that, for the File I/O benchmark, the hardware tracing overhead for the analysis

was only 1%, as compared to a similar VM analysis trace overhead with LTTng at 28.7% where tracing was activated on the host kernel[113]. This large overhead was mostly due to the trace storage competing with the benchmark for disk bandwidth. In other cases, the PT overhead was not statistically significant, and hence was neglected, while the LTTng overhead stood at around 3.6%. Even though LTTng's trace analysis could give deeper insights about the host and VM than our pure hardware trace approach, the completely non-intrusive, platform OS independent, approach of hardware tracing can yield similar end results at a much lower execution cost on host and VM.

6.6 Conclusion

The use of tracing allows cloud infrastructure providers to diagnose issues that may be hard to reproduce otherwise. As virtualization is the base layer for building up cloud services, it is important to tackle issues in VMs. We observed that most of such analyses would require gathering data from the VM, the hypervisor and the host kernel which needs agents running inside client VMs. To overcome this limitation, we propose a new hardware trace analysis based *HAVAna* algorithm that allows detailed diagnosis of CPU resource consumption by processes on VMs, their states and flow of execution, in a completely non-intrusive manner, without the involvement of any OS or VM interface. We demonstrate that our technique allows detection of resource contention in the VMs without querying the guest at all, thereby allowing infrastructure providers to meet their SLAs effectively without any support from the clients' VMs. This can also be beneficial to further analyze malicious executions in the target VMs or move them to different resource groups based on observed workloads.

Even though our approach and algorithm are independent from any VM interaction, the amount of information, such as identifying the faulty process by name or PID, or gathering instruction profile data for individual processes is reduced. Such problems can be tackled eventually by fetching minimal statistics from VMs, such as process maps from the guest kernel. Another obvious addition can be to identify executions in VMs intended to have small lifespans (such as those which mimic containers in their behavior) and compare their successive startup and teardown profiles by comparing instruction executions. This would help in clustering them and moving them to different resource groups as required.

CHAPTER 7 ARTICLE 4 : HARDWARE TRACE RECONSTRUCTION OF RUNTIME COMPILED CODE

Authors

Suchakrapani Datt Sharma and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {suchakrapani.sharma, michel.dagenais}@polymtl.ca

Submitted in ACM Transactions on Computer Systems in November 2016

Reference as S. D. Sharma and M. Dagenais, 'Hardware Trace reconstruction of Runtime Compiled Code", *Under-Review*

Abstract

Hardware tracing has emerged as a low-cost technique to analyze systems at a very fine granularity, thus mitigating the need for software-only trace approaches for performance analysis. State-of-the-art trace hardware on modern Intel and ARM processors allows recording change-of-flow instructions in executable binaries, such as branches, for offline reconstruction. This conventional userspace based trace reconstruction, however, is not robust enough in the common scenarios where runtime code is being generated, compiled and executed. We therefore propose a novel kernel-assisted mechanism called FlowJIT to reconstruct hardware traces with a low overhead of around 1.3μs per code page modification event. We further show the efficacy or our technique with the help of two illustrative usecases that cover the JIT compiled code scenario as well as a same-page instruction modification scenario. Our implementation has been open sourced as a patch for the Linux kernel.

7.1 Introduction

For developing quality software, developers rely heavily on performance analysis and debugging tools. However, interrupt driven debugging can cause unintended latency in execution, especially in situations where time-correctness is as important as the data-correctness of a software execution. Detecting such *heisenbugs* [90] in complex systems is getting challenging

on contemporary infrastructure. In addition to traditional debugging approaches, which may distort execution flow, advanced tracing tools such as LTTng [11] reduce the overhead of analysis. A similar in-kernel static and dynamic tracing infrastructure [3] also aims to gather detailed traces. These tracing tools depend on software tracing *hooks* exposed in the Linux kernel, to which they can attach in order to gather trace data and write it to special trace buffers. Similar to the software approach, almost all modern processors provide hardware tracing blocks on the chip that can also be used to gather program or data flow, along with timing information, at instruction level granularity. As the trace packets are encoded and generated directly by the processor, the overhead of hardware tracing on target software is minimal and the trace information retrieved is precise. The state-of-the-art hardware tracing approach aims to record only change-of-flow instructions such as conditional or indirect jumps, calls, asynchronous interrupts and returns. The data is transmitted in highly compressed and encoded [89] form by the processor, in order to reduce the impact on the memory bus. During the decoding phase of this trace data, the target application binary is parsed and, along with the branch instruction information from the hardware trace, the complete control flow is generated. Highly granular control-flow of programs is especially important in performance and security analysis of embedded and large scale production systems.

This approach has been used faithfully for program flow reconstruction at the kernel as well as userspace level. Introduction of hardware tracing blocks in mass-produced commodity processors will soon lead to their quicker adoption and improved analysis tools. However, an important caveat manifests in scenarios where runtime code compilation is used. While experimenting, we observed that dynamically generated and compiled code in binaries was not being faithfully reconstructed while decoding recorded hardware traces.

In this paper, we present a novel technique for reconstructing runtime generated program flow using operating system support, without any support from the JIT compilation mechanism or any other API in userspace. We were able to gather runtime compiled code with operating system support which we could then use in our offline analysis. We further propose a decoding mechanism that accounts for these code changes happening at runtime, in memory address locations that were opaque to the decoder earlier. We demonstrate the efficacy of our approach using as experimental base the extended Berkeley Packet Filter (eBPF) JIT compiler, Linux and the Intel Processor Trace (PT) hardware tracing blocks. Our experiments demonstrate that with a very low overhead of 1.1 to 2.8μs, runtime code reconstruction using hardware-only tracing is indeed possible - thus seamlessly integrating static and dynamically generated program flow, with a language agnostic approach. We further propose extensions of this technique to commonly known non-traceable regions in the Linux kernel itself, where are used runtime code patching techniques such as static-key instrumentation [114].

The rest of the paper is organized as follows : We first elaborate on the background of program flow, hardware tracing and code generation techniques in section 7.2 and present our motivation for this research. We define the problems and review the state-of-the-art techniques that have been used until now for JIT code reconstruction from hardware traces. In section 7.4 we discuss our algorithm for OS-supported JIT code access tracking and propose an architecture and implementation strategy for our approach. Thereafter, we demonstrate its use with two illustrative usecases in section 7.5. In section 7.6, we analyze the overhead incurred in OS assisted hardware tracing with Intel PT along with some typical execution profiles gathered from code reconstruction. We conclude the paper outlining our observations and identifying pointers for future work.

7.2 Literature Review

Recovering program flow traces from an execution can provide in-depth details about program executions and has been widely used in feedback based compiler optimization [106] and development of control flow analysis tools such as Callgrind and Kcachegrind [51, 49, 50]. Larus et al. have discussed about instrumenting code and injecting tracing code into function blocks or control-flow edges to keep track of instructions and get insights on their execution frequency [48, 41]. They reported an overhead of 0.2% to 5%, when the effect of the extra overhead of disk writes was not taken into consideration. Program-flow tracing can either be a very detailed low level all-instruction trace implementation, or a more lightweight branch-only control-flow trace generation. Complete instruction flow traces have been supported by various hardware architectures in the form of Coresight (ARM), NSTrace (PowerPC) and PDTrace (MIPS) [57, 58]. Instruction flow traces are gathered from specialized hardware which records each and every instruction executed by the processor in addition to its associated data such as instruction pointers or register values. This instruction and data trace provides a fine-grained view of system with details such as execution profiles and memory access patterns. The drawback of such an approach is the huge amount of data produced, which requires additional external hardware for trace analysis. The extra overhead on the memory subsystem has been observed earlier as well [48, 41]. An alternative is to record only change of flow and branch instructions, and reconstruct the flow post execution, while decoding with information about the executed binaries. Dedicated hardware blocks in the Intel architecture, such as Last Branch Record (LBR), Branch Trace Store (BTS) [59], and more recently Intel Processor Trace (PT) [89] choose to only record branches in the currently executing code on the CPU cores. Intel BTS uses 24 bits per branch, and causes the CPU to enter a special debug mode, which leads to an overhead of 20% to 100% for varying branch

profiles in the applications [62, 63]. Intel PT however records 1 bit per conditional branch, and the complete IP for indirect branches, thus reducing the trace bandwidth further. Intel PT generates this information in an encoded form, as multiple hardware packets such as Taken-Not-Taken (TNT), Target IP (TIP), Time Stamp Counter (TSC), which are then decoded post execution to generate the program flow [60]. In previous work, we observed Intel PT's low overhead as 2-3 % for common workloads, compared to other software-only program flow trace approaches which had an overhead of up to 63.5% [60]. Jörg et al. in [107] use Intel PT for improving security and dependability in software by presenting a data parallel provenance algorithm. Analysis of virtual machines using host-only Intel PT hardware traces has also been explored recently [115].

The most common candidate for runtime code generation are the process virtual machines and bytecode interpreters that use JIT compilation techniques to improve execution speed. The eBPF bytecode interpreter, originally developed for packet filtering in the OS kernel, has been recently improved to generate optimized and verified JIT code for execution [33, 116]. Modern implementations of Javascript support in browsers also leverage V8 engine's JIT capabilities to provide native execution speed for Javascript code. Debugging and analysis of runtime generated code is however still a challenge, owing to the fact that code is generated and modified on the fly.

Tikir et al. have discussed an approach where they use dynamic recompilation of Java code to lazily insert instrumentation code at runtime to gather debugging information from JIT when required [65]. Their approach is however language specific and deals with additions done to the Java Virtual Machine Debugger Interface (JVMDI).

Very recently, Hawkins et al. have proposed new techniques for improving Dynamic Binary Translation (DBT), relevant for dynamically generated code [64]. Their DynamoRIO based approach, claiming a 7.3x improvement over its predecessors, is based on keeping track of JIT compilation and dynamically instrumenting its translation. They also propose static annotation techniques for dynamically generated code.

A recent patent by Koltachev et al. [66] takes a unique approach to debug JIT applications by allowing them to execute and natively compile their code under a debugging session, thus allowing a dual-debug session with native as well as interpreted code analysis. A similar approach is taken by authors in [40] where they implement runtime interposition with the help of an *agent* that handles all language transitions and allows mix-mode debugging of code between multiple languages.

In their research on post-mortem control flow generation, Ayers et al. [67] point out the difficulty in keeping track of native code generated from Java or other such languages, as there

is no standard way to instrument such code. Thus, they provide a customized solution, which uses runtime code instrumentation, at control-flow block granularity, for binary program analysis that records traces for post-analysis. Joan Calvet et al. [117] also discusses instruction analysis based dynamic tracing techniques for inference of cryptographic primitive patterns at runtime by identifying I/O parameters of such obfuscated functions.

Intel provides a comprehensive JIT Profiling API to report information about just-in-time generated code that can be used by performance tools. The intended use is by JIT compilers themselves, where they can use the API calls to report execution profiles by sending execution traces to the Intel VTune profiler [68]. The major drawback of this approach is that this requires a very invasive JIT engine re-compilation.

In summary, most state-of-the-art tools approach the runtime compiled code analysis issue by either providing a language specific API or by instrumenting the native code generators themselves – either dynamically or statically. While this may work on a per-case basis or for use by mixed-mode debuggers, in case hardware tracing is active, to the best of our knowledge, no solution exists that is able to transparently record all runtime compiled code execution along with native code – especially when branch traces are being used to reconstruct instruction flow. We deviate from the approaches taken by these tools and move towards a more encompassing approach which uses operating system support.

7.3 Background and Motivation

Almost all the major applications used today have some form of dynamically generated code being executed inside. Web browsers aggressively use the latest Just-in-time (JIT) compilation techniques to speed up Javascript execution. Dynamic instrumentation and dynamic translation of code is also common in trace tools that inject performance analysis code for gathering metrics from running applications. However, program flow analysis with hardware trace of such code presents many challenges. To explain the current limitations, we first define its scope, some background on code execution and then move towards an example where it manifests.

For a given target process P, Figure 7.1 illustrates the operating system's view of the process memory. The virtual address space for P has some content in the form of pages, a number of which are in the executable Virtual Memory Areas (VMA), typically the `.text` sections of a process, which contains the executable code of the program and shared library code. This is shown as VMA_1 and contains executable *file-backed* pages which we name as code section CS_p. Figure 7.2 shows this small linearly executing code section CS_p of the process

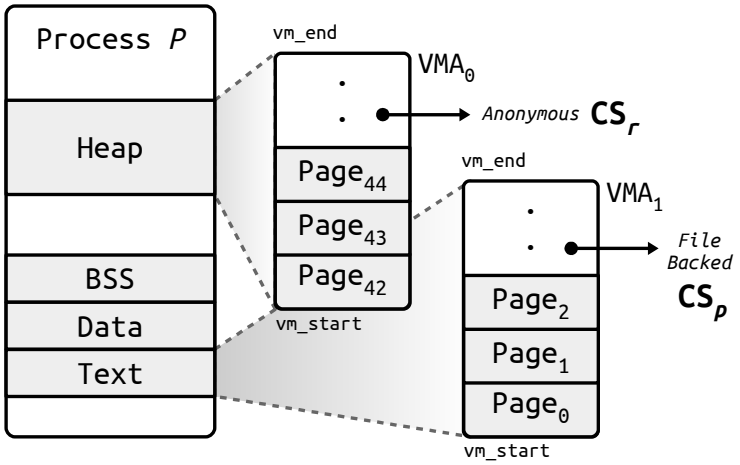

Figure 7.1 Runtime and file-backed code section for a process P as observed by the OS

P. Consider that P now generates dynamically compiled code. Typically for such code, the memory is dynamically allocated on the heap and the code copied to the assigned pages which are then marked as executable. Unlike pages in VMA_1, these pages in VMA_0 are *anonymous* and do not contain a backing file. We name these pages as part of the code section CS_r. At runtime, some of these pages may need to be modified and revised. As seen in figure 7.2, at execution, each revision or a new dynamically compiled section can be considered as a single segment CS_{rn}, where n is the number of times a new section is added or a previous section revised and rewritten at runtime. This behavior is common for every userspace and in-kernel dynamically executed code. As an example, for a userspace JIT compiled network packet filter based on eBPF, CS_r may represent a single page worth of dynamically compiled filter code which may be modified repeatedly at runtime, based on policy requirements. We elaborate more on this in section 7.5. We can now define the process control flow function $F(P)$ as,

$$F(P) = F(CS_p) \cup \sum_{i=1}^{n} F(CS_{ri})$$

where F denotes the instruction flow of a given code and $\sum$ signifies the union of individual flows of CSr_{ri}. However, a software-only approach for generating the flow $F(P)$ would also involve extra code sections before each CS_r and add additional instructions to the critical execution. As discussed, in case of JIT compiled code, this is only currently achieved using JIT compiler specific functions or language dependent APIs [65, 68].

Modern hardware tracing techniques however allow $F(P)$ to be reconstructed post-execution,

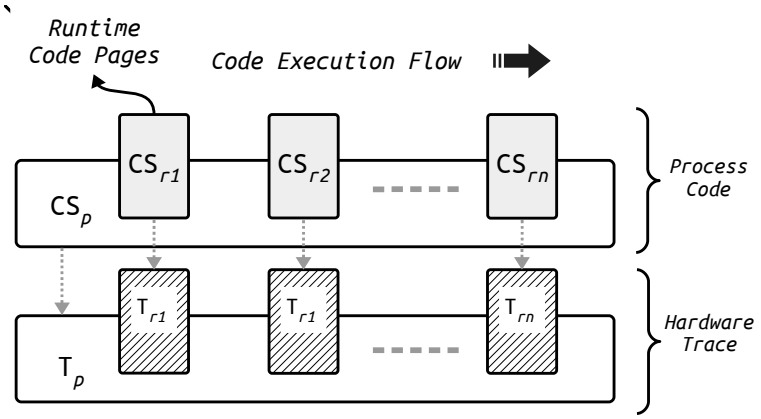

Figure 7.2 Corresponding hardware pages

thus generating true execution profiles at very low-overhead. We have discussed this in detail in our previous work [60]. Therefore, for each branch encountered in CS_p and CS_r, the processor generates encoded trace packets representing the decision on a branch taken or not taken, along with the instruction pointer (IP) if required. We represent these trace packets symbolically for CS_p and CS_r as T_p and T_r. For branch traces, the decoding of this encoded trace requires the availability on disk of the static binaries of the running process, as the pages belonging to this VMA are file-backed. Therefore when traced with hardware, the process code section control flow $F(CS_p)$ can now be derived as,

$$F(CS_p) = \Pi(CS_p, T_p)$$

where Π is a map and merge function that takes the statically available process code segment (CS_p) and the corresponding hardware trace packets (T_p) as input, and generates the flow as output. However, for dynamically generated CS_{rn} section, it is not possible to faithfully obtain $F(CS_{rn})$, as the packets T_{rn} don't map to any available code segments, since they belong to a VMA which is anonymous memory. For example, JIT compilers cause in memory execution of short sections of dynamically generated code which the hardware trace decoders fail to account for, as they expect static binaries while decoding. As discussed in the previous section, a solution to the problem of non-availability of CS_r sections is use JIT or language specific APIs that periodically dump runtime compiled code when it is generated and executed. However, this may require recompilation of the JIT supported runtime, which may not be in diagnosing production systems that don't allow code modifications. Moreover, this also adds the undesired API code in the critical flow path which we observe eventually in $F(P)$.

We observed this problem throughout in many locations in the Linux kernel where modifying

code for optimization is a fairly common occurrence in trace, network packet filter and security subsystems. The problem is more acute in userspace where multiple languages may be using JIT compilation, and APIs to dump and analyze JIT code may not always be available. This motivated us to approach the limitation of reconstruction in state-of-the-art hardware trace systems, from a different perspective. We therefore propose a kernel-assisted technique that monitors and keeps track of executable code memory to record CS_{rn} sections transparently, in order to generate accurate program flow. Therefore, to get the flow of a given dynamic code section CS_{rn} we can define $F(CS_{rn})$ as,

$$F(CS_{rn}) = \Gamma(CS_{rn}, vma(CS_{rn}), ts(CS_{rn}), T_{rn})$$

Function Γ takes as input the code section (CS_{rn}), address of the VMA in which this CS_{rn} section belongs ($vma(CS_{rn})$), along with the timestamp of the revision of this section ($ts(CS_{rn})$) and the associated hardware trace for the section (T_{rn}). We store the timestamp, address and content of each new dynamic code section revision with our FlowJIT technique, which then allows reconstruction of the hardware trace, something not otherwise possible.

7.4 Methodology

In order to monitor runtime-compiled code, our strategy is to first start tracking the processes which are expected to generate executable code at runtime. When the target bytecode is intended to be JIT compiled, or executable code is prepared in a process, a small *code cache* is prepared in the memory where the generated machine code is saved. This code cache may also be updated repeatedly with the updated CS_r sections, as seen in Figure 7.2. Our approach keeps track of each update for each page change of the code-cache. On a POSIX-compliant system, the `malloc()` syscall is used to allocate memory for this cache and its attributes flags are set to executable, using a `mprotect()` syscall and the `PROT_EXEC` flags supplied with it. We essentially keep track of access rights on this memory section and generate artificial page-faults from the kernel to extract the executable pages. The following subsection describes the FlowJIT approach in detail.

7.4.1 FlowJIT Architecure

FlowJIT begins by tracking the individual pages in the kernel for the tracked process which is runtime compiling native code for execution. As shown in Figure 7.3, a userspace program `flowjit-agent` is used to register the target process using a FlowJIT specific `ioctl()`. This sets a flag in the kernel task structure for the process, which then marks the process to

be monitored. Next, FlowJIT intercepts the access protection functions in the kernel which

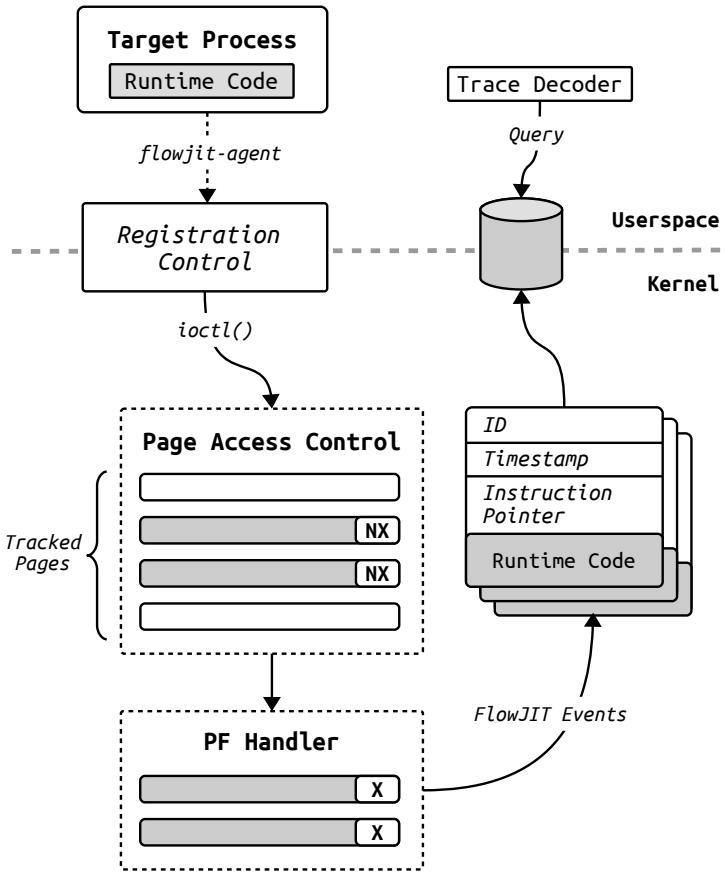

Figure 7.3 FlowJIT Architecture

identify the executable pages requested by the target process. As discussed before, these pages correspond to the CS_r code sections. The page access control module in FlowJIT sets those pages as tracked and flips their protection rights from *executable* to *non-executable* (NX). The Page Fault Handler manages access-faults on those pages and sets the access rights back to executable. At the same time, it copies the executable code from the kernel and generates a synthetic FlowJIT event by adding the current time-stamp, the faulting instruction IP, and assigning an ID. Any update to the JIT code for the same IP is stored in a linked list in the kernel. A userspace daemon, part of the hardware trace decoder framework, lazily copies the linked list contents from the exposed DebugFS file to the disk. The decoder can then query the FlowJIT events based on their IP and synchronize them with the timestamp to complete the code reconstruction. The linked list data is indexed according to the virtual address for the dynamic pages and queried by address and time when the decoder encounters a virtual address where the decoding failed. We demonstrate this in detail in section 7.5, where we

properly reconstruct the otherwise incompletely decoded hardware trace gathered from the Intel PT trace hardware.

In-Kernel Access Tracking

The access management part of the FlowJIT system can be defined using a state machine where the states denote the status and access rights of pages in a virtual memory area (VMA). In our context, the pages may be in state either *Tracked*, *Untracked* or *Fault*. In Figure 7.4, we show how these states can transition, and the access rights on the pages thereafter. Initially,

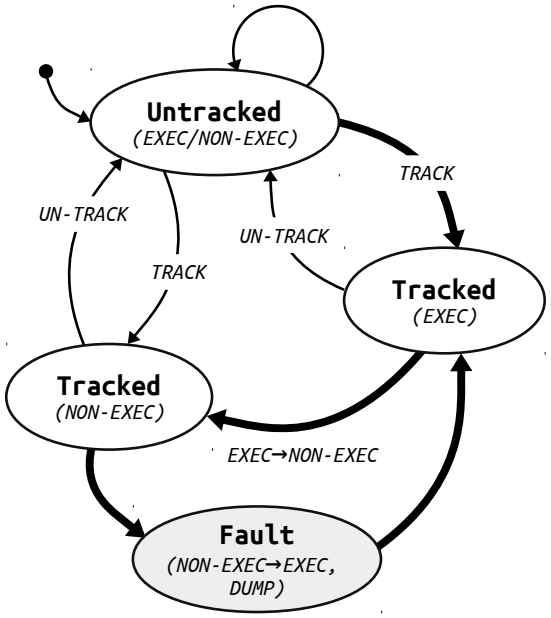

Figure 7.4 Access Tracking State Machine

when the process registers itself with the kernel using the `flowjit-agent`, all its pages, whether executable or non-executable, are marked as tracked and remain in this state until the agent requests them to be untracked. Tracked (Exec) pages which are executable are artificially marked as non-executable by FlowJIT and they enter the Tracked (Non-Exec) state. In the normal course of execution, when the application tries to execute them, they generate a page fault and enter the Fault state. We handle that page fault in the kernel and query if the page was originally tracked and its attributes changed. In such a case, the page attributes are changed back to their original values and a synthetic FlowJIT event is generated. The event is dumped and the page again enters the Tracked (Exec) state and is then untracked by the agent. The underlying process has been illustrated in Algorithm 3

Algorithm 3 Access Tracking Algorithm

Input : VMAs containing CS_r and CS_p for P
Output : Updated FlowJIT event buffer (B_{ev})

$V_r \leftarrow vma(CS_r)$
$V_p \leftarrow vma(CS_p)$
$V = \{V_r, V_p\}$
while $P.pid$ is *tracked* and *page* in $exec\_pages(V)$ **do**
    if $(page \in V_r)$ **then**
        $page.arm = armed$
        FlowjitToggle($page$, *INIT*)
    end if
    if $(page.fault_{exec})$ **then**
        FlowjitHandle($page$) ▷ Access-fault handler
    end if
end while
function FlowjitToggle($page, state$)
    if ($state$ is *INIT*) **then**
        $page.exec \leftarrow 0$ ▷ FlowJIT sets NX
    else if ($state$ is *PF*) **then**
        $ev \leftarrow \{timestamp, address(V_{page})\}$
        FlowjitEventDump(ev)
        $page.exec \leftarrow 1$ ▷ FlowJIT resets NX
    end if
end function
function FlowjitHandle($page$)
    if ($page.arm$ is *armed*) **then**
        $page.arm = disarm$
        FlowjitToggle($page$, *PF*)
    end if
end function
function FlowjitEventDump($event$)
    $B_{ev} \leftarrow event$
    $flush(B_{ev})$ ▷ Buffer copied to disk
end function

We have released FlowJIT as an open-source kernel patch [1] which can be activated by a simple configuration option in the kernel. While most of the other runtime code profiling techniques require compiler-specific APIs or manual addition of code in the target application/runtime compiler engine, the FlowJIT technique requires no other modification to the source code and tackles the problem of runtime code reconstruction at kernel level. This makes it versatile to use as well as the first approach to tackle runtime compiled code reconstruction for hardware

1. `http://github.com/tuxology/flowjit`

traces. We explain this using two illustrative examples in the next section.

7.5 Illustrative Use-Cases

7.5.1 JIT Compiled Code

Recently improved and extended, the Berkeley Packet Filter (eBPF) in the kernel is being increasingly adopted for traffic filtering, shaping and classification. Its versatile in-kernel virtual machine is also being used to improve trace aggregation and trace filtering. At the heart of eBPF's improvement is the eBPF's JIT compiler [116]. Hardware trace reconstruction for eBPF code is therefore important for a complete execution flow. To demonstrate our FlowJIT technique on a JIT compiled case, we consider a scenario where a user is executing code within a process using a userspace eBPF process VM. This is illustrative of the real-life use of eBPF to filter syscalls using `seccomp-bpf` in Chrome or userspace network packet filtering in tools like Wireshark. We use uBPF [2] to emulate eBPF's behavior in userspace while hardware tracing is activated. As an example, our target process (P) executes a dynamically compiled eBPF code section (CS_r) which is a simple loop that increments an integer 42 times. This section is different from the normal code of P (CS_p), which includes executable code in its address space such as its `.text` section, and dynamically loaded libraries. As P executes, uBPF builds the CS_r section as a identifiable function, complete with a function prologue and epilogue. This is stored in a code cache whose access permissions are set to executable and executed using a simple `call` instruction after loading the address of the code cache CS_r in the `rax` register on an x86 machine.

The process by which FlowJIT works in this context has been illustrated in pseudocode 4, where FlowJIT essentially intercepts the `mprotect()` call made by the uBPF compiler, as it intends to set executable permissions for the compiled bytecode. However, FlowJIT intercepts it in the kernel and flips the executable flags. Eventually, an access page fault is induced, which FlowJIT handles and generates a synthetic event. This FlowJIT event contains the timestamp, the virtual address of the code segment and the raw binary data which was the JIT compiled code. Eventually, we add this event to a memory mapped buffer which is lazily copied to a userspace file.

During the above execution, hardware tracing using Intel PT was enabled and the raw trace data was collected while the process P was executed. The decoded trace data consists of two interesting packets - the Target Instruction Pointer (TIP) packet and Taken-Not-Taken (TNT) packet. The TIP packet is generated when an indirect branch, exception or an in-

2. `https://github.com/iovisor/ubpf`

Algorithm 4

1: USERSPACE PROCESS P :
2: $mem \leftarrow malloc()$
3: $CS_r \leftarrow compile(bytecode, mem)$
4: $mprotect(mem)$ with $Flags = \{EXEC, !WRITE\}$
5: IN-KERNEL FLOWJIT :
6: **On** *mprotect(mem, Flags)* :
7: **if** (mem **is** *anonymous*) **then**
8: Set $Flags = \{!EXEC\}$
9: $mem.tracked = 1$
10: **end if**
11: **Continue** with original *mprotect()*
12: **if** (P executes CS_r from mem) **then**
13: **if** ({*EXEC*} **not in** $Flags$ **and** $mem.tracked$ **is** 1) **then**
14: $page \leftarrow page\_fault(mem)$ ▷ Access-fault on tracked VMA
15: $fault\_handler(page)$
16: $event \leftarrow build\_event(page, timestamp, address(mem))$
17: $list \frown dump(event)$ ▷ Add *event* to *list*
18: Set $Flags = \{EXEC\}$
19: Continue P execution
20: **else if** ($Flags$ **is** $\{EXEC\}$) **then**
21: Continue P execution
22: **end if**
23: **end if**

terrupt occurs. The TNT packets are issued at each conditional branch and loop instruction and contain one bit for each branch taken or not taken. Thus, in our case, when P tried to execute the runtime compiled CS_r section of code by issuing a `call %rax` instruction, the processor generated a TIP packet with the value of the IP of the CS_r section. While decoding the trace data, the TNT packets are associated with the binaries of P and its associated libraries on disk to complete the instruction flow in the decoded trace. However, we observed that, when the runtime compiled CS_r section address was encountered, the decoding failed as no memory was associated with that address. This is illustrated in the missing association of trace packets (T_r) in Figure 7.5.

To overcome this decoding failure, we used the dumped FlowJIT events and queried the runtime compiled images based on the IP retrieved from the TIP trace packet when the decoding failed. This IP corresponds to the virtual address which is used as an index to access the individual FlowJIT event from the dumped data. The timestamp is used to verify the version of the JIT copy (in case the runtime compiled code was updated multiple times). To reconstruct the flow of the retrieved image (I_r), we first generate the control flow graph

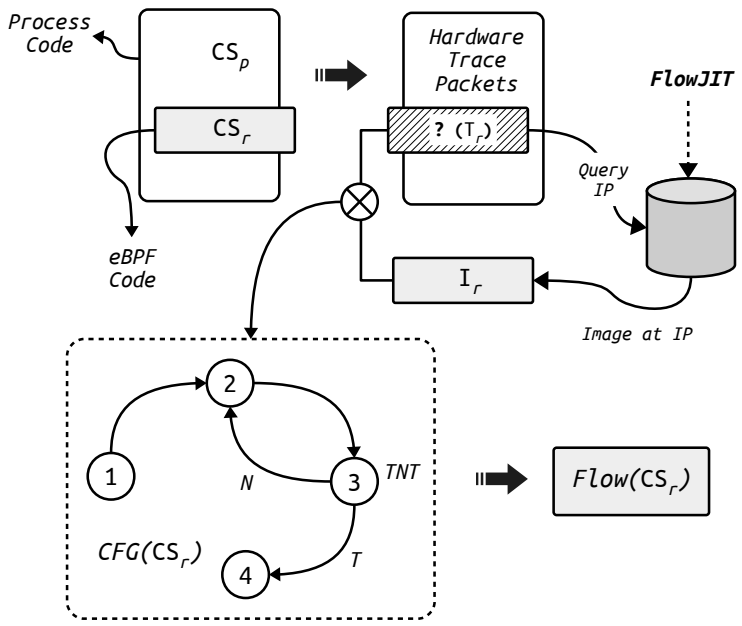

Figure 7.5 FlowJIT retrieved runtime code image of eBPF (I_r) merged with failed decoding in hardware trace (T_r) to reconstruct flow of JIT compiled code

representation of this image (CFG(CS_r)) and then merge it with the associated trace packets T_r. For this example, 1, 2 and 4 nodes represent segments which have a unconditional jump and are thus omitted from the trace T_r, whereas node 3 represents a conditional jump which generates 42 TNT packets with the first 41 bits being N and the last bit T. The TNT packets are mapped to the conditional branches in I_r sequentially, which results in the true flow of the hardware trace (Flow(CS_r)). Therefore, using FlowJIT, runtime JIT compiled eBPF code could be reconstructed successfully, without any dedicated API or invasive process, by just using kernel support.

7.5.2 Static Key Instrumentation

Another important scenario where current hardware decoding is incorrect is the commonly used static key approach in the Linux kernel, which avoids conditional branches while keeping much of their functionality [114]. Static keys are actually dynamically activated by modifying instructions inside the kernel. The technique uses the `asm goto` statements in GCC which allow branching to a label. Thus, the branches may be taken or not taken by default while avoiding a memory reference each time. In case the branch direction needs to be changed, the `nop` introduced by the static key instrumentation are patched with the targets at runtime. This technique is useful in avoiding conditional checks in performance sensitive code in the

kernel, such as tracepoints. We devised a userspace test for static key instrumentation which consisted of the first execution of a function `foo()` with the default `nop` and another execution with the branch being modified by patching the jump site on-the-fly. Listing 7.1 and 7.2 show the target function, and the corresponding assembly code and jump-site patching respectively. At runtime, the jump-site at line 3 in Listing 7.2 is patched with a relative jump to line 7 (as `e9 23 00 00 00`), which corresponds to line 7 in Listing 7.1. The Linux kernel relies heavily on static key for reducing the cost of tracepoints further and providing a zero-overhead execution when default branch conditions are intended to be taken.

Listing 7.1 The function `foo()` showing static-key being used

```
void foo()
{
    printf("foo entry\n");
    if (asm_goto_full(&key, 1)) {
        printf("STATIC_KEY IS TRUE\n");
    } else {
        printf("STATIC_KEY IS FALSE\n");
    }
    printf("foo exit\n");
}
```

Listing 7.2 The `nop` inserted with static-key is the patch site

```
4009f0 <foo>:
...
400a00: 0f 1f 44 00 00        nopl 0x0(%rax,%rax,1)
400a05: 48 8d 3d 02 01 00 00  lea 0x102(%rip),%rdi
400a0c: e8 2f fd ff ff        callq  400540
...
400a28: 48 8d 3d fc 00 00 00  lea 0xfc(%rip),%rdi
400a2f: e8 0c fd ff ff        callq  400540
...
```

The code generation and modification for static keys here happens at instruction granularity. Unlike the previous eBPF JIT usecase, the actual binary of the process already exists. Therefore, when we executed our static key test while generating hardware traces with Intel PT, the post-execution decoding worked as the target IPs can be mapped to the functions in the binary. However, on closer inspection, we observed that on both static key branching cases, only the default `nopl` was being shown in the decoded instruction trace instead of the code

from the updated jump-site. We observed this in the Linux kernel as well, where a wrong instruction flow was being reported with hardware tracing. This issue of incorrect decoding can be resolved with our FlowJIT approach. We now ran the same test on our FlowJIT enabled kernel and were able to recover both modified and unmodified static key instrumentation from the dumped FlowJIT events. We then used the extracted IP from the event and matched it with the static key sites. We could then update the incorrect instructions `nopl` in the decoded trace, based on the actual execution trace from FlowJIT, which we found out as `jmp 400a28`. Thus, we were successfully able to reconstruct CS_r sections, which were incorrectly being used for decoding. It is worth noting that the code modification was done in the same page in this case. This also relates to a similar behavior shown in self-modifying code where FlowJIT could also be eventually applied.

In addition to the above examples, there are numerous other scenarios where FlowJIT may be essential. In the Linux kernel, the function tracer uses the `nop` patched `mcount` mechanism, supported by the GCC's profiling options. Function entries can be patched at runtime with actual `mcount` calls when dynamic tracers are enabled. Apart from that, in userspace, shared libraries utilizing load time relocation would require extra decoder support for faithful decoding. In all these cases, FlowJIT can provide a portable and efficient mechanism to reconstruct the actual code flow.

7.6 Experimentation and Results

Any addition to the critical execution path in the kernel causes extra overhead in terms of time taken. In the case of FlowJIT, each runtime compilation location causes an extra access page fault. In our current implementation, the generation of each FlowJIT event causes the addition of a total of 4132 bytes (modified executable page, identifiers and timestamp data), appended to a linked list in the kernel. In this section, we provide a comprehensive study of the overhead associated with FlowJIT in terms of extra time and number of access faults associated with executions.

Test Setup Overhead tests were performed on a machine with an Intel Skylake i5-6600K processor, with Intel PT support, running a FlowJIT patched Linux kernel, version 4.7, and Fedora 23 distribution. CPU auto-scaling was disabled and the operating frequency was fixed to 3.5GHz, to minimize jitters and have reproducible results.

We first analyzed the effect of JIT compiled code sites on the number of page faults caused by the process, when FlowJIT was activated for our target process. The test target consisted of a dynamically compiled eBPF program being recompiled and re-executed multiple times from

within the process. The page faults and time in subsequent experiments were measured for the complete execution of the test target. Following the nomenclature used earlier, these JIT sites may be seen as the runtime code segments CS_r. We therefore vary the number of CS_r in a program in logarithmic steps and observe the number of page faults (PF_n) caused. Figure 7.6 shows the FlowJIT enabled and disabled cases for our eBPF test program in comparison to a controlled baseline test, where the page faults were tightly controlled.

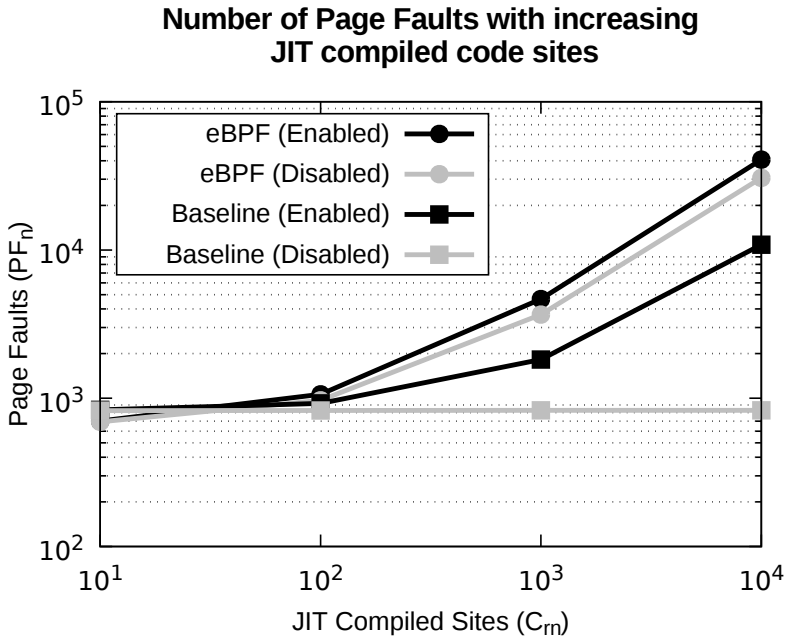

Figure 7.6 Effect of JIT compile sites on Page Faults with FlowJIT

We observe in the baseline test that the additional number of page faults, caused in the FlowJIT enabled case, account for most of the page faults. We measured that the exact amount of tracked access faults were equal to the extra JIT sites added. The overhead on top of the baseline looks magnified. Indeed, in a more realistic scenario, with multiple JIT sites executing in our eBPF enabled program, the overhead, caused by FlowJIT induced access faults, would be amortized over more numerous already existing page faults.

We also measured the absolute overhead of our test program with increasing CS_r values. We can see in Figure 7.7 that the extra time taken by the test process, while FlowJIT was enabled, remained largely consistent, even as the JIT compiled sites increased. The maximum overhead was 19.3%, as measured for 60K different JIT executions.

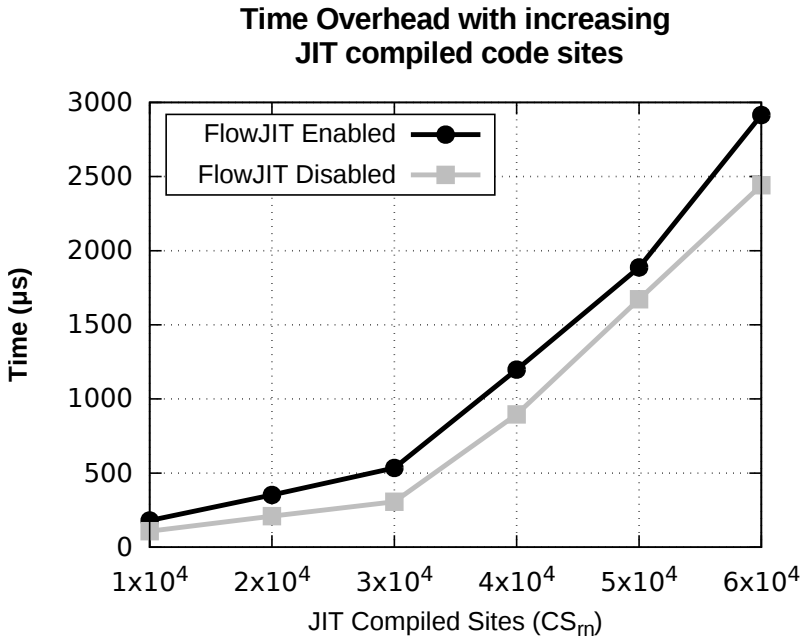

Figure 7.7 Overhead of FlowJIT enabled executions with increasing JIT compiled sites

7.6.1 Access Fault Overhead

As evident from baseline results in Figure 7.6, the number of extra CS_r sections cause almost the exact same number of access faults. This is therefore the absolute minimum overhead that FlowJIT causes. Therefore, in order to quantify this per access-fault overhead in execution, we devised a synthetic experiment that performs 20 thousand small code compilations and subsequently executes them. We then measure precisely in-kernel how much time a single FlowJIT event takes. For each execution, the overhead T can be quantified as,

$$T = t_{tr} + t_{ac} + t_{pf}$$

Here, t_{tr} denotes the tracking initiation time, t_{ac} denotes page access change time and t_{pf} denotes the access page fault time. Out of these, t_{tr} can be safely neglected as the tracking initiation is a one time cost and would be insignificant over recurring $t_{ac} + t_{pf}$ time in our experiment. Therefore, at runtime, FlowJIT's addition to the critical execution path time now is (1) modifying access permissions ; (2) allocate memory for FlowJIT event ; (3) obtain and copy timestamp and instruction pointer ; (4) copy the modified page and (5) add the FlowJIT event to the linked list. This is the flow that is of interest to us. We label this as revised overhead ($T_{obs} = t_{ac} + t_{pf}$) and monitor it by putting timestamps for the complete

T_{obs} flow inside our FlowJIT kernel implementation.

We plotted the density curve for the 20 thousand executions, with the observed overhead per access fault and page dump time (T_{obs}) as a parameter. As observed in Figure 7.8, the overhead of FlowJIT varied in our synthetic tests from 973ns to 6.8μs with three distinct peaks at 1.3μs, 2.7μs and 6.6μs. This multi-modal variation was a result of the non-uniform nature of page-faults themselves, due to the caching effects related to VMA resolution and subsequent TLB hits and misses. To minimize variations as much as possible, and ensuring that the observations were reproducible, we isolated page faults to just those access faults which were caused by our FlowJIT tracked pages. We can thus infer that, in an ideal scenario, most of the page faults took around 1.3μs. The individual values of t_{ac} obtained in our tests had an average of 20.8ns, which was very low compared to t_{pf}. Overall, the total cost is almost negligible, considering the huge number of page faults that naturally occur in the due course of process execution.

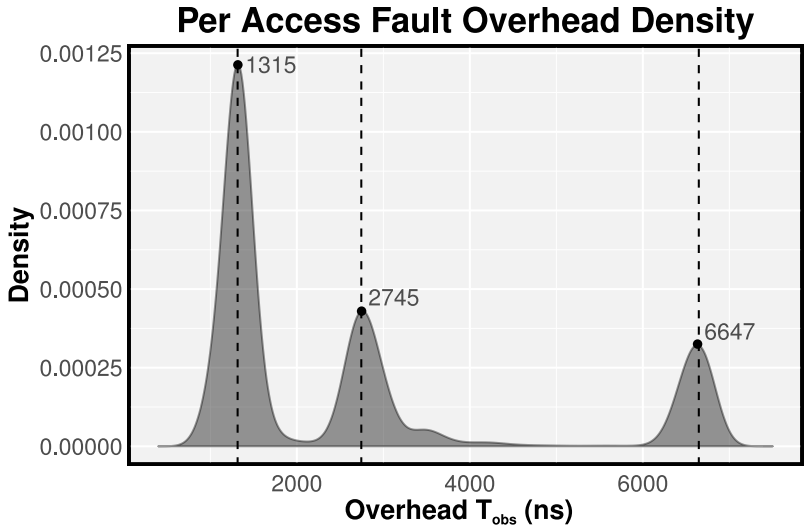

Figure 7.8 Distribution of overhead for each access fault

The current implementation of FlowJIT requires copying tracked pages between kernel and userspace. In future versions, we plan to reduce the overhead further to a few nanoseconds range by userspace-kernel shared memory implementation or the `splice()` technique used in [11], thus minimizing kernel-userspace copy costs. For our current implementation, the event copy cost is part of the t_{pf} overhead discussed before. We confirmed that our observations are in line with our synthetic tests of `memcpy()`, where 349ns were required on the test system for copying data of similar size as a single FlowJIT event.

7.7 Conclusion

Recent advancements in hardware tracing allow compact and accurate branch traces to be generated for a target process or OS kernel. However, we observed that decoding the traces post-execution either fails or produces erroneous instruction flow when runtime compiled code is generated and executed by the target. To overcome this problem, we developed FlowJIT, a kernel-assisted technique that allows tracking of pages containing runtime generated code, and dumping them for rectifying and reconstructing the hardware trace. Our OS-centric approach is language and JIT compiler agnostic, thus removes the reliance on assistance from their infrastructure. Our technique allows accurate instruction flow reconstruction, with a negligible cost of around 1.3μs of extra execution overhead per access fault event. FlowJIT solves one of the major flaws in dynamically compiled code reconstruction that has not yet been tackled until now. This approach with minimal overhead makes hardware tracing more robust and reliable.

Even though we cover the two major cases for runtime generated code – method based JIT compilation and direct code instrumentation, a natural extension of this work would be to cover the special case of self-modifying code as well. Currently, such an analysis would require single-stepping instructions in the target process to assess if they are being rewritten while recording individual changes. As we observed in Subsection 7.5.2, the modifications within the same page could indeed be observed by FlowJIT. This may induce considerable overhead but open up larger avenues for code security analysis using hardware traces. Another extension to our technique may be to identify and track all executable pages of the process, thus eliminating the need to rely on on-disk process binaries for code reconstruction. This would allow faster and efficient decoding and instruction profiling, but with a trade-off of not having function names at analysis time, since this is stored in extra sections in the binaries on disk.

CHAPTER 8 GENERAL DISCUSSION

In this chapter, we revisit the objectives set at the beginning and discuss how our research helped in achieving those. We also observe the broad impact of our work in our domain and in industry outlining specific contributions, outcomes and limitations of our research.

8.1 Revisiting Milestones

Our strategy of defining milestones and directing our research to achieve those was discussed earlier in Chapter 3. In this section, we further the discussion on our research progression identifying specific instances from our work that helped in reaching the milestones.

Conditional Tracing Enhancements A part of our work presented in the first article dealt with how filtering can be enhanced for a fast userspace tracing tool such as LTTng. It is important to note in such cases that per filter overhead may seem only a few nanoseconds each but this is of huge importance when the event frequency is high. Advancements in network packet filtering over the past years have been leading the discussion in the filtering domain. Our work with eBPF based JIT filters for providing conditional tracing to LTTng, puts the spotlight on the tracing aspect. Tracing network events at high speed is a challenge in itself. Our work aims at providing conditional support so that similar filtering techniques can be used in userspace as well as kernel space scenarios. This milestone formed the fist step towards our goal of achieving a low-overhead filtered tracing architecture where events could be detailed and recorded only when deemed necessary.

User-Kernel Co-operative Tracing The first article also discussed a novel architecture which allowed similar kernel and userspace VMs to communicate and provide a co-operative tracing approach. Our new mechanism enabled what we can call co-operative conditional tracing. A major part of this was facilitated by our new shared memory architecture which allowed userspace as well as kernel code to act on filtering conditions collectively. This is an important step in building a mechanism where we cannot just provide a fast filtering mechanism but also provide a way to control the information generated, so that the context is not lost even though the events recorded are minimized by filtering. Our contributions advanced the software based filtering approach and extended the available JIT supported bytecode mechanisms it uses.

Hardware Trace Analysis Our research continued for achieving the next milestone where we shifted our focus from a software only approach and investigated hardware tracing advances in modern processor architectures. This allowed us to provide a more fine grained view of the system at a very low overhead than a software-exclusive approach. While the hardware tracing domain is not new in itself, the advances in silicon which make it low-overhead and easily accessible in commodity processors are very recent and have not been explored earlier. We devised a new algorithm and trace analysis technique that allows instruction level analysis for latency detection of interrupts and syscalls. In addition, we provide empirical evidence of our low overhead approach through rigorous experimentation and characterization of hardware tracing techniques. This milestone was a major one in tracing - considering that advanced analysis from modern hardware tracing infrastructures is still at a nascent stage in the performance analysis domain.

Hardware-Assisted VM Analysis Owing to their importance in the modern distributed computing landscape, we set our fourth research milestone as the analysis of VMs. Earlier, it required support from the guest hypervisor traces as well as host hypervisor traces. Thus, in the third article, we have proposed a new algorithm called *HAVAna* that allows almost zero overhead analysis of VMs without any software tracing running on guest or even on the host. Our algorithm deciphers hardware trace packets without any intrusion in the VM through which we could analyze its computing resource consumption and process scheduling on the CPUs. Such an approach to tracing of VMs had not been proposed till now.

Hardware Trace Reconstruction Improvements While working on the hardware and software milestones set for our research, we observed that when runtime generated code such as JIT compiled userspace eBPF programs were used in conjunction with hardware tracing, the trace reconstruction was anomalous. Therefore, in the fourth article, we developed a new algorithm called *FlowJIT* to identify instances of runtime compiled code or instruction modification scenarios with the help of kernel assistance and tracked them so that they could be used for offline hardware trace reconstruction. Our new algorithm was able to solve the trace reconstruction failure we observed in kernel as well as userspace cases. This completed our initial goal of creating a robust, low overhead and detailed tracing technique.

8.2 Research Impact

Our work on an enhanced software tracing approache led to development of fast trace filtering techniques that we now use in userspace. Our userspace implementation of the JIT supported

eBPF VM has been open-sourced [1] as a general purpose filtering library that can be adapted in any userspace project. Further development of other userspace eBPF libraries by industry in networking domain have evolved since then for similar filtering usecases. Hardware assistance for tracing can be considered as the next paradigm in low-overhead tracing. Our work investigated this hardware-software interaction to provide a highly detailed program flow tracing output. This marks a major shift in how high-speed tracing further evolves. Our own work for hardware assisted VM analysis has been released as open source software with an interactive visualization tool based on Trace Compass [2]. Apart from that, our final research work of hardware trace reconstruction has been released as a Linux kernel patch [3]. This will help the research community and hardware developers to improve upon their designs and incorporate features to minimize errors.

8.3 Limitations

Even though we strive for a thorough software-hardware amalgamated approach to tracing for a low-overhead and fine-grained tracing approach, some limitations on our research still exist. Aggressive filtering without application and kernel context, reduces the trace size to a greater extent but can significantly reduce the trace usefulness. Even though this can be reduced by a co-operative tracing approach, the choice of filter predicates is user controlled and not automated yet. In addition to this, our userspace eBPF library is not designed to be thread safe and there are opportunities for improvements in supporting multi-threaded applications. Apart from that, hardware assistance for low overhead tracing requires specialized hardware support in the production environments which is vendor dependent and needs separate implementations for different hardware trace packet formats and protocols. In addition, the lack of standardization in hardware tracing among different silicon vendors would require multiple implementations of our approach with minor adjustments to each. However, for processors of the same architecture, design standardization such as in ARM Coresight and Intel PT are now being introduced gradually. Using raw hardware trace packets for analysis of VMs also reduces the analysis scope to the identification of processes by the associated CR3 values, and the identification of vCPUs by isolating unique VMCS values. Even though this is sufficient, there is scope for adding identification context such as process names or PIDs.

1. `http://step.polymtl.ca/~suchakra/libebpf.tar.gz`
2. `http://step.polymtl.ca/~suchakra/havana.tar.gz`
3. `https://github.com/tuxology/flowjit`

CHAPTER 9 CONCLUSION AND RECOMMENDATIONS

9.1 Concluding Remarks

Software is becoming ever more complex and decentralized by design. Monolithic software architectures are being replaced with micro-services based architectures and run on advanced and powerful modern processors. The analysis of such complex systems requires knowledge of its interactions with the operating system as well as the underlying hardware. In our research work, we had initially set out with a goal of a more fine grained as well as precise software analysis approach, with the help of tracing. Reducing the trace overhead, while maintaining relevant information in the traced data, was one of the most important areas of work we focused on. Recent developments in on-chip resources were an untapped potential in trace analysis but the lack of prior work in this area posed multiple challenges to us. In our work we have discussed in detail how we developed multiple algorithms which bridged hardware-software boundaries and helped in advancing the tracing framework further. In the first phase of research we tacked the problem of low overhead detailed tracing with a software filtering based approach, which eventually led to development of a new and efficient co-operative tracing architecture. While working on this goal, we researched how hardware tracing blocks in processors could assist and lead to an overall low overhead design. We therefore proposed new algorithms that were able to accurately profile syscall and interrupt latency with such an analysis, and use it further for our novel non-intrusive VM analysis algorithm called *HAVAna*.

In our next research phase, we worked on improving the robustness of such an analysis through a new technique of tracking runtime code pages with the help of kernel support. Our new technique called *FlowJIT* was able to accurately reconstruct anomalous hardware trace flow, which was an unsolved problem till now. The research we carried out, cuts across diverse fields in the computing domain - operating systems, hardware analysis as well as graphs and Software Engineering in general. It helped us immensely in enhancing our knowledge as well. We hope that the work presented in this thesis will further help in advancing the field of software analysis and tracing.

9.2 Recommendations for Future Research

Based on our research experience, we recommend that future researchers focus on efficient usage of hardware tracing techniques, exploring its use in other fields such as security analysis. As an extension to our work on VM analysis, process data such as PIDs, maps and names can

be recorded from the guest OS at regular intervals, which can further improve the information provided by the analysis, and help identify resource hungry processes easily. Another aspect of the future work in this area can be the analysis of boot sequences of VMs. Hardware tracing is especially useful for boot time tracing since it is difficult to have the software trace infrastructure working before a large part of kernel and devices are initialized. Thus, execution profiles of whole VMs can be generated with this approach.

We also observed that the hardware trace data generated is huge. Our analysis focused on short spans of executions only. However, this may not be true in all cases. Therefore, there is ample opportunity for research on efficient storage of hardware trace data in distributed databases and to allow parallel decoding and search using Big-data analysis techniques. In addition, to restrict the hardware trace data generation, there may be a possibility to combine native compiled filtering techniques in tracing with the hardware trace snapshot mode, where a cyclic buffer may store hardware trace data continuously. Then a conditional software-trace event is generated, the hardware trace data could also be tagged along with it, for a more accurate analysis. Even though such analysis techniques are more focused on performance domain, the analysis of individual instructions and associated data can lead to further research opportunities in reverse engineering and security analysis of binaries.

REFERENCES

[1] Timur Iskhodzhanov, Reid Kleckner, and Evgeniy Stepanov. Combining compile-time and run-time instrumentation for testing tools. *Programmnye produkty i sistemy*, 3 :224–231, 2013.

[2] M. Hiramatsu and S. Oshima. Djprobe—Kernel probing with the smallest overhead. In *Proceedings of the 2006 Linux Symposium*, Ottawa, Canada, July 2006.

[3] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel probes (kprobes). `https://www.kernel.org/doc/Documentation/kprobes.txt`, 2014. Accessed Nov 2016.

[4] A. Brown and G. Wilson. *The Architecture of Open Source Applications : Elegance, Evolution, and a Few Fearless Hacks*. The Architecture of Open Source Applications. CreativeCommons, 2011.

[5] Debugging with GDB : In-process agent. `https://sourceware.org/gdb/current/onlinedocs/gdb/In_002dProcess-Agent.html`.

[6] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4) :317–329, November 2000.

[7] Nicholas Nethercote and Julian Seward. Valgrind : A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6) :89–100, June 2007.

[8] Valgrind manual. `http://valgrind.org/docs/manual/manual.html`.

[9] ptrace(2) : process trace - linux man page. `http://linux.die.net/man/2/ptrace`.

[10] Padala, Pradeep. Playing with ptrace, Part I | linux journal. `http://www.linuxjournal.com/article/6100`.

[11] Mathieu Desnoyers. *Low-impact operating system tracing*. PhD thesis, École Polytechnique de Montréal, 2009.

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

[13] `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`. accessed March 2016.

[14] `http://lists.lttng.org/pipermail/lttng-dev/2015-October/025151.html`. accessed March 2016.

[15] David J Wilder, Michael Holzheu, and Thomas R Zanussi. Driver tracing interface. In *Linux Symposium*, page 261. Citeseer, 2007.

[16] Tim Bird. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*, pages 47–54. Citeseer, 2009.

[17] Vara Prasad, William Cohen, F. C. Eigler, Martin Hunt, Jim Keniston, and J. Chen. Locating system problems using dynamic instrumentation. In *Proc. of 2005 Ottawa Linux Symposium*, page 49–64. Citeseer, 2005.

[18] Julien Desfossez. LTTng-UST vs SystemTap userspace tracing benchmarks. `https://sourceware.org/ml/systemtap/2011-q1/msg00244.html`.

[19] Mathieu Desnoyers. Re : LTTng-UST vs SystemTap userspace tracing benchmarks. `https://sourceware.org/ml/systemtap/2011-q1/msg00247.html`.

[20] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2) :375–382, February 2012.

[21] Suchakrapani Datt Sharma and Michel Dagenais. Scalable user space dynamic tracing performance on multi-core systems [Poster]. In *ACM Symposium on Operating Systems Principles*, 2013.

[22] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 41–49, New York, NY, USA, 2003. ACM.

[23] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown : Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4) :2 :1–2 :36, January 2008.

[24] M. Anton Ertl and David Gregg. 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, pages 403–412, London, UK, UK, 2001. Springer-Verlag.

[25] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

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

[27] Sun Just-In-Time (JIT) compiler (JDK 1.1 for Solaris Developer's Guide). `http://docs.oracle.com/cd/E19455-01/806-3461/ch1intro-3/index.html`.

[28] Mark Stoodley, Kenneth Ma, Marius Lut. Real-time java, part 2 : Comparing compilation techniques. *IBM developerWorks*, April 2007. `http://www.ibm.com/developerworks/java/library/j-rtj2/index.html`.

[29] Just-in-time compiler library : 5. building and compiling functions with the JIT. `http://www.gnu.org/software/dotgnu/libjit-doc/libjit_5.html`.

[30] Steven McCanne and Van Jacobson. 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, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[31] Mary L Bailey, Burra Gopal, Michael A Pagels, Larry L Peterson, and Prasenjit Sarkar. Pathfinder : A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 115–123. Citeseerx, 1994.

[32] Dawson R Engler and M Frans Kaashoek. Dpf : Fast, flexible message demultiplexing using dynamic code generation. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 53–59. ACM, 1996.

[33] Andrew Begel, Steven McCanne, and Susan L Graham. Bpf+ : Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 123–134. ACM, 1999.

[34] Zhenyu Wu, Mengjun Xie, and Haining Wang. Design and Implementation of a Fast Dynamic Packet Filter. *Networking, IEEE/ACM Transactions on*, 19(5) :1405–1419, Oct 2011.

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

[36] C. Boogerd and L. Moonen. On the use of data flow analysis in static profiling. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 79–88, Sept 2008.

[37] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2) :69–75, Mar 1995.

[38] Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, Stanford, CA, USA, 2006.

[39] Katerina Goseva-Popstojanova and Andrei Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.*, 68(C) :18–33, December 2015.

[40] Jongeun Lee and Aviral Shrivastava. A compiler optimization to reduce soft errors in register files. *SIGPLAN Not.*, 44(7) :41–49, June 2009.

[41] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4) :1319–1360, July 1994.

[42] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling : Where have all the cycles gone ? *ACM Trans. Comput. Syst.*, 15(4) :357–390, November 1997.

[43] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme : hardware support for instruction-level profiling on out-of-order processors. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 292–302, Dec 1997.

[44] Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes, and Wenmei W. Hmu. A hardware mechanism for dynamic extraction and relayout of program hot spots. *SIGARCH Comput. Archit. News*, 28(2) :59–70, May 2000.

[45] Kapil Vaswani, Matthew J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 217–228, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. Establishing a base of trust with performance counters for enterprise workloads. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 541–548, Santa Clara, CA, July 2015. USENIX Association.

[47] Georgios Bitzes and Andrzej Nowak. The overhead of profiling using pmu hardware counters. Technical report, CERN, openlab, 2014.

[48] J. R. Larus. Efficient program tracing. *Computer*, 26(5) :52–61, May 1993.

[49] Nicholas Nethercote and Julian Seward. Valgrind : A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6) :89–100, June 2007.

[50] Nicholas Nethercote and Alan Mycroft. Redux : A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2) :149 – 170, 2003.

[51] Josef Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing*, pages 93–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[52] Nathan Froyd, Nathan Tallent, John Mellor-Crummey, and Rob Fowler. Call path profiling for unmodified, optimized binaries. In *GCC Summit*, volume 6, pages 21–36. Citeseer, 2006.

[53] `http://ds.arm.com/ds-5/debug/dstream/`. accessed March 2016.

[54] `http://www.ghs.com/products/supertraceprobe.html`. accessed March 2016.

[55] Amrish K. Tewar, Albert R. Myers, and Aleksandar Milenković. mcftraptor : Toward unobtrusive on-the-fly control-flow tracing in multicores. *J. Syst. Archit.*, 61(10) :601–614, November 2015.

[56] Sriraman Tallam and Rajiv Gupta. Unified control flow and data dependence traces. *ACM Trans. Archit. Code Optim.*, 4(3), September 2007.

[57] P. A. Sandon, Y.-C. Liao, T. E. Cook, D. M. Schultz, and P. Martin-de Nicolas. Nstrace : A bus-driven instruction trace tool for powerpc microprocessors. *IBM J. Res. Dev.*, 41(3) :331–344, May 1997.

[58] B. Vermeulen. Functional debug techniques for embedded systems. *IEEE Design Test of Computers*, 25(3) :208–215, May 2008.

[59] A. Vasudevan, N. Qu, and A. Perrig. Xtrec : Secure real-time execution trace recording on commodity platforms. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–10, Jan 2011.

[60] Suchakrapani Sharma and Michel Dagenais. Hardware-assisted instruction profiling and latency detection. *The Journal of Engineering*, August 2016.

[61] Intel. *Intel Processor Trace*. Intel Press, Denver, CO, 2015. accessed March 2016.

[62] Craig Pedersen and Jeff Acampora. Intel code execution trace resources. *Intel Technology Journal*, 16(1) :130–136, 2012.

[63] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting hardware advances for software testing and debugging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 888–891, New York, NY, USA, 2011. ACM.

[64] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 68–78, Washington, DC, USA, 2015. IEEE Computer Society.

[65] Mustafa M. Tikir, Jeffrey K. Hollingsworth, and Guei-Yuan Lueh. Recompilation for debugging support in a jit-compiler. *SIGSOFT Softw. Eng. Notes*, 28(1) :10–17, November 2002.

[66] M. Koltachev, N. Khandelwal, and A. Gandhi. Debugging native code by transitioning from execution in native mode to execution in interpreted mode, Dec 2014. US Patent App. 13/911,108.

[67] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback : First fault diagnosis by reconstruction of distributed control flow. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 201–212, New York, NY, USA, 2005. ACM.

[68] Intel. About jit profiling api, 2016. Accessed Nov 2016.

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

[70] Martin Bligh, Mathieu Desnoyers, and Rebecca Schultz. Linux kernel debugging on google-sized clusters. In *Proc. of 2007 Ottawa Linux Symposium*, page 29–40. Kernel.org, 2007.

[71] Naser Ezzati Jivan. *Multi-Level Trace Abstraction, Linking and Display.* PhD thesis, École Polytechnique de Montréal, 2014.

[72] David Goulet. Unified Kernel/User-Space Efficient Linux Tracing Architecture. Master's thesis, École Polytechnique de Montréal, Avril 2012.

[73] J. Mogul, R. Rashid, and M. Accetta. The packer filter : An efficient mechanism for user-level network code. *SIGOPS Oper. Syst. Rev.*, 21(5) :39–51, November 1987.

[74] Louis Sobel. Secure Input Overlays : increasing security for sensitive data on Android. Master's thesis, Massachusetts Institute of Technology, Massachusetts, USA, 2015.

[75] Bart Jacob, Paul Larson, Breno Henrique Leitao, and Saulo Augusto M Martins da Silva. *SystemTap : Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems.* IBM Redpaper, 2009.

[76] Masami Hiramatsu. The Enhancement of Kernel Probing - Kprobes Jump Optimization. In *LinuxCon*, 2010.

[77] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin : A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education : Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[78] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX Annual Technical Conference*, pages 139–144, 2013.

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

[80] Mohamad Gebai, Francis Giraldeau, and MichelR Dagenais. Fine-grained preemption analysis for latency investigation across virtual machines. *Journal of Cloud Computing*, 3(1), 2014.

[81] Etienne M. Gagnon and Laurie J. Hendren. 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, pages 3–3, Berkeley, CA, USA, 2001. USENIX Association.

[82] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF). `https://www.kernel.org/doc/Documentation/networking/filter.txt`.

[83] Soumya Chakraborty. *Efficiency of LTTng as a Kernel and Userspace Tracer on Multicore*. VDM Verlag Dr. Müller, Saarbrücken, Germany, 2011.

[84] Octavian Lascu, Shawn Bodily, Matti Harvala, Anil K Singh, DoYoung Song, and Frans Van Den Berg. *IBM AIX Continuous Availability Features*. IBM Redpaper, 2008.

[85] Raphaël Beamonte and Michel R. Dagenais. Linux Low-Latency Tracing for Multicore Hard Real-Time Systems. *Advances in Computer Engineering*, 2015, August 2015.

[86] Pablo Neira-Ayuso, Rafael M. Gasca, and Laurent Lefevre. Communicating Between the Kernel and User-space in Linux Using Netlink Sockets. *Softw. Pract. Exper.*, 40(9) :797–810, August 2010.

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

[88] Paul E McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It ? *Linux Technology Center, IBM Beaverton*, 2015.

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

[90] T. Ball, S. Burckhardt, J. Halleux, M. Musuvathi, and S. Qadeer. Deconstructing concurrency heisenbugs. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 403–404, May 2009.

[91] Ieee nexus 5001. `http://nexus5001.org/`. accessed March 2016.

[92] `http://ds.arm.com/ds-5/`. accessed March 2016.

[93] `http://www.ghs.com/products/timemachine.html`. accessed March 2016.

[94] Andi Kleen. Adding Processor Trace Support to Linux. `http://lwn.net/Articles/648154/`. accessed March 2016.

[95] `https://github.com/andikleen/simple-pt`. accessed March 2016.

[96] `https://github.com/01org/processor-trace`. accessed March 2016.

[97] L. Baugh and C. Zilles. An analysis of i/o and syscalls in critical sections and their implications for transactional memory. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 54–62, April 2008.

[98] Philippe Moret, Walter Binder, Alex Villazón, Danilo Ansaloni, and Abbas Heydarnoori. Visualizing and exploring profiles with calling context ring charts. *Softw. Pract. Exper.*, 40(9) :825–847, August 2010.

[99] Andrea Adamoli and Matthias Hauswirth. Trevis : A context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 73–82, New York, NY, USA, 2010. ACM.

[100] `http://julialang.org/benchmarks/`. accessed March 2016.

[101] `http://www.openblas.net`. accessed March 2016.

[102] `http://lists.lttng.org/pipermail/lttng-dev/2015-October/025151.html`. accessed March 2016.

[103] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. Cloud monitoring : A survey. *Computer Networks*, 57(9) :2093 – 2115, 2013.

[104] Mathieu Desnoyers and Michel R. Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium) 2006*, pages 209–224, 2006.

[105] Matthew C. Merten, Andrew R. Trick, Erik M. Nystrom, Ronald D. Barnes, and Wenmei W. Hmu. A hardware mechanism for dynamic extraction and relayout of program hot spots. *SIGARCH Comput. Archit. News*, 28(2) :59–70, May 2000.

[106] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-directed optimizations in gcc with estimated edge profiles from hardware event sampling. In *Proceedings of GCC Summit 2008*, pages 87–102, 2008.

[107] Joerg Thalheim, Pramod Bhatotia, and Christof Fetzer. Inspector : Data Provenance using Intel Processor Trace (PT). In *proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[108] Mohamad Gebai, Francis Giraldeau, and Michel R Dagenais. Fine-grained preemption analysis for latency investigation across virtual machines. In *Journal of Cloud Computing*, pages 1–15, December 2014.

[109] D. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut. Perfcompass : Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds. *IEEE Transactions on Parallel and Distributed Systems*, PP(99) :1–1, 2015.

[110] A. Anand, M. Dhingra, J. Lakshmi, and S. K. Nandy. Resource usage monitoring for kvm based virtual machines. In *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*, pages 66–70, Dec 2012.

[111] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang. Vmon : Monitoring and quantifying virtual machine interference via hardware performance counter. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 399–408, July 2015.

[112] K. Kouame, N. Ezzati-Jivan, and M. R. Dagenais. A flexible data-driven approach for execution trace filtering. In *2015 IEEE International Congress on Big Data*, pages 698–703, June 2015.

[113] Hani Nemati and Michel Dagenais. Virtual cpu state detection and execution flow analysis by host tracing. *The 6th IEEE International Conference on Big Data and Cloud Computing (BDCloud 2016)*, October 2016.

[114] Jason Baron. Static keys, 2012. Accessed Nov 2016.

[115] Suchakrapani Datt Sharma, Hani Nemati, Geneviéve Bastien, and Michel Dagenais. Low overhead hardware-assisted virtual machine analysis and profiling. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, Dec 2016.

[116] Suchakrapani Datt Sharma and Michel Dagenais. Enhanced userspace and in-kernel trace filtering for production systems. *Journal of Computer Science and Technology*, 2016.

[117] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot : Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 169–182, New York, NY, USA, 2012. ACM.