Ostbayerische Technische Hochschule Regensburg

Laboratory for Digitalisation

Fakultät Informatik und Mathematik
Galgenbergstraße 32
93053 Regensburg

Comparative Analysis of Tracing Mechanisms in Linux for Real-Time Systems

Bachelor Thesis

Alexander Letaief
Student ID: 3313581
Email: alexander.letaief@st.oth-regensburg.de

Supervised by:
Prof. Dr. Wolfgang Mauerer (First Examiner)
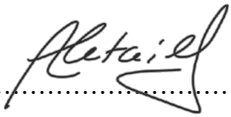Prof. Dr. Johannes Schildgen (Second Examiner)
Benno Bielmeier (Supervisor)
Dr.-Ing. Ralf Ramsauer (Supervisor)

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt gegenüber der Fakultät Informatik und Mathematik der Ostbayerischen Technischen Hochschule Regensburg, dass die vorliegende, dieser Erklärung angefügte Arbeit selbstständig und nur unter Zuhilfenahme der im Literaturverzeichnis genannten Quellen und Hilfsmittel angefertigt wurde. Alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind kenntlich gemacht. Ich reiche die Arbeit erstmals als Prüfungsleistung ein.

Regensburg, den 21.02.2025

....

Alexander Letaief

Abstract

Tracing mechanisms in real-time Linux systems enable detailed observation of application and kernel behavior while minimizing interference, aiming to capture system behavior as accurately as possible under production conditions with minimal observer effect. This thesis presents a comparative analysis of three widely used tracing frameworks, ftrace, LTTng, and eBPF, evaluating their usability, instrumentation workflow, features, performance impact, and suitability for tracing user-space applications. The study systematically measures the overhead introduced by each framework using a microbenchmark focused on User Statically-Defined Tracing (USDT). Additionally, the instrumentation workflow and feature set of each framework are examined to assess their practicality and integration usability. The impact on real-time performance is evaluated empirically using Cyclictest, the de facto standard benchmark for measuring real-time performance. The results indicate that LTTng has the least impact on real-time performance for user-space instrumentation due to its static instrumentation approach, while ftrace achieves the best stability and lowest overhead for kernel-space tracing. In contrast, eBPF introduces the highest performance impact in both kernel-space and user-space tracing. The study further highlights differences in usability, with ftrace offering the simplest setup, LTTng providing the most structured workflow for instrumenting user-space applications, and eBPF enabling advanced programming and scripting capabilities. The findings support developers and system designers in selecting a tracing framework that aligns with system constraints, enabling effective debugging and performance analysis while preserving real-time properties and ensuring accurate insights.

Contents

1 Introduction

1.1 Observability in Real-Time Systems

The increasing demands of real-time applications and the rise of mixed-criticality systems have driven interest in using Commercial-off-the-Shelf (COTS) hardware for real-time domains. This shift stems from the need to reduce costs and utilize readily available components while meeting stringent performance requirements. The Linux kernel has become a prominent solution, offering extensive hardware support and a mature programming environment. A notable advancement is the PREEMPT_RT patch, which improves the Linux kernel's predictability and responsiveness for real-time use cases [1]. As Steven Rostedt, a key contributor to PREEMPT_RT and maintainer of the ftrace framework, stated: "We actually would not push something up unless we thought it was ready" [2, 3]. After two decades of development, PREEMPT_RT has been integrated into the mainline Linux kernel for ARM64, RISC-V, and x86 architectures [4, 5]. This integration marks the growing maturity of real-time Linux for critical applications. Both the European Space Agency (ESA) and NASA have considered real-time Linux for space and ground systems [1].

The development of robust and reliable real-time systems presents significant challenges. History offers stark examples of the consequences of software defects in such systems. Between 1985 and 1987, the Therac-25 medical accelerator caused six incidents of massive radiation overdoses due to a software bug, resulting in severe injuries and fatalities [6]. A race condition in the Therac-25 software triggered X-ray emissions before the safety component, the flattener, was in position, exposing patients to hazardous levels of unfiltered radiation. Such failures could have been avoided with proper observability techniques to detect and address software defects during development. In the past decade, the development and availability of observability tools have expanded, particularly within the Linux ecosystem [7]. As a result, Linux has become a suitable platform for building and analyzing real-time systems, offering the necessary instrumentation to support safe and reliable operation.

1.2 Problem Statement and Motivation

Observing and analyzing systems introduces a fundamental challenge: the observer effect [8]. This phenomenon occurs when the observation process alters the system's characteristics, potentially causing Heisenbugs, elusive performance anomalies that shift or vanish during observation [9, 10]. In real-time applications, which operate under strict timing constraints, even minor perturbations can significantly affect performance. Identifying the root causes of latency in such systems is particularly difficult, as performance issues may stem from hardware limitations, kernel latency, scheduling delays, interrupt handling, or preemption mechanisms. To address these complexities, tracing frameworks provide diagnostic capabilities with the granularity needed to analyze the sources of unknown latencies [11]. However, these tools often introduce overhead, potentially masking the performance issues they are meant to diagnose. This creates a paradox where the diagnostic process itself disrupts accurate system analysis. The key challenge lies in developing tracing methodologies that capture meaningful observations while minimizing disruption to the system's natural behavior.

1.3 Research Scope, Objectives and Questions

This bachelor's thesis provides a detailed overview of tracing frameworks and tools within the Linux ecosystem, focusing on tracing real-time applications executed in user-space. The research explores the fundamental concepts of tracing and examines three well-established frameworks, ftrace, LTTng, and eBPF-based, with particular attention to user-space application tracing methods.

The primary objectives of this study are:

- To explain the core concepts of tracing in Linux systems

- To explore the user-space instrumentation process for each framework, presenting the integration options available to development teams for instrumenting their applications

- To evaluate the overhead introduced by the tracing routines of ftrace, LTTng, and eBPF

- To evaluate the overall impact of tracing on real-time systems

Based on these objectives, the following research questions have been formulated:

1. Which framework offers the highest usability for instrumenting an application?

2. Which framework provides the most extensive and effective features for user-space tracing?

3. Which framework introduces the lowest performance overhead when instrumenting a user-space application?

4. What effect does tracing have on real-time performance?

To address these questions, this study will offer a comparative analysis of the three selected tracing frameworks. The analysis will examine their instrumentation workflow, overhead implications, and overall impact on real-time performance. Empirical data will be collected through controlled experiments, utilizing benchmarking tools and custom-designed microbenchmarks to quantify performance characteristics. Additionally, the study will include a qualitative assessment of the frameworks' usability and feature sets, providing a detailed evaluation of their suitability for various observability requirements.

By investigating these aspects, this thesis aims to contribute findings to the field of Linux system tracing, supporting developers and system administrators in selecting the most suitable tracing framework while balancing functionality, performance, and ease of use.

1.4 Structure of the Thesis

Chapter 2 presents the theoretical foundations necessary for understanding the subject matter. Chapter 3 introduces the most relevant tracing tools and frameworks, followed by an analysis of three selected frameworks, focusing on their user-space application capabilities. This chapter also examines the impact of tracing on real-time system performance. Chapter 4 provides a detailed discussion of the differences identified between the frameworks. The thesis concludes in Chapter 5 with the main findings and a summary of the previous chapters.

2 Fundamentals

2.1 Observability Techniques in Computer Systems

Observability is the practice of understanding a system through observation, using tools such as counters, profilers, and tracers. Although tracing will be discussed in detail later, it is essential to distinguish observability tools from benchmarking tools, which actively alter the system's state. Observability tools can be classified based on their scope (system-wide or per-process) and their data collection method (counter-based or event-based) [7]. Event-based tools, such as profilers and tracers, capture dynamic system behavior, while counters, commonly used for monitoring, track specific metrics.

Applications and the kernel generate data on their state and activity, including operation counts, byte counts, latency measurements, resource utilization, and error rates. This data is typically exposed through integer variables known as counters, some of which are cumulative. Performance tools read these counters to compute statistics such as rates of change, averages, and percentiles. When these statistics are selected to evaluate or monitor a specific target, they become metrics [7]. Advanced monitoring software can also trigger alerts based on these metrics, notifying staff of potential issues.

While time-series metrics can sometimes identify the exact cause of a performance issue, such as a recent software or configuration change, they often provide only a general indication. In such cases, profiling or tracing tools become essential for deeper analysis. Profiling, within the context of system observability, typically involves sampling—collecting a subset of measurements to form an overview of the target system. CPUs are common profiling targets, often using timed-interval samples of on-CPU code paths [12]. Flame graphs, a powerful visualisation of CPU profiles, can identify various performance bottlenecks, including lock contention, memory inefficiencies, and networking delays, by analyzing CPU usage patterns.

2.2 Historical Background and Evolution of Tracing Technologies

With the increasing complexity of modern computer systems, performance and reliability often depend on a complex interaction of factors, including I/O subsystems, device drivers, interrupts, lock contention, scheduling, and memory management [13]. This complexity has driven significant growth in the development of tracing tools across the software stack over the past decade. While traditional debugging methods, such as printf() statements, offer basic insights, they lack the granularity and efficiency needed to diagnose performance bottlenecks in complex systems. Tracing, by contrast, provides a powerful approach for debugging and reverse-engineering intricate systems, offering a configurable and efficient methodology for detailed logging [14].

A significant milestone in the evolution of tracing was the introduction of DTrace by Sun Microsystems in 2005 [15]. This technology enabled dynamic tracing of both kernel and user-space environments without requiring system restarts or code modifications, an achievement nearly impossible at the time. DTrace's success extended beyond its technical capabilities. Its widespread adoption was driven by a well-supported ecosystem that included marketing, training resources, and user-friendly scripting tools. This contributed to an "observability gap" within the Linux ecosystem, as no comparable tool was available at the time [16].

This gap has led to the development of several tracing tools for Linux, including SystemTap, LTTng, and ftrace. A primary objective of these tools has been to minimize the overhead introduced by the tracing process, as excessive overhead can distort system behavior and cause elusive "Heisenbugs" [13, 17]. The introduction of eBPF (Extended Berkeley Packet Filter) in 2014 marked a major advancement in tracing technology [18]. eBPF enables developers to dynamically inject custom code into the kernel, unlocking new possibilities for high-performance networking, observability, and security tools.

2.3 Mechanics and Operation of Tracing Techniques

Tracing, at its core, involves recording events. This process captures essential event data, which is either stored for later analysis or processed immediately to generate summaries and trigger actions. Specialized tracing tools serve specific purposes, such as Linux's

strace for system calls and tcpdump for network packets [7]. General-purpose tools like SystemTap, ftrace, and bpftrace offer broader capabilities, enabling the analysis of various software and hardware events. These tools will be examined in detail later in this work. Tracing provides information about both the timing and location of events, identifying where they occur in the source code and which process is responsible.

An event is typically represented as an ordered tuple containing the event identifier, a timestamp indicating its occurrence, its location (often specified by node and thread identifiers), and an optional field for event-specific details [19]. This section will further examine the mechanics of tracing by defining key terms and concepts, offering a detailed overview of the tracing process.

2.3.1 Core Terminology of Tracing

Understanding the core terminology is essential for effectively utilizing tracing tools and frameworks. This section defines several key terms necessary for comprehending the mechanics and applications of tracing:

- **Tracepoint**: A tracepoint is a specific location in a program's code that allows probes to be attached. It acts as a hook, triggering probe execution when the program reaches that point. Tracepoints can be enabled or disabled dynamically, offering control over tracing activity [20, 7].

- **Probe**: A probe is a function or code segment associated with a tracepoint. When program execution reaches an active tracepoint, the corresponding probe is activated. Probes can perform tasks such as recording data, capturing timestamps, or modifying program behavior. They are generally designed to be lightweight and efficient to reduce their impact on the system under observation [14].

- **Event**: An event marks the occurrence of a specific action or condition within a program. In tracing, an event is commonly linked to the activation of a tracepoint. Events can represent various activities, including function calls, system calls, or context switches [21, 14]. Each event is usually recorded with a timestamp, offering an exact record of its occurrence [22].

- **Payload**: The payload refers to the data associated with an event. This data can provide essential context and information about the event's significance. For

example, the payload of a function call event may include the function's arguments and return value [22].

2.3.2 Callback Mechanism in Tracing

With the concept of tracepoints established, we now examine the mechanisms used to implement probe callbacks. First, it is essential to understand that instrumentation can follow two distinct approaches: static or dynamic [23, 14]. Static instrumentation embeds tracepoints directly into the binary at compile time, with predetermined locations. Dynamic instrumentation, on the other hand, inserts tracepoints at runtime based on user-defined locations. This distinction should not be confused with dynamic tracing, which refers to the ability to activate or deactivate tracing functionality during program execution, regardless of whether the instrumentation is static or dynamic.

A mechanism provides a conceptual model for implementing a callback, while the tracing infrastructure determines its specific implementation [14]. This relationship resembles that between an architectural blueprint and a completed structure: the blueprint represents the design (mechanism), while the structure (implementation) is the concrete realization of that design. For example, a common mechanism is the use of a trampoline, which allows code to be dynamically inserted at a tracepoint. The implementation of the trampoline varies depending on the tracing framework and target.

Before examining the underlying mechanisms, it is essential to differentiate between tracers and aggregators, as they represent distinct approaches to analyzing program behavior. Tracers follow a callback-serialize-write pattern: they trigger a probe when a tracepoint is reached, convert the event data into a storable format, and write this data to persistent storage [14]. A ring buffer is often used to temporarily hold events before writing them, allowing the tracer to manage incoming event streams efficiently and prevent data loss during bursts of activity. The resulting output is a trace, a detailed record of events for later analysis. Tracers are designed to minimize overhead, preserving system performance during observation.

Aggregators, in contrast, follow a callback-compute-update pattern. When a tracepoint is encountered, they trigger a probe, perform computations on the collected data, and update program state or variables accordingly. This approach prioritizes real-time analysis

and immediate action. Tools such as SystemTap and bpftrace exemplify the aggregator model, offering scripting capabilities to support custom aggregation logic [14].

Unlike tracers, which produce detailed event logs, aggregators generate condensed outputs tailored to specific needs. These outputs may include aggregated metrics or updated program variables [24].

Function Instrumentation

Function instrumentation, a type of static instrumentation, relies on compiler support. This method automatically inserts a call to a generic tracing probe before each function execution. The compiled binary embeds explicit calls to a designated routine at every function entry and, optionally, upon exit.

For example GCC provides various mechanisms for this callback, such as the -pg flag, which generates a binary where each function includes an mcount routine call as a preamble [25, 14].

```
echo 'main(){}' | gcc -x c -S -o - - -pg | grep mcount
# Output: "1: call *mcount@GOTPCREL(%rip)"
```

The implementation of this routine, which may include tracing, profiling, or other monitoring functions, is defined by the developer or tracer [26].

Static Tracepoints

A tracepoint, a form of static instrumentation, is manually inserted into application code by developers to enable the tracing of specific events [27]. In contrast to conventional function calls, tracepoint statements in the Linux kernel are optimized to minimize performance overhead. As these statements remain in the compiled binary unless explicitly disabled, their efficiency is critical, especially in production environments where tracing is generally inactive [14].

Several optimizations are employed to achieve this. First, the compiler is directed to position tracepoint instructions outside the cache lines used by the regular code execution path. This preserves the cache efficiency of the fast path, ensuring it remains unaffected

by inactive tracepoint code. Second, tracepoints are implemented as C macros rather than function calls. This approach reduces stack operations and eliminates function call overhead, further minimizing performance impact [24].

Although tracepoints are primarily used during development, their impact on runtime performance when disabled is limited to a single condition check. This minimal check ensures that the normal flow of execution remains efficient and uninterrupted. Despite these optimizations, a slight overhead persists, as the system must read the condition operand from memory.

To address this issue, the Linux Trace Toolkit Next Generation (LTTng) project introduced the Immediate Value infrastructure [14]. This mechanism embeds a constant directly within the instruction, eliminating the need for a memory read and improving tracepoint handling efficiency. By storing the tracepoint's enabled or disabled status directly in the instruction, Immediate Values remove memory access, enabling the CPU to check the status instantly. This optimization is especially valuable in high-frequency or real-time environments. As a result, it reduces overhead, making disabled tracepoints nearly invisible to the system.

Within the Linux kernel, tracepoints are implemented using the TRACE_EVENT macro, developed by Steven Rostedt. This macro simplifies the definition and insertion of tracepoints, allowing developers to use functions such as tracepoint_name() within their kernel code. Additionally, the TRACE_EVENT infrastructure provides a standardized interface for various kernel tracers to connect their probes. This standardization simplifies the development of custom tracers, promoting a more accessible and versatile tracing ecosystem [14, 24].

Trap

Trap-based instrumentation provides a method for dynamically instrumenting applications at runtime. This technique utilizes operating system support for traps to insert and execute custom probes at nearly any point within the kernel or application code.

In the Linux kernel, this mechanism is implemented through the Kprobes infrastructure. Kprobes use a trap-based approach to dynamically intercept kernel code execution [14]. When a Kprobe is registered at a specific instruction, the instruction is replaced with a breakpoint instruction. Upon reaching the breakpoint, the kernel's breakpoint handler

is invoked. The handler saves the application's state (including registers and stack) and transfers control to the Kprobes infrastructure, triggering the registered tracing probe.

After the probe executes and the trap is handled, the original instruction, previously replaced by the breakpoint, is executed, restoring normal control flow. Tracing infrastructures can be built on top of Kprobes, offering a dynamic alternative to manually inserting trace_tracepoint_name() statements in the code. Rather than modifying the source code, a Kprobe can be registered at the desired location during runtime. Tracers use this approach to attach their probes to Kprobes instead of relying on the TRACE_EVENT macro.

Abstracting the callback mechanism increases flexibility in probe management. By linking a tracer's probe to various backends, users gain more control over the tracing process. Although the resulting trace is functionally identical to one produced with the TRACE_EVENT macro, differences in the underlying callback mechanisms may affect performance.

When a Kprobe is unloaded, the breakpoint instruction is replaced with the original instruction, removing the instrumentation [24]. This dynamic approach enables temporary instrumentation and provides flexible tracing capabilities.

Trampoline

Trampolines provide a dynamic, jump-based method for patching or instrumenting applications at runtime [24]. This approach offers a lower-overhead alternative to trap-based mechanisms, although it involves a more complex implementation. Recent versions of the Linux kernel employ this technique to optimize registered Kprobes, replacing costly breakpoints with jump-based trampolines.

This optimization uses a detour buffer, referred to as the optimized region, to reduce overhead compared to the breakpoint approach. Instead of inserting a breakpoint instruction to trigger a trap, the original instruction is replaced with a direct jump to the optimized region. The jump-based mechanism starts by saving the CPU's registers to the stack. It then transfers control to a trampoline, which redirects execution to the user-defined probe.

Upon probe completion, the execution flow is restored. Control returns from the optimized region, the CPU registers are reloaded from the stack, and the original execution

path resumes. This technique effectively balances performance efficiency with the flexibility of dynamic instrumentation [14].

2.4 Definition and Characteristics of Real-Time Systems

In real-time systems, program correctness depends not only on the logical validity of its output but also on delivering results within predefined time constraints or deadlines. Temporal determinism is essential for ensuring predictable task execution [1]. Although throughput and low latency are desirable, they remain secondary to the system's ability to meet timing requirements. Consequently, real-time signifies "as fast as required", not necessarily "as fast as possible" [1].

Real-time systems are generally categorised into three types [28]:

- **Hard real-time systems:** In these systems, even a single missed deadline can lead to catastrophic failure. Consider an avionics weapons system: failing to launch a missile within the specified time window after pressing the button could have disastrous consequences.

- **Firm real-time systems:** These systems can tolerate a limited number of missed deadlines, but exceeding this threshold can lead to failure. For example, a navigation controller in an autonomous robot might tolerate occasional missed deadlines, causing minor deviations from the planned path, but repeated failures could lead to significant damage.

- **Soft real-time systems:** These systems tolerate occasional deadline misses without catastrophic consequences. Performance may degrade, but the system continues to function. For instance, in a console hockey game, occasional missed deadlines might cause choppy gameplay but will not result in complete failure.

Real-time systems are not simply 'fast' systems, and no universally accepted methodology governs their specification and design. The consequences of missing a deadline must be clearly defined [28]. Each real-time system presents distinct challenges and requires careful consideration of its specific timing requirements.

2.5 Real-Time Operating Systems (RTOS)

To address real-time demands, specialized operating systems known as Real-Time Operating Systems (RTOS) have been developed. These systems extend the functionality of general-purpose operating systems to handle real-time events effectively [29]. Unlike generic operating systems, which balance performance across metrics such as throughput, resource utilization, and responsiveness, RTOS focus on meeting task deadlines through scheduling policies that prioritize tasks with the most urgent timing requirements. Key features of an RTOS include minimal interrupt latency, ensuring rapid response to external events, and preemptive task scheduling, which allows high-priority tasks to interrupt lower-priority ones at any time [30]. These capabilities enable RTOS to meet strict deadlines in firm and hard real-time applications, making them essential in fields such as automation, communications, entertainment, and defense. While commercial RTOS solutions such as VxWorks, QNX, and LynxOS dominate the embedded systems market [31], Real-Time Linux offers a strong alternative. It combines real-time performance with Linux's robust ecosystem, extensive hardware support, cost efficiency, flexibility, and excellent development environment. This makes Real-Time Linux a well-suited choice for a wide range of industrial and embedded applications [32].

2.6 Related Work

Previous research has primarily focused on the performance characteristics of individual frameworks or conducted comparisons with a limited scope.

Gebai and Dagenais [14] analyzed kernel and user-space tracers, evaluating their design, implementation, and overhead. Their study employed microbenchmarks to quantify the performance costs of various tracing frameworks, identifying key sources of overhead. Although they compared various tracers, their focus was limited to tracing overhead and did not address the instrumentation workflow for User Statically-Defined Tracing and feature sets.

Piotrowski [33] compared the performance of DTrace on FreeBSD and eBPF on Linux, assessing their observability capabilities and overhead. The study employed both microbenchmarks and application benchmarks, including the dd workload and kernel compilation, to evaluate per-event costs and overall system impact. The results highlighted

trade-offs between static and dynamic instrumentation, demonstrating scenarios where DTrace outperformed eBPF. Despite these insights, the analysis focused on cross-OS comparisons rather than assessing the frameworks' suitability for real-time Linux.

Beamonte et al. [34] examined the impact of LTTng on real-time systems, assessing its applicability for tracing multicore Linux environments. Using cyclictest and hwlat-detector, they measured LTTng's effect on maximum latency, identified performance bottlenecks, and proposed modifications to LTTng-UST to reduce overhead. Their findings indicated that kernel-space tracing with LTTng introduced minimal overhead, while user-space tracing required improvements for deterministic real-time performance. Despite these findings, their analysis did not compare LTTng with other Linux tracing frameworks, such as eBPF-based or ftrace.

Bird [35] examined function duration measurement with ftrace, focusing on function graph tracing to capture function entry and exit timestamps. The study described the implementation of function graph tracing on ARM, addressing challenges such as stack manipulation, recursion prevention, and runtime filtering of short-duration functions. An optimized mechanism for discarding trace events was introduced, improving ftrace's ring buffer efficiency and extending trace coverage. Performance measurements indicated that function graph tracing incurs substantial overhead, particularly with active tracing. The study did not compare ftrace with other tracing frameworks, assess USDT, or evaluate its applicability to real-time Linux use cases.

Although these studies offer important perspectives on individual tracing frameworks, a comparison of their suitability for real-time systems—specifically for USDT tracing—remains absent. This thesis addresses this gap by systematically analyzing three well-known tracing technologies, ftrace, LTTng, and eBPF-based frameworks, under both isolated and realistic real-time conditions, while also evaluating their usability for instrumenting user-space applications

13

3 Comparison

3.1 Overview of Tracing Frameworks for Linux

A comprehensive comparison of Linux tracing tools and frameworks requires examining the most widely used solutions. These tools can be categorized along two primary axes: their domain of operation and their scope of functionality.

First, tracing frameworks are typically classified by their ability to collect metrics in either kernel or user-space. Kernel-space tracing tools operate within the OS kernel, capturing low-level events such as resource management and process scheduling. User-space tools, conversely, focus on monitoring applications running outside the privileged kernel-space, capturing their specific execution patterns.

Second, the scope and complexity of these observability tools must be considered. Tools like strace and tcpdump are lightweight utilities tailored for specific tasks, providing quick insights with minimal setup, making them ideal for focused debugging or profiling. In contrast, frameworks like DTrace, LTTng, and ftrace offer broader functionality, enabling real-time tracing across both kernel and user-space. These frameworks offer greater flexibility, allowing users to define custom probes, handle complex data, and monitor entire systems.

This section provides a concise overview of the most prevalent tracing tools and frameworks in Linux. Beginning with pioneering solutions like DTrace and moving to modern eBPF-based tools, we examine their core functionalities, underlying principles, and unique contributions to system observability.

DTrace

Introduced by Sun Microsystems in 2005, DTrace was the first widely recognized and adopted dynamic tracing tool, marking a significant advancement in the field. It enabled real-time system observation in both kernel and user-space environments without requiring system restarts or code modifications, a feat previously considered difficult or

impossible [15]. Initially developed for Solaris, DTrace was later ported to FreeBSD, macOS, and Linux, expanding its influence [36, 37, 38].

DTrace introduced the 'D' scripting language, inspired by AWK and C, with a focus on efficiency and minimalism. It encourages concise one-liners for quick system tracing without complex scripts. For example, the one-liner below counts read() system calls per executable:

```
dtrace -n 'syscall::read:entry { @[execname] = count(); }'
```

These one-liners became popular for their simplicity and ability to provide immediate insights into system performance [39].

Systemtap

Introduced by Red Hat in 2005, SystemTap is a tracing framework that provides real-time observability into both Linux kernel and user-space activity [40]. Although dynamic instrumentation capabilities like Kprobes were available in Linux since 2004, their complexity posed challenges for many developers. Sun's release of DTrace for Solaris in 2005, with its user-friendly interface and powerful tracing features, led Linux users to seek a comparable solution. SystemTap emerged as a strong contender, offering tracing capabilities similar to DTrace. In contrast to DTrace's widely praised simplicity, SystemTap initially faced criticism for its complexity and less intuitive user experience [41, 42]. Despite early challenges, SystemTap evolved into a flexible tool, offering real-time insights into system events. It features a scripting language inspired by DTrace's 'D', enabling users to define probes that are automatically translated into C code, compiled into a kernel module, and dynamically loaded into the running kernel. The example below performs the same function as the previous DTrace one-liner, counting read() system calls per executable:

```
stap -e 'global counts;
    probe syscall.read { counts[execname()] += 1 }
    probe end {
        foreach (name in counts) {
            printf("%s: %d\n", name, counts[name])
        }
    }'
```

Furthermore, SystemTap offers "tapsets", pre-written libraries of common probe functions, which simplify script creation and make observability accessible to a wider audience [43]. Managed by the stap command-line tool, SystemTap remains an efficient tool for tracing complex system behavior.

ftrace

Introduced in 2008 by Steven Rostedt, ftrace is a Linux kernel tracing framework that provides extensive insights into kernel functions [7]. It quickly became indispensable for developers seeking to understand and monitor kernel behavior [35]. Over time, ftrace evolved to include advanced features like function graph tracing and event tracing [7].

ftrace is managed through files in the debugfs pseudo-filesystem. These files configure various aspects of ftrace, including trace buffer size, event timestamp clock source, and the enabling or disabling of specific events. This flexibility enables ftrace to support function tracing, tracepoints, system call tracing, and dynamic instrumentation. ftrace can leverage both the TRACE_EVENT infrastructure for static instrumentation tracing and the Kprobe infrastructure for dynamically hooking into various parts of the kernel [14].

The development of tools like trace-cmd, a command-line utility for managing ftrace, has streamlined trace configuration and control, enhancing usability. Furthermore, KernelShark, a graphical tool for visualizing trace data, simplifies the analysis of complex kernel interactions through intuitive visualizations [44].

LTTng (Linux Trace Toolkit Next Generation)

Introduced in 2006 as an extension of the original Linux Trace Toolkit (LTT) [13], LTTng (Linux Trace Toolkit: Next Generation) is an open-source toolkit for tracing both the Linux kernel and user applications simultaneously. Unlike ftrace, which is built directly into the Linux kernel, LTTng relies on loadable kernel modules for kernel-space tracing and user-space components for instrumentation, control, and data collection. [14].

LTTng offers numerous features, including multiple recording sessions, dynamic event rule management, and efficient event filtering using custom expressions. Traces can be saved to the file system or streamed over a network, either entirely or selectively. LTTng supports multiple programming languages, including Python, Java, and C++, through user-space libraries that enable direct integration of tracing functionality into applications [22].

eBPF (Extended Berkeley Packet Filter)

A major advancement in system observability, eBPF enables the dynamic execution of code within kernel space. eBPF traces its origins to the BSD Packet Filter (BPF), first introduced in a 1993 paper by the Lawrence Berkeley National Laboratory. BPF was integrated into Linux in 1997 with kernel version 2.1.75.2 and was used in the tcpdump utility.

In 2014, BPF evolved into the extended Berkeley Packet Filter (eBPF). This evolution introduced several enhancements, including a significantly expanded instruction set, enhanced security measures, and improved performance. By 2016, eBPF had gained significant traction, with its adoption in production systems and the rise of influential figures like Brendan Gregg, whose work on tracing at Netflix significantly popularized eBPF within infrastructure and operations circles [45]. eBPF allows developers to write custom code dynamically loaded into the kernel, enabling real-time modifications to its behavior. This capability enabled a new generation of high-performance tools in networking, monitoring, and security [18].

Enhancing its accessibility, eBPF includes bpftrace, a command-line tool designed for ease of use. As described in the project's README, "bpftrace is a high-level tracing language for Linux eBPF, inspired by awk and C, and predecessor tracers such as DTrace and SystemTap." bpftrace translates high-level programs into eBPF kernel code and outputs formatted results in the terminal [46].

Perf

Perf, the official Linux profiling tool, is integrated into the mainline kernel. This multifaceted tool provides profiling, tracing, and scripting functions, serving as the front-end to the kernel's perf_events observability subsystem. Initially focused on performance monitoring counters, perf now supports event-based tracing sources. Its capabilities in CPU analysis are particularly noteworthy [7].

perf's typical use case differs from that of ftrace or LTTng. While it can interface with the kernel's tracepoint infrastructure and record tracepoints, including system calls, perf is primarily used for sampling and profiling applications. Additionally, perf monitors only a single process at a time. Perf reports only the events and counters that occurred within the traced process. This makes perf particularly effective for analyzing a specific program's behavior [14].

strace

Originally developed by Paul Kranenburg in 1991 for Sun Systems [47], strace is an open-source Linux utility used for diagnostics, debugging, and instruction in user space. It traces interactions between processes and the Linux kernel, including system calls, signal deliveries, and process state changes.

strace uses the kernel's ptrace feature to hook into a process and intercept all its system calls along with their arguments. The trace data is then written to a file descriptor for later analysis. Because strace relies on ptrace and incurs scheduling overhead from managing multiple processes, it typically introduces significant performance overhead [14].

System administrators, diagnosticians, and troubleshooters find strace particularly valuable for diagnosing issues in programs without access to source code. Its ability to trace programs without requiring recompilation makes it invaluable for analyzing program behavior [48].

Sysdig

Sysdig, initially developed as a traffic-monitoring system, integrates a kernel module to capture all network traffic, particularly between containers. Its support for Lua-based filters, known as "chisels", provides extensive flexibility for network traffic analysis and custom system monitoring, offering capabilities similar to eBPF and SystemTap [14]. Beyond network monitoring, Sysdig instruments systems at the OS level by capturing system calls and other OS events [49]. This functionality has made it a widely used commercial tool for monitoring diverse environments, from containers to web services [50, 51].

3.2 Evaluation of USDT-Based Instrumentation

Testing real-time systems in environments that closely resemble real-world conditions is essential to obtaining accurate insights, as discussed in Chapter 1. To address this challenge while enabling precise analysis, static code instrumentation concepts emerge as an effective solution. Specifically, the use of User Statically-defined Tracing (USDT) offers substantially reduced overhead compared to conventional logging approaches [14]. This instrumentation methodology enables the integration of efficient logging mechanisms

while maintaining the integrity of time-critical operations. The minimal performance impact of USDT allows for meaningful testing that accurately reflects production environment behavior.

The subsequent section examines the underlying mechanics of USDT functionality. Multiple established frameworks provide USDT support, among which this analysis focuses on three distinct tracing frameworks: ftrace, LTTng, and eBPF-based. The selection of ftrace is justified by its position as one of Linux's pioneering comprehensive tracing frameworks, alongside LTTng, and its robust contemporary community support. eBPF, as a modern tracing solution, offers exceptional adaptability and sophisticated capabilities, making it particularly relevant for contemporary tracing applications [18]. Our focus will not only be on measuring overhead but also on identifying the strengths and weaknesses of these tracing methods in terms of their instrumentation and feature sets. It is equally important to explore the level of effort required for development teams to implement and utilize these tools effectively.

3.2.1 User Statically-Defined Tracing: Concept and Integration

User Statically-Defined Tracing represents a mechanism for adding static tracing capabilities to user-space applications. As previously discussed in Chapter 2, the implementation of tracing mechanisms varies across different tracing frameworks, allowing for diverse design approaches to User Statically-Defined Tracing (USDTs). A critical distinction exists between tracepoint insertion in the application's source code and the implementation of the actual probe attachment to this tracepoint.

A well-known dynamically probe-attached USDT implementation utilizes the trap mechanism, where tracepoints are embedded as metadata within the binary during compilation rather than being compiled into executable instructions. When tracing is enabled, the system dynamically inserts a trap instruction, typically implemented as a breakpoint, at each USDT location. This trap temporarily halts the normal program flow and redirects execution to a handler routine provided by the tracing framework. To achieve this, the tracing framework introspects the ELF section of the binary to identify the location of the tracepoint. A breakpoint is then placed at the tracepoint's marker, which translates into an interrupt. When program execution reaches this marker, the interrupt handler is triggered, invoking the kernel's uprobe mechanism. The kernel processes the event and broadcasts it to user-space for further handling [52, 53]. Handler

routines are designed to efficiently collect relevant data with minimal execution overhead. Once the required information is gathered, control promptly returns to the main program flow, ensuring normal execution resumes with minimal delay.

The core advantage of USDT is their negligible performance impact when inactive. In their dormant state, USDT impose virtually no overhead on program execution. When disabled, system performance is nearly identical to that of an uninstrumented binary. Once activated, USDT provide an efficient mechanism for targeted data collection at strategic locations. This combination of low inactive overhead and precise active tracing makes them particularly well-suited for production-near environments where performance minimization is critical [54].

To leverage USDTs in practice, we now explore the integration workflow for developers using ftrace, eBPF, and LTTng.

3.2.2 User Statically-Defined Tracing Instrumentation Workflow Across Frameworks

ftrace

ftrace can be used to trace statically instrumented applications through the use of its uprobe capabilities [55]. To implement static instrumentation in applications, you first need to add tracepoints to the application. The "sys/sdt.h" header file, which is part of the systemtap-sdt-devel package, is required.

Therefore, statically coded instrumentation is achieved using DTRACE_PROBEn macros, such as DTRACE_PROBE1(provider, event, arg1), where the macro specifies the provider, event name, and up to 12 arguments that define the tracepoint parameters [56]. During compilation, USDTs are translated into NOP (No Operation) instructions, while the associated metadata is stored in the .note.stapsdt section of the ELF file.

Locating the memory addresses of the tracepoints in the .note.stapsdt section of the ELF file requires manual inspection or the use of a script to automate the process. Tools such as readelf or objdump can help extract this information. Once the memory addresses are identified, the actual probe attachment is implemented using uprobe, which can be activated using ftrace's debugFS interface. After activation, ftrace monitors the provided address and allows you to extract specific arguments by accessing the relevant register

contents. The exact registers used for argument passing must also be determined from the ELF file [57].

eBPF

The use of eBPF shares the same prerequisites as ftrace. Applications must also be instrumented using the DTRACE_PROBEn macros provided in the "sys/sdt.h" header file. Unlike ftrace, the activation of the tracing mechanism with eBPF involves significantly less effort. bpftrace, a tracing tool built on top of eBPF, simplifies this process by automatically detecting all tracepoints from the application's binary, making it well-suited for instrumenting user-space applications [53]. Unlike a tracing framework, eBPF is a technology that enables the dynamic insertion of code into kernel space at runtime, providing the necessary infrastructure for bpftrace. Since bpftrace utilizes eBPF for dynamic instrumentation, and eBPF provides the foundation for its functionality, we will use the terms eBPF and bpftrace synonymously for simplicity in this context.

A major advantage of bpftrace is its adherence to the callback, compute, update pattern, as discussed in Chapter 2, categorizing it as an aggregator. This enables real-time evaluations based on tracepoint information, making it particularly effective for live analysis. Alternatively, the BPF Compiler Collection (BCC) can be used instead of bpftrace to implement a custom instrumentation solution. While BCC provides finer control over tracepoint handling and data processing, it requires more complex implementation compared to bpftrace.

The following script demonstrates how to capture user-space tracepoints with nanosecond precision, access provided arguments, and aggregate executions. Its structure closely resembles awk, with distinct BEGIN, main processing, and END blocks, making it intuitive for users familiar with text processing tools.

It initializes a counter in BEGIN, increments it on each tracepoint hit, and prints execution details, including process ID, CPU ID, timestamp, and arguments. In END, it reports the total number of probe hits.

```
#!/usr/bin/env bpftrace

BEGIN {
    @count = 0;
}
```

```
usdt:/opt/trace/tests/main:* {
        printf("%u [%u] %u %d\n",
                pid,
                cpu,
                nsecs,
                arg0
            );
        @count++;
    }


END {
    printf("Total USDT probe hits: %d\n", @count);
}
```

LTTng

LTTng provides a more flexible instrumentation mechanism compared to ftrace and eBPF by allowing the creation of custom tracepoint providers using C macros. This flexibility enables developers to tailor tracing functionality to their specific needs, offering advanced options for passing arguments from the application to the tracing system at runtime. Complex data structures can be transmitted as payloads, facilitating more detailed analysis in the resulting trace logs [58].

For simpler tracing scenarios, LTTng offers an alternative via the "lttng/tracef.h" header This functionality mirrors the simplicity of ftrace and eBPF, allowing developers to quickly implement basic trace logs. For example:

```
lttng_ust_tracef("my message: %s (%d)", my_string, my_integer);
```

To leverage LTTng's features, applications have to be dynamically linked against the lttng-ust library.

LTTng also includes a command-line tool for managing tracing sessions and enabling tracepoints. For instance, developers can activate tracepoints in user-space applications with commands such as:

```
lttng enable-event --userspace 'lttng_ust_tracef:*'
```

3.3 Assessment of Tracing Framework Overhead

With the options for instrumenting user-space applications now outlined, the following sections focus on evaluating the overhead introduced by instrumentation and tracing on an application's runtime behavior, as well as identifying the mechanisms that minimize this impact.

3.3.1 Methodology for Measuring Tracing Overhead

The idea is to test a simple application under different conditions using each tracing mechanism. The goal is to measure the time with nanosecond accuracy from the start of the routine until it returns to the program flow. This time will serve as a measure of how much time is consumed by a tracepoint hook.

Several scenarios are of interest:

Baseline (Case 0): Measure execution time when no tracing mechanism is activated and the application is not instrumented. This will serve as the baseline for comparison.

Full Tracing Implementation (Case 1): Measure the performance impact when the application is instrumented and the tracing mechanism is active while storing all logs produced by the application for later analysis.

Inactive Tracing (Case 2): Measure the performance impact when the application is instrumented and the tracing mechanism is active, but no trace routines are triggered. This tests the impact of the full tracing implementation without any actual tracing activity.

Instrumentation Impact Alone (Case 3): Measure the performance impact when the application is instrumented but the tracing mechanism is inactive, i.e., the tracer is disabled. This isolates the effect of the instrumentation code itself.

3.3.2 Microbenchmark Design and Execution Environment

For the baseline (Case 0), the cpu_relax function was implemented to insert a null assembly instruction, ensuring that the execution is not optimized away by the compiler.

```
static inline void cpu_relax() {
    asm("" ::: "memory");
}
```

For the microbenchmark, a C program was designed that, depending on whether ftrace, eBPF, or LTTng is used, provides a corresponding function to trigger a simple trace event.

```
static inline void event() {
    #ifdef USE_LTTNG
        lttng_ust_tracef("%s (%d)", "event", 0); // LTTng (Case 1, Case 3)
    #elif defined(USE_DTRACE)
        DTRACE_PROBE1(workload, event, 0); // ftrace and eBPF (Case 1, Case 3)
    #else
        cpu_relax(); // Case 2
    #endif
}
```

Depending on the previously described cases, the execution time of the routine is measured using the corresponding mechanism:

```
static inline void run() {
    #ifdef TRACING_ON
        event();
    #else
        cpu_relax(); // (Case 0)
    #endif
}
```

```
// In Main
/*
In Case 2, tracepoints were defined elsewhere in
the code to serve as attachment point for the tracer
*/
while (runs--) {
    ticks start_time = getticks();
    run();
    ticks end_time = getticks();
}
```

The microbenchmark was executed on hardware with an Intel® Core™ i7-1065G7 CPU @ 1.30 GHz, capable of turbo boosting up to 3.9 GHz, running Ubuntu 24.04.1 LTS with the latest available kernel (6.12.11) and the RT preemption model. To ensure stable and repeatable benchmarking results, several system modifications were applied, some of which are also recommended by Denis Bakhvalov, a senior developer at Intel [59].

Disabling Turbo Boost (Dynamic Frequency Scaling)

Turbo Boost is a feature in Intel CPUs that dynamically increases the clock speed to improve performance during demanding tasks. This can lead to inconsistent benchmark results, as the CPU frequency might vary throughout the test. To avoid this, Turbo Boost is disabled to ensure that the CPU operates at a consistent frequency. This is achieved by writing 1 to the no_turbo file in the Intel P-state directory, as shown below:

```
echo 1 | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo
```

Setting the CPU Scaling Governor to "Performance"

The Linux kernel uses a CPU scaling governor to adjust the processor's frequency based on workload, with the "performance" governor keeping the CPU at its maximum frequency. In power-saving modes, the CPU frequency can be reduced, which might affect benchmark results. To prevent any reduction in performance, the CPU scaling governor was set to "performance" across all cores using the following commands:

```
for i in /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor; do
    echo performance > $i
done
```

Clearing File System Cache

The Linux kernel caches file system contents in memory to reduce disk access times. This can distort benchmark results by causing file operations to be faster than they would be if the data had to be read from the disk. To eliminate the influence of file system caching, the cache was cleared by writing 3 to the drop_caches file and executing the sync command to ensure all changes were written to disk:

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
sync
```

Disabling Hyperthreading

Hyperthreading, Intel's implementation of Simultaneous Multi-Threading (SMT), can introduce performance variability during benchmarking by allowing two logical threads to share a single physical core's resources [60]. This resource sharing can lead to inconsistent performance measurements in certain scenarios. To achieve more stable and reproducible core performance, Hyperthreading has been disabled through the BIOS settings.

CPU Isolation

To minimize scheduler-induced disturbances and ensure consistent benchmarking conditions, CPU isolation was enforced by configuring the kernel boot parameters. The system was booted with GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=3 nohz_full=3", which prevents the Linux scheduler from migrating tasks onto CPU 3. This isolation minimizes latency spikes and prevents interference from background processes during benchmarking

3.3.3 Results and Analysis of Microbenchmark Experiments

Case	None				ftrace			
	min	max	$\bar{x}$	s	min	max	$\bar{x}$	s
Case 0	13	5189	15.55	12.27	-	-	-	-
Case 1	-	-	-	-	506	58 365	607.26	889.06
Case 2	-	-	-	-	13	5180	15.62	9.59
Case 3	-	-	-	-	13	5102	15.66	8.90

Case	eBPF				LTTng			
	min	max	$\bar{x}$	s	min	max	$\bar{x}$	s
Case 0	-	-	-	-	-	-	-	-
Case 1	450	58 709	564.67	1446.99	171	22 404	185.75	43.27
Case 2	12	32 926	15.60	17.15	13	5185	15.52	10.26
Case 3	12	39 352	15.70	22.11	12	7053	15.83	11.77

Table 3.1: **Comparison of routine duration across tracing frameworks.** The table presents the minimum, maximum, mean ($\bar{x}$), and standard deviation (s) of routine execution times (in nanoseconds) for different cases under four configurations: no tracing, ftrace, eBPF, and LTTng.

Table 3.1 presents the measured execution times for user-space instrumentation in nanoseconds. Measurements were performed over 10 million iterations. As shown in Figure 3.1 and Figure 3.2, the distribution of these measurements reveals several key insights about the performance characteristics of different tracing frameworks.
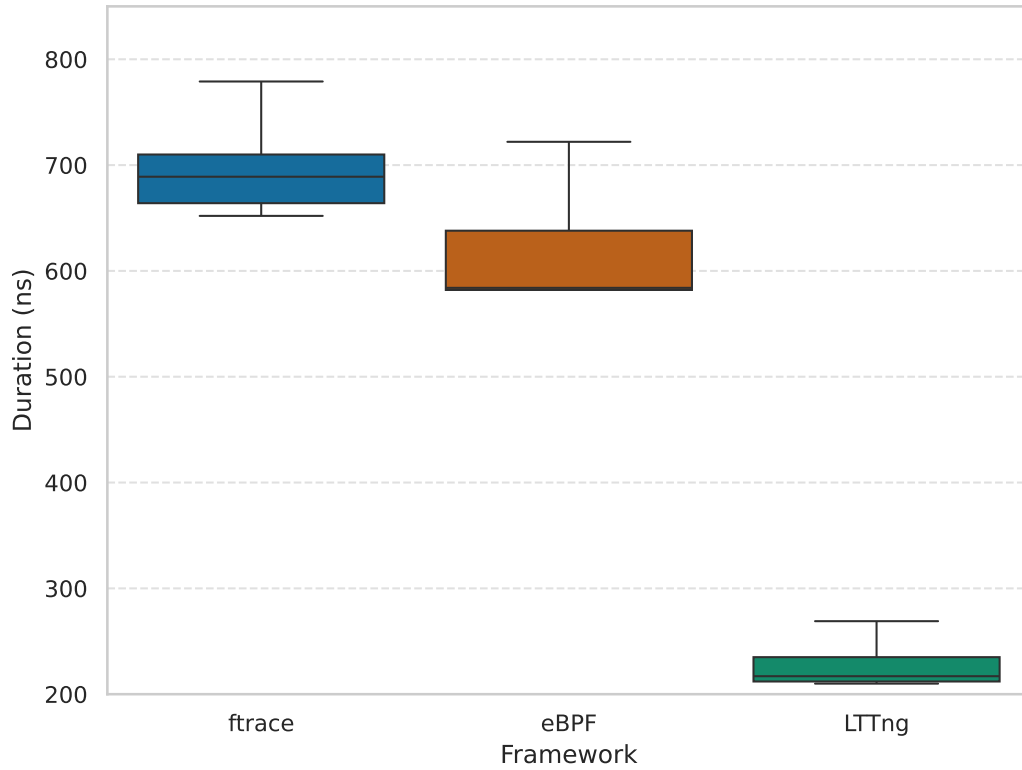
Figure 3.1: **Probe duration across frameworks (Case 1).** The box plot compares probe durations for ftrace, eBPF, and LTTng, highlighting differences in execution time and variability.

LTTng demonstrates notably different behavior compared to ftrace and eBPF, particularly in Case 1, where tracing is actively performed. The box plot in Figure 3.1 shows that LTTng's execution time is concentrated in a narrower range (approximately 210–230 nanoseconds), indicating a more compact distribution. In contrast, ftrace and eBPF show higher median execution times, clustering around 590–700 nanoseconds.
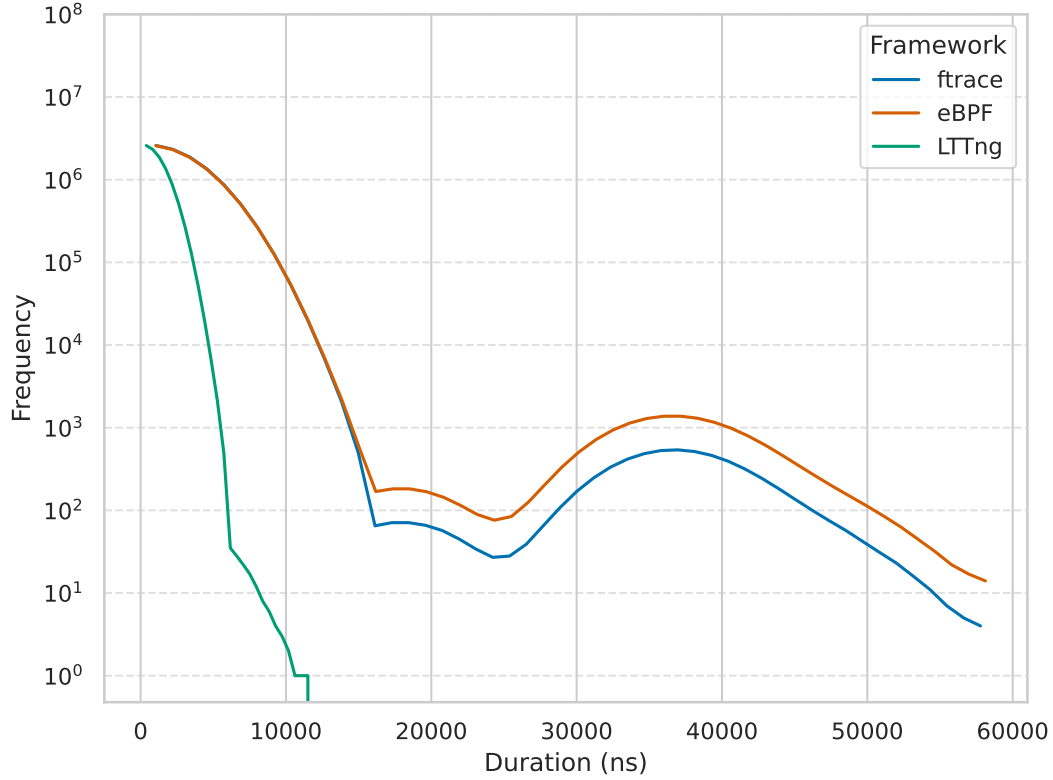
Figure 3.2: **Probe duration frequency distribution (Case 1).** The frequency polygon represents the spread of probe durations across frameworks, showing how frequently specific duration values occur.

The frequency polygon in Figure 3.2 presents the distribution of USDT execution times across the three frameworks. eBPF exhibits a broader spread of execution times, with higher frequencies in the mid-to-upper range, suggesting a greater performance impact for some probes. ftrace maintains a more consistent distribution, closely mirroring eBPF in some regions but with fewer spikes in execution time. LTTng, in contrast, consistently records lower execution times, maintaining a tighter and more stable distribution with minimal variation. This indicates that LTTng imposes the least overhead and provides the highest stability for user-space tracing.

Cases 2 and 3, where tracing is disabled or partially active, exhibit remarkably similar performance across all frameworks, with execution times comparable to the baseline Case 0. Only eBPF showed a single maximum latency spike of 32 µs in Case 2 and two spikes of 32 µs and 39 µs in Case 3. This indicates that the overhead of inactive tracepoints is

minimal to negligible, regardless of the framework. The box plots for these cases show nearly identical shapes and distributions, reflecting consistent behavior across different scenarios where tracing is inactive.

These observations indicate that while ftrace and eBPF exhibit similar performance characteristics due to their shared underlying instrumentation approach, LTTng utilizes a more efficient mechanism for active tracing, leading to significantly lower overhead and more stable tracing behavior.

3.4 Benchmarking the Impact of Tracing on Real-Time Performance

Cyclictest, the most widely used benchmark for evaluating real-time system latency, was used to measure the impact of tracing. [61]. It provides a precise mechanism for measuring system responsiveness with nanosecond accuracy by calculating the time difference between a thread's expected and actual wakeup time. The test utilizes an advanced measurement approach involving a non-real-time master thread that creates multiple measurement threads with real-time priority. These measurement threads are periodically awakened by a cyclic timer, and the difference between their defined and actual wake-up times is calculated and passed to the master thread via shared memory [62]. One key advantage of Cyclictest is its integration with ftrace [63]. This integration allows users to define upper latency limits, automatically stop the test when these thresholds are exceeded, and generate a trace marker. This capability enables root cause investigation by capturing and analyzing surrounding events.

3.4.1 Experimental Setup: Configuration of Cyclictest

For the following measurements, Cyclictest from the rt-tests package (v2.80) was used. The test was conducted using the following configuration:

```
./cyclictest \
    --affinity=3 \
    --mainaffinity=2 \
    --threads=1 \
```

```
--mlockall \
--loops=10000000 \
--histogram=10000000 \
--distance=0 \
--nsecs \
--quiet \
--priority=99 \
--interval=30 \
--json="$RESULT_DIR/$1"
```

This setup ensures that a single real-time measurement thread is bound to a dedicated CPU (CPU 3), while the main computation thread executes on a separate CPU (CPU 2). The test runs at a fixed interval of 30 µs for 10 million iterations.

For all tests, the microbenchmark environment was used, following best practices from the Linux Foundation to ensure reliable and stable execution [64]. The main computation thread was explicitly assigned to a separate CPU to prevent resource contention between the real-time measurement process and background system operations. Additionally, to eliminate potential delays from memory paging, the –mlockall option was enabled to lock all current and future memory allocations in RAM. This prevents memory pages from being swapped to disk, ensuring consistent execution latency. Cyclictest results are stored in a JSON-formatted latency histogram, forming the basis for subsequent analysis.

3.4.2 Kernel and User-space Instrumentation of Cyclictest

To evaluate the impact of tracing on real-time performance, both user-space and kernel-space instrumentation in Cyclictest were implemented. For user-space instrumentation, Cyclictest's source code was modified to insert a tracepoint between the expected and actual timing measurements. This modification ensured that a tracepoint was triggered for each latency calculation and recorded in the trace buffer. In contrast, kernel-space instrumentation leveraged the sys_exit_clock_nanosleep system call as a tracing hook to capture each wake-up event in the test cycle. This approach provided insights into Cyclictest's real-time behavior without modifying its source code.

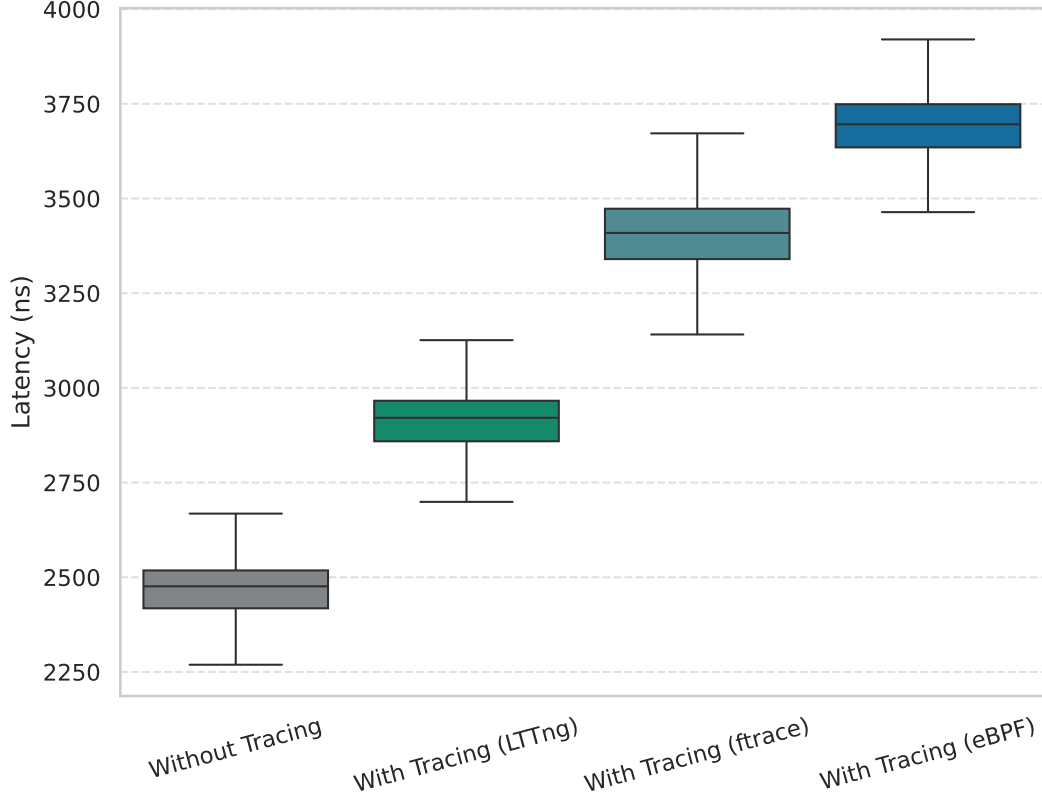### 3.4.3 Results of Kernel and User-space Cyclictest Benchmark

Figure 3.3: **Comparison of latencies with and without tracing.** The figure illustrates the impact of tracing on latency. The box plot presents the latency distributions for configurations without tracing, with static instrumentation (LTTng), and with dynamic instrumentation (ftrace and eBPF).

Figure 3.3 presents the latency for USDT tracing, showing that static instrumentation (LTTng) incurs less latency than dynamic instrumentation (ftrace and eBPF). Both static and dynamic tracing increase system latency, with dynamic tracing exhibiting slightly greater variability. While eBPF and ftrace rely on the same dynamic instrumentation method via uprobes, eBPF results in a higher median latency.
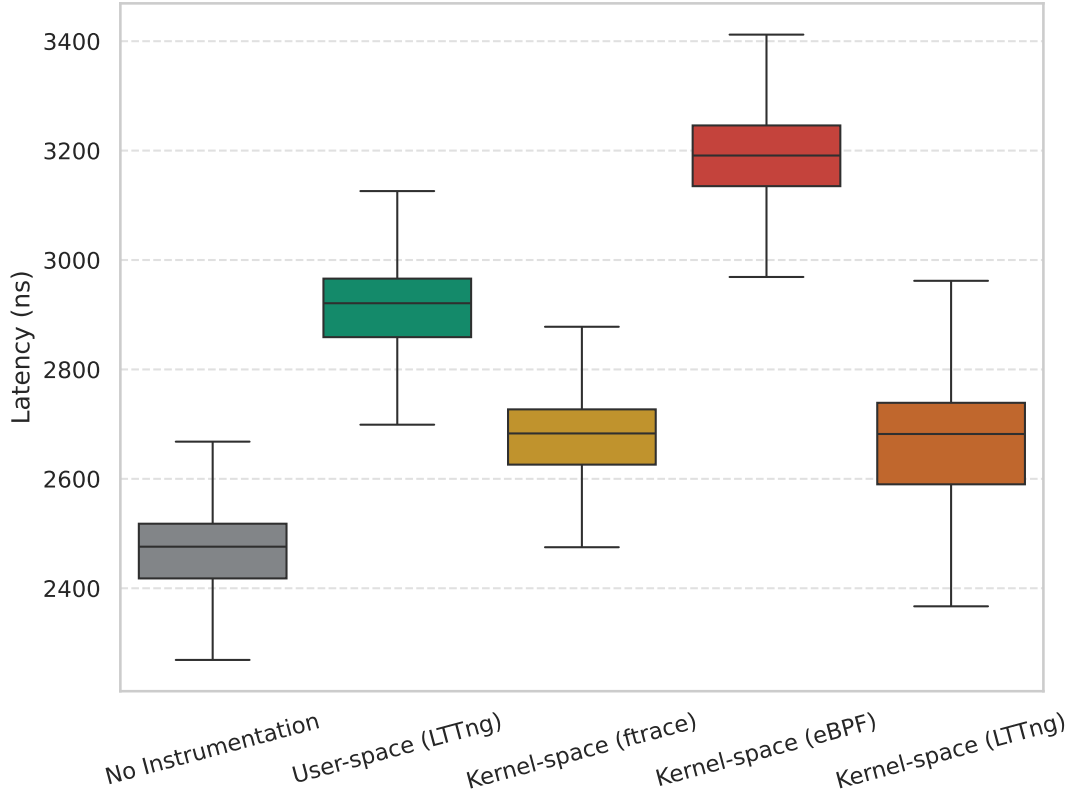
Figure 3.4: **Latency comparison of user-space and kernel-space instrumentation.** The box plot shows the latency distributions for configurations without instrumentation, with user-space instrumentation using LTTng, and with kernel-space instrumentation using ftrace, eBPF, or LTTng.

Figure 3.4 compares the latency impact of user-space and kernel-space instrumentation, with LTTng as the reference for user-space tracing. Tracing the sys_exit_clock_nanosleep system call in kernel-space results in lower latency with ftrace or LTTng compared to user-space instrumentation with LTTng. Among kernel-space methods, ftrace and LTTng have similar median latencies, but LTTng exhibits greater variability in its latency distribution. In contrast, eBPF introduces a higher latency impact than both ftrace and LTTng in kernel-space, as well as LTTng in user-space.

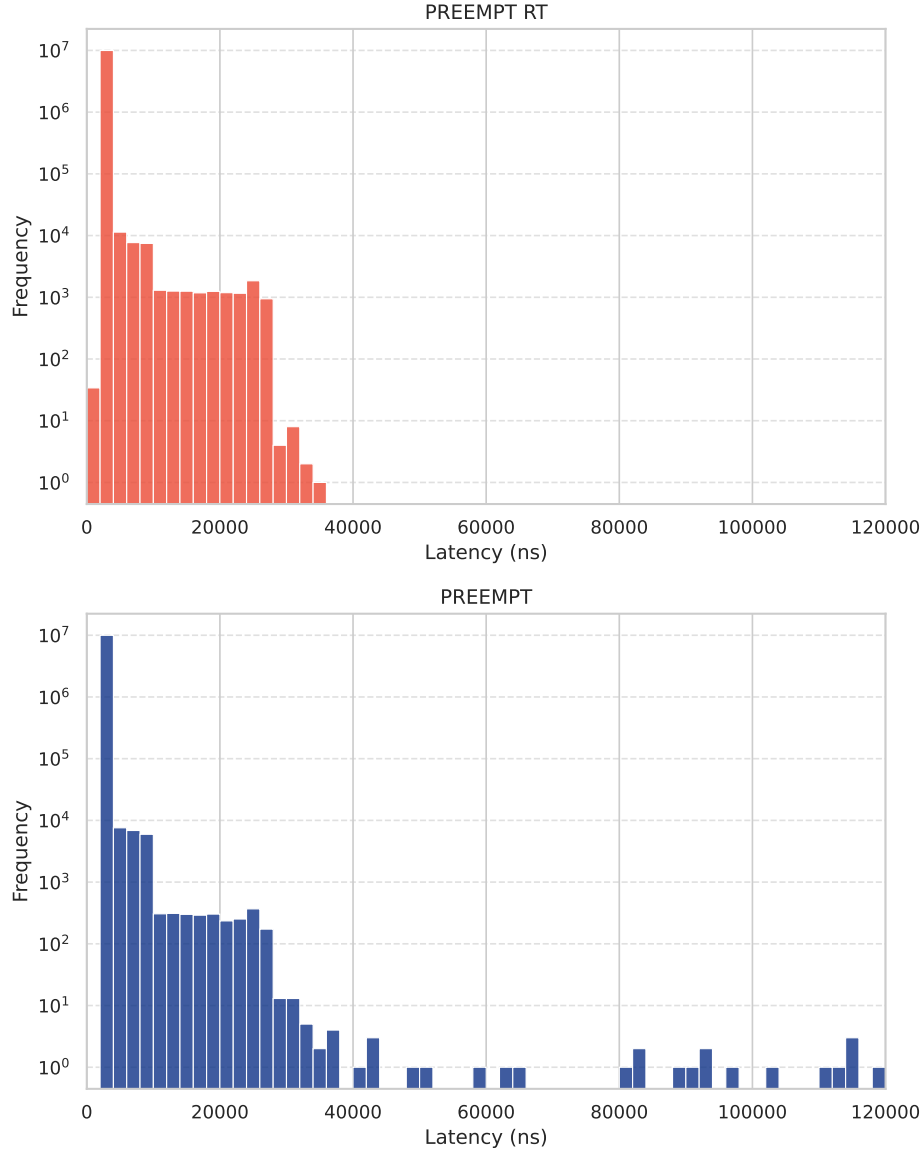### 3.4.4 Impact of Preemption Models on Real-Time Performance

Figure 3.5: **Latency distribution comparison between PREEMPT_RT and PREEMPT.** The histrogram illustrates the distribution of latencies under the PREEMPT and PREEMPT_RT configurations, highlighting differences in tracing stability.

This experiment evaluates Cyclictest execution with user-space instrumentation under the PREEMPT_RT and PREEMPT models to validate the real-time capabilities of PREEMPT_RT. LTTng was selected for this analysis as it demonstrated the most stable user-space performance among the evaluated tracing frameworks. The PREEMPT

model allows kernel task preemption to improve responsiveness, while PREEMPT_RT transforms the kernel into a fully preemptible environment for real-time systems. The visualization shows that both models exhibit similar latency patterns up to 30 µs. Beyond this point, PREEMPT_RT results in fewer high-latency occurrences, with most latencies remaining below this threshold and a maximum latency of 34 µs. In contrast, the PREEMPT model continues to show latency variations, reaching nearly 120 µs, with more than 40 occurrences exceeding 35 µs recorded in 10 million cycles, while PREEMPT_RT experiences a sharp decline in frequency beyond 25 µs.

4 Discussion

4.1 Evaluation and Discussion of Framework Comparison Results

The comparative analysis of ftrace, eBPF, and LTTng highlights significant differences in usability, features, and performance characteristics. This section synthesizes these findings to provide a clear understanding of each framework's strengths and limitations in user-space tracing.

4.1.1 Discussion on Usability and Features

Setup and Configuration

ftrace, integrated into the Linux kernel, offers the most straightforward setup among the three. It requires no separate installation, as the Linux kernel must simply be configured with the appropriate kernel options enabled [65]. Users interact with ftrace through the debugFS filesystem, where they configure tracing parameters and access trace outputs. For a more user-friendly interface, the trace-cmd CLI tool serves as a frontend to ftrace. It simplifies operations such as starting tracing sessions and listing available events but provides limited configuration options. eBPF, while also a kernel feature, has stricter version requirements, requiring at least Linux kernel version 4.4 [66]. To use eBPF effectively for tracing, users must install either the BPF Compiler Collection (BCC) framework to write custom eBPF programs or the bpftrace CLI tool for quick tracing tasks. LTTng differs in that it requires separate installation of both kernel-space and user-space components [22]. It features a modular setup, allowing users to install only kernel tracing, user-space tracing (supporting languages like C/C++, Java, and Python), or both. LTTng's configuration is managed via a dedicated CLI tool that oversees recording sessions and various components, providing a unified interface for both kernel and user-space tracing.

The choice between these tools depends on specific use cases and system requirements. ftrace, with its kernel integration, is well-suited for constrained Linux environments. eBPF provides flexible configuration options, making it ideal for both quick insights and custom tracing tool development. LTTng, with its modular design, offers a comprehensive tracing solution for both kernel and user space but comes with a more complex setup process.

Instrumentation Workflow

The instrumentation process for ftrace, eBPF, and LTTng involves distinct trade-offs in complexity and flexibility. Both ftrace and eBPF use DTRACE_PROBEn macros to add tracepoints. ftrace requires manual steps or the use of scripts to locate tracepoint addresses and identify argument registers, making iterative development cumbersome. In contrast, eBPF streamlines this process by automatically detecting tracepoints, requiring only the application's binary location when using it. For more complex use cases, BCC provides greater control over instrumentation and functionality but involves a more complex implementation. LTTng differs by providing custom tracepoint providers via C macros, enabling tailored tracing solutions. These tracepoints can be automatically located using the CLI tool without manually identifying the binary's location or extracting tracepoint addresses from the source code. While ftrace requires manual activation, eBPF allows automated attachment through bpftrace scripts. LTTng offers a command-line tool that balances ease of use with control. In summary, while ftrace has the simplest setup, its instrumentation workflow demands the most manual effort. eBPF balances usability with real-time aggregation capabilities, while LTTng provides unmatched flexibility.

Scripting capabilities

bpftrace, built on eBPF, provides a high-level scripting language inspired by AWK and DTrace's D language [67]. It enables users to write scripts that combine probes, predicates, and actions to dynamically trace kernel and user-space events while performing real-time aggregations. bpftrace can attach to probes and conditionally execute actions such as printing formatted output or maintaining counters, making it ideal for quick monitoring tasks. In contrast, ftrace functions as a traditional tracing tool without native scripting or aggregation capabilities. Users interact with it through the debugFS filesystem or trace-cmd, which provide basic functionality such as enabling tracepoints and collecting logs. While ftrace effectively captures raw trace data, it lacks a programmable interface, making it less suitable for advanced data manipulation. Similarly,

LTTng focuses on capturing and recording trace data but does not include an embedded scripting language for processing or aggregation. Instead, it relies on external tools to analyze its output, typically stored in the Common Trace Format (CTF) [68]. In summary, bpftrace excels with its AWK-like scripting capabilities for real-time aggregation and analysis, while ftrace and LTTng primarily serve as tracing tools for capturing raw data without built-in aggregation features.

Analysis and visualization

ftrace benefits from strong visualization capabilities through integration with tools like KernelShark, which provides interactive visualizations of ftrace data, including timing and delay analysis. eBPF, particularly with bpftrace, offers advanced real-time analysis but lacks direct visualization support. Instead, its scripting capabilities enable complex data aggregation and analysis. LTTng distinguishes itself with a comprehensive trace analysis and visualization approach. It outputs binary trace data, which can be analyzed using tools like Babeltrace for text conversion or more advanced solutions such as Tracealyzer and Trace Compass. Trace Compass, in particular, provides an intuitive interface for importing and analyzing LTTng traces and supports remote trace retrieval, a valuable feature for embedded systems where traces are often collected on resource-constrained devices. Depending on the analysis approach, ftrace and LTTng offer dedicated visualization tools, while eBPF relies on CLI-based aggregation and evaluation.

Documentation and Community Support

ftrace's documentation is primarily integrated into the Linux kernel documentation [69], providing detailed explanations of its functionality, available tracers, and configuration options. While updated with each kernel release, it may be less accessible to newcomers unfamiliar with kernel documentation. Community support is mainly available through Linux kernel mailing lists and forums. eBPF, as a more recent technology, has a growing ecosystem of documentation and community resources. The BPF Compiler Collection (BCC) project offers extensive documentation and examples for eBPF programming, complemented by books, articles, and online tutorials [70]. Additionally, various online resources provide one-liners and guidance for writing short bpftrace scripts. LTTng distinguishes itself with comprehensive and well-structured official documentation [22], including detailed user manuals, API references, and version-specific guides. Regular updates cover installation, usage, and advanced topics. Among the three, LTTng offers the most structured and beginner-friendly documentation with dedicated community

support. ftrace, despite its detailed documentation, relies on general Linux kernel community channels. eBPF, while rapidly expanding, has a diverse but less centralized documentation ecosystem.

4.1.2 Discussion on Overhead and Stability

The performance analysis in Section 3.3.3 highlights differences in execution overhead and stability across the evaluated frameworks. LTTng demonstrates superior performance in active tracing scenarios, achieving average routine execution times approximately 60% lower than both ftrace and eBPF. This performance advantage stems from fundamental architectural differences. While ftrace and eBPF use trap-based instrumentation with "sys/sdt.h" macros to detect tracepoints in ELF files, representing a dynamic instrumentation approach using uprobes, LTTng employs static instrumentation [34]. By integrating with the UST library, LTTng leverages static function call instrumentation, bypassing system calls entirely [14]. This elimination of system calls removes a significant source of overhead and enhances stability. In contrast, the higher overhead and greater variability in routine durations observed in ftrace and eBPF primarily result from their reliance on kernel infrastructure. When tracing is inactive or disabled, all three frameworks exhibit similar performance, with overhead approaching that of uninstrumented code. This finding indicates that dormant tracepoints impose minimal to negligible performance impact, regardless of the chosen framework.

4.2 Impact of Tracing on Real-Time Performance

The evaluation of tracing mechanisms in real-time systems highlights the trade-offs between overhead, stability, and instrumentation flexibility. The Cyclictest benchmark, executed under carefully controlled conditions, provides quantitative insights into the performance impact of different tracing frameworks. The results show that ftrace and LTTng in kernel-space tracing have a lower impact on latency compared to user-space instrumentation. ftrace demonstrates greater stability than LTTng in kernel-space, which may be attributed to its out-of-the-box integration within the Linux kernel, whereas LTTng requires loading its own kernel modules for kernel-space tracing [71]. In contrast, eBPF has a greater performance impact on kernel-space tracing than both ftrace

and LTTng, and even surpasses LTTng's effect on user-space instrumentation (Figure 3.4).

In terms of user-space effects on real-time performance, LTTng's static instrumentation exhibited lower latencies than dynamic tracing with ftrace or eBPF. This aligns with expectations, as static tracepoints avoid the runtime overhead of dynamic symbol resolution and breakpoint handling. The box plot analysis (Figure 3.3) confirms that static instrumentation maintains more consistent latency bounds, whereas dynamic approaches display a broader latency distribution. The comparison between PREEMPT and PREEMPT-RT revealed that while both models exhibit similar latency distributions up to 30 µs, PREEMPT-RT enforces tighter latency bounds beyond this threshold (Figure 3.5). This behavior aligns with PREEMPT-RT's design goal of ensuring predictable real-time performance and validates its effectiveness in achieving it.

4.3 Practical Recommendations for Developers and System Designers

Choosing the right tracing solution is fundamental to effective performance analysis and debugging. The optimal choice depends on factors such as the target environment, required tracing depth, and acceptable performance overhead.

Table 4.1 provides a reference for determining whether a tracing framework supports a particular instrumentation method, without detailing the underlying mechanisms or implementation specifics. It categorises frameworks and tools based on their ability to perform static and dynamic instrumentation in both kernel and user-space.

	Kernel-space		User-space	
	Static Instrumentation	Dynamic Instrumentation	Static Instrumentation	Dynamic Instrumentation
SystemTap	✗	✔	✗	✔
ftrace	✔	✔	✗	✔
LTTng	✔	✔	✔	✔
eBPF	✔	✔	✗	✔
perf	✔	✔	✗	✔
strace	✗	✗	✗	✔

Table 4.1: **Supported Instrumentation Mechanisms in Common Linux Tracing Frameworks/Tools.** The table categorises widely used tracing frameworks by their support for static and dynamic instrumentation in kernel and user-space.

For quick insights, lightweight tools like strace and perf require minimal setup and effectively identify performance bottlenecks in individual processes. In kernel-space tracing, ftrace provides efficient function-level tracing with low overhead, while the BCC framework enables flexible, programmable instrumentation via eBPF. For user-space applications, LTTng is a strong choice due to its efficient event tracing, broad language support, and extensive documentation. When short, script-based tracing is required, the eBPF-based bpftrace offers an expressive, awk-like syntax for both user-space and kernel-space insights.

To minimize overhead and enhance stability, static tracepoints are preferable. ftrace leverages predefined tracepoints within the kernel, while LTTng supports static user-space instrumentation, with both introducing lower runtime overhead than dynamic trap-based techniques like kprobes and uprobes. Restricting traced events to those required and avoiding full system tracing further reduces the impact on the observed system. Buffered tracing solutions, such as LTTng's ring buffers, are more efficient than direct logging to files or consoles. Finally, tools relying on ptrace, such as strace, should be used cautiously due to the substantial scheduling overhead they impose.

5 Conclusion

This thesis presents a comparative analysis of tracing mechanisms in Linux, evaluating their applicability to real-time systems. The study examines three major tracing frameworks, ftrace, LTTng, and eBPF-based solutions, assessing their usability, features, performance overhead, and impact on real-time behavior.

Regarding usability and features, ftrace offers the simplest setup due to its native kernel integration, making it accessible to system developers. Overall, ftrace instrumentation workflow for user-space applications is cumbersome, and the lack of built-in scripting capabilities limits its flexibility. In contrast, eBPF provides extensive programmability and real-time data aggregation through the BCC framework and bpftrace, making it well-suited for advanced tracing. LTTng, though requiring a more complex setup, is highly modular and supports both kernel-space and user-space tracing with structured instrumentation.

The evaluation of user-space instrumentation showed that LTTng introduces the lowest overhead due to its static instrumentation methodology, outperforming ftrace and eBPF-based solutions. While ftrace and eBPF rely on dynamic instrumentation via uprobes, resulting in performance penalties from context switching, LTTng's integration with user-space libraries eliminates these sources of overhead. The choice between dynamic and static instrumentation, represented by ftrace/eBPF and LTTng, presents a fundamental trade-off. Performance benchmarks indicated that active user-space tracing imposes measurable overhead, with LTTng's static approach achieving the lowest execution times and the highest stability. All three frameworks exhibited negligible impact when tracing was disabled, which is the default state in production systems.

The impact of tracing on real-time performance was evaluated using Cyclictest, revealing that kernel-space instrumentation with ftrace or LTTng introduces less latency than user-space instrumentation. While eBPF's kernel-space instrumentation has less impact than its user-space counterpart, it was the only kernel-space method that had a greater impact on real-time performance than user-space instrumentation with LTTng.

The findings of this thesis emphasise the importance of selecting an appropriate tracing framework based on specific use case requirements. While tracing frameworks provide useful insights for debugging and performance analysis, their overhead can disrupt time-sensitive applications. Developers must carefully balance tracing granularity with performance constraints to prevent interference with the system under observation, which could distort the application's actual behavior. Prioritizing static instrumentation and limiting tracing to essential events helps minimize overhead while preserving the accuracy of performance evaluations.

Future research could explore the functional differences between ftrace and eBPF in user-space tracing to identify factors influencing their performance variations, despite both relying on the same kernel infrastructure. Expanding the evaluation to include additional tools, such as SystemTap, which also offers scripting capabilities but uses a different approach by loading custom kernel modules instead of eBPF's method, would also provide valuable insights.

List of Figures

List of Tables

Bibliography

[1] Reghenzani, Federico; Giuseppe Massari; and William Fornaciari (2019): "The Real-Time Linux Kernel: A Survey on PREEMPT_RT." In: *ACM Comput. Surv.*, **52**(1). doi:10.1145/3297714. URL `https://doi.org/10.1145/3297714`.

[2] BeauHD (2024): "20 Years Later, Real-Time Linux Makes It To the Kernel." `https://linux.slashdot.org/story/24/09/18/2151240/`. Accessed: 2024-11-11.

[3] (2022): "Open Source Summit - Steven Rostedt." `https://ossna2022.sched.com/speaker/steven_rostedt.1uvjfs44`. Accessed: 2024-11-11.

[4] Torvalds, Linus (2024): "Merge tag 'sched-rt-2024-09-17'." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=baeb9a7d8b60b021d907127509c44507539c15e5`. Accessed: 2024-11-11.

[5] Altenberg, Jan (2024): "PREEMPT_RT is mainline - What's next?" `https://www.osadl.org/Single-View.111+M5a552108278.0.html`. Accessed: 2025-22-01.

[6] Leveson, N.G. and C.S. Turner (1993): "An investigation of the Therac-25 accidents." In: *Computer*, **26**(7), pp. 18–41. doi:10.1109/MC.1993.274940.

[7] Gregg, Brendan (2021): *Systems performance.* 2. Upper Saddle River, NJ: Pearson.

[8] Baclawski, Kenneth (2018): "The Observer Effect." pp. 83–89. doi:10.1109/COGSIMA.2018.8423983.

[9] Beningo (2016): "Embedded Basics – Classifying Software Bugs." `https://www.beningo.com/embedded-basics-classifying-software-bugs/#`. Accessed: 2024-11-11.

[10] Weis, Or (2021): "Heisenbug 101: Guide to Resolving Heisenbugs." `https://www.rookout.com/blog/fantastic-bugs-and-how-to-resolve-them-ep1-heisenbugs/`. Accessed: 2025-23-01.

[11] Rostedt, Steven (2009): "Finding Origins of Latencies Using Ftrace." In: , p. 14.

[12] Gregg, Brendan (2020): "perf Examples." `https://www.brendangregg.com/perf.html`. Accessed: 2025-15-01.

[13] Desnoyers, Mathieu and Michel R. Dagenais (2006): "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux." URL `https://api.semanticscholar.org/CorpusID:11300732`.

[14] Gebai, Mohamad and Michel R. Dagenais (2018): "Survey and Analysis of Kernel and Userspace Tracers on Linux: Design, Implementation, and Overhead." In: *ACM Comput. Surv.*, **51**(2). doi:10.1145/3158644. URL `https://doi.org/10.1145/3158644`.

[15] Gregg, Brendan; Jim Mauro; Chad Mynhier; and Tariq Magdon-Ismail (2011): *DTrace.* Philadelphia, PA: Prentice Hall.

[16] Gregg, Brendan (2015): "Tracing Summit 2014: From DTrace To Linux." `https://www.brendangregg.com/blog/2015-02-28/from-dtrace-to-linux.html`. Accessed: 2024-11-11.

[17] Gray, Jim (1986): "Why Do Computers Stop and What Can Be Done About It?" In: *Symposium on Reliability in Distributed Software and Database Systems*. URL `https://api.semanticscholar.org/CorpusID:10364748`.

[18] Rice, Liz (2023): *Learning eBPF.* Sebastopol, CA: O'Reilly Media.

[19] Shende, Sameer S. (1999): "Profiling and Tracing in Linux." URL `https://api.semanticscholar.org/CorpusID:1436837`.

[20] Desnoyers, Mathieu (2025): "Using the Linux Kernel Tracepoints." `https://docs.kernel.org/trace/tracepoints.html`. Accessed: 2025-20-02.

[21] Theodore Ts'o, Li Zefan and Tom Zanussi (2025): "Event Tracing." `https://www.kernel.org/doc/html/v5.1/trace/events.html#`. Accessed: 2025-20-02.

[22] Proulx, Philippe (2024): "LTTng: an open source tracing framework for Linux." `https://lttng.org/docs/v2.13/`. Accessed: 2024-11-11.

[23] Cantrill, Bryan M.; Michael W. Shapiro; and Adam H. Leventhal (2004): "Dynamic instrumentation of production systems." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04. USA: USENIX Association, p. 2.

[24] Harper-Cyr, Christian; Michel R Dagenais; and Ahmad S Bushehri (2019): "Fast and flexible tracepoints in x86." In: *Softw. Pract. Exp.*, **49**(12), pp. 1712–1727.

[25] Frysinger, Mike (2024): "Function Tracer Design." `https://www.kernel.org/doc/html/v5.1/trace/ftrace-design.html#have-function-tracer`. Accessed: 2025-20-02.

[26] Rostedt, Steven (2009): "Debugging the kernel using Ftrace - part 1." `https://lwn.net/Articles/365835/`. Accessed: 2025-15-01.

[27] Toupin, Dominique (2011): "Using tracing to diagnose or monitor systems." In: *IEEE Softw.*, **28**(1), pp. 87–91.

[28] Laplante, Phillip A and Seppo J Ovaska (2011): *Real-time systems design and analysis*. 4. New York, NY: Wiley-IEEE Press.

[29] Hambarde, Prasanna; Rachit Varma; and Shivani Jha (2014): "The Survey of Real Time Operating System: RTOS." In: *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. pp. 34–39. doi:10.1109/ICESC.2014.15.

[30] Wang, K. C. (2023): *Embedded Real-Time Operating Systems*. Cham: Springer International Publishing, pp. 429–503. doi:10.1007/978-3-031-28701-5_10. URL `https://doi.org/10.1007/978-3-031-28701-5_10`.

[31] Murti, KCS (2022): *Real-Time Operating Systems (RTOS)*. Singapore: Springer Singapore, pp. 189–224. doi:10.1007/978-981-16-3293-8_7. URL `https://doi.org/10.1007/978-981-16-3293-8_7`.

[32] Aroca, Rafael Vidal and Glauco A. P. Caurin (2009): "A Real Time Operating Systems (RTOS) Comparison." URL `https://api.semanticscholar.org/CorpusID:8033045`.

[33] Piotrowski, Mateusz (2024): "Benchmarking Performance Overhead of DTrace on FreeBSD and eBPF on Linux." URL `https://papers.freebsd.org/2024/asiabsdcon/piotrowski-Benchmarking-Performance-Overhead-of-DTrace-on-FreeBSD-and-eBPF-on-Linux.files/piotrowski-Benchmarking-Performance-Overhead-of-DTrace-on-FreeBSD-and-eBPF-on-Linux-paper.pdf`.

[34] Raphael Beamonte, Michel Dagenais, Francis Giraldeau (2012): "High Performance Tracing Tools for Multicore Linux Hard Real-Time Systems." URL `https://onlinelibrary.wiley.com/doi/10.1155/2015/261094`.

[35] Bird, Tim (2009): "Measuring Function Duration with Ftrace." In: *Proceedings of the Linux Symposium*, p. 10.

[36] Community, FreeBSD (2018): "DTrace on FreeBSD." `https://wiki.freebsd.org/DTrace`. Accessed: 2024-11-11.

[37] Jones, Colin (2015): "Dtrace even better than strace for OSX." `https://8thlight.com/insights/dtrace-even-better-than-strace-for-os-x`. Accessed: 2024-11-11.

[38] Fox, Paul (2015): "Dtrace4Linux Github." `https://github.com/dtrace4linux/linux`. Accessed: 2024-11-11.

[39] George Neville-Neil, Graeme Jenkinson, Jonathan Anderson (2018): "OpenDTrace Specification version 1.0." `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-924.pdf`. Accessed: 2024-11-11.

[40] Frank Ch. Eigler, Will Cohen, Vara Prasad (2005): "Architecture of systemtap: a Linux trace/probe tool." In: . Accessed: 2024-11-11.

[41] Rocca, Emanuele (2021): "Comparing SystemTap and bpftrace." `https://lwn.net/Articles/852112/`. Accessed: 2024-11-11.

[42] Corbet, Jonathan (2007): "On DTrace envy." `https://lwn.net/Articles/244536/`. Accessed: 2024-11-11.

[43] Team, SystemTap Development (2014): "SystemTap Tapset Reference Manual." `https://sourceware.org/systemtap/tapsets/`. Accessed: 2024-11-11.

[44] Lawall, Julia; Himadri Chhaya-Shailesh; Jean-Pierre Lozi; and Gilles Muller (2023): *Graphing Tools for Scheduler Tracing*. Tech. Rep. RR-9498. URL `https://inria.hal.science/hal-04001993`.

[45] Gregg, Brendan (2022): "Netflix End of Series 1." `https://www.brendangregg.com/blog/2022-04-15/netflix-farewell-1.html`. Accessed: 2024-11-11.

[46] Gregg, Brendan (2018): "bpftrace (DTrace 2.0) for Linux 2018." `https://www.br` `endangregg.com/blog/2018-10-08/dtrace-for-linux-2018.html`. Accessed: 2024-11-11.

[47] Kranenburg, Paul (1991): "Sun Strace Inital Message." `https://stuff.mit.` `edu/afs/sipb/project/eichin/cruft/machine/sun/sun-Strace`. Accessed: 2024-11-11.

[48] strace developers (2024): "strace — strace.io." `https://strace.io/`. Accessed: 2024-11-11.

[49] 03cranec (2024): "Sysdig Github." `https://github.com/draios/sysdig/?tab=` `readme-ov-file`. Accessed: 2024-11-11.

[50] Shukla, Durgesh (2025): "Sysdig is recognized as a Customers' Choice in Gartner® Voice of the Customer for Cloud-Native Application Protection Platforms." `https:` `//sysdig.com/blog/sysdig-recognized-as-customers-choice-in-gartner-v` `oice-of-the-customer-for-cnapp/`. Accessed: 2025-20-02.

[51] Contributors, Peer (2024): "Voice of the Customer for Cloud-Native Application Protection Platforms." `https://www.gartner.com/doc/reprints?id=1-2JU4Q` `PLM&ct=241230&st=sb`. Accessed: 2025-20-02.

[52] eunomia (2024): "eBPF Practice: Tracing User Space Rust Applications with Up-robe." `https://eunomia.dev/tutorials/37-uprobe-rust/`. Accessed: 2025-16-01.

[53] sematext (2020): "How to Instrument UserLand Apps with eBPF." `https://sema` `text.com/blog/ebpf-userland-apps/`. Accessed: 2025-13-01.

[54] Fleming, Matt (2018): "Using user-space tracepoints with BPF." `https://lwn.ne` `t/Articles/753601/`. Accessed: 2025-16-01.

[55] Rostedt, Steven (2017): "Uprobe-tracer: Uprobe-based Event Tracing." `https:` `//docs.kernel.org/trace/uprobetracer.html`. Accessed: 2025-16-01.

[56] Wielaard, Mark (2012): "Github - sdt.h." `https://github.com/jav/systemtap` `/blob/master/includes/sys/sdt.h`. Accessed: 2025-16-01.

[57] Gregg, Brendan (2015): "Hacking Linux USDT with Ftrace." `https://www.bren dangregg.com/blog/2015-07-03/hacking-linux-usdt-ftrace.html`. Accessed: 2025-16-01.

[58] Proulx, Philippe (2024): "Instrument a C/C++ user application." `https://lttn g.org/docs/v2.13/#doc-c-application`. Accessed: 2025-16-01.

[59] Bakhvalov, Denis (2020): *Performance Analysis and Tuning on Modern CPUs - Squeeze the Last Bit of Performance from Your Application.* Seattle: Amazon Digital Services LLC - Kdp.

[60] Saini, Subhash; et al. (2011): "The impact of hyper-threading on processor resource utilization in production applications." In: *2011 18th International Conference on High Performance Computing.* pp. 1–10. doi:10.1109/HiPC.2011.6152743.

[61] de Oliveira, Daniel Bristot (2020): "Demystifying the Real-Time Linux Scheduling Latency." `https://bristot.me/demystifying-the-real-time-linux-latency /`. Accessed: 2025-23-01.

[62] Shul, Costa (2025): "Cyclictest - The Linux Foundation." `https://wiki.linux foundation.org/realtime/documentation/howto/tools/cyclictest/start`. Accessed: 2025-23-01.

[63] Wensley, Bart (2022): "Cyclictest - Latency debugging with tracing." `https: //wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclic test/tracing`. Accessed: 2025-23-01.

[64] Wensley, Bart (2021): "Cyclictest - Test Design." `https://wiki.linuxfounda tion.org/realtime/documentation/howto/tools/cyclictest/test-design`. Accessed: 2025-23-01.

[65] Roesch, Stefan (2024): "Introduction to kernel tracing with ftrace." `https://de vkernel.io/posts/ftrace_intro/#overview`. Accessed: 2025-08-02.

[66] Traiaia, Karim (2023): "Issue 002: Programming the Kernel with eBPF." `https: //www.kerno.io/blog/programming-the-kernel-with-ebpf`. Accessed: 2025-08-02.

[67] danobi (2024): "bpftrace - Github." https://github.com/bpftrace/bpftrace. Accessed: 2025-08-02.

[68] Proulx, Philippe (2024): "CTF2-SPEC-2.0rA: Common Trace Format version 2." `https://diamon.org/ctf/`. Accessed: 2025-08-02.

[69] Rostedt, Steven (2017): "ftrace - Function Tracer." `https://docs.kernel.org/trace/ftrace.html`. Accessed: 2025-08-02.

[70] eBPF.io authors (2025): "eBPF Documentation." `https://ebpf.io/what-is-ebpf/#documentation`. Accessed: 2025-08-02.

[71] Proulx, Philippe (2024): "LTTng - Build from source." `https://lttng.org/docs/v2.13/#doc-building-from-source`. Accessed: 2025-20-02.