System and Application Performance Analysis Patterns Using Software Tracing

Sauradip Ghosh

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

June 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sauradip Ghosh**

Entitled: **System and Application Performance Analysis Patterns Using Software Tracing**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________ Chair
Dr. Nematollaah Shiri

______________________ Examiner
Dr. Nematollaah Shiri

______________________ Examiner
Dr. Juergen Rilling

______________________ Supervisor
Dr. Abdelwahab Hamou-Lhadj

______________________ Co-supervisor
Dr. Naser Ezzati-Jivan

Approved by ______________________

__________ 2022

____
Dr. Mourad Debbabi, Interim Dean
Gina Cody School of Engineering & Computer Science

Abstract

System and Application Performance Analysis Patterns Using Software Tracing

Sauradip Ghosh

Software systems have become increasingly complex, which makes it difficult to detect the root causes of performance degradation. Software tracing has been used extensively to analyze the system at run-time to detect performance issues and uncover the causes. There exist several studies that use tracing and other dynamic analysis techniques for performance analysis. These studies focus on specific system characteristics such as latency, performance bugs, etc. In this thesis, we review the literature to build a catalogue of performance analysis patterns that can be detected using trace data. The goal is to help developers debug run-time and performance issues more efficiently. The patterns are formalized and implemented so that they can be readily referred to by developers while analyzing large execution traces. The thesis focuses on the traces of system calls generated by the Linux kernel. This is because no application is an island and that we cannot ignore the complex interactions that an application has with the operating system kernel if we are to detect potential performance issues.

Acknowledgments

I would like to convey my deepest thanks to my supervisor Dr. Abdelwahab Hamou-Lhadj for having considered me worthy enough to be a part of his research team and assisting me in every possible way. I would not have been able to complete this thesis without his expertise, support and guidance. I would also like to thank my co-supervisor Dr. Naser Ezatti-Jivan for all the technical help and guidance he gave me which gave me the confidence to dive deeper into my research. His timely feedback served as an anchor to this project and made sure it did not drift off-track. A million thanks to Dr. Raphaël Khoury for his help in the detection and formalization of the patterns and his quick E-mail replies to whatever queries that I might have had. This project wouldn't have been what it is without his expertise. And lastly, I want to convey all my gratitude to Riley who came up with the scripts and the traces and patiently accommodated each and every change request that I had. It has been my great pleasure and privilege to work with all of you. Lastly, I would like to thank my parents for believing in me and supporting me throughout this journey.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction & Motivation **1**

1.1 Introduction 1

1.2 Problem Statement & Motivation 3

1.3 Contributions of the Thesis 5

1.4 Organization of the Thesis 5

2 Background & Literature Review **6**

2.1 Background 6

2.1.1 Patterns in Software Engineering 6

2.1.2 Software Tracing 7

2.1.3 Linux Kernel Tracing 8

2.1.4 Trace Analysis Tools 13

2.1.5 Run-time Verification Tools 13

2.1.6 An Introduction to Temporal Logic 15

2.2 A Review of Existing Trace Analysis Studies 18

2.2.1 Detecting patterns in system traces 18

2.2.2 Performance Analysis using Tracing 25

2.3 Discussion 29

3 A Catalogue of Performance Analysis Patterns 30

3.1 Introduction . . . 30

3.2 Memory Management . . . 32

3.2.1 Memory Fragmentation . . . 32

3.2.2 Memory Leaks . . . 33

3.2.3 Tracing Processes Affected by the OOM Killer . . . 35

3.2.4 Memory Compaction Overhead . . . 37

3.3 Thread Scheduling . . . 39

3.3.1 Thread Swamping . . . 39

3.3.2 Unfair Scheduling . . . 40

3.3.3 Priority Inversion . . . 42

3.3.4 Sub-Optimal Threading . . . 43

3.3.5 Thread Scheduler Latency . . . 45

3.4 Disk Scheduling (I/O) . . . 45

3.4.1 I/O Scheduler Overhead . . . 45

3.4.2 Improper I/O Workload . . . 47

3.5 Network Stack . . . 49

3.5.1 TCP Re-Transmits . . . 49

3.6 Synchronization Context . . . 50

3.6.1 Excessive Busy-waiting . . . 50

3.6.2 Excessive synchronization . . . 51

3.6.3 Critical Lock Bottleneck . . . 52

3.6.4 Asymmetric Contention . . . 53

3.7 Hardware Context . . . 54

3.7.1 TLB Performance . . . 54

3.7.2 Cache Performance . . . 56

3.7.3 Branch Performance . . . 57

3.8 Application Context . . . 59

3.8.1 Page Writeback Latency . . . 59

3.8.2 Abnormally Long Application Sleep Duration 61
3.9 Overview & summary of discussed patterns 62

4 Evaluation **65**

4.1 Implementation of pattern detection methods 65
4.2 Evaluation Protocol 65
4.3 Evaluation Setup 66
4.4 Memory Management 66
4.4.1 Memory Fragmentation 67
4.4.2 Tracing Processes Affected by the OOM Killer 71
4.4.3 Memory Compaction Latency 73
4.5 Thread Scheduling 75
4.5.1 Thread Swamping 75
4.5.2 Unfair Scheduling 78
4.5.3 Priority Inversion 81
4.5.4 Sub-Optimal Threading 84
4.5.5 Thread Scheduler Latency 87
4.6 Disk Scheduling (I/O) 88
4.6.1 I/O Scheduler Overhead 89
4.7 Network Stack 91
4.7.1 TCP Re-Transmits 91
4.8 Synchronization Context 93
4.8.1 Excessive Busywaiting 93
4.8.2 Asymmetrical Contention 95
4.8.3 Excessive Synchronization 97
4.9 Application Context 99
4.9.1 Page Writeback Latency 99
4.9.2 Abnormally Long Application Sleep Duration 101
4.10 Discussion 102

5 Conclusion & Future Work **104**

5.1 Conclusion 104

5.2 Future Work 105

5.3 Concluding Remarks 105

Bibliography **106**

List of Figures

Figure 2.1 Semantics of LTL-FO+ [1] 17

Figure 3.1 A broad classification of Performance Analysis Patterns 31

Figure 3.2 Patterns under the Linux subsystem 31

Figure 3.3 Patterns under a context scope 32

Figure 3.4 Branch statistics via perf 59

Figure 4.1 Loading the fragmentation trace onto Trace Compass 67

Figure 4.2 Trace in XML format 69

Figure 4.3 Verifying LTL-FO+ specification using the BeepBeep monitor 70

Figure 4.4 Trace from the bpftrace tool oomkill.bt is converted to XML 72

Figure 4.5 Setting a high OOM score of 15 ensures that the process is killed off in case of low memory 72

Figure 4.6 Memory Compaction Latency Histogram 73

Figure 4.7 bpftrace script to display memory compaction histogram 74

Figure 4.8 Colour coded segments showing various thread states on the Trace Compass UI 76

Figure 4.9 Yellow indicates a blocked thread 76

Figure 4.10 Defining TCD and TD using timestamps A,B,C & D 77

Figure 4.11 A timeline of a thread which isn't swamped by stress threads 77

Figure 4.12 A high execution ratio in the absence of stress threads 77

Figure 4.13 Timeline of the thread when it is swamped by 40 stress tests 78

Figure 4.14 A low execution ratio which means that our application has been swamped . 78

Figure 4.15 A visual graph showing all the thread migrations 79
Figure 4.16 Highlighting how a single thread gets migrated repeatedly between CPUs . 79
Figure 4.17 The higher priority thread requesting a lock already held by a lower priority thread 81
Figure 4.18 Trace Compass timeline showing the relationship between the three threads 82
Figure 4.19 High Priority thread getting CPU access after the Low Priority thread relinquishes lock 83
Figure 4.20 The application with 25 threads sees a large number of threads being swamped. 86
Figure 4.21 The application with 4 threads sees no swamped threads 86
Figure 4.22 Latency histogram of a relatively idle system 87
Figure 4.23 Latency histogram of a system under stress 88
Figure 4.24 Updated IOSCHED code 89
Figure 4.25 Latency Distribution of the mq-deadline I/O scheduler 90
Figure 4.26 An updated tcpretrans bpftrace script to show timestamps in nanoseconds . 92
Figure 4.27 Output of the RTLola specification for Th = 9 92
Figure 4.28 User Space instrumentation helps trace spin lock requests, acquisitions and releases 94
Figure 4.29 Trace Compass resource view shows Thread 5008 spending considerable time on CPU 94
Figure 4.30 Our EASE script calculates spin-lock duration 95
Figure 4.31 Our EASE script showing lock contention statistics 96
Figure 4.32 Our EASE script shows the number of requests each lock witnesses 98
Figure 4.33 Our custom RTLola specification that triggers if a spike in writeback activity is observed 99

List of Tables

Table 3.1 A brief summary and overview of Performance Analysis Patterns 63

Table 3.2 A brief summary and overview of Performance Analysis Patterns (contd.) . . 64

Chapter 1

Introduction & Motivation

1.1 Introduction

Marc Andreessen wrote in an article in The Wall Street Journal that ”software is eating the world”[1]. What he implied by this statement was that software technologies are disruptive technologies, transforming the economy and existing industries. This rapid transformation was possible due to many factors including breakthroughs in hardware capability such as multi-core processors [2] and the adoption of the World Wide Web among consumers, thus setting the ground up for software to direct and control end-to-end supply chains. Industries that were relying less on technology had to harness the power of software to stay profitable and relevant[2].

This dramatic shift meant that many critical systems have to depend on software to function, which raises expectations when it comes to software performance and correctness [3]. There are many real life examples on how software bugs and crashes can harm the economy. Take for example, Facebook's massive outage in October 2021[3], which witnessed all Facebook owned apps crashing and not letting users log-in to access services. This downtime lasted for seven to eight hours. Users who rely on Facebook and its family of apps could not conduct business and had to look for alternatives. Facebook blamed it on a series of faulty configuration changes in the router[4].

[1] https://www.wsj.com/articles/SB10001424053111903480904576512250915629460

[2] https://techcrunch.com/2013/12/14/as-software-eats-the-world-non-tech-corporations-are-eating-startups/

[3] https://www.nytimes.com/2021/10/04/technology/facebook-down.html

[4] https://www.facebook.com/business/news/update-about-the-october-4th-outage

As time goes by, customer requirements from a software usually change which might lead to developers adding features on top of existing codebase without taking the time to sufficiently refactor or redesign so that the complexity remains manageable. This may lead to technical debt [4]. Unlike physical systems, software - which is essentially text in the form of logical instructions - is unbounded, on which multiple authors can work on for years and therefore it keeps changing, mutating and increasingly becoming too gargantuan to even comprehend; let alone maintain. This is in accordance with Lehman's second law which states that as projects grow, they tend to increase in complexity [5]. Several empirical studies have been done on the relationship between software complexity and maintainability [6]. The introduction of multi-core systems and the various parallel programming paradigms [7] make performance problems challenging to debug and fix as designing, writing, testing, analyzing, and optimizing multithreaded code can be a very challenging task [8].

Software tracing is one way to achieve visibility in code which can become extremely abstract and difficult to analyze over the years [9]. Tracing can be used to track the flow of program data and execution in real-time. Trace files show an execution snapshot of the application. Tracing is a dynamic analysis technique, which applies to the execution of a program. This is contrasted with static analysis techniques that work on the source code without executing it [10][11]. Tracing can provide programmers with a wealth of execution data which can be analyzed and processed via a suite of tools and techniques suited for the purpose [12]. The advantage of dynamic analysis is that the data collected pertains to real-life execution conditions which helps developers understand system behaviour either online or offline and detect unforeseen problems caused by complex interactions between applications and their run-time environments.

Software tracing techniques can be divided into two categories: static and dynamic tracing. Static tracing depends on *tracepoints* [13] compiled into the code. Tracepoints in code are locations which act as hooks to which a function or a probe (e.g., a print out statement) can be attached. Dynamic tracing, on the other hand, injects probe points into the system during run-time which act as break-point instructions [14] [15].

At any given time, millions of events can occur inside a system: including system calls, I/O processing, memory allocation or block device operations. Therefore, software traces can contain a large amount of information which makes processing and parsing traces a complex task [16] [17]

[18]. However, considering how useful tracing is and the industry support tracing infrastructure receives, trace analysis can be an important (and sometimes the only) tool to detect all kinds of performance problems.

1.2 Problem Statement & Motivation

The crux of the problem can be stated as thus: *How can we extract actionable performance patterns from trace files so that programmers can quickly detect resource bottlenecks, latencies and avenues for further optimization?* We were inspired by our study of code smells where the presence of bad smells could indicate deeper problems in the codebase. In the same way, we asked ourselves if we could, similarly, extract dynamic performance related smells from trace files which underline deeper problems in application performance.

In this thesis, we shall detect, categorize and formalize several performance patterns in order of their complexity. We call these patterns *Performance Analysis Patterns.* These point to performance problems such as synchronization issues, latency inducing block device operations and memory-related performance issues.

There is a wealth of research regarding performance analysis using tracing. Côté et al. [19] have worked on analyzing execution traces in real-time systems by defining an execution model of real-time tasks, running a pattern detection (MANEPI) algorithm and then executing complex analyses on only selected parts of the trace instead of the whole trace. They also used critical path analysis coupled with scheduling information to quickly detect scheduling issues. Giraldeau et al. [16] recover insightful system metrics such as CPU, block device and network usage from Linux kernel traces. They use LTTng [20] to generate traces based on static tracepoints compiled into the kernel. Daoud et al. [21] recover advanced block device level metrics from tracing events by developing efficient data structures and algorithms using a state-based approach. They have also developed a visualization model based on state history trees so that latencies in I/O request life-cycle can be visualized. Temporal languages such as LTL-FO+ have also been used to verify security related properties such as Integer overflow detection and call sequence profiling from assembly traces by Khoury et al [22]. Francois Doray and Michel Dagenais [23] introduce TraceCompare in where they detect performance variations by comparing multi-level execution traces using custom data-structures and a comparison algorithm. The goal was to highlight differences in execution by comparing traces generated by two execution instances while performing the same task. LTTV Delay Analyzer [24] analyzes blocking events and helps developers understand blocking dependencies in the form of dividing process waiting time into its constituting components. The authors develop an algorithm to unpack the blocking dependencies from traces generated by existing kernel instrumentation. DepGraph [25] uses tracing to create dependency graphs of threads to detect bottlenecks. These graphs are constructed by building a state database and extracting dependencies such as blocking states. Then, multiple graphs arising from different executions are combined and clustered using the k-means algorithm to identify performance outliers. LaRosa et al. [26] employ data mining techniques such as frequent pattern mining to detect excessive inter-process communication. We will review these works in detail in the next chapter.

Additionally, Dmitry Vostokov has presented a catalogue of crash and dump analysis patterns in his book, *Encyclopedia of Crash Dump Analysis Patterns* [27]. This category mostly consists of performance patterns detected in Windows memory and crash dumps with a few from Linux and Mac OS as well. Most of these patterns are from WinDbg logs and memory dumps. However, these performance patterns are identified through manual dump inspection, mostly on a case by case basis with no attempt to formalize them or automatically extract them. However, our work goes much farther in detecting and formalizing these performance patterns from trace files as we attempt to extract these patterns using scripting, run-time monitoring and formalize them using temporal languages. His work on trace and log analysis delves into identifying and detecting patterns in trace files itself such as Corrupt Messages, Circular Traces and Discontinuity [28].

Our approach relies on an extensive review of the literature related to system performance to determine performance patterns and use tracing to detect them. We have also formalized these patterns and detected them using scripting and run-time monitoring techniques. Our approach leads to more patterns being detected to help developers quickly find out performance bottlenecks using our tools. To our knowledge, no work has attempted to come up with a categorization, formalization, and implementation of performance analysis patterns on the basis of trace files. Vostokov's work [27] comes somewhat close. However, most of his patterns are based on crash and dump analysis and there is no attempt to automatically detect and formalize them.

1.3 Contributions of the Thesis

Our thesis contributions are as follows:

- We have developed a catalogue of performance analysis patterns for the Linux operating system, which cover a wide range of performance issues varying from memory management to network latency. To our knowledge, this is the first time that such a comprehensive catalogue is presented.
- We use a formal language to represent the patterns. This allows analyst to use formal method tools to detect performance issues in traces.
- We have developed several scripts to support performance analysis patterns described in this thesis. The scripts are available as an an open source and can readily be embedded in trace analysis tools.

1.4 Organization of the Thesis

This thesis is organized into five chapters. Chapter 1 provides an introduction and a brief discussion of related work. Chapter 2 introduces some relevant technical background, an extensive literature review and a brief discussion. Chapter 3 features the catalogue of performance analysis patterns and a summary. Chapter 4 features an evaluation along with the implementation details of selected patterns. Chapter 5 ends with a conclusion and future work.

Chapter 2

Background & Literature Review

2.1 Background

In this section, we present background concepts needed to understand the content of the thesis. We begin with a general introduction to patterns in software engineering and their utility. We then move on to software tracing in general and how it can be employed to achieve performance gains and detect difficult-to-detect bugs. A thorough review of the Linux tracing infrastructure follows including the tools we use in this thesis. Further, we introduce temporal logic and run-time verification techniques we use for the formalization and detection of these performance analysis patterns.

2.1.1 Patterns in Software Engineering

Christopher Alexander, an architect by profession, started working on patterns by developing a pattern language to design town layouts [29]. He realized that by doing so, he and his peers could detect, identify and document common problems and best solutions into patterns that can be reused by other practitioners. He proposed to catalogue the patterns in the form of best practices in order to build a database of dos-and-donts. In software engineering, a pattern can be defined as a general, reusable solution to a design problem that manifests repeatedly across multiple projects [30]. An example of patterns would be design patterns, which are reusable solutions to common software design problems [31]. The patterns can be described using formal languages and documented accordingly so that they can be consulted in the future.

Design patterns became popular after the publication of *Design Patterns: Elements of Reusable Object Oriented Software* by the "Gang of Four" - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [32] in 1985. The book was a compilation of 23 software design patterns to address common design problems in object-oriented programming. The patterns were classified into categories such as creational, structural and behavioral. Creational patterns are about object creation in place of direct instantiation. Some examples are Abstract Factory (grouping object factories together on the basis of commonality) and Singleton (which ensures that only a single instance is created for a class). Structural patterns are those that are concerned with class and object composition such as a Facade (which offers a simple interface to complex codebases) and Flyweight (which makes it easier to manipulate a large number of objects). Behavioral patterns are about object communication such as Chain of Responsibility and Observer (which allows certain objects to listen to an event).

There are performance patterns in software engineering as well, which assist in building scalable and responsive software. Examples of performance patterns including the Coupling pattern -using coarse grained objects to limit excessive interactions and aggregation of data which is frequently accessed, and the First Things First pattern, which ensures that tasks are prioritized using centering principle to ensure maximal scalability[1].

In this thesis, we present performance analysis patterns to capture good practices for analyzing execution traces with the objective of detecting performance issues. The patterns presented in this thesis should not be confused with performance code smells, which are places in the code that may indicate performance issues. Our patterns emphasize best practices for the 'analysis' of execution traces with the objective of detecting performance problems that may (or may not) be caused by code smells.

2.1.2 Software Tracing

Software tracing provides a deep insight into program execution [33]. As modern software systems become larger, more complex and distributed, tracing provides system administrators and programmers observability in order to diagnose performance issues and bugs [34]. In order for

[1]https://research.cs.queensu.ca/home/elgazzar/soft437/lecture-notes/Chapter10.pdf

tracing to make sense, instrumentation, data collection, visualization and analysis tools are crucial. They must also be effectively leveraged across different software layers for developers to be able to track the entire life-cycle of a process or a request. The amount of information in trace files can be considerably high [33][35]. They can contain information about system calls, routine calls, interrupts, memory allocation events, etc. In this thesis, we leverage the Linux tracing architecture in order to trace and profile applications with the intent of detecting performance patterns in traces.

2.1.3 Linux Kernel Tracing

The Linux Kernel Tracing framework sources are as follows:

- **Tracepoints:** Tracepoints[2] are essentially hooks that are placed in the code that a function probe can attach to during run-time. Tracepoints can be turned on or off. A tracepoint which has been turned off has a very low overhead. A tracepoint being 'on' essentially means that a probe has been attached to it. When a tracepoint is reached, the probe is called in the execution context of the caller. When the probe has finished execution, control returns to the caller. Adding a probe to the tracepoint is done via register_trace_subsys_eventname(). Tracepoints can be placed in important Linux subsystems to fetch valuable run-time information. They can also pass parameters. In order to insert a tracepoint, a tracepoint definition must be placed in a header file and the tracepoint statement itself must be present in the C code. An example of a tracepoint would be the *mm_page_alloc_extfrag* which provides information regarding how fragmented the memory has become under current operating conditions. This tracepoint is placed in the memory management subsystem of Linux.

- **Kprobes:** Kprobes[3] allow one to dynamically trace any Linux kernel routine in order to collect run-time information in a non-invasive manner. They are not as stable as tracepoints. If the name of the routine changes with subsequent kernel versions, then the kprobe would also not work. However, kprobes offer a much deeper instrumentation than tracepoints as almost any routine in the Linux kernel can be traced and investigated. When the probe is hit, a handler routine is invoked. There are two types of probes - kprobes and kretprobes. A kretprobe is hit

[2] https://www.kernel.org/doc/html/latest/trace/tracepoints.html
[3] https://www.kernel.org/doc/html/latest/trace/kprobes.html

when the handler routine returns. Kprobe based instrumentation comes in the form of a kernel module which 'registers' the probes and the return probe 'unregisters' them. When a kprobe is registered, a copy of the probed instruction is made and a break-point instruction is inserted into the first byte of the instruction. When the break-point instruction is reached by the CPU, a trap occurs and control passes to the kprobe. The pre_handler is executed, the copy of the probed instruction is single-stepped and then the post_handler is executed. Kprobes perform a variety of safety checks such as making sure the function doesn't contain an indirect jump or an exception causing instruction. There are some functions that kprobes cannot probe as they would cause a recursive trap. The list of these functions is called a kprobe blacklist. An example of a kprobe would be *vfs:read* which instruments the read function in the virtual file-system.

- **Uprobes:** Uprobes [36] allows one to to trace user-level functions in the form of user-level dynamic tracing. For example, tracing the return values of a user-level function from running bash shells. Even library functions can be traced.

There are tracers that help collect tracing data from the sources discussed above. There are two such tracers built in the kernel itself - perf-tools and ftrace. Other tracers include sysdig, SystemTap and LTTng. Other tools such as eBPF and LTTng can also be used to trace and profile the kernel, however they are not built into the kernel. In this thesis, we use LTTng and perf-tools.

- **ftrace:** ftrace[4] is an in-built kernel tracer used by developers in order to analyze latency. It was developed and is maintained by Steven Rostedt and was merged into the mainline in kernel version 2.6.27. It is essentially a function tracer but can also be considered a framework of tracing tools. ftrace can be used to perform tracing operations like fetching information from tracepoints, detecting interrupt latency, attaching kprobes, recording function calls, generating call graphs, stack reports and many more.

- **perf-tools:** perf[5] was originally referred to as Performance Counters for Linux. It is a performance analysis tool integrated into the Linux kernel version 2.6.31 in 2009. It has its own

[4] https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[5] https://github.com/brendangregg/perf-tools

command line and can perform statistical profiling of both kernel and user-space. It can work with hardware performance counters, static tracepoints and kprobes. perf-tools are useful because engineers can quickly get a system snapshot when investigating faults or latency. Unlike other tracers such as LTTng, perf does not require daemons. It relies on a system call function. perf is one of the most used profiling tools and is integrated in many popular Linux distributions. It has several sub-commands such as stat, top, record, report and annotate which can be used to extract useful system metrics such as event counts and top functions, record and report sampled data in the form of graphs and trace scheduler events.

There are also front-end and visualization tools which helps configure tracing sessions, make sense of trace data and to come up with useful analyses and visualization aids. Front end tools are also used to configure tracers. However, they are add-ons and not built in the kernel. For example, trace-cmd[6], kernelshark[7], Trace Compass[8] and Flame Graph[9].

In this thesis, we work with LTTng and eBPF as tracing tools while using Trace Compass and Trace Compass EASE scripting as a trace analysis tool. We shall also be using temporal logic and run-time verification in order to formalize and detect some of these performance patterns. The following subsections will elaborate on these technologies.

Linux Trace Toolkit Next Generation (LTTng)

The Linux Trace Toolkit Next Generation is a low overhead, fully re-entrant and extremely scalable tracer for the Linux kernel as well as the user space. LTTng can be used to collect crucial run-time information in order to analyze the performance of large complex software systems. Having a low overhead would ensure that the studied system is not effected by the tracing infrastructure itself. The tracer has an easy to use interface and is memory efficient. It is architecture-independent and can generate traces across different architectures with precise timestamps across multiple processors.

LTTng has a command-line interface which operates on the user space (*lttctl*), a user-space

[6]https://linux.die.net/man/1/trace-cmd
[7]https://kernelshark.org/
[8]https://www.eclipse.org/tracecompass/
[9]https://www.brendangregg.com/flamegraphs.html

daemon (*lttd*) that writes trace data to disk and a kernel component which carries out kernel tracing. The user can control the tracing via *lttctl* which in turn starts *lttd*. *ltt-core* controls helper modules such as *ltt-heartbeat* (which detects cycle counter overflows), *ltt-facilities* (which lists the event types) and *ltt-statedump* (which describes the kernel state during tracing). *ltt-base* facilitates data transfer into circular buffers and once the sub-buffers are full, the disk writer daemon is called (*lttd*).

LTTng uses an XML event description that processes user-space and kernel level instrumentation to automatically generate trace headers and metadata in the traces. When control reaches the instrumentation site, these headers activate the functions that collect the relevant tracing information. *ltt-facilities* has information registered into by the information traced. The tracer uses a lockless re-entrancy mechanism and can handle non-maskable interrupts (NMI) handlers. A *call site* is the location in the original code where the tracing function is called whereas the *instrumentation site* is the site of the tracing function [20].

LTTng can interact with a variety of Linux kernels and user applications via instrumentation. Each LTTng session records all the instrumentation events and stores the information in circular buffers. It has a set of modules which does the work of kernel instrumentation as well. It can trace kernel tracepoints, system calls, kprobes and user-space probes and can create rules which are set into motion when certain specified tracepoints and/or system calls are encountered. It can also trace user applications provided a tracepoint header file is created. Triggers can also be created which executes a set of actions if a particular condition is satisfied such as when the buffer space consumed during a session becomes greater than a threshold.

LTTng records its data into *ring buffers*. A channel object oversees a set of such buffers. Each buffer has a number of *sub-buffers* where data is recorded when an event rule is matched. Channels can be created in order to dictate the size of ring buffers, set the number of sub-buffers under each ring buffer, define a set of tasks when there is no buffer space and so on.

As the name suggests, LTTng is a toolkit that contains a set of LTTng-tools, a user-space tracing library (liblttng-ust) and LTTng-modules that instrument and trace a kernel. It is open source, powerful, is able to produce correlated trace data including both kernel traces and user-space traces and is capable of handling multi-gigabyte sized trace data. For more, the reader can refer to the official documentation[10].

BPF

Berkeley Packet Filter [37] (BPF) was initially a technology developed to improve the performance of packet capture tools. It was then completely re-written by Alexei Starovoitov which transformed it into an execution engine which can be safely run inside the Linux kernel [36]. BPF is tremendously powerful and consists of its own helper functions, instruction sets and variables. The BPF runtime executes these instructions and they are subsequently turned into native instruction by the JIT compiler and an interpreter. There is also a verifier that ensures that the code doesn't crash the kernel. Hence, BPF is safe and will not crash the system.

BCC and bpftrace are high level front ends that abstract the process of writing BPF code which, otherwise, can be very difficult.

BCC(BPF Compiler Collection) is a high level tracing framework which provides an environment where a user can code in languages such as C and Python. The BCC repository on GitHub has more than 70 performance measuring tools which are ready to use[11]. It can make good use of a number of kernel features such as supporting kernel level dynamic and static instrumentation, Performance Monitoring Counter (PMC) events [38], filtering, stack-traces and the ability to overwrite ring buffers.

bpftrace on the other hand is a newer, high level language to develop BPF tools. We have used a few bpftrace scripts in this work in order to trace the system running some application. It allows full visibility to the entire software stack which includes all the major subsystems such as memory, block I/O, VFS and so on. bpftrace one-liners are powerful lines of code using which the user can trace executing events, fetch arguments from method calls, generate a stack trace, show latency distributions, access PMCs and so on. bpftrace scripts can be written to carry out more complex tasks such as calculating the latency between a kprobe and its kretprobe and displaying them in the form of a histogram using in-built variables and arrays.

[10] https://lttng.org/docs/v2.13/

[11] https://github.com/iovisor/bcc

2.1.4 Trace Analysis Tools

TraceCompass

TraceCompass[12] is a powerful trace analysis and visualization tool with the primary goal of extracting useful information from large trace files in the form of graphs, statistics, and views. It allows support many useful features including parsing large trace files, time correlation of multiple trace files, zooming into trace events at nanosecond level and inserting bookmarks in traces. Add to this, TraceCompass works with both system and application traces. It allows multiple trace formats and fast filtering and searching for trace events. TraceCompass can also be extended by adding support to new trace types.

TraceCompass has four main views: Project Explorer, Events, Statistics and Histogram. TraceCompass supports LTTng integration and useful plug-ins for viewing control flows, Resources, CPU usage, and so on.

Scripting plugins can be installed in TraceCompass to manipulate trace information. The plugins make use of Eclipse EASE Project which supports Javascript, Python, Ruby and many more languages. The EASE framework supports several scripting engines such as Nashorn for Javascript, Jython for Python and so on whereas providing an API for various TraceCompass modules such as Analysis, View, DataProvider and Filters. EASE scripting is extremely powerful as it can manipulate trace attributes and extract actionable metrics from traces[13].

2.1.5 Run-time Verification Tools

RTLola - a stream based specification language

Stream based monitoring consists of input streams being analyzed run-time and their data (such as readings from sensors) converted into statistics such as a running average, counters and integrals. Then, triggers can be set up to raise an alarm or carry out a predetermined task if a condition becomes true.

Stream based languages such as LOLA [39] are used to process synchronous, real-time traces

[12]https://www.eclipse.org/tracecompass/
[13]https://archive.eclipse.org/tracecompass.incubator/doc/org.eclipse.tracecompass.incubator.scripting.doc.user

whereas RTLola [40][41] - an extension, can be used to process variable-rate input streams and aggregate data over sliding windows. RTLola specifications can be determined via static analysis if it can be executed on a system with limited memory. For example, in order to calculate summations, the values can be pre-aggregated according to time intervals defined by the rate of output streams.

StreamLab is the monitoring framework for RTLola [40]. Stream based monitoring languages are much more expressive than temporal logic [40]. Lookup expressions in RTLola are used to compute the value of output streams by taking into account previous stream values. There are no assumptions made on the frequency of input streams. RTLola categorizes streams into *Event-based* - computed whenever new values arrive, and *Periodic-based* which are evaluated based on a time window - independent of arriving input values. RTLola is a strongly typed language where each expression has a value type (Boolean, String, Integer, Float) and a stream type (Event-based, Periodic). Techniques such as the ones presented by Meertens [42] and Li et al. [43] are used to ensure bounded memory consumption to compute sliding windows. StreamLab supports several predefined aggregation functions such as count, integration, summation, product, etc. The processing engine comprises of the Event Manager(EM) which reads inputs such as CSV files and translates into an internal representation and then mapped into corresponding input stream. The EM pushes the event on a working queue. After that, the Time Manager(TM) is responsible for the scheduling of the periodic streams - grouping them by their deadlines. An approaching deadline means the pushing of the streams into the working queues using the same channels as that of the EM. The third component of engine is the Evaluator(Eval) which evaluates the streams and store the computed values. Evaluated items are then popped off the working queue[40]. StreamLab uses Rust [14] to implement the monitor [44]. The monitor assesses the specifications against the system performance. Once the specifications are translated to Rust, it can be guaranteed that the monitor will not crash, provide the correct output - assuring that the platform satisfies the requirements. RTLola has the ability to type check streams and validate its correctness. It also performs static checks by generating a dependency graph where each stream is a node and each stream access is an edge. This is used to analyze memory and run-times. The memory consumption can be evaluated using a formula. However, the running time cannot be determined via specification alone. Hence, preliminary analysis provided

[14] https://www.rust-lang.org/

which calculates the complexity of each evaluation cycle in a certain point in time and how parallellized the specification is. The advantages of Rust are that it is LLVM backed, they are compatible with multiple platforms, enforces static checks and allows memory management. Rust compiler for LOLA injects verification annotations into the code which allows the static verifier Viper to prove functional correctness. StreamLab detects inconsistencies in the specification such as type errors (lossy conversion of floats to integers) and timing errors. It has two modes - interpreter and FPGA compilation. The interpreter mode allows the specifier to validate their specification which requires a trace. It checks if the trace complies with the specification. For more details the reader can refer to [45].

BeepBeep

BeepBeep [46][47] is a run-time monitor for AJAX web applications and JAVA programs. It routinely checks in real-time if events sent or received by an application satisfy a predetermined specification. These specifications can dictate the message ordering or the parameters inside messages in order to make sure that the application does not send incorrect messages or perform improper method calls, thus saving bandwidth and server processing time. BeepBeep comprises of a JavaScript file and a .jar applet file. BeepBeep supports linear temporal logic.

BeepBeep can only perform run-time verification and is restricted to trace files in the XML format. BeepBeep 3, the successor of BeepBeep, is a complete redesign. It does not mandate users to use any query language. Instead, the tool encourages them to build their own custom processor objects. It is customizable, modular and versatile.

2.1.6 An Introduction to Temporal Logic

Linear Temporal Logic (LTL)

Temporal logic deals with truth values that evolve over time. Amir Pnueli [48] introduced it to computer science and can be used to reason about reactive systems and is the basis of model checking. It uses modal logic such as $\diamond$ or **F** (for possibility) and $\Box$ or **G** (for necessity).These two operators are duals. Temporal logic has derived from Modal logic where $\diamond\Phi$ means that Φ will hold to be true eventually at some point in time whereas $\Box\Phi$ implies that Φ will always hold true at any time. **F** and **G** are written as *Eventually in the Future* and *Always in the Future* respectively[15]. Other temporal operators are $\bigcirc\phi$ or **X** which implies that ϕ is true in the next moment in time and $\phi\mathbf{U}\psi$ which implies that ϕ will hold until ψ does.

The temporal operators can also be re-written using **U** and **X** operators. One can combine atomic propositions with Boolean operators $(\wedge, \vee, \neg, \rightarrow)$ with the aforementioned temporal operators to design specifications which may evaluate to T (True) F (False) or ?. The specification will evaluate to T when every possible continuation of the trace will respect the property. It will evaluate to F when every possible continuation will violate the property and will result in ? when the property might be violated / matched later in the future.

LTL is used in model checking in order to verify safety, liveness and fairness properties[16]. For example, let us take into consideration a property where every request will eventually guarantee a response. Hence, this can be formalized in LTL as:

$$\mathbf{G}(\textit{request} \rightarrow \mathbf{F}(\textit{response}))$$

We shall be using BeepBeep, a lightweight monitor to check whether the traces that we obtain satisfy a given specification. Our specifications will be in the form of execution trace patterns.

LTL-FO+

LTL-FO+ [1][47] is the Fully First Order Quantification of LTL which is appropriate to model "data-aware" properties. Along with atomic propositions, Boolean operators and temporal operators, LTL-FO+ adds *quantifiers* that work with trace parameters. These quantifiers are $\exists$ and $\forall$. The semantics are shown in Figure 2.1

$\bar{m}$ represents the execution trace and $\bar{m}_0$, $\bar{m}_1$ represent the first and second elements of the trace.

[15]Nicolas Markey. Temporal logics. 2015. ffhal-01194612

[16]https://web.iitd.ac.in/ sumeet/slide3.pdf

$$
\begin{aligned}
\overline{m} &\models x = y \equiv x \text{ equals } y \\
\overline{m} &\models \neg\varphi \equiv \overline{m} \not\models \varphi \\
\overline{m} &\models \varphi \wedge \psi \equiv \overline{m} \models \varphi \text{ and } \overline{m} \models \psi \\
\overline{m} &\models \varphi \rightarrow \psi \equiv \overline{m} \not\models \varphi \text{ or } \overline{m} \models \psi \\
\overline{m} &\models \mathbf{G}\, \varphi \equiv m_0 \models \varphi \text{ and } \overline{m}^1 \models \mathbf{G}\, \varphi \\
\overline{m} &\models \mathbf{X}\, \varphi \equiv \overline{m}_1 \models \varphi \\
\overline{m} &\models \varphi\, \mathbf{U}\, \psi \equiv \overline{m}_0 \models \psi, \text{ or both } \overline{m}_0 \models \varphi \text{ and } \overline{m}^1 \models \varphi\, \mathbf{U}\, \psi \\
\overline{m} &\models \forall x \in \pi : \varphi \equiv \overline{m} \models \varphi[x/v] \text{ for all } v \in m_0(\pi)
\end{aligned}
$$

Figure 2.1: Semantics of LTL-FO+ [1]

LTL - Tally Keeping

Tally Keeping-LTL (TK-LTL) is a novel extension of First Order LTL (LTL-FO) proposed in order to go beyond the reductionist 3-valued judgement that temporal languages evaluating specifications come up with. For instance, consider the property where every open file must be eventually closed. It would be useful to count the number of execution steps that occur between the opening and closing of the file. TK-LTL allows users to craft formulae that evaluate numeric properties found in traces and come up with a quantitative answer instead of the usual True, False or ?. It can be used to count the number of specific formal patterns in a trace or to find out the duration between each such pattern. Other questions such as the longest, shortest and average time from open to close can also be accurately answered. The authors were motivated to work on this extension from the limitation of formal logic in specifying properties in the software industry. A quantitative verdict instead of the standard Boolean would provide better feedback to the developers. TK-LTL extends LTL-FO by introducing three sets of semantic structures. They are counters, quantifiers and filters. Counters count the number of prefixes in a sequence that evaluates to a given truth value or the trace index at which the condition holds. For example, $\mathcal{C}_\phi^\nu$ where ϕ is an LTL formula and ν ranges over truth values T,F or ?, returns the number of prefixes in the input trace where ϕ evaluates to ν. The syntax of TK-LTL also contains counters $\mathcal{D}_\phi^\nu$ and $\mathcal{L}_\phi^\nu$ where $\mathcal{D}_\phi^\nu$ returns the first prefix of the trace where ϕ evaluates to ν and $\mathcal{L}_\phi^\nu$ returns the last instance. The quantifier K can reason over parameterized events and filters which are functions that can be applied on maps to extract relevant data.

Some useful filters could be *max(), average().* For definitions, semantics and proofs, the reader can refer to [49].

2.2 A Review of Existing Trace Analysis Studies

We present a review of related work covering trace analysis techniques that extract useful performance metrics from software executions, patterns in the system traces themselves and performance analysis of software using tracing.

2.2.1 Detecting patterns in system traces

Vostokov's work features a categorization patterns divided into software trace analysis patterns and memory dump analysis patterns [28][50]. The former entails detecting and classifying patterns in the trace content itself whereas for the latter the objective is to analyze memory dumps and crashes to uncover performance problems. Most of these patterns apply to traces of the Windows system, with a few for MacOS and Linux. Vostokov uses concepts of Narratology [51] to approach the topic where he views software traces as narratives which can be classified into patterns and adapted for other systems as well.

Vostokov states that patterns can be either domain-dependent - such as patterns detected in Windows logs and network traces, as well as domain independent where analyses are independent of the hardware or the software running it. He then classifies these patterns into categories such as Vocabulary Patterns (deals with descriptions), Error Patterns, Trace as a whole, Activity Patterns, etc [52].

His work on crash dump analysis patterns deals with abnormal software behavior reflected in memory. Most of the patterns are detected in Windows using WinDbg. There are more than 200 patterns currently and more are being added and catalogued in the Encyclopedia of Crash Dump Analysis Patterns[27]. Some of these patterns include critical section deadlocks (threads in a critical section mutually waiting for each other and obstructing progress), unloaded modules (access violations at addresses belonging to unloaded modules), accidental locks (appearing in crash dumps) and wait chains (causal waiting relations between tasks and resources). He classifies these patterns into categories such as contention patterns, deadlock and livelock patterns, memory consumption patterns, process patterns, and so on.

Some of the performance patterns included in the memory dump analysis anthology [50] also feature in this thesis. For example memory leaks, thread starvation, critical section contention, and swarm of shared locks. We have included these performance patterns in the sections on Memory Leaks, Thread Swamping, Critical Section Bottleneck and Over-synchronization, respectively. We have also used the concept of an *affine thread* or thread affinity in one of our implementations. Although the performance patterns may be similar, but our method of detecting, extracting and formalizing these performance patterns is totally different from Vostokov's dump analyses. Our sources are also different. We extract these patterns from traces whereas Vostokov manually detects them from crash and memory dumps.

LaRosa et al. [26] designed a framework that applied data mining techniques to kernel traces. They successfully identified inter-process patterns in noisy trace files. In order to achieve this, they transformed the problem of kernel trace data mining to maximal frequency itemset mining. Window folding and window slicing were employed to aggregate trace events using their respective timestamps. Appropriate pre-processing techniques such as bit-packing and data filtering were also developed. Kernel trace data can be set to have interesting temporal characteristics as the scheduler can re-order the partially ordered execution. Therefore, the concept of frequent itemset mining comes handy. Window folding helped generate a series of parallel events by easing up the ordering requirement of events to include events at a certain temporal radius. The ordering requirement could not be too strict as the scheduler itself did not guarantee process order. Once a set of parallel events was obtained, window slicing was used to find out frequent parallel episodes which were temporally close. It converted a long sequence into a database. Events grouped together in a window formed a single sequence record in the event database. The authors employed fold and slice techniques in tandem to get an unordered database of parallel events which ultimately became a frequency itemset mining problem. Then, they looked out for maximal frequent itemsets so that the output was human-readable. These techniques were then used in a real-life scenario where the authors traced a GNOME stock ticker app with a known issue of generating expensive X programming system calls.

Côté et al. [19] applied a pattern discovery algorithm alongside creating an execution model on RT-Linux to detect scheduling problems. They used critical path analysis and scheduling information to achieve this. They also developed plug-in views for TraceCompass to support their analyses. The authors first let the user provide a list of ordered tracing events and selected a pattern which was used by the algorithm to find the jobs to be analyzed. The authors chose the MANEPI algorithm that extracted patterns which were more frequent than a given support threshold. This threshold could be selected by the user. A high threshold meant that the algorithm was faster as it skipped a majority of the detected episodes. The user could also mention the number of basic elements (an episode of one event type where an episode is a group of event definitions). An interface was provided to design or edit the pattern. This pattern was then loaded onto the execution model. Once the traces were loaded in they were displayed in specially designed views such as Comparison View (to compare different executions of the same job to spot irregularities), Time Perspective View (which showed the job durations as a function of their starting time so that the time distribution could be studied), Critical Path Complement View (to uncover significant thread dependencies) and the Extended Time View.

Khoury et al. [22] analyzed assembly traces using LTL-FO+, which is a first order extension of Linear Temporal Logic (LTL). The reason for choosing assembly traces was that it provided a detailed view into the applications functioning which higher level traces fail to provide. The authors verified security properties in assembly traces via run-time monitoring with BeepBeep being the monitor used. This technique was used to detect call sequence profiling, error conditions and malicious activity. LTL-FO+ had been shown to be appropriate for data aware properties. In addition to Boolean operators, temporal operators, LTL-FO+ added quantifiers such as EXISTS which can be used to access parameters inside events.The authors detected five properties in assembly trace to demonstrate the possibilities of temporal logic verification in assembly traces. The properties were Integer Overflow Detection, Call Sequence Profiling (used in Software comprehension and maintenance), Return address protection, Pointer subterfuge detection and Malicious pattern detection. The traces were converted to XML with a pre-processing script and then processed using BeepBeep.

Idris et al. [53] used Trace Correlation to achieve software comprehension and compare software versions. This helped developers to differentiate different software versions and implemented features as well as to estimate resources and requirements needed for future versions. The approach consisted of two steps. The first step was generating traces of the target features in the system in order to identify implementation differences via trace correlation. The second stage was source code (static) analysis to understand the changes. The authors introduced two novel trace correlation techniques. Before correlation, the traces which were generated from different versions of the same system were pre-processed to eliminate noise such as repetitions owing to loops and recursions and accessor methods. After that, similar patterns in the traces were extracted for comparison. These extracted patterns would determine if the traces exhibit similar behaviour. A threshold was set such that similarity can be established if the threshold is crossed. The result ranged from 0-1 with 0 indicating complete dissimilarity and vice-versa. The two metrics were the non-weighted trace correlation metric and the weighted trace correlation metric. The former compared two traces based on the proportion of similar extracted patterns that they share and the latter modified this metric by taking into account the frequency of these patterns. Choosing which metric to use depended upon the task at hand as these metrics resulted in different similarity scores.

Matni et al. [54] used an automata based approach to detect problematic behaviour via traces generated from the system kernel. They used state machine language to achieve the same alongside LTT (Linux Tracing Toolkit) to generate the kernel traces. A variety of problems such as excessive thread migration, disk swapping, locking problems and security issues could be detected using this approach. The overhead was minimal as well. The three classes of problematic behavior were security threats, software testing and performance debugging. The patterns were described via state machine (SM) language which is simple, expressive and domain independent. The state transition approach allowed easy generation of synthetic events. The state machine language lets one declare a state and all the transitions which emerge from it. These transitions have a name, a destination state, transition action and an optional argument list. If an expression evaluates to true then the transition is triggered and the transition action follows suit. The destination state can be defined in another state machine for simplicity. When it came to implementation, the authors designed automatons to detect chroot jail escapes, validate locking and check real-time constraints.

Tate et al. [55] presented a survey of data mining algorithms used for extracting patterns from a database. The authors performed a comparative study of algorithms like Apriori, Tree-Projection, FP-Growth, RARM, ASPMS and Eclat. The modes of comparison were storage structure, search technique, database scan times, advantages, drawbacks and execution time. The authors found FP-Growth to be the best algorithm to detect patterns based on execution time, advantages and drawbacks. It uses a FP-tree as a storage structure and uses divide and conquer as the search technique. The authors then used LTTng to generate traces, converted them into human readable format using Babeltrace and ran the FP algorithm on the trace file. The authors then used the algorithm to show uneven workload distribution across CPU cores.

Giraldeau et al. [16] extracted meaningful performance metrics from Linux kernel traces such as CPU, disk and network usage using the LTTng kernel tracer. They defined a trace as a series of ordered events with each event comprising of a timestamp, event type and event payload. The payload was an ordered set of event fields. Metrics could be recovered from trace metadata (which was used to declare event types) and system state dump (a description of system state at the beginning of the trace which can contain static and variable events with the former remaining unchanged during the trace duration such as system call tables and the latter being subject to change such as number of active processes). Scheduling events could be used to recover CPU usage metrics. If no process ran on a CPU, then the kernel made the *swapper* process run on the CPU and the CPU was considered to be idle. Total CPU time could be calculated as the time when the CPU was not idle. Tracepoints related to memory usage could be used to derive memory usage metrics. *vm_state.vm_map*, for instance stores the system memory usage at the beginning of the trace. *page_alloc* and *page_free* indicates allocation activity. Similarly, file manipulation activity could be monitored via open, read, write and close system calls. Network usage metrics could be derived from network event payloads and individual packets could be traced. Block I/O scheduling could be observed via block level tracepoints. Request additions, merging and completion times indicated block I/O activity. High disk offset could affect performance and could be calculated via *fs_issue_rq* event. Individual requests could also be mapped to the inode via *bio_remap* event.

Daoud et al. [21] carried out a stateful analysis of block device performance by introducing a framework to compute meaningful storage performance metrics from low level kernel trace events. The trace events were fed into a model generator which then computed the necessary metrics and fed them into a visualization system. They introduced an LTTng module based on Kprobes and introduced three static tracepoints into the kernel. All the required computations were done when the trace was read for the first time. A state history database stored the metrics and then generated the statistics for an arbitrary period in the trace. An attribute tree and a state history tree wer used as data structures to store attributes such as process name, PID, CPU etc. They also developed two views for TraceCompass - latency distribution view and latency scatter chart. The requests view showed the contents of the waiting queue as well as the dispatch queue. The metrics computed were disk utilization, latency (time required by the disk to process an I/O request), seek time (time taken by the mechanical head to reach a sector) and queue length. Disk utilization is the amount of time the disk is busy where business refers to at least one I/O request being processed.

Francisco de Melo jr. et al. [56] introduced an automatic solution to group metrics using a call context tree data structure and find performance issues. The first step was to trace the program execution using LTTng including performance metrics such as cache-misses, page faults and scheduling switches using perf counter tools in Linux. Then, the trace was divided into segments containing different instances so that they could be compared. File openings could be delimited by sys_open and sys_exit. Comparable information was stored in ECCTs (Enhanced Calling Context Tree) where individual nodes represented a call and its related information was stored in the nodes. This also enabled offline analysis. Existing Linux kernel events or LTTng UST probes included in the code could be used to identify the start and endpoints of each execution. Then, the number of groups of execution was measured using the elbow method which compared the sum of the squared errors (SSEs). Heuristic evaluations could also compare different SSE values. Several runs of classifications wer made and the one with the least number of SSEs was extracted. Then association rules wer run to find the root cause. The authors used this technique to optimize the cache in a server application and to detect regressions in OpenCV.

Rezazdeh et al. [57] proposed a multi-level critical path analysis algorithm to capture thread dependencies causing lock contention and present tools and visual analyses in order to visualize lock contentions. They recorded traces from the kernel as well as the user-space in order to achieve this as spin locks were implemented in the user space. The authors instrumented Pthread libraries using the LD_PRELOAD technique so that applications could be instrumented non-invasively. They instrumented mutex lock requests, acquisitions alongside spin-lock and semaphores. Wrappers replaced the lock functions which contained the tracepoints and called the replaced functions. The authors developed a stateful model with the states being stored in the state database system. The trace events wer converted into state values using the State Provider. The authors then extended the critical path algorithm in Giraldeau et al. [58] to analyze user level events as well. Their modified algorithm built an execution graph using locking data which was a two dimensional sparse matrix. The horizontal edges showed the thread's state changes whereas the vertical edges showed links between threads and their blocking dependencies. The critical path was extracted recursively by replacing the waiting edges of a thread with the edges of the waking thread. The authors developed views like the wait-block analysis view which displayed locking dependencies and helped detect bottlenecks. The critical flow view showed execution dependencies between participating tasks. Using theses analyses, the authors evaluated contention in the Apache web server and zero in on OPCache which caused contention in cache memory.

Doray et al. [23] introduced a framework called TraceCompare that automatically detected execution differences between several executions of the same task at the user-space and kernel levels. Their comparison algorithm took into account all factors that affected performance such as on-CPU and off-CPU wait times and process interference. The problems detected were also mapped to the source code. A data model was constructed which would summarize all the performance characteristics of task executions alongside algorithms that built a database from the traces. The authors introduced a data structure called enhanced calling context tree (ECCT) which represented the performance characteristics. It represented all the latency that affected task duration as well as the captured call stacks for each thread which connected the latency to the task. Existing Linux events wer used as delimiters along with statically inserted LTTng-UST probes. TraceCompare used a critical path algorithm developed by the authors themselves in order to find all the latency dependencies of a task. Once the critical path had been computed, an ECCT was generated and global execution metrics calculated. After that, the ECCT was stored in the database for comparison. The metrics could be context specific like page faults or memory consumption. Once the execution ECCT had been saved to a database, comparisons had to be made in order to determine problems. Filters, flame graphs and execution lists were used for this purpose. Filters were used to compare two groups of executions with reference to one or more metrics. Differential flame graphs were generated from the ECCTs and showed call stacks of every thread in the chain of dependencies.

Call stacks were useful as it helped developers quickly find the area of the source code responsible for the latency. The authors used TraceCompare to detect disk contention in a server application, CPU contention in a real time application (by comparing slower executions with the normal ones) and lock contention in MongoDB.

2.2.2 Performance Analysis using Tracing

Fournier et al. [24] introduced an approach for analyzing blocking in multi core applications using LTTV Delay Analyzer in order to see how the time elapsed could be divided into its constituent waiting components. The authors instrumented scheduling changes (preemption, blocking), process wake ups, IRQ entry/exit, softIRQ entry/exit and process forks. The architecture comprised of two state machines - Control Flow State Stack and Working/Interrupted/Blocked State. The former is a stack of execution states that the control flow passes through such as running in user-space, in another interrupt or in a system call. The stack helped deduce the control flow states while reading the trace. The WIB State is a higher level abstraction that can take values like Working (when the task is executing), Interrupted (if an IRQ interrupts the task) or Blocked. The framework also required a state hold-back mechanism as the information associated with some events came after the event had taken place. The LTTV Delay Analyzer produced reports based on the state machine which helped developers analyze latency.

Ezzati-Jivan et al. [25] used system level tracing to extract a dependency graph of a task in its critical path to show its breakdown among all interleaving threads and resources. The trace files were processed to construct a state database and then a novel algorithm was used to construct the dependency graph from the state database. Finally, multiple dependency graphs belonging to different executions were analyzed, grouped and merged to find out the root cause of the latency. The authors used LTTng to collect the traces. The state database comprised of states like running, runnable, preempted and blocked. When it comes to waiting dependency, there can be two types of dependencies : direct and indirect. Direct dependency is when a thread explicitly waits for another thread to wake up the former. Indirect dependency is when threads compete with each other for accessing a resource without any direct causal relationship between them. When multiple depGraphs wer generated corresponding to the execution traces, they wer grouped according to certain parameters such as number of page faults, number of disk accesses, etc. The authors used k-means clustering to group these graphs so that the outliers could be studied. The root causes could be found out by comparing graphs from different clusters such as normal executions versus abnormal ones. Bold lines and dashed lines were used to visualize similar calls and dissimilar calls respectively.The authors used this approach to detect lock contention in an Apache server and CPU contention in a real time task.

Nair et al. [59] introduced GAPP, which is a tool that identifies serialization bottlenecks in parallel Linux applications from execution traces. It makes use of the eBPF framework as it utilizes its probes to trace context switching events. Serialization bottlenecks wer detected using the thread criticality metric. A thread with a high criticality metric suggested serialization which could be a cause of bottleneck. The criticality metric was calculated by weighing every running thread's execution time by the number of active threads. This metric was calculated at the end of every execution timeslice. The criticality metric was the sum of contributions from every interval that took place during the timeslice. The authors used eBPF framework to keep track of all active threads. Whenever the parallelism for a given timeslice went below a certain threshold, the tool recorded the stack trace so that developers could exactly pinpoint the bottleneck in the source code. Lyons et al. extended GAPP by augmenting the stack traces generated by the tool and classifying the type of bottleneck detected. They also tracked kernel level synchronization calls (futexes) to analyze locks which were critical. This extension then provided a summary of the most critical individual file and synchronizations. All this was done without needing source code instrumentation[17].

Dean et al. [60] introduced PerfCompass, which is an online anomaly fault localization tool designed to categorize global faults and local faults in IaaS (Infrastructure as a Service) clouds. Faults having a global impact can affect almost all the threads of an application when it surfaces whereas faults with local impact will only affect a subset of the threads. The tool uses lightweight system call tracing to continuously record system calls. Fault localization was performed using four main steps. Initially, the trace was segmented into groups of closely related system calls called *execution units*. These units were then processed to extract a set of fine-grained fault features (for

[17]https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-ug-projects/Davies-LyonsA-IO-bottleneck-detection-in-the-Linux-kernel.pdf

example, threads that were affected by the fault and how quickly the fault propagated). After that, these features were used to categorize local and global faults and the root cause was determined to be either an external environment issue or an internal software bug. PerfCompass also detects the top affected system calls and ranks them. The tool has four components: execution unit extraction, fault onset time identification, fault differentiation and affected system call ranking. The fault is initially quantified as global or local. Then, PerCompass suggests if the root cause is external or internal and ultimately provides a ranked system call analysis for developers and system administrators. Suitable metrics are calculated to answer questions such as the threads affected by the fault and how quickly they are affected. The metrics are *fault onset time* and *fault impact factor*. The former answers how quickly a thread is affected by a fault and the latter computes the percentage of threads affected by the fault. Another metric, fault onset time dispersion metric, quantifies the differences among the fault onset time values. Some of the external faults detected were CPU cap problem, memory cap problem and packet loss whereas some of the internal faults detected were deadlocks, infinite wait bug and data flushing bug.

Alam et al. [61] categorized, detected and diagnosed synchronization issues using SyncPerf. It processes information based on call-sites, lock variables and thread types to identify problems. It also provides a secondary analysis tool that collects information from critical sections to perform root cause analysis. The paper divided synchronization issues into 5 categories: improper primitives, improper granularity, over-synchronization, asymmetric contention and load imbalance. The study found out that multiple root causes could cause a single symptom. Two of the synchronization issues in this paper (Asymmetric Contention and Excessive synchronization) feature in our thesis. However our detection method differs from those of the authors. The authors showed that performance problems could also occur in locks that were not excessively acquired or highly contented. The length of a critical section could slow down execution even if the lock guarding it was not excessively acquired. Locks which were excessively acquired could cause overhead even though they were not be highly contended. SyncPerf provides two tools: a detection tool and a diagnostic tool. The detection tool uses lightweight profiling to detect issues and diagnose root cause for asymmetric contention, try lock failures and load imbalance problems. The diagnosis tool checks memory accesses in critical sections using binary instrumentation. This analysis is somewhat heavy-weight and is used only when the detection tool cannot diagnose problems easily. Data collection is performed by instrumenting *pthread* synchronization primitives. It uses indirection to avoid lookup overhead and a per thread data structure to avoid cache-coherence traffic as using a global hash table would induce significant overhead. It also collects call-site information for synchronization operations for developers to localize the issue in the source code. After the analysis is completed, SyncPerf generates a report in two steps. The first step is to combine all thread-wise data of a synchronization operation together and check for lock contentions, acquisitions, and try lock failures and report potential problems. The second step is to check call-sites with similar lock behaviour.

Orero et al. [62] designed a PoC (Proof of Concept) in order to remedy writeback cache latency and lockups. Linux uses a writeback mechanism in order to commit to disk all the dirty pages at once instead of writing files to disk every time a change is made. The changes are, meanwhile, stored in the cache which is located in the main memory. This writeback mechanism caused cache flushes which were responsible for considerable latency and lags in the presence of processes that wrote to the cache heavily. Writing a lot of data to cache forced it to 'evict' pages which have not been used in a while and thus initiate writeback. In order to ensure that the cache doesn't fill up quickly, the kernel 'throttles' the task. However, the authors found out that this throttling was indiscriminate and many tasks which were not write-heavy ended up being throttled as well. In order to study this phenomena, the authors decided to interface with the kernel tracer, simulate heavy write behaviour and analyze the results. They used the *global_dirty_state* tracepoint to get a snapshot of the cache state. Then, they wrote a tool using NodeJS which would plot all the important variables in real-time such as number of dirty pages, global dirty states and thresholds. Then, the authors setup a variety of loads including offending (write heavy) and innocent processes. After the experiments were run, the data was analyzed using a Jupyter, NumPy and MatPlotLib setup. The authors realized that the long pauses could be mitigated by some extent by simply using the BFQ scheduler. The PoC was designed by creating a daemon which would monitor the tasks in the system, identify the offending processes based on a criteria and lower their I/O priority so that they do not affect other tasks.

Matias et al. [63] have used the Linux kernel instrumentation to study software aging. Software aging occurs in continuously running software systems which may lead to failures owing to aging related reasons. These problems could be data inconsistency, numerical errors and saturation of resources. The authors performed two types of experiments in order to demonstrate how kernel instrumentation could be leveraged to measure aging related effects. The first one involved memory leaks and the second experiment involved memory fragmentation. In order to detect memory leaks, the authors wrote a SystemTap script to count the number of times the Socket Buffer (SKB) allocation and free functions were called and displayed the information every second. If the difference between allocations and frees was more than zero then it suggested a memory leak. In case of memory fragmentation, the authors used the *mm_page_alloc_extfrag* tracepoint in order to determine the degree of fragmentation. They used a SystemTap script to determine if the allocation order was greater than the fallback order. The authors then devised several workloads and monitored the fragmentation tracepoint. Some of these workloads belonged to a set of scripts called the Linux Test Project (LTP). Based on their experiments the authors found out that memory allocation operations and filesystems showed the most fragmentation.

2.3 Discussion

The literature review delves deep into the different trace analysis approaches taken in previous work. As per our knowledge, no comprehensive study exists on detecting, extracting, cataloguing and formalizing performance analysis patterns from trace files. Vostokov's work comes close, however, his collection of trace analysis patterns deal with patterns present in the traces themselves alongside memory and crash dump analysis. The patterns that we present here represent various performance issues and avenues of optimization in the Linux operating system. We also use temporal logic, run-time monitoring and EASE scripting to detect these patterns which has not been attempted before. We also leverage the power of eBPF tools and bpftrace scripting to generate traces which are in turn fed to run-time monitoring frameworks to detect performance anomalies. We hope that more patterns will be added to this catalog as sophisticated tracing frameworks are developed along with stream processing tools.

Chapter 3

A Catalogue of Performance Analysis Patterns

3.1 Introduction

In this chapter, we present the catalogue of performance analysis patterns which can be classified into two types (Figure 3.1). The first type of classification is contingent upon the scope at which these patterns occur generally. The scope may be an entire Linux subsystem, the hardware level or the application level. The second type of classification is based on the complexity of the patterns themselves. Some patterns such as memory fragmentation and TCP packet drops are simple enough to be detected and formalized by temporal language and run-time monitoring. Other patterns are more complex and needs scripting using a trace analysis framework or stream processing frameworks.

The categorization of performance analysis patterns is as follows:

- Patterns falling under the scope of an individual Linux Subsystem (Figure 3.2)
  - Memory Management
  - Thread Scheduling
  - Disk Scheduling (I/O)
  - Network Stack

• Patterns falling under the scope of a context (Figure 3.3)

  ◦ Synchronization Context (Regulating resource access in multi-threaded applications)

  ◦ Application Context

  ◦ Hardware Context (Hardware Performance Counters)

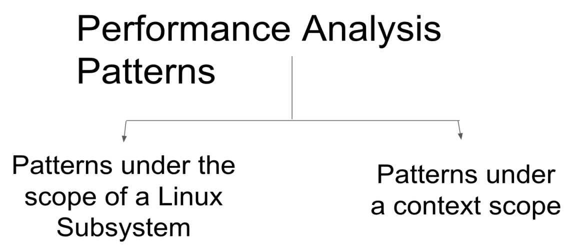


Figure 3.1: A broad classification of Performance Analysis Patterns

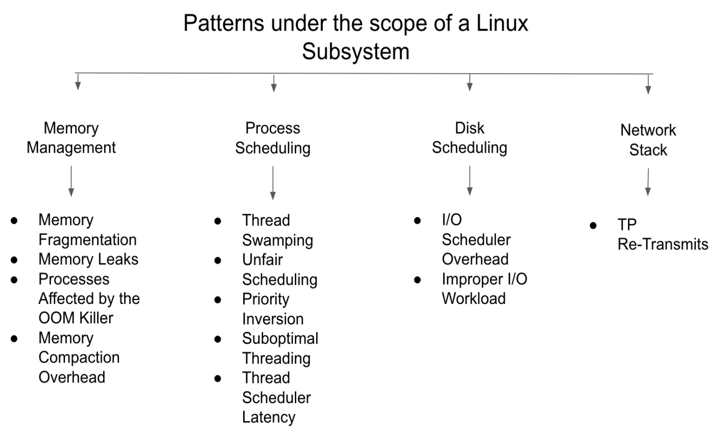


Figure 3.2: Patterns under the Linux subsystem

The individual patterns are described in the following template: a brief summary which describes the objective of the pattern, how it is situated in its subsystem or context scope, how it can be detected, its formalization (if applicable) and the data source such as tracepoints or kprobes that can be used to detect the pattern.

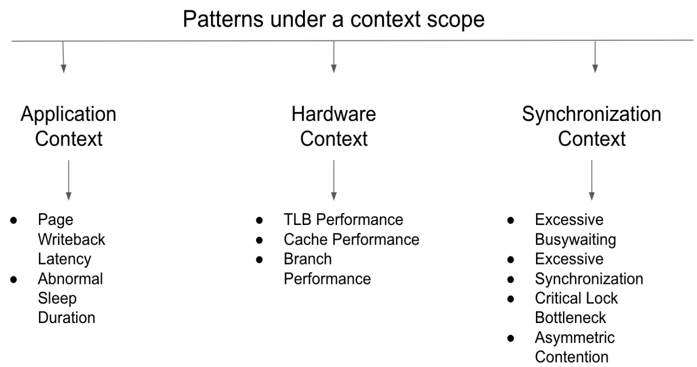


Figure 3.3: Patterns under a context scope

3.2 Memory Management

3.2.1 Memory Fragmentation

Summary

Memory fragmentation is a phenomenon that affects dynamic memory allocators in operating systems and leads to loss of efficient memory utilization. Fragmentation can be of two types: internal and external. Internal fragmentation can be said to occur when a larger-than-required free block is allocated by the allocator instead of allocating the exact requested block size. Linux makes use of the slab allocator in order to tackle this feature [64]. External memory fragmentation occurs when a high order memory allocation request fails owing to the fragmentation of memory into smaller blocks even though there is enough free memory in totality [65].

Once reason why memory fragmentation and memory leaks (featured in the next section) must be understood and dealt with accordingly is because these memory related defects contribute to software aging [66]. Software aging is a phenomenon where long running software systems witness gradual performance degradation [67]. Aging software may not be able to keep up with client requirements and their performance may dip considerably causing the software system to respond slowly, crash or fail altogether.

Since Linux uses virtual memory [68], fragmentation can be difficult to observe as non contiguous memory blocks can be arranged via page tables in the form of virtually contiguous blocks.

However, the problem starts when physically contiguous memory blocks are required. Memory fragmentation can occur both in user-space and in the kernel-space. Memory fragmentation in user-space affects only the user applications, however kernel level fragmentation can affect the whole system [69], [66]. Processes need memory allocators to handle dynamic memory allocation requests using functions such as malloc(), realloc() and free(). These functions belong to a standard library which is linked to applications. When a memory request exceeds the available memory in the heap (the part of the main memory pre-reserved for the process which can dynamically grow and shrink as per program requirements) the allocator asks for more memory from the OS [70]. The allocator uses the system call *sbrk()* to provide new arenas (sub-heaps) for the caller process [71]. We can use LTTng to trace these system calls and check the amount of memory that the process has asked for. If the available heap memory is greater than the amount of memory requested, then we can be sure that fragmentation has occurred in the heap.

Detection

Some services and I/O devices require contiguous memory blocks [72]. If the device does not have IOMMU (which allocates contiguous buffers to non contiguous physical memory blocks) then fragmentation can be a real concern. In this case, we make use of the *mm_page_alloc_extfrag* tracepoint to detect fragmentation status. The number of times this event occurs shows how severe the fragmentation is. We have access to two variables in this tracepoint: *alloc_order* and *fallback_order*. If the latter is lesser than the former then fragmentation has reached unhealthy levels and further higher order allocations would not be possible. Matias et al. [63] have also performed similar experiments on fragmentation as a yardstick of software aging using this tracepoint.

3.2.2 Memory Leaks

Summary

Memory leaks occur when allocated memory is not freed after usage. It can also happen when the allocation pointer is deleted which renders the corresponding block of memory unusable. Memory leaks can cause many performance problems as they lead to spikes in paging activities, thrashing and an eventual memory shortage. Leaks are especially prone to occur in C/C++ programs since programmers can allocate/de-allocate memory dynamically [73].

Detection

There are several tracing based tools to detect memory leaks such as Valgrind's memcheck[1]. It basically keeps track of all calls to memory allocating functions like *malloc()* and reports instances where memory is not freed after usage and classifies them into leak categories such as "Still reachable", "Definitely lost", "Indirectly lost" and "Possibly lost". However, the use of Valgrind's memcheck can slow the application.

Memory allocator functions such as *malloc(), calloc(), realloc()* can also be traced directly using eBPF and/or perf and their stack traces studied for leak code paths. However, this might have a high overhead as these functions can be called many thousands of times per second [74]. A BCC tool *memleak*[2] also exists which checks for allocations that have not been freed in a given time interval. It hooks on to memory sub-system tracepoints such as *kmalloc* and *kfree*. It also attaches probes to allocator functions.

Memory leaks can be very common in applications and are easy to replicate. Tracing allocator functions is one way to detect these leaks. Memory leaks can be formalized as a specification using LTL-TK. A memory leak occurs when allocated memory blocks are not freed. Therefore, every allocated memory block should have a corresponding event where the block is freed. We use this specification written in LTL-TK:

$$(\exists_{>0}(\mathcal{C}_{\phi}^{\top} - \mathcal{C}_{\psi}^{\top})) \rightarrow \mathbf{F}log$$

where ϕ denotes a *malloc()* operation and ψ denotes a *free()* operation. We use the existential quantifier $\exists$ to count the number of times *malloc()* operations exceed *free()* operations and log it when the count is greater than 0. However, this can only be an indication of a leak. It is not abnormal for memory allocations to not have a corresponding de-allocation. Hence, this is an undecidable pattern where a complete and correct solution is not possible. Advanced tools can

[1] https://valgrind.org/docs/manual/mc-manual.html
[2] https://github.com/iovisor/bcc/blob/master/tools/memleak.py

present call stacks where allocated memory has not been freed and the rest is up to the user to determine if memory has leaked.

Therefore, detecting memory leaks is not a trivial task. It can slow down applications as memory allocations and de-allocations occur many times per second. Once memory which has not been freed after allocation is found, a call stack analysis is conducted to find out the code-path where the leaked pages originated from.

3.2.3 Tracing Processes Affected by the OOM Killer

Summary

Every process has its own *address space* containing all the addresses that the process may legally use. Sometimes, processes do not require all the addresses included in their address space and it is possible that some addresses may never be used. In order to utilize free memory efficiently, the Linux kernel employs a dynamic memory allocation technique called *demand paging* where page frame allocation is deferred till the last moment when the process actually 'touches' or utilizes the memory it has been allocated. If the process tries to access an address that is not present in the RAM, a page fault exception occurs and then the page frame is allocated [70]. Since, the Linux virtual memory can be much larger than the actual physical memory [75], it may lead to a situation where the kernel commits memory greater than the actual physical memory of the system.

The kernel over-committing memory does not generally cause system slowdowns. However, if more and more processes actually start using the memory allocated to them then the system will find itself in a critical situation where it will be rapidly running out of free memory as the processes would need more memory than the available physical memory. The kernel is now unable to free memory as the disk caches are already depleted and swap memory is full. This could freeze the entire system and needs to be resolved as soon as possible. When the kernel Page Frame Reclaiming Algorithm (PFRA) is unable to allocate free page frames, it calls the *out_of_memory()* function which in turn, selects a *sacrificial* process amongst the other running processes which must be killed off in order to reclaim free memory critical for the system to function [70]. Before that, the *out_of_memory()* function performs a series of checks to validate if the system has indeed run out of memory such as checking for available swap space, the number of process failures in the last 5 seconds and so on [68]. Once all these checks are passed, the *oom_kill()* is called. This function has two tasks - to select a process which must be killed off to reclaim memory and to call the *oom_kill_task* process a.k.a the Linux OOM(Out of Memory) Killer which kills off the chosen sacrificial task.

Selecting the sacrificial process is carried out via *select_bad_process* which assigns a badness score to each process [70]. The higher the badness score, the greater the chance to be killed off by the OOM killer. This score is dependent upon certain factors such as short-lived processes hogging a large amount of memory and making sure that the minimum number of processes are killed. This situation is, obviously, not ideal. However, it is a sacrifice that must be made for the sake of overall system stability. Considering all factors, the OOM killer sets an OOM score for every task. This score can be manipulated to ensure that certain tasks are not killed off under any circumstances or vice-versa - where a certain task is chosen to be killed off first whenever free memory is critically low. The OOM score of any process can be easily found out via the /proc interface using the task's process ID. The OOM killer can itself be disabled. However, it must be done with caution. It can also be explicitly called by the user.

The OOM killer checks the adjusted OOM score of every process before going ahead with killing the process. The *oom_adj* score is scaled from -17 to 15 with the positive scores implying greater liability to be killed off by the OOM killer [76]. Hence, if the oom_adj score is -17, it implies that the task will never be killed off by the OOM killer as it will have a score outside the acceptable range. In order to check why a particular task has been killed by the killer, the user can refer to system logs.

Detection

The OOM killer function can be traced using the EBPF infrastructure in the Linux kernel with the help of BCC/bpftrace tools and scripts. Tools based off dynamic EBPF tracing to detect OOM killings are also available in the market such as Instana's Crash Detector [3]. Brendan Gregg has also created a bpftrace tool - *killsnoop* [36] for the same purpose which can be found in the bpftools suite

[3]https://www.instana.com/blog/solving-the-out-of-memory-killer-puzzle/

which can be installed in the system. Using bpftrace scripting, kprobes can be used in order to trace out-of-memory process killings. In this case, the kprobe will trace the oom_kill_process function, and with the help of powerful bpftrace builtin variables, a wealth of information can be obtained in real-time such as the sacrificial process as well as the process that initiated the OOM killing. A bpftrace script can be written which hooks into the oom_kill_process function and a real-time trace can be generated as well which can be fed into run-time verification tools using specification languages. The details of the sacrificial process would be found in the struct oom_control which forms an argument to the oom_kill_process function. The process that triggers the OOM killing can be fetched via built-in bpftrace variables which would be able to access the said details from the kprobe itself.

3.2.4 Memory Compaction Overhead

Summary

As Linux supports virtual memory, the translation of virtual memory address to its corresponding physical memory address can be sped up by using the *Translation Lookaside Buffer* [77]. This ensures that the master tables in the main memory do not have to be referred which can save a considerable number of CPU cycles. If the master table has to be referred, then a TLB miss occurs which may have as significant an impact as CPU cache misses [78]. In order to increase the rate of TLB hits, modern processors support large page sizes which can be as large as a gigabyte. However, the problem is that it would require the physical memory for the page entry to be contiguous. This is where memory compaction comes into the picture. Memory compaction is a process which ensures the presence of fewer, but larger contiguous, free memory blocks [79].

With time, memory tends to fragment as the OS allocates blocks of memory to many different tasks and large contiguous blocks get split into smaller and smaller blocks of memory. If the system is using hugepages then the problem is compounded as hugepages need large contiguous blocks of memories. Gorman et al. [79] show how previous techniques used (such as memory reclaim) to curb fragmentation nullify the advantages of using huge pages and introduce a memory compaction algorithm which would ensure that contiguous memory allocations are possible.

The compaction algorithm features two scanners. The *free page scanner* moves from right to left looking for MOVABLE pages and storing the free pages in a private list. The *migration scanner* moves from left to right and looks for the least recently used (LRU) pages which have a higher chance of being migratable. Compaction ends as the two scanners encounter each other. After that, the free pages are moved to the left whereas the allocated pages are coalesced together to the right to allow for the creation of a contiguous chunk of free pages. This process can be triggered manually via the */proc/sys/vm/compact_memory* command.

In kernel versions 5 and above, proactive compaction is introduced which performs memory compaction in the background. The problem with on-demand compaction is that it compacts only the required memory so that a single page is available of the required size. This can hurt applications that need a large number of huge pages in a small amount of time [80]. Sometimes, the compaction process loops several times in order to achieve a suitable compaction degree and can cause high CPU utilization and CPU spikes. The latency can also be compounded if the system is trying to directly reclaim memory at the same time.

Detection

There are several kernel tracepoints which can be hooked into in order to provide compaction statistics and thus determine if compaction is responsible for slow workloads. There is also a BCC tool *compactsnoop* [36] that provides compaction statistics. For tracing compaction latency we wrote a simple bpftrace script that traces the compaction time and outputs a histogram according according to the latency. The tracepoints that we used are *tracepoint:compaction:mm_compaction_begin* and *tracepoint:compaction:mm_compaction_end*.

3.3 Thread Scheduling

3.3.1 Thread Swamping

Summary

Computing involves a great deal of resource management. Resources can be shared in time and space. This act of sharing is known as multiplexing. A system having a single CPU running multiple programs must ensure that each program gets to run on the CPU for a certain time period. The CPU scheduler is responsible for determining how CPU time is shared. When a CPU is running a task, it might need to process tasks of higher priority such as hardware interrupts. Instead of stopping the execution of the task, multitasking ensures that the state of the program is saved so that it can resume processing later. This process is known as a *context switch* and it requires registers and memory maps to be loaded, various tables and lists and the memory cache to be updated [81]. If a program needs to access a peripheral, or to wait for a request to be completed, it can relinquish CPU access so that the CPU can switch to another task instead of waiting for the program to resume. For optimal performance, CPU timeslice management between multiple programs must be efficient.

In this performance pattern we shall be examining a specific case where application performance suffers because its thread spends a lot of time waiting for CPU time slices. This is because another application, that is running concurrently is swamping the CPU with too many threads. This can be categorized as an external fault because it stems from interference from other co-located applications. A more complex case is investigated in 3.3.4 where we witness threads of a single application swamping each other; holding up progress.

Detection

We shall be using two metrics - TD and TCD, in order to analyze this pattern.

Thread duration [TD]: The total duration of the thread in the trace

Thread On CPU duration [TCD]: The total duration of time the thread gets to run on CPU (Ignoring periods when the thread voluntarily yields to the scheduler)

Execution Ratio:

$$\frac{TCD}{TD}$$

A higher Execution Ratio implies a lesser degree of thread swamping.

A sched_switch is a Linux kernel event that is triggered when a new process executes on the CPU. It represents a context switch. The contents are as follows: *prev_comm=Thread1,prev_tid=57062,prev_prio=20 next_comm=stress, next_tid=57079,next_prio=20*

We can get the thread name and ID of the thread that is being stopped and the next thread that will run on the CPU. We also get the priority of the thread. In this particular case, Thread1 is our Java application thread which is being blocked so that the stress thread can be executed on the CPU. It is also important to note the *prev_state* parameter which tells us the state of the thread when it was interrupted such as TASK_RUNNING, TASK_INTERRUPTED and TASK_WAKING. In this scenario we are only interested in the state TASK_RUNNING because this suggests that the thread was blocked while doing useful work. Not all preemptions are forced. Sometimes threads are blocked when they are waiting for I/O or data as well and they yield to the scheduler so that another thread can run in its stead.

3.3.2 Unfair Scheduling

Summary

Linux is a multitasking operating system. This means that several processes can run on a single processor which has finite processor time. The process scheduler is responsible for ensuring that multitasking is possible by governing which process can run on the CPU and for how long. The scheduler must ensure that as long as there are runnable processes, a process must be running on the CPU. When a process runs on the CPU, several processes would be waiting to run next. The scheduler decides which processor would be allotted the next CPU timeslice. Most modern operating systems implement preemptive multitasking where the scheduler has the power to preempt (involuntary suspension) a running process as well. In order to maintain system responsiveness and throughput, scheduler behaviour can be dictated via policies. A process is scheduled to be executed based on its *priority* and *nice* value. Thread priority is simply how important the thread is. The nice value, on the other hand, is the degree of accommodation that a thread is willing to grant other threads. These values can be changed by the scheduler anytime to account for changes in execution conditions. [82].

Modern multiprocessor systems can have multiple workloads running in parallel - many times in conflict (considering limited system resources) and often supervised by different users. This can create complications for the scheduler when it comes to *fairness.* The idea of fairness when it comes to process scheduling implies that all runnable threads in parallel are allocated a fair share of the system resources. However, this can be difficult to achieve in practice.

In order to achieve a degree of fairness in task scheduling, the scheduler takes in to account several load metrics. One such example would be per-entity load tracking (PELT) where individual scheduling entities such as a process or a process control group is tracked instead of tracking individual per-cpu run queues [83], [84].

The scheduler has a load balancing algorithm that is tasked with placing tasks on CPUs in such a way that the overall throughput of the system is stable and none of the CPU cores are overloaded. It monitors the system periodically in order to determine which tasks have to be migrated in order to ensure fair distribution of system resources. However, this can be difficult to achieve as the algorithm has to take into account asymmetrical CPU topologies and outdated load metrics which need to be updated [85].

Biased statistics can show a current group as overloaded when in reality it is not and the scheduler initiates a migration as a result. This would result in a task placement that is not optimal and affect system performance. This could lead to situations where certain CPU cores are overloaded despite the presence of one or two idle cores. The presence of out-of-date heuristics can also muddle up the process. An example would be the removal of the *runnable_load_avg* signal in favor of *runnable_avg* by Vincent Guittot (whose session at the 2020 Power Management and Scheduling in the Linux Kernel Summit (OSPM) inspired this performance pattern [4]) to address a problem seen in capacity tracking during task migration. Another scenario where scheduling can be unfair would be a workload profile that inherently cannot be balanced. For example a 9 threads running on an 8 core system, or 3 tasks running on 2 CPUs which would be a mismatch between load granularity and

[4]http://retis.sssup.it/ospm-summit/

CPU topology. Sometimes the load balancing results in a single task being unfairly singled out for migration which would result in the task getting less CPU time and conversely, other tasks getting more than their fair share of allotted CPU time-slices. This 'unfair' task choice occurs because of a synchronization between the scheduling period and the load-balancer which results in the same task waiting to run. This imbalance can be fixed by tweaking the load-balancing intervals. Active load-balancing should also be reduced as they invariably end up migrating tasks that should not be removed after failed attempts at load-balancing [86].

Detection

Scheduling problems can be detected using tracing. There are several tracepoints in the Linux kernel related to scheduling which can be activated such as *sched_migrate_task* and the run-time execution traced. Trace Compass has several views such as the Resource view which can help developers identify tasks that were repeatedly and/or unfairly migrated. On CPU time can also be calculated using an EASE script.

3.3.3 Priority Inversion

Summary

Multi-threaded applications are often synchronized to ensure proper access to shared resources [87]. In such a scenario it is critical to ensure that the threads are sequenced in the order of their individual priorities. This is known as *priority scheduling* [88]. Individual threads have priorities assigned to them based on their importance. The idea is to make sure that higher priority threads get more CPU time slices to run. A good example of priority assignment heuristic would be to assign high priorities to threads that need to respond in a short time. Therefore, priority scheduling ensures that the system executes the task with the highest priority first.

However, this does not always happen. Circumstances leading to the violation of priority scheduling - such that a thread with a high priority is blocked from accessing a shared resource by a thread of lower priority, are referred to as *priority inversion* events[89]. This happens when tasks of different priorities contend for shared resources. Although it is not possible to completely eliminate priority inversion, it is important to make sure that the correctness of the application is not affected.

Priority inversion can be either *Bounded* or *Unbounded* [90]. A bounded priority inversion is a scenario where a higher priority thread is blocked by a lower priority thread for a bounded time after which access to the resource is guaranteed. An unbounded priority scenario, however, is when the high priority thread is blocked indefinitely by a lower priority thread which can lead to deadlock and subsequent system unresponsiveness. Priority inversion cannot be completely eliminated, however it can be minimized. There are several well documented strategies to avoid priority inversion such as *Priority Inheritance Protocol* and disabling interrupts [91].

Detection

Priority inversion can be detected from application traces loaded onto Trace Compass. Having said that, programmers do not usually go looking for instances of priority inversion unless there is a good reason to do so. A common symptom of unbounded priority inversion is threads slowing down or being preempted for abnormally long times. Trace Compass has a critical path view which can be used to isolate the said thread and find out whether it has been blocked by a thread of a lower priority. One can also find the precise time during which the thread was blocked.

The *sched_switch* event stores the priorities for the previous and next events in the fields *prev_prio* and *next_prio*. This priority information is crucial to understanding instances of priority inversion. Using this information in the context of an isolated thread in the Critical Path View will help programmers quickly understand the root cause of performance issues as they realize that a priority inversion scenario has occurred and they can deploy strategies to fix it.

3.3.4 Sub-Optimal Threading

Summary

Thread level parallelism is often used to take advantage of modern multi-core processors in order to eke out a greater performance boost such as improved responsiveness and better throughput. Processes can spawn multiple threads that divide tasks among themselves in order to achieve a high degree of parallelism [78].

It is important for applications to strike a right balance when it comes to the number of threads being used. If an application uses a single thread, then it cannot take advantage of multi-core processors, thus leaving processor cores unutilized. However, using too many threads can produce its own set of unique challenges. Researchers [92], [93] have shown that using an excessive number of threads can make the application run slower owing to contention of shared resources. Multi-threading can also be very difficult to debug and is prone to race conditions [94]. Too many worker threads can swamp other application threads as well. Threads have their own stacks [95]. Therefore lots of threads can, in theory, consume a lot of memory. Threads can even exhaust the system's virtual memory if they are improperly used. Schedulers allocate CPU time slices for individual threads. A sub-optimal number of threads may start competing with each other against limited CPU time. Improper lock granularity can cause numerous threads to wait for one lock to be accessible. Tools such as Thread Tailor [96] have been proposed to dynamically adjusts the number of threads in an application to achieve optimal efficiency. The experiments carried on the fluidanimate benchmark carried out by the authors demonstrate that the application performs best with 4 threads - 12% better than 8 threads and a whopping 75% better than 2.

Thus, using too few threads - which leads to unutilized CPU cores- and using too many threads - which can quickly overwhelm limited system resources - is a performance pattern that we can term as *Sub-optimal Threading*.

Detection

Application tracing is a dynamic snapshot of the run-time environment of an application. *sched* tracepoints such as *sched_switch, sched_waking, sched_wakeup, sched_process_fork* can be used to delineate thread related events. The trace can then be loaded onto Trace Compass and individual threads can be studied. Views can also help in analyzing thread resource usage. Based off this information, developers can decide if threading has been optimally used or not. Context switching overhead can also be individually calculated using appropriate scripts.

3.3.5 Thread Scheduler Latency

Summary

CPU scheduler latency can be an indicator of CPU saturation [36]. If there are more tasks than the total number of cores then tasks can witness a long waiting period till they get a chance to run on the CPU. If CPU utilization is low, then tasks are quickly serviced with low wait periods. Otherwise, there can be considerable latency up to millisecond level. This can happen in scenarios where the number of parallel tasks is greater than the number of CPU cores.

Detection

Scheduler latency can be determined by tracing the moment when a thread is waken up till it gets a chance to run on a CPU core. This can be done by instrumenting scheduler wakeup events and scheduler switch events.

3.4 Disk Scheduling (I/O)

3.4.1 I/O Scheduler Overhead

Summary

The I/O (Input-Output) process encapsulates data transfer between memory and peripheral devices such as secondary storage and I/O devices such as a mouse and a keyboard, printers, etc. Disk I/O - which refers to input/output operations performed on a physical hard disk drive, can often bottleneck performance owing to the considerably slower rate of data transfer compared to that of a standard CPU. If I/O transactions are not efficiently managed, then these waiting times can snowball into major performance hits [97].

Thus, the I/O scheduler is employed to streamline this process. Its main object is to minimize I/O request latency and ensure that the I/O requests are fairly and quickly met, thus increasing disk throughput. Schedulers have the ability to store and reorder events to improve performance. They use techniques such as merging adjacent requests to reduce seeking and expanding I/O system call sizes, ordering requests on the basis of physical location and prioritizing requests. There is no one-size-fits-all scheduler that can be used regardless of the load profile and/or the hardware[98].

Schedulers could be chosen according to specific system performance goals. Embedded systems, for example, require low input latency. An absence of any kind of I/O scheduler would result in every I/O request interrupting the kernel so that the request could be fulfilled. This would slow down the system. Using the (Budget Fair Queuing) BFQ scheduler (which has considerable overhead owing to its complexity) for fast multi-queue systems with SSDs can be counter-productive. 'NONE' would be a better option to increase throughput. On the other hand, slower systems using disks with mechanical rotating parts could do well with a scheduler that supports request merging and enforcing request deadlines which 'NONE' does not support. Deadline has queues for both read and write requests and it enforces a 'deadline' on those requests which, on reaching, would ensure that the requests are completed. A comparative study on Linux I/O schedulers on SSDs has been done by Yunus et al. [99].

Therefore, it is important to choose the correct scheduler based upon the load profile. A wrongly chosen I/O scheduler could slow down performance. The simplest scheduler that can be employed in order to compare performance with other complex schedulers would be the NOOP scheduler. It has no ordered queues and requests are processed first in, first out(FIFO). Therefore, there is little to no scheduler overhead. Although the scheduler does merge adjacent requests to reduce seek time. This scheduler leaves the optimization to some other device. Therefore storage devices such as USB sticks, flash drives and SSD disks could benefit from this scheduler [100].

Detection

Tracing can be used to detect I/O latency and measure throughput as well. There are several static I/O tracepoints in the linux kernel such as block:block_rq_abort, block:block_rq_requeue, block:block_rq_complete, etc where abort, requeue and complete indicates the requests aborted, requeued and completed respectively [36]. Several LTTng analyses exist as well[5]. The I/O scheduler latency can be found out using the bpftrace tool iosched [36] which we implement in the next chapter.

[5]https://github.com/LTTng/LTTng-analyses

3.4.2 Improper I/O Workload

Summary

I/O workload profiles have a huge influence on system performance as disk I/O can cause application latency. They affect multiple users and thus affect business. An improper workload profile generally results from less-than-optimal I/O size (the size of the input or output request), IOPS (Input/Output per second), Throughput, Response Time, Read/Write ratio and whether the I/O is sequential or random. I/O operations can have significant overhead as they involve context switching, making system calls, address mapping and other such examples of ”initialization tax” [97]. Hence, it is critical that the data transfer in the process is as efficient as possible. For example, a large I/O size has its own unique advantages but it cannot be used in smaller applications with databases performing small reads. Hence, I/O workload profiles must be tailored according to the evolving needs of the application.

The response time is the total time taken to process a request starting from its acknowledgement till its dispatch. Read/Write ratio is self-explanatory and different workloads may have different ratios. I/O writes are generally asynchronous while reads are synchronous. This is because an application can block on a read request therefore read requests must have priority. Read requests also have a shorter deadline. Write requests can be cached to be written to disk later [82]. The Read/Write ratio is therefore an important metric that must be considered when designing the system as write requests generally have greater overhead (as they need to be reordered and serialized for efficiency) and slower response times.

Sequential I/O - as the name suggests - is when adjacent blocks of data are requested as opposed to random I/O where requests do not follow any predictable pattern. Random I/O requests may spike latency as hard disk drives with rotating media tend to witness extended seek times. Flash drives and RAM tends to work better with random I/O patterns as there are no moving parts [101].

When it comes to performance, a large I/O size results in longer response times. Similarly, writes too have additional overhead because of the nature of spinning disks as well as data protection technologies such as RAID [102].

The system must also ensure that it is adequately prepared to tackle a high workload peak such as peak response time during busy sessions and peak throughput during heavy writeback activities. I/O solutions should ensure that these peaks can be sustained for a long time with guaranteed performance stability.

Detection

Multiple I/O heavy applications running on a common database can introduce latency. Tracing can help detect improper I/O workload parameters. System throughput and IOPS can also be easily determined using tools such as Trace Compass and iostat [36]. Several tracepoints can be leveraged in order to trace I/O performance. Most of these tracepoints furnish information regarding individual block requests and how they're generated, merged, queued, re-queued and dispatched. Several views can be created once a state history tree is constructed from large trace files.

Specialized views such as latency distribution, disk utilization and throughput can be constructed from the metrics in the state database. These metrics can be calculated from tracepoint events and mapped onto a graph. Several kprobes can also be tapped into in order to fetch real-time information about I/O workload profiles. For example, the bpftrace tool *biopattern* can be used to find out the degree of randomness of I/O. It traces the block_rq_complete tracepoint and compares the previous disk address with the next address in order to determine if the profile is random or sequential. Similarly, another BCC/bpftrace tool, Bitesize can determine the per-process I/O size [36].

Applications with sequential workloads can afford large I/O sizes. However, they can occupy sizeable cache space. On the other hand, small I/O sizes can have a considerable overhead.

Real life examples of latency and their cause can be detected and analyzed in this way. For example, unexpectedly high response times for particular server requests.

I/O block tracepoints such as *LTTng_statedump_block_device, block_rq_insert, block_rq_merge, block_issue and block_complete* can be used to calculate aforementioned metrics.

3.5 Network Stack

3.5.1 TCP Re-Transmits

Summary

Computer networking takes place via certain data transfer protocols. Transmission Control Protocol (TCP) is one such example which establishes two endpoints in a network and allows two-way data transmission. TCP is reliable as it has safeguards against data loss. However, it comes at a certain cost in latency. TCP along with IP (Internet Protocol) - which are responsible for data transfer based on IP addresses, form the TCP/IP protocol stack which powers much of the internet. Data is transferred in the form of *packets* which contain several fields in their headers to store connection metadata as well as the payload (the data itself). These packets are transferred from the source to the destination once the TCP connection is established via a three-way SYN/ACK handshake [103].

As TCP is a reliable protocol, it ensures that lost data packets are re-transmitted. There can be several reasons for network packet losses such as network congestion or frame corruption. It is evident that packet loses can pose many problems as they indicate data loss and can slow applications down. TCP will keep re-transmitting any lost packet that it detects until the destination acknowledges. These acknowledgements are tracked via acknowledgement numbers in the packet headers which serves as a proof of receipt. When the source sends a packet, there is a timeout window within which it must receive a receipt of acknowledgement. Failure to do so would mean that the source will re-transmit the packet. Packets are transmitted in series and selective acknowledgements are used to indicate the packet numbers which are lost.

Tracing can help diagnosing network issues. Several TCP tracepoints have been added to the kernel to enable tracing the network stack. Before tracepoints, kprobes were used. However, its implementation would vary from one kernel version to another which would make matters difficult. Tracepoints, on the other hand, provide a ”stable API” and make maintenance and testing easier. There are tracepoints used to determine TCP session states such as *sock:inet_sock_set_state* and tracepoints available for tracing TCP congestion window such as *tcp:tcp_probe* [36].

Detection

For TCP re-transmits, the tracepoint *tcp:tcp_retransmit_skb* can be hooked into via bpftrace scripting. TCP re-transmit data may indicate network congestion and data packet loss. This tracepoint has been used in utilities such as *tcpretrans*[6]. This tracepoint in particular traces the *tcp_retransmit_skb()* kernel function with negligible overhead. It also does not trace every single packet and traces the kernel re-transmit code path instead.

Re-transmissions in themselves are no cause for alarm as it only demonstrates how reliable the TCP protocol is. But the rate of re-transmits going above a certain threshold can be. A high number of re-transmits can merit further investigation and network optimization.

3.6 Synchronization Context

3.6.1 Excessive Busy-waiting

Summary

On systems with shared memory, processes use synchronization constructs along with atomic operations in order to ensure data consistency. Synchronization constructs can be of two types: blocking constructs that block waiting processes and busy-wait constructs where a process repeatedly tests a shared variable in order to determine when to proceed. Busy-waiting is an important aspect of parallel programming and is preferred to blocking owing to its lower wake-up latency. Blocked processes take considerably more time to wake-up and they have to be scheduled to access the CPU as well. The spin-lock is a widely used implementation of busy-waiting. They are used to protect small critical sections where the waiting period is low. However, busy-waiting can cause performance bottlenecks in the form of memory and interconnection network contention [104]. They also can make debugging difficult and can cause race conditions known as synchronization races [105]. Another drawback of busy-waiting can be unnecessary wastage of CPU cycles if the waiting period turns out to be longer than expected. In this case, the process simply consumes CPU cycles and does not do any useful work. This is why programmers should be careful when using

[6]https://github.com/brendangregg/perf-tools/blob/master/net/tcpretrans

busy-waiting constructs.

Detection

Excessive busy-waiting can be detected by tracing. Once the trace is loaded onto Trace Compass, the resources view can highlight if a particular thread is spinning on a CPU for a particularly long time without doing any useful work.

3.6.2 Excessive synchronization

Summary

While synchronization is necessary in order to ensure that shared resources are used correctly, sometimes developers may use excessive synchronization primitives. So much so that even non conflicting processes end up being serialized. For synchronization to be efficient, it must be ensured that only the *critical* sections of the thread are synchronized. There can be several cases where excessive synchronization may harm performance. A lock can be unnecessary if the critical section does not access shared data, or the instructions are already atomic. Sometimes programmers do not realize that the computations are already protected by a lock and implement further unnecessary synchronization. This may happen if there is not enough familiarity with the code base [61].

Detection

A good way to detect possible instances of excessive synchronization is to look for high rates of lock acquisition with zero contention rates as detailed in [61] where a function uses a lock to synchronize removal operations which were already atomic. Another scenario could be when a single lock is used to guard several other locks. This may show very high rate of contention and acquisition as we show in our implementation.

Once detected, these issues can be easily fixed by removing the unnecessary locks. Tracing can help us identify potential cases of excessive synchronization. User space POSIX thread libraries can be instrumented alongside *futex()* system calls to analyze contention patterns related to excessive synchronization as discussed above. A manual inspection would be preferable as removing locks must be done after careful consideration so that the correctness of the application is not compromised.

3.6.3 Critical Lock Bottleneck

Summary

Multi-threaded applications, which rely upon a shared memory architecture, involve synchronization which regulates access to shared resources in order to prevent concurrent modification. *Critical sections* are such protected resources [106]. They often feature serialized access and reduced parallelism as only one thread can access the critical section at a time. These can become sites of bottlenecks. A *critical path* [107], on the other hand is the longest set of inter-dependent events that span from the beginning of the process till the very end - taking the longest time to complete and serves as a baseline for how fast the process would take to execute. If one is able to optimize the critical path and reduce its duration by a certain degree, then that would be reflected in the overall process completion time as well. The presence of critical sections in critical paths, therefore, require to be optimized to achieve substantial speedups.

Critical lock analysis, as investigated in [108] is a method by which locks in the critical path are detected and checked if they can be further optimized. This results in smarter optimizations. Many a time, engineers simply rank locks by order of contention and try to optimize the locks which show the most contention. However, their efforts will largely go to waste if the locks are not part of the critical path. The authors term critical sections that directly affect the critical path length as *hot critical sections* and the locks in these sections are called *critical locks*. Optimizing critical lock durations can cause a ripple effect and shorten overall execution time by a considerable magnitude. Critical sections that are off the critical path can execute in parallel. Hence, focusing optimization efforts on such sections will not result in performance gains. When it comes to the multiple critical locks on the critical path, the size of the hot critical section (the time spent executing) and the contention probability of the critical lock can help determine which lock to target first. The contention probability of the lock is the ratio of the number of contented invocations of the lock to the total number of lock invocations. A lock with a high probability of contention will witness many threads blocking on it and a long critical section in the critical path will result in the section taking up a sizeable portion of the path.

Detection

In order to detect critical locks in an application, the application must be traced accordingly. Also, the trace data must be fed to the algorithm which identifies the critical locks appearing on the critical path. Synchronization primitives can be traced using instrumentation modules which override the thread library routines and insert additional code so that run-time synchronization information can be collected. Data regarding lock acquisitions, lock contentions and lock releases must be collected along with their timestamps and stored as a trace for later processing. Once the critical path has been established, one can look at the lock contention activity in the critical path and determine if these locks can be better optimized. This trace pattern can help programmers know about potential performance boosting lock optimizations in their code.

A Linux test application with a variety of critical and non-critical locks can be written with varying idleness factors. Optimizing a non-critical lock with a high contention can be compared with optimizing a critical lock with little to no idleness and the results can be compared. The main objective being to demonstrate that developers should devote valuable time in identifying and optimizing critical locks *first* before focusing on non-critical locks. The algorithm provided in [108] can be used in tandem with EASE scripting to generate critical lock statistics. The locks with the most contention probability ratio should be targets of optimization.

3.6.4 Asymmetric Contention

Summary

Multi-threaded applications need synchronization to ensure serialized access to shared resources. Locking is one way to achieve that. A thread locks a particular shared resource to ensure atomic operation [109]. However, if another thread or process tries to access the lock at the same time then it would fail. This is called *contention.* The factors that affect contention are the frequency of lock requests and the duration of the lock acquirement. If a lock is highly contented then this will cause scalability problems where an increase of physical CPU cores will not translate to a linear performance boost. This is because lock contention affects parallelism [110].

Asymmetrical contention can be said to occur when locks protecting similar data show a skewed contention ratio. In theory these locks should have more or less same contention but in real-life scenarios this cannot be guaranteed. A good example of asymmetric contention according to [61] would be a situation where a hash table implementation of locks is used with locks for each bucket. If the hash function is unable to distribute the locks equally then some locks will be accessed more than the others which can lead to asymmetrical lock contention. The authors observed a 12% performance boost after optimizing this contention.

Detection

Locks can be traced by instrumenting the threading libraries as well as non-invasively. EASE scripting can be used to calculate lock statistics such as the number of lock requests, average waiting time and average lock retention time. If certain locks from a data structure witness a skewed contention ratio then it is clearly a case of asymmetric contention.

3.7 Hardware Context

3.7.1 TLB Performance

Summary

Operating systems such as Linux use virtual memory as compensation for limited RAM in order to increase the effective usable memory. The Linux kernel swaps unused blocks to secondary storage so that the main memory can be freed. When required, it can swap back those blocks onto main memory. Virtual memory creates an impression of a large, contiguous memory space for the running applications [68].

As a result of using virtual memory, addresses can be either *physical* or *virtual.* The virtual address space can be divided into several small *pages* so that they can be swapped in and out of swap space. Page tables are data structures that map the relationship between virtual and physical addresses. Virtual addresses need to be translated into physical addresses and this requires access to physical memory to fetch the page table as well as data. The *Translational Lookaside Buffer (TLB)* is high-speed cache for storing the most recently used page table entries so that page table entries are not needed to be accessed from the comparatively slower main memory. When a virtual address is to be mapped, the TLB checks if the page table entry is present. If it is, then it is a TLB *hit.* Otherwise, a *miss.* Once the TLB is hit, the virtual address is translated to a physical address which is then looked up in the cache. Like all fast memories, TLB size is limited and TLB misses can impact performance as it increases physical memory access [111].

Modern CPUs have TLBs for each core and hardware performance counters [38] can be used to determine iTLB (instruction) and dTLB (data) misses. TLB bottlenecks can be easily detected from monitoring counters. TLB misses result in cache misses and generally occur when the application working set is large. A working set is basically the memory range that an application is using [112]. If an application has a huge working set and on the other hand the page size is that of the default 4kB, then it results into a large TLB miss to LLC miss ratio.

In order to redress this, the page size can be increased. Linux supports *Transparent Huge Pages (THP)* as large as 1GB [79]. Large pages ensure that the TLB can map more pages and thus results in lower TLB hits.

Detection

TLB performance as well as THP performance can be detected using performance monitoring counters such as *dTLB-loads, dTLB-loads-misses, iTLB-load, iTLB-load-misses*. Important metrics such as CPU cycles spent in page table walking and number of highly expensive main memory reads owing to TLB misses can also be ascertained using *-e cache-misses*

It is important to balance TLB performance with that of using large pages. Using large pages can reduce CPU cycles wasted on page walks and the number of RAM reads. Even a tiny reduction in the TLB miss rate can provide a considerable performance boost. On the other hand, tracing kernel functions *__alloc_pages_slowpath* and *khugepaged_scan_mm_slot* - which looks for huge pages to collapse, can help control memory fragmentation that comes about as a result of using large pages.

3.7.2 Cache Performance

Summary

Modern CPUs use caches to ensure that the disparity between slow main memory access (which takes several hundred CPU cycles) and increasingly high CPU speeds is reduced. CPU caches are very fast memory units requiring access latency of only a few CPU cycles. Caches use the property of temporal and spatial locality. Temporal locality means that the information required in the future is already in use. For example, in program loops where variables and instructions are reused. Spatial locality, on the other hand implies related data items and instructions are usually stored and executed sequentially. When the CPU tries to read or write to main memory, it checks if the data is already in the cache. If the data is found to be present in the cache, then valuable CPU cycles are saved as the CPU does not have to look for the data in the considerably slower main memory. This is called a *cache hit*. Caches are generally multi-layered with the L1 cache being the smallest, yet the fastest following the L2 and L3 cache which increasingly grow in size but slower in operation. If the data or instruction is not found in the L1 cache then the CPU will look for them in the L2 cache and then the L3 ache. The CPU being unable to fetch data from the cache is called a *cache miss* [113].

Cache misses are expensive and exact a toll on overall performance. A cache miss on every level (L1, L2 and L3) implies that data has to be fetched from the RAM (which can take several thousand cycles) or worse, the secondary storage which can take millions of CPU cycles.

The cache stores data in the form of blocks of fixed size called *cache lines.* When a cache miss occurs, the cache copies the requested data from the main memory and then the request is fulfilled. Because caches have low memory capacities on account of their fast retrievals, room must be made to include the new entries. Therefore, existing data on the cache must be replaced. This is called cache replacement.

It is important to streamline application cache performance. The cache misses to instructions ratio is a good metric to evaluate cache performance with even a tiny improvement in the ratio resulting in significant performance boosts.

Detection

Hardware Performance Monitoring Units (PMUs) [38] can be used to gauge cache performance. These can be traced using profilers such as *perf* and tracers such as LTTng. A BCC tool, *cachestat* also exists which shows page cache hits and miss statistics. It does so by using kprobes to trace the **mark_page**, **mark_buffer_dirty**, **add_to_page_cache_lru** and **account_page_dirtied** kernel functions [36]. The event *cache-misses* records the total number of cache misses that occurred during a particular window of the application. A high cache-miss to instruction ratio should indicate to the user that cache friendly optimizations could be considered in order to improve performance.

3.7.3 Branch Performance

Summary

Instructional pipelining is an important aspect of RISC architectures where instructions are prefetched and parallelized in order to increase the efficiency of processing. Pipelining sees instructions flow in stages which lets the CPU process an instruction per clock cycle which results in a higher throughput [114].

However, if a particular instruction is a conditional branch then two paths open up. In this case, the processor tries to predict the path that would be taken so that stalls can be avoided. Prediction is crucial because the processor cannot wait till the execution unit computes the path to be followed. Owing to the speculative nature of the whole undertaking, it is wholly possible for the branch predictor circuit to guess wrongly the path that would be taken. If the CPU prediction is correct then the execution continues unabated. If not, then the processor has to flush the instructions originating from the wrong path, go back, and execute on the right path. This is a very expensive operation and is termed as a *branch path miss* [115].

Branch path prediction is contingent upon program behaviour. It has become increasingly critical as conditional branches have greatly increased in codebases. The prediction is based on the history of the particular branch. If a particular branch takes a certain path then it will, more likely than not, take the same path the next time around. Correct branch predictions will result in performance gains whereas branch-misses can slow down an application. Therefore, branches should be designed to be consistently predictable by reducing dependency chains and minimizing conditional statements.

In such a scenario, tracing the application using performance monitoring counters would provide a good indication of branch performance.

Detection

LTTng allows contexts to be added to events and channels. Contexts are additional nuggets of information that can be attached to channels such as Process ID and Thread ID. Performance Monitoring Counters (PMU) [38] can also be added using the perf kernel API. There are performance counters that detail the number of instructions processed per cycle, CPUs utilized, page faults, cache misses and branch misses. Analyzing this pattern would require the use of the branch misses performance counter. An abnormally high count would indicate poor branch prediction. Once detected, this can be mitigated by several approaches. Fixing the bottleneck can result in an increase in instruction per cycles as the number of stalls will be dramatically reduced.

We used a C++ code that manipulates an array twice:once on an unsorted array and once on a sorted one[7]. We notice that the code manipulating the sorted array takes 5 seconds as opposed to a whooping 17 seconds when the array is unsorted. This is a clear example of how branch mispredictions can slow down code-paths and tank performance. While the program was running we recorded branch statistics using perf and found that the code run with the unsorted array accounted for a whooping 15.45% of all branch misses. Once the array was sorted,the branch miss percentage dropped down to 0.13% as shown in the Figure 3.4.

[7]https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array

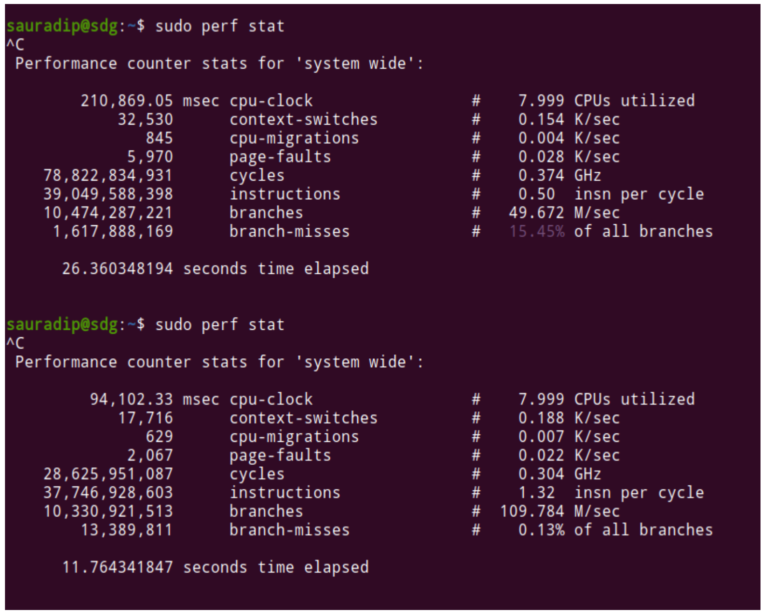


Figure 3.4: Branch statistics via perf

3.8 Application Context

3.8.1 Page Writeback Latency

Summary

Owing to the tremendous difference (literally orders of magnitude) in access speeds between physical memory and storage disks along with the principle of temporal locality - which stipulates that data once accessed will be, in all probability, accessed again, Linux maintains a page cache in order to reduce disk I/O. Instead of writing files to the disk every time, the changes are committed to the cache located in the main memory with the intent of *writing back* to the disk. This propagation of changes in the page cache back to the disk is called page writeback. The cache itself consists of RAM pages and can grow and shrink according to memory pressure. A *cache hit* occurs when the kernel finds the relevant data in the cache and does not need to access disk blocks and a *cache miss* occurs when the kernel has to access the disk in the form of block I/O in order to fetch the data. A cache miss results in the the data being loaded onto the page cache for subsequent accesses [82].

When the Linux kernel has to perform write operation, it does so on the cache data instead on the disk blocks. The updated pages are termed as *dirty* - meaning that the changes committed on these pages need to be written back to the disk in order to maintain data coherency [116]. The writeback operation can be scheduled accordingly so that a bulk of dirty pages are written back to the disk at some point in the future. After writeback, the pages are marked clean. Essentially, this is a synchronization process which can be complex, but efficient considering how disk writeback activities can be coalesced and performed in batches instead of writing to disk every single time there is a kernel write activity. These batches can also take advantage of different I/O scheduler algorithms to maintain fast and efficient transfers. When the pages are in the process of being copied to the disk, they are said to be in ”writeback” state.

The cache content frequently changes, either to free more memory or to load other relevant data which could be accessed short. When the cache has to ’evict’ data, it chooses only clean pages to do so in order to make sure that the data is synchronized with the storage. If there are no clean pages available, then it forces a writeback operation [82].

Sometimes, write heavy tasks fill up the cache quickly and thus, the kernel has to throttle those tasks to avoid them from continuing to fill up the cache. However, as shown by [62], the throttling can be unfair and many innocent tasks can also end up being throttled - tasks which do not fill up the cache. The writeback cache and its throttling can lead to system unresponsiveness. This is why, it could be useful to constantly monitor writeback activity during write heavy tasks in order to anticipate any system slowdown. The author(s) found out that once the cache gets full, all writing tasks are throttled equally which slows down the system. This is due to the fact that during writing, the file system inodes are locked until they are written to disk. The author(s) used tracing to detect throttling along with isolate the write heavy tasks so that their I/O priority could be reduced - allowing innocent processes to complete their write activities first. They devised a real-time monitor application and used the *global_dirty_state* tracepoint as it provides a snapshot of cache state such as number of dirty pages and thresholds and realized that long, unwanted pauses are because of large system caches which cause large amounts of data to be queued for writeback during cache flushes and locked inodes during writeback.

Detection

For this pattern, we concentrate on the writeback activity itself instead of kernel throttling. Brendan Gregg wrote a bpftrace tool called writeback [36] which dynamically traces tracepoints *writeback_start* and *writeback_written* which fetches real-time information on kernel writeback activities. The information fetched includes block device name, number of pages written, timestamp, latency and reason for writeback. The latency is found out by subtracting the timestamps in the *writeback_written* and *writeback_start* tracepoints whereas the number of pages written is determined by the difference in count of the parameter *nr_pages* which mandates the minimum amount of pages that must be flushed to the disk [70].

3.8.2 Abnormally Long Application Sleep Duration

Summary

Multi-threaded programs involving concurrent threads are often designed such that threads which do not require the processor or threads waiting for access to a particular resource (in the case of locks) or signal are made to sleep [117]. This is to ensure that processor time-slices are not wasted and other threads can access the CPU when needed. A sleeping thread implies that it will stop execution for a certain period of time and will wake up whenever the sleep period is over or when the scheduler wakes the thread up in case the kernel itself has induced the sleep.

Sometimes, programmers themselves add sleep durations in strategic code-paths while debugging or recreating a conditions contingent to a long running process. However, unintended sleep durations creeping in can cause latency which might be hard to debug. It is possible for programmers to have forgotten to remove the sleep period after maintenance activity is completed. It is also possible that certain threads may sleep for abnormally long durations because of badly written/unoptimized code or in some cases, execution can be suspended to facilitate polling. In either case, abnormal sleep times must be investigated and one way to do it in run-time is to trace the Linux syscalls:sys_enter_nanosleep tracepoint which instruments the Linux nanosleep() function [36].

The nanosleep() function puts the current thread to sleep until the specified duration in the *struct timespec *rqtp* argument has elapsed or the invocation of a signal that terminates the process or invokes the handler in the calling thread[8].

Detection

Since we have tracepoints already compiled into the kernel which instruments the nanosleep(3p) function, all we need to do is write a simple bpftrace script in order to access the necessary information that we would require for the trace in order to run it through a run-time verification/specification language parser.

3.9 Overview & summary of discussed patterns

The following table consists of all the performance analysis patterns that we have discussed till now along with a brief summary of the pattern. In the next chapter, we shall detect these patterns and subsequently, evaluate them.

[8] https://man7.org/linux/man-pages/man3/nanosleep.3p.html

Memory Management	
Performance Analysis Pattern	*Brief Summary*
Memory Fragmentation	Failure of high order memory block allocation owing to a lack of high order contiguous memory blocks.
Memory Leaks	Failure to de-allocate memory blocks after use.
Tracing Processes affected by the OOM killer	When a process of interest is killed off owing to Out Of Memory (OOM) condition
Memory Compaction Overhead	System latency induced by the memory compaction process.
Thread Scheduling	
Performance Analysis Pattern	*Brief Summary*
Thread Swamping	Threads of target application getting less On CPU time due to interference by other processes.
Unfair Scheduling	Thread being repeatedly migrated or unfairly scheduled.
Priority Inversion	A high priority thread waiting indirectly for a lower priority thread when a medium priority thread preempts the low priority thread.
Sub-optimal Threading	An application having too many or too little threads to make optimal use of multi-core processors.
Thread Scheduler Latency	The latency between a thread being woken up and scheduled on a CPU by the scheduler
Disk Scheduling (I/O)	
Performance Analysis Pattern	*Brief Summary*
I/O Scheduler Overhead	Latency introduced by the I/O Scheduler
Improper I/O Workload	Latency introduced by the nature of the I/O workload
Network Stack	
Performance Analysis Pattern	*Brief Summary*
TCP Re-transmits	Rate of TCP packet drops
Synchronization Context	
Performance Analysis Pattern	*Brief Summary*
Excessive Busywaiting	Detection of excessively long spin-locks which waste CPU cycles.
Excessive-synchronization	Too many synchronization primitives leading to performance hits.
Critical Lock Bottleneck	Optimizing critical locks for performance gains
Asymmetrical Contention	Uneven spread of synchronization primitives.

Table 3.1: A brief summary and overview of Performance Analysis Patterns

Hardware Context	
Performance Analysis Pattern	*Brief Summary*
TLB Performance	TLB hits and misses. CPU cycles wasted.
Cache Performance	Cache hits and misses.
Branch Performance	Degree of instruction pipelining and predicting code branches.
Application Context	
Performance Analysis Pattern	*Brief Summary*
Page Writeback Latency	Tracing significant cache writeback activity to disk by applications.
Abnormally Long Sleep Duration	Applications sleeping for abnormally long durations.

Table 3.2: A brief summary and overview of Performance Analysis Patterns (contd.)

Chapter 4

Evaluation

4.1 Implementation of pattern detection methods

In this chapter, we evaluate the different proposed techniques of detecting the performance analysis patterns. We ask ourselves if there would be a way to formalize these patterns using some kind of specification or temporal language. Some of the patterns that we found were too complex to formalize. In that case, we used trace analysis tools such as Trace Compass EASE scripting to detect and identify and evaluate those patterns.

To briefly summarize, we used Linear Temporal Language (LTL) and its extensions TK-LTL and LTL-FO+, an extension of a stream based specification language RT-LOLA, bpftrace scripting and tools along with EASE [1] scripting in order to generate and analyze the trace files with the purpose of detecting the patterns.

4.2 Evaluation Protocol

The evaluation protocol shall consist of the following headers:

- Data Source: How we obtained the trace data
- Detection Method: The technique(s) we employed to detect and identify the performance analysis pattern

[1] https://bit.ly/3KlxjrU

- Evaluation Metrics: The metrics used during pattern analysis
- Results: Our overall findings

Our goal in this section is to evaluate the usefulness of the proposed methods to detect the performance analysis patterns as well as to demonstrate their correctness. We also provide implementation details regarding how we conducted our experiments and how we evaluated the results. For certain patterns, we employ more than one detection method as well.

4.3 Evaluation Setup

Our system specifications are as follows:

- Host operating system: Windows 10 version 20H2
- Guest operating system: Ubuntu 20.04.2
- VM: Oracle VM VirtualBox 6.1.16
- VM Memory: 2048 Mb
- Host CPU: quad-core Intel Core i7-7700HQ CPU @ 2.80GHz (guest allocated two of four)
- Guest hard drive: 500 Gb Virtual Disk Image file stored on 1 Tb external hard drive
- LTTng version: 2.12.3
- TraceCompass version: 6.2.1
- Java version: openjdk 11.0.13

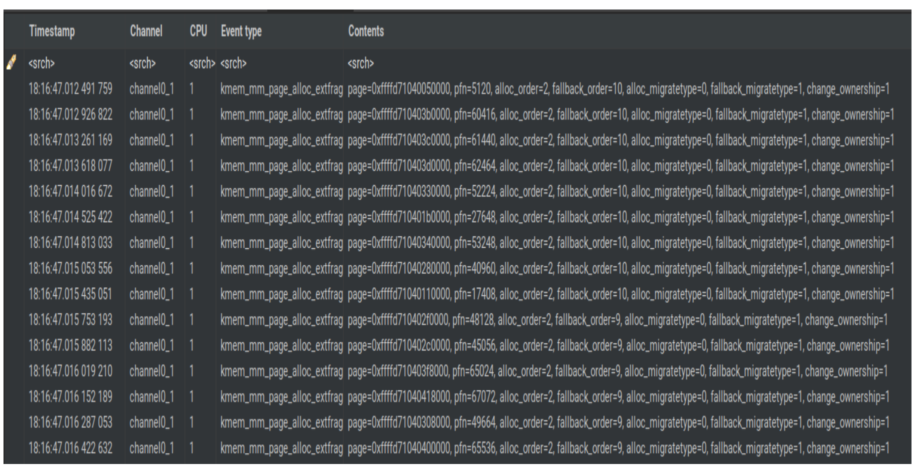
4.4 Memory Management

In the following section, we evaluate performance analysis patterns related to the Linux memory management system and find out if performance hits are because of memory issues.

4.4.1 Memory Fragmentation

Data Source

We generated a system trace using LTTng after having loaded and unloaded a memory fragmentation module [2]. This module creates fragmentation in kernel memory. This code basically allocates 100k blocks of memory of size 16,384 bytes using *kzalloc()*[3] and the GFP_KERNEL flag (which provides the caller full freedom in the allocation process). After that, every alternate block is freed as the module is unloaded. This causes significant memory fragmentation and higher order allocations will not be possible after some time. We use */proc/buddyinfo* to check the number of available higher order blocks and find that the number of lower order blocks has significantly increased after the module was loaded. This indicates that free memory is available, but it is fragmented and cannot accommodate high order allocations. In the meantime, we trace the system using LTTng and activate the *kmem_mm_page_alloc_extfrag* trace point as it indicates memory fragmentation. We could have activated other tracepoints as well but it only adds to the size of the trace file.



Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>
18:16:47.012 491 759	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040050000, pfn=5120, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.012 926 822	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710403b0000, pfn=60416, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.013 261 169	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710403c0000, pfn=61440, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.013 618 077	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710403d0000, pfn=62464, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.014 016 672	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040330000, pfn=52224, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.014 525 422	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710401b0000, pfn=27648, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.014 813 033	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040340000, pfn=53248, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.015 053 556	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040280000, pfn=40960, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.015 435 051	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040110000, pfn=17408, alloc_order=2, fallback_order=10, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.015 753 193	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710402f0000, pfn=48128, alloc_order=2, fallback_order=9, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.015 882 113	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710402c0000, pfn=45056, alloc_order=2, fallback_order=9, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.016 019 210	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd710403f8000, pfn=65024, alloc_order=2, fallback_order=9, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.016 152 189	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040418000, pfn=67072, alloc_order=2, fallback_order=9, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.016 287 053	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040308000, pfn=49664, alloc_order=2, fallback_order=9, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1
18:16:47.016 422 632	channel0_1	1	kmem_mm_page_alloc_extfrag	page=0xffffd71040400000, pfn=65536, alloc_order=2, fallback_order=9, alloc_migratetype=0, fallback_migratetype=1, change_ownership=1

Figure 4.1: Loading the fragmentation trace onto Trace Compass

Detection Method I

We use a specification written in LTL-FO+ to detect fragmentation. The specification is run once the LTTng trace is pre-processed. We use the BeepBeep monitor to evaluate the specification

[2] https://www.uninformativ.de/blog/postings/2017-12-23/0/POSTING-en.html
[3] https://www.kernel.org/doc/htmldocs/kernel-api/API-kzalloc.html

against the trace.

Once we load the trace onto Trace Compass (Figure 4.1) we can see that the fallback order is initially high at 10. Memory blocks are requested and serviced in multiples of powers of 2 [68]. An allocation order 0 means that a single contiguous page has been requested. Similarly, an allocation order 2 means 4 contiguous pages have been requested. The fallback order is the allocation order which the allocator falls back to if there are no blocks present to allocate for a particular order. The fallback order keeps changing with respect to memory fragmentation levels. Let's say, an allocation of order 5 (2^5 = 32 pages or 128KB) fails, then the buddy allocator [68] will fetch a block of order 10 (a block of order 10 would be equal to 2^{10} pages of 4kb each), split it into two buddy blocks of order 5 and complete the allocation. A non-fragmented memory implies the existence of higher-order blocks such as 8, 9 and 10 and subsequently, a higher fallback order. As external memory fragments, the fallback order starts dipping as the allocator is unable to find higher order blocks to split. We observe in our trace that the order starts with 10 and goes down to as low as 2.

Physical memory is divided into zones which are in turn divided into pageblocks. X-86 architecture features a 2MB (order 9) pageblock [118]. Pageblocks can be of migratetype. Free lists containing pages of each migratetype are present. An allocation can declare the migratetype using GFP (Get Free Pages) flag. If the allocation cannot be satisfied, it falls back to other types and 'steals' free pages from the other migratetype. Generally the largest free page is stolen from the other migratetype. If the fallback order becomes lesser than the pageblock order, it means that allocation cannot fallback to higher order blocks as there are none. This indicates memory fragmentation.

We process this trace to include only the fallback order and the constant pageblock order and then convert it to XML as shown in Fig 4.2

In order to detect fragmentation, we write a specification using LTL-FO+ and run it against the trace once it has been pre-processed. The specification is as follows:

$$\mathbf{F}(\exists a \in /message/FallbackOrder :$$

$$\mathbf{F}(\exists b \in /message/PageblockOrder : ((\neg(a = b)) \wedge (\neg(a > b))))))$$

The specification translates to: *Eventually, there exists variable a in FallbackOrder such that*

```
<trace>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
<message>
<Timestamp>1622931407</Timestamp>
<PageblockOrder>9</PageblockOrder>
<FallbackOrder>10</FallbackOrder>
</message>
```


Figure 4.2: Trace in XML format

eventually there exists variable b in PageblockOrder such that $a < b$*.* The BeepBeep monitor version we used does not support the $<$ sign, hence we used a simple walk-around.

If this specification, using the parameters from the trace, is evaluated as True, then we can say that Fragmentation has occurred. We use the BeepBeep monitor to implement our specification. Our specification was evaluated to be *True* as shown in Figure 4.3 , as there was a point in the trace where FallbackOrder was less than PageblockOrder.

Detection Method II

Another way of detecting memory fragmentation is to simply track the number of times the *mm_page_alloc_extfrag* event takes place in a given duration. A high number of these events can indicate possible memory fragmentation and incoming memory reclaims and/or compaction events. The timestamp can be extracted from the trace along with the trace-event itself and with the use of RTLola aggregate count function, the number of events in a sliding window of say, 2 seconds, can be counted. If the number of such events is greater than a given threshold **Th**, then a trigger can be generated. The computation frequency itself can be adjusted as per requirement. For example: output extFragCount @0.5Hz := extFrag.aggregate(over 1s, using: count) trigger extFragCount $>$

Th

This specification would calculate once every two seconds the number of extFrag events in a sliding window of 1sec and alert if the number of extFrag events become greater than a given threshold **Th**. In order to use RTLola specification on the trace, the trace would have to be converted to CSV format.

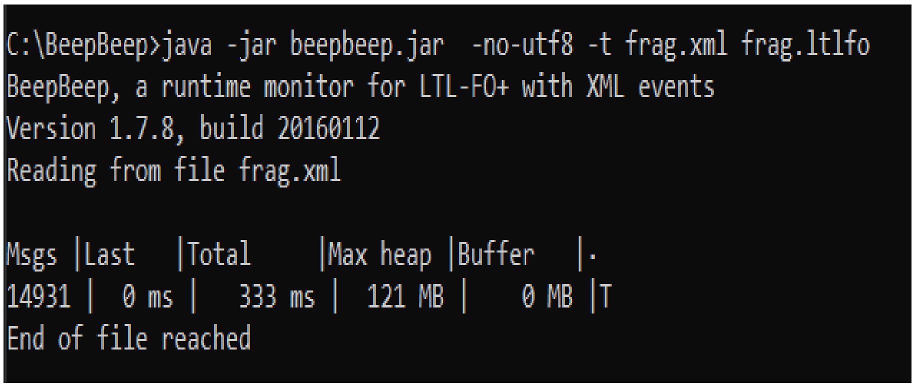


Figure 4.3: Verifying LTL-FO+ specification using the BeepBeep monitor

Detection Metrics

As seen in Fig 4.1, the tracepoint *kmem_mm_page_alloc_extfrag* has several parameters out of which we are interested only in the fallback order and the pageblock order (which is equal to 9 in X-86 architectures [118]. We appended the pageblock order to the trace and converted the whole trace to XML format. To detect memory fragmentation we look for instances where the fallback order is $<$ than pageblock order.

Another metric is the frequency of the tracepoint itself in a given interval. If the frequency of the event becomes higher than a given threshold **Th**, then we can say that the memory is fragmented.

Results

We were successful in fragmenting kernel memory in our VM using the kernel module. After that, we were also able to detect fragmentation in the trace files using an LTL-FO+ and RTLola specification. Detecting memory fragmentation can be helpful in investigating system stalls and unresponsiveness.

4.4.2 Tracing Processes Affected by the OOM Killer

Data Source

oomkill [36] is a bpftrace tool created by Brendan Gregg which traces the kprobe *oom_kill_process*. Data fetched from this kprobe will provide us the details of the application killed off by the OOM killer, the process that initiated the kill as well as other useful details. This output is converted into an XML file as shown in Figure 4.4. We modified the code slightly to include the timestamps in nanoseconds and fixed an error in the code which was present in the book. We can also include a cat command to display the oom_adj_score [76] of the sacrificial process so that users can know the OOM score of the process which was killed off.

Detection Method

This performance analysis pattern can be detected using a specification written in Linear Temporal Language. If we are to ensure that any process of PID = x of our choosing must be safe from being killed off by the OOM Killer, we must use the following specification:-

$$\mathbf{G}\neg(SACRIFICIALPID = x)$$

which translates to: it is always the case that the sacrificial process ID is NOT x or it is never the case that the process X is the sacrificial process.

The XML file is fed to the BeepBeep LTL monitor which checks our specification against the trace and outputs a value. It can be True, False or ? (Undetermined). If the evaluation is True, then our process hasn't been killed.

When it comes to a process that the user is willing to allow to be killed off in case of an out-of-memory situation, they can set a very high adjusted OOM score, such as 15, which will ensure that the process is killed off first. In our experiment, we assigned the maximum possible OOM score of 15 to a P2P client and once we triggered the OOM Killer using the sysreq-trigger, we observed that the client was killed off first. (Figure 4.5)

Detection Metrics

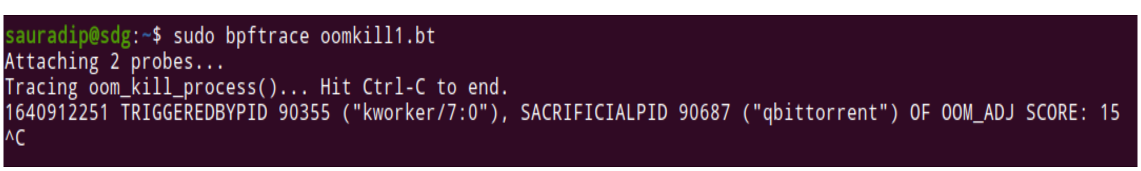
We know that a particular process has been killed in real-time from the arguments of the oom_kill_process kprobe. The argument is a struct of type oom_control [36]. This struct contains the PID and process name of the chosen sacrificial process. The process which triggers the OOM kill can also be traced using the bpftrace built-in variables. If we are to ensure that our process hasn't been killed off, we must make sure that the process PID is never equal to the sacrificial PID.

Result

We assigned a very high OOM score to a process (15) and manually triggered the OOM killer to see if the process is killed off. As per our expectations, the OOM_Killer killed off the process immediately and our specification was evaluated to False. In order to prevent a process from being killed off, one could assign a very low OOM score to the process. However, it is not advisable as poorly optimized code can eat up system resources and the OOM killer would not terminate the process which can lead to system hangups.


```xml
<trace>
  <message>
    <TIME>1633394699</TIME>
    <TRIGGEREDBYPID>18601</TRIGGEREDBYPID>
    <SACRIFICIALPID>1165</SACRIFICIALPID>
  </message>
  <message>
    <TIME>1733394701</TIME>
    <TRIGGEREDBYPID>18601</TRIGGEREDBYPID>
    <SACRIFICIALPID>48556</SACRIFICIALPID>
  </message>
</trace>
```


Figure 4.4: Trace from the bpftrace tool oomkill.bt is converted to XML



```
sauradip@sdg:~$ sudo bpftrace oomkill1.bt
Attaching 2 probes...
Tracing oom_kill_process()... Hit Ctrl-C to end.
1640912251 TRIGGEREDBYPID 90355 ("kworker/7:0"), SACRIFICIALPID 90687 ("qbittorrent") OF OOM_ADJ SCORE: 15
^C
```

Figure 4.5: Setting a high OOM score of 15 ensures that the process is killed off in case of low memory

4.4.3 Memory Compaction Latency

Figure 4.6: Memory Compaction Latency Histogram

Data Source

In order to calculate the amount of time it takes for one compaction process to complete, we need to hook into two tracepoints. *mm_compaction_begin* and *mm_compaction_end*. We do this using a custom bpftrace script. (Fig 4.7)

Detection Method

In the bpftrace script, we make use of an array @start which stores the starting timestamp when compaction begins and then subtract it from the ending timestamp which we obtain in the compaction_end tracepoint for every TID. After that, we display the histogram of the compaction durations. The durations are in microseconds.(Figure 4.6). Higher compaction latency may be the cause for slow system response, CPU usage spikes and high CPU utilization.

We can formalize this pattern as a LTL-TK specification. We use the structural semantic counter ${}_{\phi}\mathcal{D}_{\psi}^{\top}$ which identifies the first position in a trace where the condition ψ holds true $\top$ from the position where ϕ is satisfied. In this case, we denote ϕ as the tracepoint *compaction:mm_compaction_begin*

```
#!/usr/local/bin/bpftrace

tracepoint:compaction:mm_compaction_begin
{
 @start[tid]= nsecs;
}

tracepoint:compaction:mm_compaction_end
/@start[tid]/
{
$temp = (nsecs-@start[tid]) / 1000;
@us = hist($temp);
delete(@start[tid]);
}
```

Figure 4.7: bpftrace script to display memory compaction histogram

for a given thread *tid* and ψ as the tracepoint *compaction:mm_compaction_end* for the same thread *tid*. Therefore, the compaction latency can be denoted as ${}_{\phi}\mathcal{D}_{\psi}^{\top}$ which states how long after the initial *compaction_begin* tracepoint do we hit *compaction_end* tracepoint. This period is the compaction latency.

Detection Metrics

A histogram of compaction latency in the form of intervals can be seen after the execution of the script. The hist function comes built-in with bpftrace and it indicates the number of times a compaction process occurred and its corresponding latency. The latency brackets are displayed in multiples of 8. Given that it is a rough approximation, it is a good visual indicator of compaction times.

Result

We can detect abnormal compaction latency using a bpftrace script which hooks into compaction_begin and compaction_end tracepoints and displays a latency histogram.

4.5 Thread Scheduling

In the following section, we evaluate several performance analysis patterns which relate to thread scheduling. Improper scheduling of threads may lead to thread starvation and priority inversion which can introduce latency and system hangups.

4.5.1 Thread Swamping

Data Source

We conducted several experiments where a single-threaded Java application runs alongside several stress worker threads. The Java application uses a thread to print a message on loop to the console. We used a tool called *stress*[4] which would generate the required workload. Our Java application has a thread named ”Thread 1” whereas the stress threads are named as ”stress”. These are essentially worker threads which spin on the sqrt() function. After running both the Java application and the worker threads in parallel, we wanted to quantify the effect of the worker threads on our application thread. The trace file was recorded using LTTng which was then analyzed to detect the degree of swamping that our Java application thread witnessed owing to the stress threads.

Detection Method

We used a custom EASE script to calculate the execution ratio of Thread 1 with respect to a variable number of stress threads. We begin the experiment by simply running the Java program without the stress threads and as expected, we get a high execution ratio. The ratio starts decreasing as we increase the number of stress threads. The EASE script was run on the trace file after it was loaded onto Trace Compass.

[4]https://linux.die.net/man/1/stress

Detection Metrics

A thread can be inactive if it is blocked from running or if it is waiting for CPU. (Figure 4.8) We use the information available from the sched_switch tracepoint in order to calculate the amount of time a thread spends executing on a CPU. We make sure that we calculate the timestamps only when the thread was blocked while it was doing useful work. We define 2 metrics in this section: TCD and TD. These metrics would determine the amount of swamping.

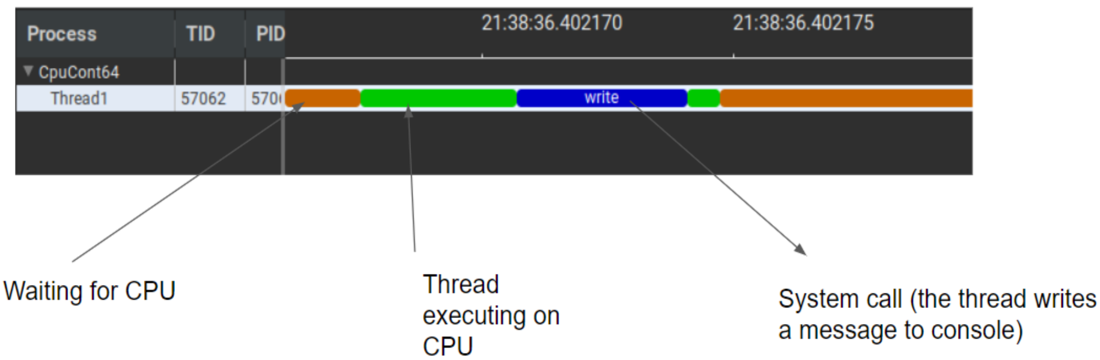


Figure 4.8: Colour coded segments showing various thread states on the Trace Compass UI

Figure 4.9: Yellow indicates a blocked thread

Let A, B, C and D be the timestamps at each context switch respectively. We define thread execution time by summing up the thread execution time (in green) and system calls made by the thread (in blue). Therefore, the *Thread On CPU duration [TCD]* would be:

$$(B - A) + (D - C)$$

The *Thread duration [TD]* can simply be calculated by subtracting the timestamp of the last event of the trace with the first event of the same trace. (Figure 4.10)

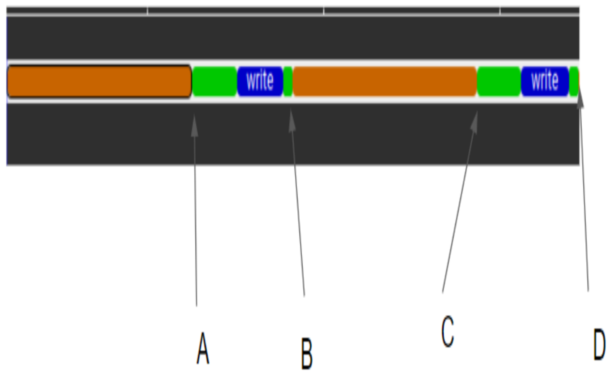


Figure 4.10: Defining TCD and TD using timestamps A,B,C & D

Result

At the beginning of the experiment, the thread timeline is filled with greens and blues- implying that the thread runs uninterrupted in a single CPU in the absence of any stress threads. On running the script with the TID of Thread1, we get a 96.3% execution ratio. (Figure 4.11 and Figure 4.12)

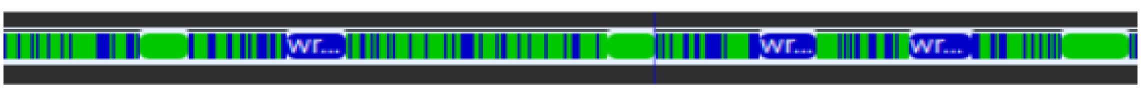


Figure 4.11: A timeline of a thread which isn't swamped by stress threads

The hit on the execution ratio is evident when we generate a very high number of stress threads. 40 in this case. As we see the thread timeline we see a large number of orange and yellow patches which indicate a high level of blocking. (Figure 4.13)

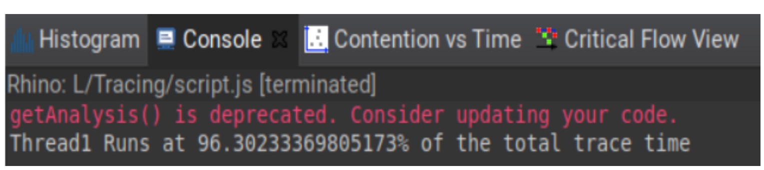


Figure 4.12: A high execution ratio in the absence of stress threads

We see the execution ratio go down to 56.44%, which implies that the thread waits nearly half its time waiting for CPU. (Figure 4.14)

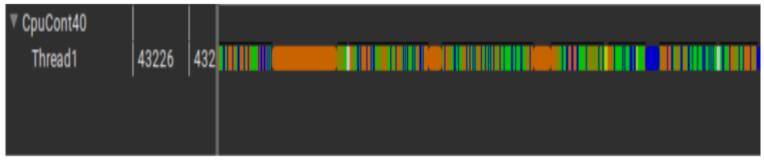


Figure 4.13: Timeline of the thread when it is swamped by 40 stress tests

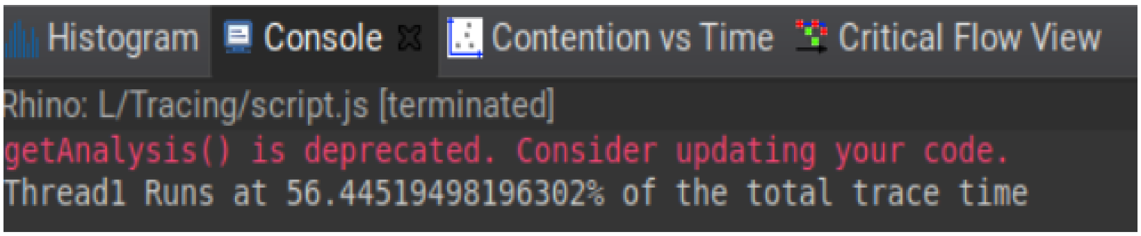


Figure 4.14: A low execution ratio which means that our application has been swamped

4.5.2 Unfair Scheduling

Data Source

In our experiment, we have simulated a situation where a thread is repeatedly migrated by the scheduler from one CPU to another. This may not happen in production environment but we wanted to demonstrate how tracing can lead us to discover repeated thread migrations by the scheduler. The idea was to execute a Java application which would spin endlessly in a method and to write a shell script which would use the taskset command in order to manually change CPU affinity of the thread spawned by the application. This would lead the thread to be migrated repeatedly from one CPU to another by the scheduler and highlight the problem of unfair scheduling. The application was traced using LTTng and further analysis was carried out via EASE scripting.

Detection Method I

We activated tracepoints *sched_switch*, *sched_migrate*, and *sched_migrate_task*. An EASE script was written to sort the system threads in descending order based on the number of times the thread was migrated. This was done using the tracepoint *sched_migrate_task* which has parameters like origin CPU and destination CPU. The data was mapped on a graph which clearly shows the thread being repeatedly migrated from one CPU to another. As we can see in the Figure 4.16 Thread 3483 is repeatedly migrated from one CPU to another with red denoting CPU 1 and blue denoting CPU

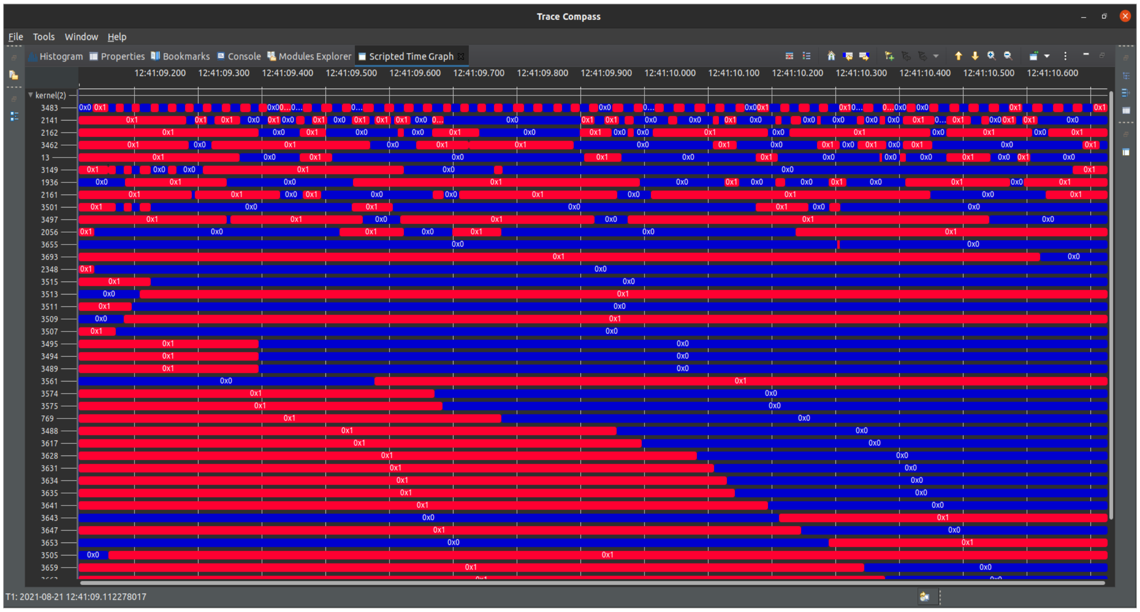


Figure 4.15: A visual graph showing all the thread migrations

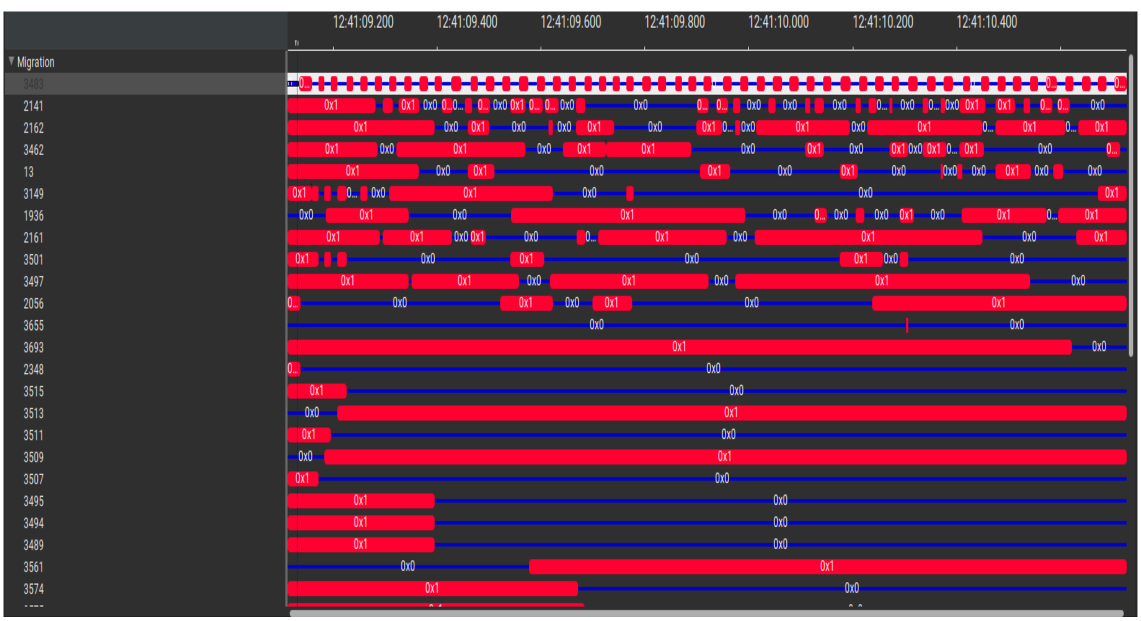


Figure 4.16: Highlighting how a single thread gets migrated repeatedly between CPUs

2.

Detection Method II

We use the semantic structures provided to us by LTL-TK such as the existential quantifier and the counter in order to design a specification to detect this pattern. Assuming a threshold *Th* beyond which a thread should not be migrated by the scheduler, we write the following specification in LTL-TK:

$$\exists_{>Th}\mathcal{C}_{\phi}^{\top} \rightarrow \mathbf{F}log$$

where ϕ denotes the condition *sched_migrate_task.tid* = *tid* where *tid* is the thread ID of the thread we are analyzing. The existential quantifier determines if the count of every instant in the trace where ϕ evaluates to $\top$ is greater than the threshold *Th*. If the count is greater than *Th*, it is logged.

Detection Method III

We tried to formalize this pattern using LTL in the following manner:

$$\mathbf{G}\neg(\mathbf{GF}(\textit{sched\_migrate\_task.tid} = \textit{tid}))$$

which can be translated as it should never be the case that a given thread *tid* will always eventually be migrated to another another CPU. However, implementing this specification would result in the verdict ? (indeterminable) as **GF**(*sched_migrate_task.tid* = *tid*) would imply that there will always be a migration of thread *tid* at an undisclosed time in the future. Hence, this specification will not result in a definitive Truth or Falsity. Hence, we will stick to EASE scripting as it shows us exactly how many times individual threads get migrated by the scheduler and sorts them in descending order.

Detection Metrics

For this pattern, the number of CPU migrations per thread was calculated from the data furnished by tracepoint *sched_migrate_task*. If the number of such migrations for a particular thread is abnormally higher than all other threads, then we will have detected this pattern. Our code displays migration information in descending order with the thread with the most migrations appearing first.

Result

We showed how we can detect unfair thread migrations using tracing and then processing the trace file using EASE scripting. Our code migrates the thread repeatedly by design and real life production instances will not see such extreme to-and-fro migration. However, our script demonstrates the ability to detect abnormal migrations in case they ever occur.

4.5.3 Priority Inversion

Data Source

We simulated an instance of priority inversion by tracing a custom Java program where we created three threads of high, medium and low priority respectively. We used the Java Virtual Machine to set the priority of these threads. The high, medium and low priority threads are named High Priority P, Sorter and Low Priority Pr respectively. JVM provides a TID of 13 to High Priority P and 15 to Low Priority Pr. We traced this application from the kernel space as well as the user-space.

<srch>	<srch>	<srch>	<srch>
18:57:58.530 285 520	lttng_jul_channel_1	lttng_jul:event	msg=Lock requested, logger_name=jello, class_name=PriorityInversionDemo, method_name=printData, long_millis=1623970678519, int_loglevel=800, int_threadid=15
18:57:58.620 161 876	lttng_jul_channel_1	lttng_jul:event	msg=Lock acquired, logger_name=jello, class_name=PriorityInversionDemo, method_name=printData, long_millis=1623970678619, int_loglevel=800, int_threadid=15
18:57:59.019 472 891	lttng_jul_channel_0	lttng_jul:event	msg=Lock requested, logger_name=jello, class_name=PriorityInversionDemo, method_name=printData, long_millis=1623970679019, int_loglevel=800, int_threadid=13
18:58:01.646 609 877	lttng_jul_channel_1	lttng_jul:event	msg=Lock released, logger_name=jello, class_name=PriorityInversionDemo, method_name=printData, long_millis=1623970681646, int_loglevel=800, int_threadid=15
18:58:01.649 753 676	lttng_jul_channel_0	lttng_jul:event	msg=Lock acquired, logger_name=jello, class_name=PriorityInversionDemo, method_name=printData, long_millis=1623970681649, int_loglevel=800, int_threadid=13
18:58:04.652 632 169	lttng_jul_channel_0	lttng_jul:event	msg=Lock released, logger_name=jello, class_name=PriorityInversionDemo, method_name=printData, long_millis=1623970684652, int_loglevel=800, int_threadid=13

Figure 4.17: The higher priority thread requesting a lock already held by a lower priority thread

Detection Method I

We use Trace Compass to manually detect priority inversion. Instrumenting the code in the user-space allows us to determine exactly when the high priority thread requests a lock already held by the lower priority thread as shown in Fig 4.17. We can see that the higher priority thread (TID=13) requests the lock at 18.57.59.019 approximately but the lower priority thread (TID=15) releases the lock much later, at around 18.58.01.646.

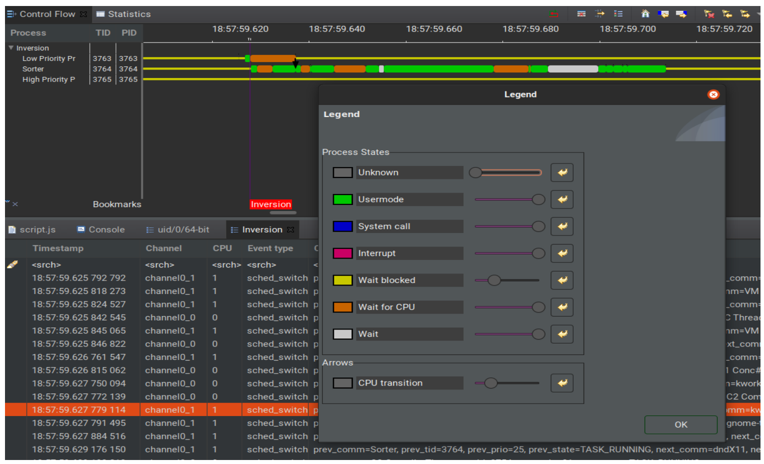


Figure 4.18: Trace Compass timeline showing the relationship between the three threads

Let's look at the resource timeline on Trace Compass. (Figure 4.18) If we observe the timeline at approximately 18.57.59.620 (bookmarked by the red vertical line), we shall see that the low priority thread is blocked (a state transition from green to orange) and the middle priority thread is scheduled on the CPU. By the time the low priority thread gets CPU time, thread Sorter has been scheduled on CPU at least thrice. One can recall from the user-space trace that the high priority thread is still waiting to acquire the lock which is acquired presently by the lower priority thread. This is an instance of priority inversion where a thread with higher priority is essentially waiting to access a resource currently being accessed by a thread of lower priority. In the Figure 4.19 we shall see that the high priority thread gets CPU access after Low Priority Pr relinquishes the lock.

In Figure 4.19 , we can see that High Priority P finally gets scheduled on CPU (state change from yellow to green) when it acquires the lock at 18.58.01.649

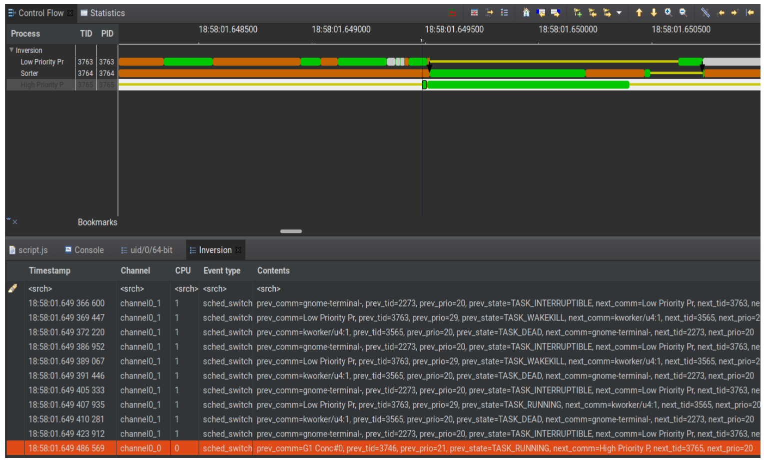


Figure 4.19: High Priority thread getting CPU access after the Low Priority thread relinquishes lock

Detection Method II

We wrote an EASE script to try to detect priority inversion. It worked by looking at the sched_switch tracepoint to see which threads occupied the CPU, what state the threads were in when they were taken off the CPU, and what priority the threads were when they were placed on the CPU. If the thread was in the state TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE, then we assigned the thread state as blocked and put them into a list. Every time another thread was scheduled on to the CPU with a lower priority than a thread in the blocked list, we flagged the thread as priority inverted. We kept track of the number of times each thread was priority inverted. The problem with this implementation was two-fold. First, threads can be indirectly blocked for many reasons. Some examples include waiting for user input, disk I/O, and network. We could have used the futex system call instead. This would show periods of time where threads are indirectly blocked by resources implemented with a futex. However, this leads to the second problem. Thread A, blocked by resource R, can only be priority inverted if Thread B, holding resource R, is preempted by Thread C, which has a lower priority than Thread A. Our script did not check for this requirement because we could not tell which thread was the one holding the resource blocking the blocked thread. Even with the futex system call, we would not be able discover this. Futexes work by only entering kernel mode (emitting a futex system call for us to trace) when there is lock contention on the futex. There is no way to tell which thread is holding the lock because futex system calls are not emitted when a futex is acquired. The system call is only emitted when the futex is requested or released and there is contention. As such, given these tracepoints, priority inversion cannot be detected.

Detection Metrics

In our experiment using Method I, we had to manually detect priority inversion by checking if a thread of higher priority is indirectly waiting to access a resource which is currently being accessed by a thread of lower priority. We find out the instance when the lower priority thread is preempted by another higher (in this case, a middle priority thread) priority thread which leads to the inversion.

Result

We use tracing and the Trace Compass UI after loading both user-space and kernel-space trace files in order to manually detect priority inversion.

4.5.4 Sub-Optimal Threading

Data Source

In our experiment, we designed two Java applications with 25 threads and 4 threads respectively and then traced their executions. Both applications do identical tasks - increase a counter until it reaches a large number. The threads in the application try to increase the integer which is common to all threads and belongs to a synchronized class so that only one thread can access the integer at a time. This creates a lot of contention between the threads as seen in the application with 25 threads. In contrast, the application with 4 threads does not see any contention. Tracepoints sched_switch, sched_process_exit, sched_process_fork were activated during the tracing session.

Detection Method I

We used an EASE script to compute the statistics which was applied to the trace after being loaded onto Trace Compass. For the application with 25 threads we see a majority percentage of threads wait for CPU for at least 65% of their lifetime. The other application sees no such thread. Thus, dynamic tracing can help us detect at least one type of sub-optimal threading. Similarly, a situation where an application using fewer threads than what is optimal could be identified by looking for free CPU cores. Performance monitoring counters can also be employed to check cache statistics such as hits and misses for an overall greater picture of resource usage.

Detection Method II

We can use LTL to formalize this performance analysis pattern with the help of EASE scripting. For a given process PID, let the total number of threads spawned by that process be **Nth** and the number of threads waiting for CPU time for more than or equal to about 65% of its lifespan be **Nwait(65%)**. The ideal threshold for the number of threads that can run optimally on a system is equal to the number of CPU cores/hardware threads present. So taking that as the threshold **Th** we come up with the formalization

$$\mathbf{G}\neg((\mathbf{Nth}>\mathbf{Th}) \wedge (\mathbf{Nwait(65\%)}/\mathbf{Nth} > 50\%)$$

which can be summarized as :- it should never be the case that the number of threads spawned by a program exceeds the number of CPU hardware threads and where at least half of those threads are waiting for CPU for 65% or more of their lifetime. The percentage ratios can, of course be modified as per requirement. The percentage of lifetime waiting for CPU was chosen to be aggressive at 65.

Detection Metrics

We use the metric **Nwait(65%)** to denote the number of threads which are waiting for CPU for 65% of their lifetime. This is an aggressive percentage which can be tweaked with as per requirement. Figure 4.20 shows how 23 threads wait for CPU for 65% of their lifetime and are hence considerably swamped. Figure 4.21 which has only 4 threads (the number of threads showing is more as there are other system and helper threads too) sees no such swamped thread.

Result

We use tracing to determine the optimal number of threads that an application should have in order to leverage multitasking on multi-core CPUs. Using tracing data we can detect threads that

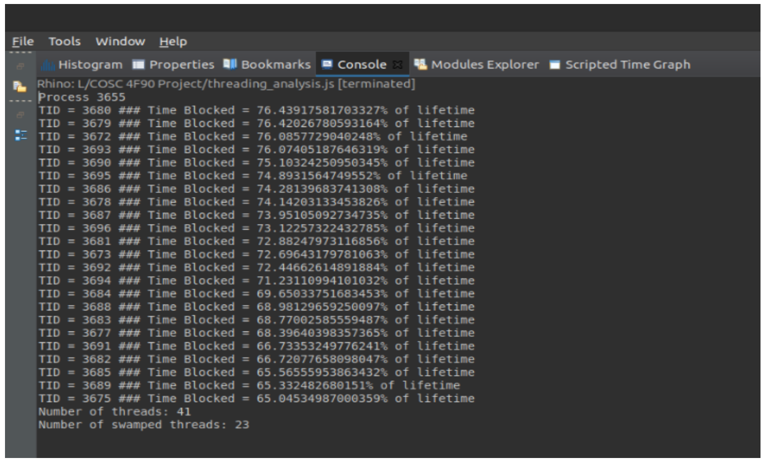


Figure 4.20: The application with 25 threads sees a large number of threads being swamped.

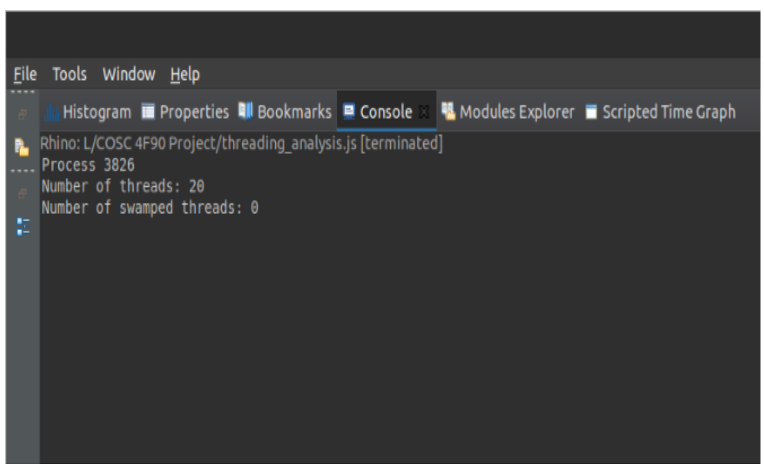


Figure 4.21: The application with 4 threads sees no swamped threads

are frequently blocked so that we can reduce the number of threads. Similarly, if we find multiple idle CPU-cores, more threads can be added to the application to boost performance.

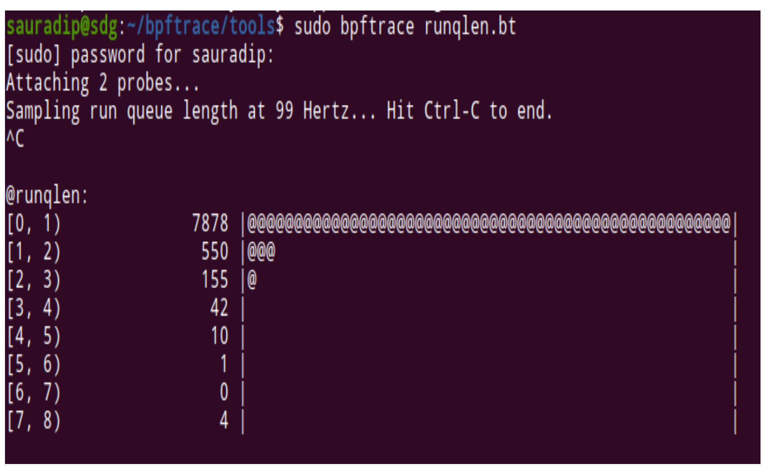
4.5.5 Thread Scheduler Latency

Data Source

Brendan Gregg's bpftrace tool runqlat [36] can be used to determine thread scheduler latency. This tool hooks into *sched:sched_wakeup*, *sched:sched_wakeup_new* and *sched:sched_switch* in order to calculate the duration between thread wakeup and thread running.

Detection Method

We use the tool to obtain two latency histograms (in microseconds) which show the average wait time taken for a thread to get CPU access. The first histogram was generated when the system was relatively idle (a few open Chrome tabs) (Figure 4.22) whereas the second histogram was after having spawned 40 worker threads using the application *stress*. (Figure 4.23).



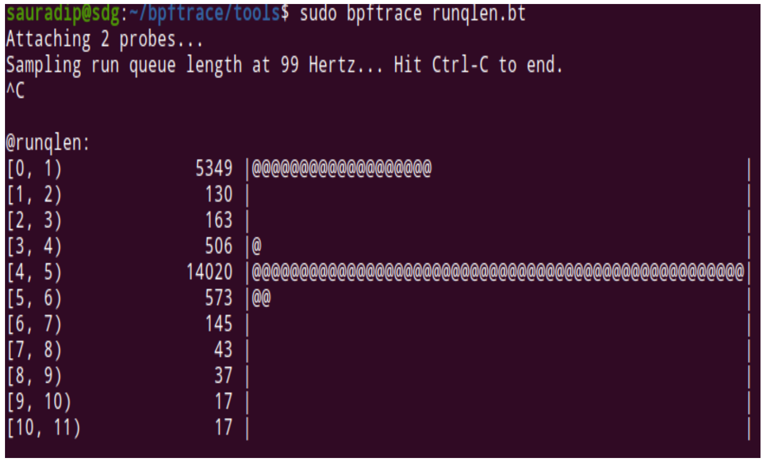
```
sauradip@sdg:~/bpftrace/tools$ sudo bpftrace runqlen.bt
[sudo] password for sauradip:
Attaching 2 probes...
Sampling run queue length at 99 Hertz... Hit Ctrl-C to end.
^C

@runqlen:
[0, 1)            7878 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[1, 2)             550 |@@@                                                 |
[2, 3)             155 |@                                                   |
[3, 4)              42 |                                                    |
[4, 5)              10 |                                                    |
[5, 6)               1 |                                                    |
[6, 7)               0 |                                                    |
[7, 8)               4 |                                                    |
```

Figure 4.22: Latency histogram of a relatively idle system

Detection Metrics

The detection metric used here is the average wait time for a thread from the moment it is woken up (sched_wakeup) till it is scheduled on a CPU (sched_switch). As we see in Figure 4.22, a



```
sauradip@sdg:~/bpftrace/tools$ sudo bpftrace runqlen.bt
Attaching 2 probes...
Sampling run queue length at 99 Hertz... Hit Ctrl-C to end.
^C

@runqlen:
[0, 1)              5349 |@@@@@@@@@@@@@@@@@@@                                 |
[1, 2)               130 |                                                    |
[2, 3)               163 |                                                    |
[3, 4)               506 |@                                                   |
[4, 5)             14020 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[5, 6)               573 |@@                                                  |
[6, 7)               145 |                                                    |
[7, 8)                43 |                                                    |
[8, 9)                37 |                                                    |
[9, 10)               17 |                                                    |
[10, 11)              17 |                                                    |
```

Figure 4.23: Latency histogram of a system under stress

relatively idle system sees a very short wait-period (0-1 microseconds) whereas a busy system can see that time go up considerably. (Figure 4.23). Systems running at capacity can see latency at the millisecond level as well.

Result

We use the bpftrace tool runqlat, in order to trace the average scheduler latency by dynamically tracing the sched_wakeup and sched_switch tracepoints.

4.6 Disk Scheduling (I/O)

In this section, we shall evaluate a single performance analysis pattern that relates to the Linux disk scheduler (I/O). We shall use bpftrace scripting in order to find individual I/O request latency in the form of a histogram to determine if the application is witnessing any lags due to I/O scheduling overhead.

4.6.1 I/O Scheduler Overhead

Data Source

When it comes to the I/O schedulers themselves, their latency can be traced using a bpftrace script. Brendan Gregg's tool iosched [36] does precisely that. The script measures the time difference between adding a new request to the scheduler and the time when the request was issued. Essentially, it traces the amount of time a request waits in the queue. The tool uses kprobes __elev_add_request, blk_start_request and blk_mq_start_request to achieve that. However, as stated earlier, kprobes are unstable and will not work if the function is changed, renamed or deleted in subsequent kernel versions and that is what exactly happened. Refer to Fig 4.24 for the updated iosched code.

```
#!/usr/local/bin/bpftrace
#include <linux/blkdev.h>
BEGIN
{
 printf("Tracing block I/O schedulers. Hit Ctrl-C to end.\n");
}
kprobe:elv_rb_add
{
 @start[arg1] = nsecs;
}

kprobe:blk_mq_start_request
/@start[arg0]/
{
 $r = (struct request *)arg0;
 @usecs[$r->q->elevator->type->elevator_alias] =
 hist((nsecs - @start[arg0]) / 1000);
 delete(@start[arg0]);
}
END
{
 clear(@start);
}
```

Figure 4.24: Updated IOSCHED code

Detection Method

Linux kernel 5.0 and onward removed all redundant elevator code and the kprobe __elev_add_request failed to attach considering the function itself was deleted from the code[5]. Similarly the probe for blk_start_request could not be attached. Hence, the script had to be slightly modified for it to work. Instead of the __elev_add_request function, a probe was attached to elv_rb_add where I/O requests are added in a red black tree. The probe to blk_start_request was removed and the script worked.

Detection Metrics

Heavy I/O activity was simulated on our system while the script was running which results in higher latency (16 to 64 milliseconds) for several requests. Our modified script (Fig 4.25) shows a histogram of queuing time durations for mq_deadline (as it was the only I/O scheduler present in the test system apart from none). This tool can be used as a complement to decide if the I/O scheduler needs to be changed and/or tuned to ensure optimal I/O.

```
#!/usr/local/bin/bpftrace
#include <linux/blkdev.h>
BEGIN
{
 printf("Tracing block I/O schedulers. Hit Ctrl-C to end.\n");
}
kprobe:elv_rb_add
{
 @start[arg1] = nsecs;
}

kprobe:blk_mq_start_request
/@start[arg0]/
{
 $r = (struct request *)arg0;
 @usecs[$r->q->elevator->type->elevator_alias] =
 hist((nsecs - @start[arg0]) / 1000);
 delete(@start[arg0]);
}
END
{
 clear(@start);
}
```

Figure 4.25: Latency Distribution of the mq-deadline I/O scheduler

Result

Using an updated version of Brendan Gregg's tool iosched we were able to detect I/O scheduler latency. The individual latency of I/O requests were displayed as a histogram.

[5] https://elixir.bootlin.com/linux/latest/source

4.7 Network Stack

In this section we shall investigate a case of performance analysis pattern that relates to the TCP/IP network stack and evaluate how it affects application performance.

4.7.1 TCP Re-Transmits

Data Source

A trace obtained from Gregg's bpftrace tool tcpretrans [36] fetches TCP retransmit information such as the timestamp, PID, local address(LADDR), local TCP port(LPORT), remote IP address(RADDR) and remote TCP port(RPORT) along with the connection state. The tool uses the tcp_retransmit_skb tracepoint in order to achieve this. The argument obtained by the script from the probe is of type *struct sock**[6] which is the Linux network layer abstraction of sockets. Based on the IPv4 address family, the script determines if the socket is communicating with a AF_INET (IPv4) or AF_INET6 (IPv6) address and uses the inbuilt bpftrace function *ntop* to convert the IP address to text for printing. Similarly, the connection state is fetched from the struct sock *__sk_common.skc_state* attribute. The states can be ESTABLISHED, SYN_SENT, SYN_RECV and so on. The states could also provide clues regarding network issues. For example, a high rate of SYN_SENT could indicate a server which is unable to clear its SYN backlogs [36].

Detection Method

The rate of TCP re-transmits can be monitored via StreamLab. After making a few adjustments in Gregg's bpftrace script which included changing the time format to nanoseconds (Fig 4.26) and to ensure that the output is written to a file in CSV (Comma-separated-value) format, the file is fed to the monitor and the rate of retransmit events is counted over a period of time. The period can be set by the user to be in seconds or minutes. A threshold can be fixed which, upon exceeding, can fire a trigger leading to the following specification.

input PID: UInt64

[6] https://www.kernel.org/doc/htmldocs/networking/API-struct-socket.html

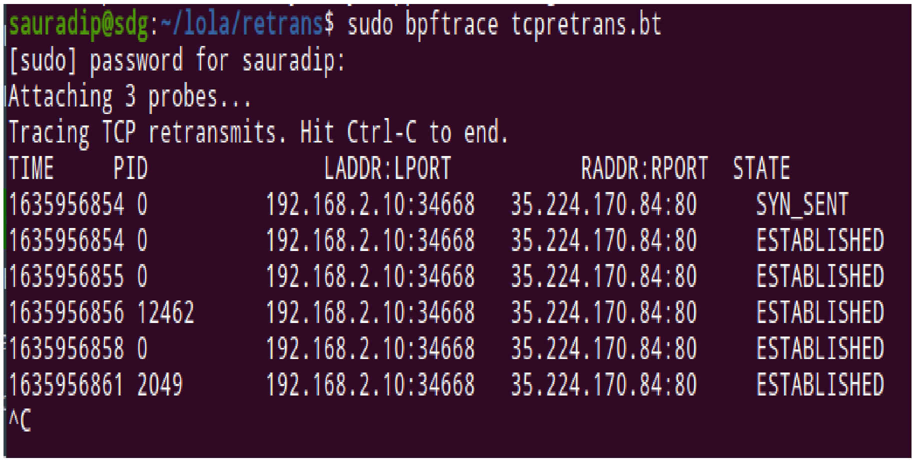


Figure 4.26: An updated tcpretrans bpftrace script to show timestamps in nanoseconds

output retransEventCount @1Hz := PID.aggregate(over: 1.1s, using: count)

trigger count >**Th** "TCP Retransmit count exceeded given threshold".

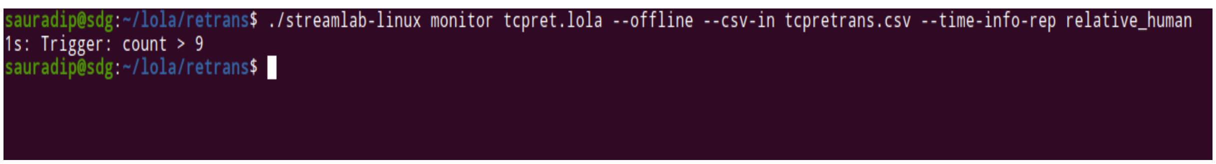


Figure 4.27: Output of the RTLola specification for Th = 9

Detection Metrics

This specification in RTLola will trigger an alert every time the number of TCP Retransmit events is greater than **Th**. The input stream is taken as the stream of PIDs that witnesses re-transmits and a periodic output stream retranEventCount is initialized which would count the number of retransmit events in a sliding window of 1.1sec, at a frequency of 1Hz. The frequency and the duration of the sliding window can be changed as per requirement. In our experiment, we set the threshold to be 9 (Fig 4.27) and RTLola triggers the alert.

Result

We used a slightly modified version of Gregg's tool tcpretrans in order to generate a CSV file on which we run a RTLola specification to determine the frequency of TCP re-transmit events.

If the number of such events over a sliding period exceeds a given threshold then one can start investigating network congestion.

4.8 Synchronization Context

In this section, we shall be evaluating performance analysis patterns related to synchronization (usage of locks in order to control access to shared resources) in multi-threaded applications. Writing efficient multi-threaded code can be difficult. Therefore, tracing can provide us visibility into synchronized code and help detect performance patterns due to improper or badly optimized synchronization.

4.8.1 Excessive Busywaiting

Data Source

Busy waiting can be brought about by the use of spin locks [104]. A spin lock is a type of lock which makes a thread 'spin', i.e. wait in a loop till the lock is available. While the thread spins, it appears to the OS that it is doing busy work but in reality it is just wasting CPU cycles. Spin locks are used to avoid the overhead of invoking function calls for mutex synchronization and context switches in situations when it is known that the length of the critical section will be short. Spin locks can be harmful too if it makes a thread busy-wait for long durations. Especially if the OS interrupts the thread holding the lock, in which case the waiting threads will keep spinning indefinitely. User-space spin locks are also prone to race conditions and long spin times [105].

We wrote a custom java code with two threads where the first thread (TID=5007) acquires a spin lock and goes to sleep for 1 second.

Detection Method

In order to detect excessive busy-waiting in user-space applications, the user space must be instrumented along with kernel space. In order to do that the Pthreads library can be instrumented using the LD_PRELOAD technique with a pthread_spinlock wrapper class which traces spin lock request, acquisition and release.(Fig 4.28) We trace the spinning time of the second thread (TID=5008)

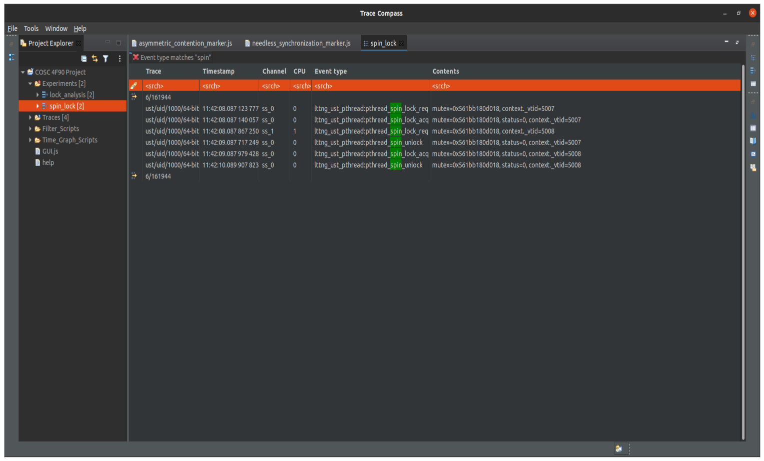


Figure 4.28: User Space instrumentation helps trace spin lock requests, acquisitions and releases

as it tries to acquire the lock shortly after the first thread does the same. We generated a user-space trace as well as a kernel trace in order to trace lock acquisition and the busy-waiting. We can see in the resource view (Fig 4.29) that Thread 5008 spends a lot of time on CPU (horizontal green bar) We also wrote an EASE script which shows the spinning time of both threads where we can see how Thread 5008 spends 1 second spinning in Figure 4.30.

Detection Metrics

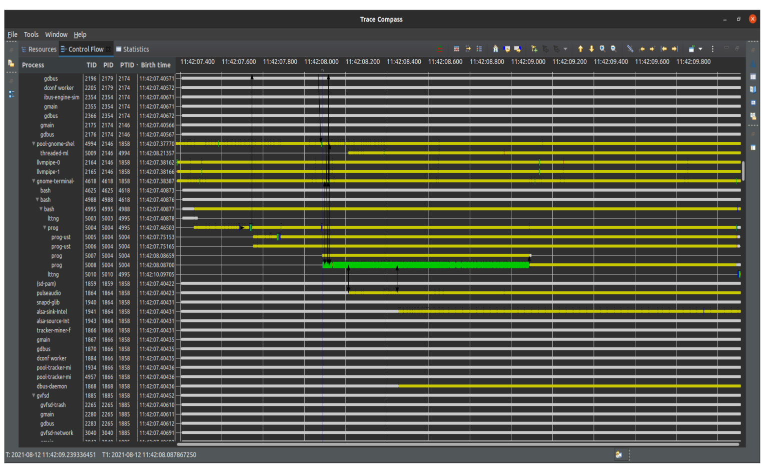


Figure 4.29: Trace Compass resource view shows Thread 5008 spending considerable time on CPU

The primary detection metric used here is the amount of time a thread spins, waiting to access the spin lock held by another thread. We calculate this time by subtracting the time when the thread gives up the spin lock from the time when the thread acquires it.

Result

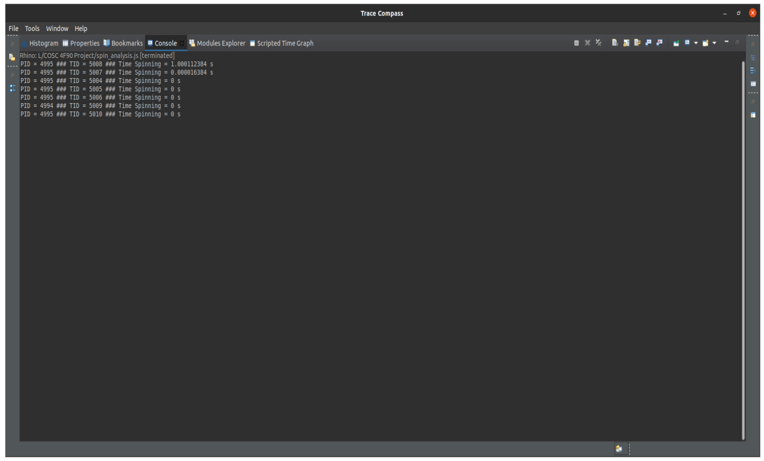


Figure 4.30: Our EASE script calculates spin-lock duration

Hence, we use Trace Compass and a complementary EASE script to detect busy waiting times. If an application sees too many threads busy waiting then its performance can be improved upon by modifying the locking design.

4.8.2 Asymmetrical Contention

Data Source

We wrote a custom Java application featuring a data structure containing ten lists of integers. List 0 is for integers 0 - 9, List 1 is for integers 10 - 19, List 2 is for integers 20 - 29 and so on. Each list has its own lock. The code creates 10 threads, named Thread 0 to Thread 9. The first five threads put random integers between 0 and 99 into the data structure. The second five threads put random integers between 0 and 9 into the data structure.

This results in a significantly higher contention rate for lock 0 as more threads are trying to access list 0 concurrently. The other locks in comparison do not see much contention activity at all. This is asymmetrical contention where locks synchronizing similar data see a skewed contention ratio.

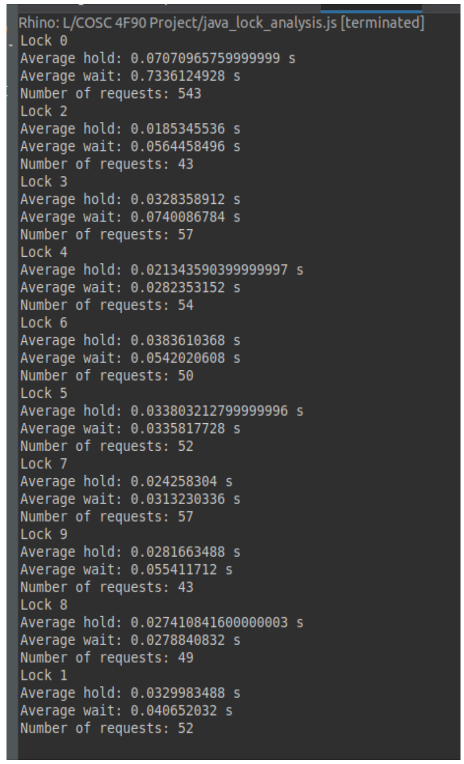


Figure 4.31: Our EASE script showing lock contention statistics

Detection Method

Our application was instrumented using the Java logging API and LTTng user space tracing. There are other techniques to instrument code in non-invasive ways as well but that is not the scope of this work. After the user-space traces were collected, we wrote an EASE script which parses the trace information to print lock contention statistics such as average wait times, average hold times and number of lock acquisition requests. We see that lock 0 (Fig 4.31) is requested twelve times more than all the other locks in the data structure. Naturally, the wait times were significantly higher as well. LTTng user space tracing allows us a fine-grained view of the codebase. The method name and class name of the locks feature in the trace information which makes it clear that the locks are from the same data-structure.

Detection Metrics

The detection metrics used here are average lock hold duration, average lock wait duration and number of lock requests. These metrics are calculated from the trace file using EASE scripting. If the number of lock requests for a particular lock is way higher than other locks, then asymmetrical contention has been detected.

Result

We use LTTng user space tracing along with Java logging API to detect asymmetric contention. We extract some useful synchronization metrics which would help us monitor lock requests, long holds and average lock wait-times.

4.8.3 Excessive Synchronization

Data Source

We modified the Java application we wrote to demonstrate asymmetric contention by simply adding another lock which protects the entire data structure. It would be helpful to recall that the data structure is comprised of 10 lists with one lock per list and 5 threads are trying to insert numbers into these lists. Each thread has to acquire, access and release the lock for every operation. When we inserted another lock which regulated access to the entire data structure (let us call this lock Main), we observed that lock X has a tremendously high rate of contention compared to the other locks guarding the individual lists. This is an example of excessive synchronization where well meaning synchronization primitives are used inefficiently which leads to performance hits.

Detection Method

We used the same EASE script that we wrote to generate lock statistics and as the Figure 4.32 shows, the lock Main witnesses an extremely high number of requests. Since the thread has to acquire another lock in order to push a number in the list, the average waiting time for other threads looking to acquire lock Main becomes very high as well.

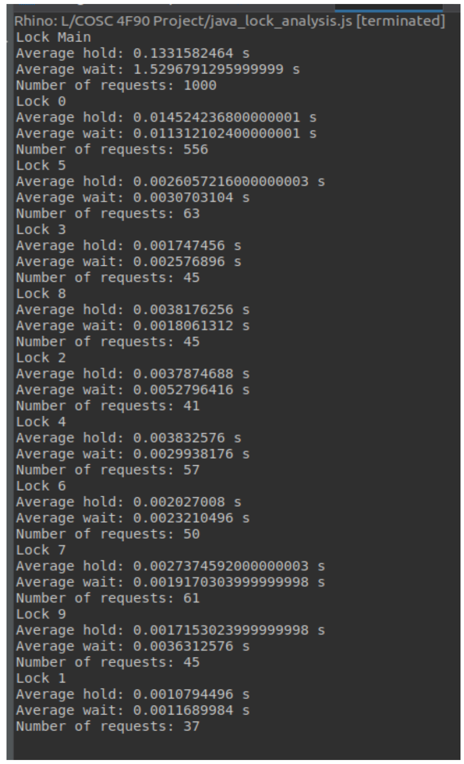


Figure 4.32: Our EASE script shows the number of requests each lock witnesses

Detection Metrics

The main detection metric used here is the number of lock requests for each lock. If an individual lock shows a high number of requests compared to other locks in the same module then it could be an instance of excessive synchronization and the lock can be potentially removed after manual inspection.

Result

We show how redundant locks can affect performance using tracing. Therefore these locks must be removed. It is necessary to maintain a fine balance during synchronization when it comes to lock granularity.

4.9 Application Context

In the following section we evaluate performance analysis patterns by tracing the application context itself. We monitor the application performance with respect to its environment and detect performance issues and avenues of optimization.

4.9.1 Page Writeback Latency

Data Source

Detecting abnormal write activities can involve keeping an eye on writeback latency and/or number of pages written. Any abnormal spike can be monitored and their timestamps noted and correlated with any application delays. The reason why the writeback occurred can also provide a clue. Normally synchronous writeback operations induce latency because it could mean that the *dirty_limit* is surpassed which leads to all writeback operations surpassing the cache and becoming synchronous [82][62].

Brendan Gregg's bpftrace tool writeback's output [36] can be modified to generate a trace which can be fed to a run-time monitoring framework such as RTLola.

```
Lola thriceavg final

input PAGES: UInt64

output count:= (if (PAGES > 0) then 1 else 0) + count.offset(by: -1).defaults(to:
0)
output sum:= PAGES + sum.offset(by: -1).defaults(to: 0)
output avg:= sum/count
output thriceAvg:= 3 * avg.offset(by: -1).defaults(to: 1)

output spike:= (if(PAGES > thriceAvg) then 1 else 0)

trigger spike > 0
"Spike in written pages detected!"
```

Figure 4.33: Our custom RTLola specification that triggers if a spike in writeback activity is observed

Detection Method

RTLola can read the trace as input streams and calculate a running average. The user can set a trigger when the latency or the number of pages written suddenly spike up with respect to the running average. These spikes can be immediately investigated. The RTLola specification is as shown in Fig 4.33 where the input is a stream of integer numbers called PAGES which we extract from the run-time trace of writeback activities. Variable *count* is used to count the number of non zero entries in the stream and a running average is calculated by dividing the sum of the pages with the count. Note the use of the offset function. It implies that any given entry in the stream is mathematically computed with the previous stream entry. If the given entry is the first entry, then a default value is chosen. For example, in the output stream count, the first non-zero entry is added to default value 0. The subsequent entries are added to the previous entry. Thus, for every entry in the PAGES stream, we have a running average as well as a stream variable thriceAvg which stores a value thrice the running average. If any item PAGES exceeds thriceAvg, then we witness a spike in the number of pages written with respect to concurrent page writing activities. This immediately raises a trigger. The trigger threshold can be changed to any factor as per requirement.

Detection Metrics

A stream thriceAvg is initialized which checks if the current number of pages in writeback exceeds thrice the running average. If that is the case, then a spike alert is triggered. The factor of 3 can be modified as per the user's requirement.

Result

We use the bpftrace tool writeback in tandem with our RTLola specification in order to detect sudden spikes in page writeback activity which might cause system freezes.

4.9.2 Abnormally Long Application Sleep Duration

Data Source

The bpftrace tool *naptime* written by Brendan Gregg [36] can be used with some minor modifications to the output such as changing the timestamp format and parameter order. In this case, we would need the name of the process (COMM), the process ID (PID), the parent process ID (PPID) and the parent process name (PCOMM). The sleep duration can be obtained from the function argument which is a struct of type timespec [7]. This struct has two variables tv_sec which is the time duration in seconds and tv_nsec which is the duration in nanoseconds which can be printed by the script along with the other relevant information listed above. If we are to use LTL with BeepBeep monitor in order to detect abnormally long sleep duration then the trace must be converted to XML [46]. If we use RTLola for detecting abnormal spikes in sleep durations then the trace must be in CSV format[8].

Detection Method I

We use LTL as the first detection method, where we specify that for a given process with ID = x, no sleep duration must exceed a given upper time limit Y. The formalization is:

$$\mathbf{G}\neg(PID = x \wedge tv\_sec > Y)$$

Detection Method II

The second formalization can be done via RTLola using the same algorithm as discussed in 4.9.1 where we can simply replace the input stream with the sleep duration in seconds.

We can use RTLola to monitor the sleep durations in real-time and raise an alert whenever the algorithm comes across a duration which is significantly greater than the average sleep times detected by the bpftrace script. We can term these anomalous sleep durations as *spikes.* The quantification of these spikes can be chosen by the user. For example, we can use RTLola to trigger sleep durations which are thrice the average sleep duration times captured by the tracing infrastructure.

[7] https://en.cppreference.com/w/c/chrono/timespec
[8] https://www.react.uni-saarland.de/tools/rtlola/

Detection Metrics

The bpftrace script fetches application sleep durations from the nanosleep system call[9]. The sleep time can be in seconds of nanoseconds. We can trace application sleep times in order to check for any anomalous entries.

Result

We trace the syscalls:sys_enter_nanosleep tracepoint to detect abnormally large application sleep durations. We use an LTL specification as well as an RTLola specification to check for abnormal sleep durations.

4.10 Discussion

In this chapter, we detected and extracted some of the performance analysis patterns that we discussed in Chapter 3. We used different tools in order to achieve this, based on the complexity of the pattern. Some patterns were simple and could be detected using a simple LTL specification. Detecting other patterns were more complex- requiring sophisticated scripting on the trace files in order to extract useful metrics which would characterize the pattern.

We can, therefore, classify these performance analysis patterns in order of their increasing complexity as well.

- Patterns detected and formalized using Linear Temporal Language (LTL)
- Patterns detected and formalized using LTL-FO+ and LTL-TK
- Patterns detected and formalized using run-time monitoring (RTLola)
- Patterns detected using bpftrace scripting
- Patterns detected and analysed using Trace Compass & EASE scripting

Most of the traces that we worked with were generated by us, ensuring that the performance pattern that we were trying to detect was present during the recording of the trace. This could be

[9] https://man7.org/linux/man-pages/man3/nanosleep.3p.html

a limitation as we have not used traces from real-life open source applications. Using traces from real-life applications would be a part of our future work.

Chapter 5

Conclusion & Future Work

5.1 Conclusion

In this thesis, we detected, extracted, categorized and formalized several performance analysis patterns from various Linux subsystems. In order to detect these patterns we first traced the system using LTTng while making sure that the performance issue we were trying to detect occurred while the tracing session was on. In some cases, we designed custom applications which would simulate said performance issues while in other cases we traced Linux routines themselves such as *oom_kill()*.

Once we had the trace file we extracted the patterns using different techniques such as applying temporal logic, run-time monitoring and scripting. In order to achieve this, we wrote custom specifications in LTL, LTL-FO+, LTL-TK and RTLola and ran those specifications against the traces. We had to pre-process the trace files so that it had only the necessary metadata in a given format. In case of LTL, the format was XML whereas for RTLola it was CSV. We demonstrated that it was possible to detect and formalize simpler patterns using temporal logic and run-time monitoring alone. For more complex patterns, we used EASE scripting in order to leverage its state system properties to extract and visualize the patterns. We also used bpftrace tools and custom bpftrace scripting in order to detect some of the patterns involving scheduler and compaction latencies.

We categorized these patterns based on the Linux subsystems they primarily effect. We also wrote about the detection and formalization of these patterns in the increasing order of complexity. Some patterns such as memory fragmentation or abnormal application sleep durations can be easily detected using run-time monitoring and temporal logic. Other patterns such as excessive busy-waiting and asynchronous contention requires advanced scripting and visualization graphs in order to be detected. Also, they are too complex to be formalized into a specification. We would require advanced stream processing logic and scripting for the same.

5.2 Future Work

This catalogue is by no means exhaustive. New patterns would be added to the existing catalogue in the future as tracing systems become more robust and developers detect new bugs and optimization tricks. Some of the possible directions that this project can take are as follows:

- Addition of new performance analysis patterns along with their formalizations.
- Exploring other temporal logic to express these patterns.
- Using the work done on detecting these patterns to build tools that can be integrated with Trace Compass so that developers can use them in order to build better software.
- Using trace files from real-life open source projects in order to detect patterns.
- Using deep learning techniques to mine traces for performance issues.
- Generalizing our patterns so that they can be valid for other operating systems as well.

5.3 Concluding Remarks

Personally speaking, the COVID 19 pandemic made it difficult to conduct research. Classrooms and seminars were replaced by Zoom meetings and break-out rooms. It made me realize how important physical presence and personal interaction are when it comes to learning and research. Nevertheless, this project gave me an invaluable opportunity to deep dive into operating systems and system performance engineering. Prior to this, I had never worked with Linux. But now, I can confidently say that I have a reasonable understanding of how Linux works and I look forward to working on the biggest open source project there is.

Bibliography

[1] R. Khoury, S. Hallé, and Y. Lebrun, "Automata-based monitoring for ltl-fo+," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 2, pp. 137–154, 2021.

[2] P. Gepner and M. Kowalik, "Multi-core processors: New way to achieve high system performance," in *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, 2006, pp. 9–13.

[3] J. C. Knight, "Safety critical systems: Challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering*. Association for Computing Machinery, 2002, p. 547–550.

[4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.

[5] M. M. Lehman, "Laws of software evolution revisited," in *European Workshop on Software Process Technology*. Springer, 1996, pp. 108–124.

[6] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering*, vol. 1, no. 1, pp. 1–22, 1995.

[7] M. J. Quinn, "Parallel programming," *The McGraw-Hill Companies*, vol. 526, p. 105, 2003.

[8] J. L. Ortega-Arjona, *Patterns for parallel software design*. John Wiley & Sons, 2010.

[9] C. Sridharan, *Distributed systems observability: a guide to building robust systems*. O'Reilly Media, 2018.

[10] A. Hamou-Lhadj and T. C. Lethbridge, "Compression techniques to simplify the analysis of large execution traces," in *Proceedings of 10th International Workshop on Program Comprehension*. IEEE, 2002, pp. 159–168.

[11] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 181–190.

[12] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, 2004, pp. 42–55.

[13] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–33, 2018.

[14] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[15] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, "Probing the guts of kprobes," in *Linux Symposium*, vol. 6, 2006, p. 5.

[16] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, and M. Desnoyers, "Recovering system metrics from kernel trace," in *Linux Symposium*, vol. 109, 2011.

[17] L. Alawneh, A. Hamou-Lhadj, and J. Hassine, "Segmenting large traces of inter-process communication with a focus on high performance computing systems," *Journal of Systems and Software*, vol. 120, pp. 1–16, 2016.

[18] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, and A. Mehrabian, "The concept of stratified sampling of execution traces," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 225–226.

[19] M. Côté and M. R. Dagenais, "Problem detection in real-time systems by trace analysis," *Advances in Computer Engineering*, vol. 2016, 2016.

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

[21] H. Daoud and M. R. Dagenais, "Recovering disk storage metrics from low-level trace events," *Software: Practice and experience*, vol. 48, no. 5, pp. 1019–1041, 2018.

[22] R. Khoury, S. Hallé, and O. Waldmann, "Execution trace analysis using ltl-fo+," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 356–362.

[23] F. Doray and M. Dagenais, "Diagnosing performance variations by comparing multi-level execution traces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, 2016.

[24] P.-M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 77–87, 2010.

[25] N. Ezzati-Jivan, Q. Fournier, M. R. Dagenais, and A. Hamou-Lhadj, "Depgraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 149–159.

[26] C. LaRosa, L. Xiong, and K. Mandelberg, "Frequent pattern mining for kernel trace data," in *Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 880–885.

[27] S. D. I. Dmitry Vostokov, *Encyclopedia of Crash Dump Analysis Patterns, Third Edition*. OpenTask, 2020.

[28] D. Vostokov and S. Institute, *Software Trace and Log Analysis: A Pattern Reference*. Opentask, 2015.

[29] C. Alexander, *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[30] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *Tapos*, vol. 2, no. 1, pp. 3–13, 1996.

[31] J. Heer and M. Agrawala, "Software design patterns for information visualization," *IEEE transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 853–860, 2006.

[32] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts, 1995, vol. 99.

[33] A. Hamou-Lhadj, "Techniques to simplify the analysis of execution traces for program comprehension," *Ph.D. Dissertation, School of Information Technology and Engineering (SITE), University of Ottawa*, 2005.

[34] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah, "Exploiting text mining techniques in the analysis of execution traces," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 223–232.

[35] H. Pirzadeh, S. Shaniam, A. Hamou-Lhadj, L. Alawneh, and A. Sharifee, "Stratified sampling of execution traces: Execution phases serving as strata," *Elsevier Journal of Science of Computer Programming*, vol. 78, no. 8, pp. 1099–1118, 2013.

[36] B. Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[37] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[38] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 56–65, 2009.

[39] B. d'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "Lola: runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. IEEE, 2005, pp. 166–174.

[40] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah, “Streamlab: stream-based monitoring of cyber-physical systems,” in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 421–431.

[41] P. Faymonville, B. Finkbeiner, M. Schwenger, and H. Torfah, “Real-time stream-based monitoring,” *arXiv preprint arXiv:1711.03829*, 2017.

[42] L. Meertens, “Algorithmics: Towards programming as a mathematical activity,” 1986.

[43] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “No pane, no gain: efficient evaluation of sliding-window aggregates over data streams,” *Acm Sigmod Record*, vol. 34, no. 1, pp. 39–44, 2005.

[44] B. Finkbeiner, S. Oswald, N. Passing, and M. Schwenger, “Verified rust monitors for lola specifications,” in *International Conference on Runtime Verification*. Springer, 2020, pp. 431–450.

[45] M. Schwenger, “Monitoring cyber-physical systems: From design to integration,” in *International Conference on Runtime Verification*. Springer, 2020, pp. 87–106.

[46] S. Hallé and R. Villemaire, “Browser-based enforcement of interface contracts in web applications with beepbeep,” in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 648–653.

[47] S. Hallé and R. Villemaire, “Runtime enforcement of web service message contracts with data,” *IEEE Trans. Serv. Comput.*, vol. 5, no. 2, pp. 192–206, 2012.

[48] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[49] R. Khoury and S. Hallé, “Tally keeping-ltl: An ltl semantics for quantitative evaluation of ltl specifications,” in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 2018, pp. 495–502.

[50] D. Vostokov, *Memory Dump Analysis Anthology*. OpenTask, 2008, vol. 1.

[51] ——, *Software narratology: introduction version 1.0*. OpenTask, 2012.

[52] ——, *Windows Software Trace Analysis Accelerated*. OpenTask, 2013.

[53] M. Idris, A. Mehrabian, A. Hamou-Lhadj, and R. Khoury, "Pattern-based trace correlation technique to compare software versions," in *International Conference on Autonomous and Intelligent Systems*. Springer, 2012, pp. 159–166.

[54] G. Matni, "Detecting problematic execution patterns through automatic kernel trace analysis," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[55] A. Tate and L. Bewoor, "Survey on frequent pattern mining algorithm for kernel trace," in *2017 IEEE 7th International Advance Computing Conference (IACC)*. IEEE, 2017, pp. 793–798.

[56] I. F. D. M. Junior, A. Benbachir, and M. Dagenais, "Performance analysis using automatic grouping," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 381–387.

[57] M. Rezazadeh, N. Ezzati-Jivan, E. Galea, and M. R. Dagenais, "Multi-level execution trace based lock contention analysis," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 177–182.

[58] F. Giraldeau and M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2450–2461, 2015.

[59] R. Nair and T. Field, "Gapp: A fast profiler for detecting serialization bottlenecks in parallel linux applications," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 257–264.

[60] D. J. Dean, H. Nguyen, P. Wang, and X. Gu, "Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds," in *6th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[61] M. M. U. Alam, T. Liu, G. Zeng, and A. Muzahid, "Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 298–313.

[62] A. Méndez Orero, "Analysis and mitigation of writeback cache lock-ups in linux," B.S. thesis, Universitat Politècnica de Catalunya, 2020.

[63] R. Matias, I. Beicker, B. Leitão, and P. R. Maciel, "Measuring software aging effects through os kernel instrumentation," in *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*. IEEE, 2010, pp. 1–6.

[64] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Ottawa Linux Symposium*, vol. 1, 2006, pp. 369–384.

[65] B. Randell, "A note on storage fragmentation and program segmentation," *Communications of the ACM*, vol. 12, no. 7, pp. 365–ff, 1969.

[66] A. Macêdo, T. B. Ferreira, and R. Matias, "The mechanics of memory-related software aging," in *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*. Ieee, 2010, pp. 1–5.

[67] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 279–287.

[68] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[69] U. Vahalia, *UNIX internals: the new frontiers*. Pearson Education India, 1996.

[70] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[71] W. Gloger, "Ptmalloc (2006)," 2006.

[72] A. Suryavanshi and S. Sharma, “An approach towards improvement of contiguous memory allocation linux kernel: a review,” *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 25, no. 3, pp. 1607–1614, 2022.

[73] Q. Ni, W. Sun, and S. Ma, “Memory leak detection in sun solaris os,” in *2008 International Symposium on Computer Science and Computational Technology*, vol. 2. IEEE, 2008, pp. 703–707.

[74] H. Daoud, N. Ezzati-jivan, and M. R. Dagenais, “Dynamic trace-based sampling algorithm for memory usage tracking of enterprise applications,” in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[75] D. A. Rusling, “The linux kernel,” 1999.

[76] L. Brindley, “Red hat enterprise mrg 1.1 realtime tuning guide,” 2008.

[77] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, “Translation lookaside buffer consistency: A software approach,” *ACM SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 113–122, 1989.

[78] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[79] M. Gorman and A. Whitcroft, “Supporting the allocation of large contiguous regions of memory,” in *Ottawa Linux Symposium (OLS)*, 2007, pp. 141–152.

[80] N. Gupta, “Proactive compaction for the kernel,” Apr 2020. [Online]. Available: https://lwn.net/Articles/817905/

[81] A. Tanenbaum, *Modern operating systems*. Pearson Education, Inc.,, 2009.

[82] R. Love, *Linux kernel development*. Pearson Education, 2010.

[83] J. Corbet, “Per-entity load tracking,” Jan 2013. [Online]. Available: https://lwn.net/Articles/531853/

[84] P. U. Murthy, "Per-entity load tracking," Apr 2015. [Online]. Available: https://lwn.net/Articles/639543/

[85] OSPM, "Reworking cfs load balancing," Jul 2019. [Online]. Available: https://lwn.net/Articles/793427/

[86] J. Corbet, "Imbalance detection and fairness in the cpu scheduler," May 2020. [Online]. Available: https://lwn.net/Articles/821123/

[87] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2015, pp. 1–10.

[88] J. Xu and D. L. Parnas, "Priority scheduling versus pre-run-time scheduling," *Real-time systems*, vol. 18, no. 1, pp. 7–23, 2000.

[89] J. Saltzer and M. F. Kaashoek, *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[90] B. P. Douglass, *Real-time UML workshop for embedded systems*. Newnes, 2014.

[91] S. Davari and L. Sha, "Sources of unbounded priority inversions in real-time systems and a comparative study of possible solutions," *ACM SIGOPS Operating Systems Review*, vol. 26, no. 2, pp. 110–120, 1992.

[92] J. Nieplocha, A. Márquez, J. Feo, D. Chavarría-Miranda, G. Chin, C. Scherrer, and N. Beagley, "Evaluating the potential of multithreaded platforms for irregular scientific computations," in *Proceedings of the 4th International Conference on Computing Frontiers*. New York, NY, USA: Association for Computing Machinery, 2007, p. 47–58.

[93] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps," *ACM Sigplan Notices*, vol. 43, no. 3, pp. 277–286, 2008.

[94] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[95] U. Drepper and I. Molnar, "The native posix thread library for linux," 2003.

[96] J. Lee, H. Wu, M. Ravichandran, and N. Clark, "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 270–279.

[97] B. Gregg, *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[98] R. Love, "Kernel korner - i/o schedulers," Feb 2004. [Online]. Available: https://www.linuxjournal.com/article/6931

[99] O. Yunus and A. Yildirim, "Performance comparison and analysis of linux block i/o schedulers on ssd," *Sakarya University Journal of Science*, vol. 23, no. 1, pp. 106–112, 2019.

[100] D. A. Heger and R. Quinn, "Linux 2.6 io performance analysis, quantification, and optimization." in *International CMG Conference*, 2010.

[101] B. N. K. Reddy, N. Venktram, and T. Sireesha, "An efficient data transmission by using modern usb flash drive," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 9, no. 10, pp. 271–282, 2014.

[102] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Computing Surveys (CSUR)*, vol. 26, no. 2, pp. 145–185, 1994.

[103] B. A. Forouzan, *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[104] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[105] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, “Automated dynamic detection of busy–wait synchronizations,” *Software: Practice and Experience*, vol. 39, no. 11, pp. 947–972, 2009.

[106] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, “Accelerating critical section execution with asymmetric multi-core architectures,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, pp. 253–264, 2009.

[107] K. G. Lockyer, “Introduction to critical path analysis,” 1969.

[108] G. Chen and P. Stenstrom, “Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications,” in *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[109] V. Alessandrini, *Shared Memory Application Programming: Concepts and strategies in multicore application programming*. Morgan Kaufmann, 2015.

[110] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, “Analyzing lock contention in multithreaded applications,” in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010, pp. 269–280.

[111] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[112] P. J. Denning, “The working set model for program behavior,” *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[113] A. J. Smith, “Cache memories,” *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[114] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[115] J. E. Smith, “A study of branch prediction strategies,” in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 202–215.

[116] P. Zhang, *Advanced industrial control technology*. William Andrew, 2010.

[117] B. Lewis and D. J. Berg, *Multithreaded programming with Java technology*. Prentice Hall Professional, 2000.

[118] A. Panwar, N. Patel, and K. Gopinath, "A case for protecting huge pages from the kernel," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2016, pp. 1–8.