**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

# Performance analysis of complex multi-thread applications through critical path analysis

**MAJID REZAZADEH**
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Novembre 2019

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

Ce mémoire intitulé :

**Performance analysis of complex multi-thread applications through critical path analysis**

présenté par **Majid REZAZADEH**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

**Gilles PESANT**, président
**Michel DAGENAIS**, membre et directeur de recherche
**Michel GAGNON**, membre

## DEDICATION

*I would like to dedicate this thesis to my parents,*
*my lovely wife and*
*my brothers...*

# ACKNOWLEDGEMENTS

I would like to thank all those who have supported me during my studies. I am deeply grateful to my supervisor, Professor Michel Dagenais, for his continuous encouragement and understanding. I am really proud of having such a great supervisor. I learned a lot from his nice personality which will help me in the future.

I acknowledge all my committee members, Professors Gilles Pesant and Michel Gagnon, to review and evaluate my research contributions.

My time at Polytechnique Montreal was a memorable journey with many adventures which can help me in the future. I would like to praise Polytechnique Montreal and the department of Computer and Software Engineering, with its helpful employees, for giving me the opportunity of studying in this prestigious institute.

I would like also to thank my colleagues at the DORSAL laboratory, for their valuable comments on my development, especially Vahid, Geneviève and Naser. They made my journey possible with their precious comments.

Most importantly, I want to thank my parents for providing me the opportunity to pursue my studies and also for all of their years of encouragement, support, patience and love which are undoubtedly the basis of all of my successes in my life.

Finally, my heartfelt and warm thanks to Mahshid, my lovely wife, for dreaming with me, for her constant encouragement and support, for being patient when I was busy and not available for her, and for providing me her pure love, support and loyalty.

# RÉSUMÉ

La performance est un sujet critique pour les applications multi-fils énormes et complexes. De nombreux facteurs tels que le conflit de ressources, les algorithmes de synchronisation inadéquats, les opérations de disque lentes, etc. peuvent affecter la performance. De nombreux développeurs ne connaissent pas la nature de la cause fondamentale des latences inattendues, en raison de la grande base de code des applications et des multiples niveaux d'abstraction entre le code de l'application et le matériel.

Le traçage est une technique qui enregistre les activités dans un système. De nombreux problèmes de performances peuvent être diagnostiqués à l'aide des informations collectées par une trace d'exécution. En raison du surcoût négligeable des traceurs reconnus, nous pouvons les déployer sur des systèmes en production, pour capturer les bogues qui se produisent rarement. Nous pouvons naviguer efficacement dans le grand nombre d'événements contenus dans les traces d'exécution à l'aide d'outils spécialisés. Cependant, même avec ces outils, il est difficile d'identifier la cause fondamentale des problèmes de performance sans une connaissance approfondie du système analysé. Dans les applications complexes telles que le navigateur Chromium, qui est notre objectif dans ce projet de recherche, le premier défi consiste à pouvoir collecter des informations unifiées et précises à partir de plusieurs couches de l'application et du système.

L'objectif de ce projet de recherche est de proposer une solution pouvant faciliter le diagnostic des problèmes de performance en utilisant une analyse du chemin critique des exécutions. Après avoir expliqué notre méthode de traçage pour collecter simultanément les données de l'application et du noyau du système d'exploitation, nous compilons les données de traçage, collectées à partir de plusieurs couches du système, dans un modèle unifié et les utilisons pour analyser les flux de contrôle au niveau du noyau pour les actions du navigateur au niveau utilisateur. Ceci sert afin de détecter les problèmes de latence et d'identifier leurs causes principales. Nous montrons l'efficacité de la méthode proposée en détectant trois problèmes de performance de Chromium et en trouvant leurs causes fondamentales. Bien que nos cas d'utilisation soient basés sur le navigateur Chromium, la méthode proposée et les leçons apprises sont directement applicables à d'autres applications complexes multi-fils. En outre, nous proposons notre méthode d'extraction de flux critiques des actions des utilisateurs du navigateur, qui est directement utilisée dans la détection et l'analyse de problèmes de latence. À l'aide de l'analyse du chemin critique, nous pouvons identifier les goulots d'étranglement des exécutions qui affectent directement le temps d'exécution total. Par conséquent, ce genre

de problèmes méritent d'être analysés.

À notre connaissance, il s'agit de la première tentative d'extraction automatique de données multi-niveau à partir du navigateur Chromium. Dans ce projet, la plupart des données utilisées pour les évaluations et les expériences ont été collectées par le Linux Trace Toolkit Next Generation (LTTng), un outil très flexible et avec peu de surcoût.

# ABSTRACT

Performance is a critical subject for huge and complex multi-thread applications. Many factors such as resource contention, inadequate synchronization algorithms, slow disk operations, etc. can affect performance. Many developers are not aware of the existence of the root cause of unexpected latencies, due to the large codebase of applications and the multiple levels of abstraction between the application code and hardware.

Tracing is a technique which records events occurring in the system. Many performance issues can be diagnosed using the information gathered by an execution trace. Because of the negligible overhead of popular tracers, we can deploy them on production systems to capture bugs that occur infrequently. We can efficiently navigate in the large number of events, contained in execution traces, using specialized tools. However, even with these tools, it is difficult to identify the real root cause of performance bugs, without a deep knowledge of the analyzed system. In complex applications like the Chromium browser, which is our focus in this research project, being able to collect precise unified information from several layers of the application/system is the first challenge.

The objective of this research project is to propose a solution which can facilitate the diagnosis of performance issues using the critical path analysis of executions. After explaining our tracing method for collecting data, from the application and the operating system kernel simultaneously, we compile the tracing data collected from several layers of the system into a unified model and use it to perform kernel-level control flow analysis for user-level browser actions. This serves to detect latency issues and identify their main causes. We show the effectiveness of the proposed method by detecting three Chromium performance bugs and finding their root-causes. Although our use-cases are based on the Chromium browser, the proposed method and lessons learned are directly applicable to other complex multi-thread applications. In addition, we propose our method on critical flow extraction of browser user actions, which is used directly in latency problem detection and analysis. Using critical path analysis, we can identify the bottlenecks of executions which directly affect the total execution time. Hence, these kinds of problems are worth analysing.

To our knowledge, this is the first attempt to automatically extract multi-level data from the Chromium browser. In this project, most of the data used for evaluations and experiments were collected by the Linux Trace Toolkit Next Generation (LTTng), which is a very flexible tool with low overhead.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS AND ACRONYMS

| | |
|---|---|
| API | Application Programming Interface |
| CPU | Central Processing Unit |
| LTTng | Linux Trace Toolkit Next Generation |
| ETW | Event Tracing for Windows |
| SSH | State History Tree |
| LLC | Logical link Control |
| RCU | Read-Copy Update |
| CPA | Critical Path Analysis |
| CPM | Critical Path Method |
| RCL | Remote Core Locking |
| POSIX | Portable Operating System Interface |
| IPC | Inter-Process Communication |

# CHAPTER 1 INTRODUCTION

The emergence of multi-core processors, the benefits of using multi-thread programming and the arrival of new dynamic programming languages facilitate the development of huge, complex and powerful multi-thread applications. This complexity in different layers of the software architecture makes the debugging task difficult. Developers need to be aware of several problems with multi-thread systems such as critical sections, deadlocking, contention for access to a resource. This task, however, can be complex because of the multiple levels of abstraction that separate the application code from the hardware. In addition, these problems are usually difficult or impossible to diagnose with traditional debuggers and profilers. Tracing is an effective technique for collecting information on problems affecting parallel systems or multi-thread applications. However, it is difficult to analyze a large volume of information in an execution trace without the appropriate tools. This thesis proposes a solution to automatically identify the performance issues of complex multi-thread applications. We focus specifically on Chromium, the open source version of the Google Chrome web browser, to detect and analyze unexpected latencies affecting the user interface at run time. We try to demonstrate that this technique allows developers to detect, diagnose and correct a wide variety of performance problems.

In the following, we define some basic concepts used in this thesis. We introduce the technique used to perform our analysis. We explain the elements of the problem and some causes resulting in performance issues. We then come up with the research question and objectives. At the end, we explain the outline of this study and briefly introduce the remaining chapters of this thesis.

## 1.1 Definitions and basic concepts

### 1.1.1 User action

The term "user action" means the state of Chromium in which the user is interacting with the page through some input mechanism (typing, clicking mouse buttons etc.)[3]. In other words, a user action is an interaction between the user and Chromium that involves potentially multiple nested function calls, resulting in the generation of several tasks sequences. Each user action can result in several interactions with different threads to be accomplished.

### 1.1.2 Task

A task is the smallest work unit to be processed [4]. Inside the scheduler, a task is considered as an object which has a closure, traits and a post timestamp. Tasks are placed in sequences. The order in which different tasks are added to a given sequence determines their execution order. Sequences have per-priority task counters. There are different types of tasks such as default_tq, control_tq, frame_loading_tq, input_tq and etc. Depending on the type of each task, it may be run by a thread in a thread pool. Thus, different threads may interact with each other, to accomplish a user action involving several tasks [4].

### 1.1.3 Jank

The term "jank" denotes an unexpected latency or performance degradation which directly influences the user interface of Chromium [5]. Performance is a highly-cited problem area for web developers. They spend a lot of time and effort to understand the root cause of these problems so that they can improve performance of the platform by eliminating janks [5]. A jank may happen within one or several user actions, impacting the usability of the browser. Many janks have been reported by Chromium users on the issue tracker [6].

### 1.1.4 Tracing

Understanding the exact interactions between system components requires an approach named tracing. This technique consists of recording the time, the type of event and optional data, which form an ordered sequence. The trace can then be analyzed to determine certain properties [7]. A tracer is a tool which is capable of recording events occurring in a computer system. An event consists of a type, a time stamp and data. An event is generated once a program encounters a trace point. A trace point can be inserted statically or dynamically in a user space application or at the kernel level, in the operating system. A user space trace provides a view of the behavior of a system close to the code application. To generate such a trace, it is necessary to instrument each application. A kernel level trace provides a lower level view of the system on which the application is running. It can reveal all the interactions between the applications and the operating system [8]. The authors of [9] denote that a large proportion of performance issues are related to interactions with the operating system. Logging can be considered as a form of tracing. However, tracing generally refers to recording events at a much higher rate. Context switches, receiving network packets, or the occurrence of page faults, are examples of relevant events to be traced for diagnosing a performance problem.

A strength of the main tracers is that they have a minimal impact on the behavior of the systems on which they are used. This feature makes them excellent tools for debugging performance issues happening on complex systems, where the slightest disruption of the system can perturbate their correct behavior. Tracers are also useful to diagnose intermittent errors that occur only in production. Because of their low overhead, they can monitor a system in production for a long period of time, until the error happens again. Like profilers, tracers can reveal the functions of an application which consume processor time, memory, network access, etc. However, they also provide details for each occurrence of an event. This allows, for example, to identify transactions whose execution time distribution is multi-modal. It is also possible to know what is happening on the whole system, when a long delay happens [8].

### 1.1.5 State System

Each state can be defined as the time duration between two state modifying events. We store all the states in the State System. Our State System is an efficient tree-based structure that stores the different states of all attributes. The State History Tree stores the same information historically and can be queried to obtain the state of each attribute, at any point in time during a trace. The general idea of our model is to extract and record the intervals of different state values, from trace events, for system resources (processes, CPUs, disks, etc.). A tree-based organization is used to store the state values [10].

## 1.2 Elements of the problem

In recent years, the developed applications have become ever so powerful and complex, with multiple levels of abstraction and multiple parallel processes and threads. Search engines use thousands of machines to return the results more relevant to each received request in a fraction of a second [11]. Smart phones have multi-core processors allowing them to be both more efficient and less energy intensive. Dynamic languages such as Java, Javascript and Python simplify the development by offering multi-purpose libraries, automatic memory management and portable binaries. In such a context, identifying all the factors which may affect the performance of an application can be a serious challenge for a single developer.

However, performance is extremely important for a large number of applications. Long response times are among the main causes of user frustration identified by [12]. The conversion rate of a commercial website can be reduced by 7% with a latency of 1 second [13]. An excessive latency can even result in the failure of a project [14].

Profilers help diagnose the simplest performance problems. However, we have seen in Section 1.1.4 that for problems that stem from the interaction between several components, or that occur intermittently, using a tracer is more helpful. Trace analysis, however, is expensive because of the large the amount of information contained in traces. In addition, a large proportion of this information is not often related to the target problem. The trace viewers simplify the navigation through a large number of events. However, they require a good knowledge of the system analyzed, to judge whether the results obtained correspond to the expectations or not. It is also necessary to have a preliminary idea of the location of the problem in the trace. There are expert systems which are capable of automatically detecting problems. However, the detection of component-specific problems is possible only if some rules have been written for it. Writing and maintaining these rules is expensive. There are also chances of missing some components, leading to an incomplete diagnosis.

Several performance issues stem from the variations between duration of several executions of the same user action. These variations can occur in several situations:

- **Updating** an application, dynamic libraries or the operating system.
  *Example:* A MySQL server is migrated from a SmartOS machine to a Linux machine. The query performance for the database is 30% lower on the new machine [15].

- Sporadic **interaction** between multiple tasks sharing the same resource.
  *Example:* Two repetitive tasks are confined to the same processor. The first runs at 1 Hz and the second at 50 Hz. Occasionally, both tasks attempt to run simultaneously, increasing their duration.

- Programming **error**.
  *Example:* Items are added regularly to a cache that is never cleaned. The duration of executions increases gradually, because they must browse the cache linearly [16].

- Different **system load**.
  *Example:* Writing to a database is protected by a lock. The duration of a query to a website increases when there is a large number of concurrent users, due to the contention on the write lock.

This research assumes that the traces of a long and a normal execution exhibit differences which are sufficient to explain the cause of the variation in performance. We are looking for a solution that can automatically extract information to analyze the abnormal executions, in order to significantly accelerate the diagnosis of a wide variety of performance issues.

The trace of a huge multi-thread application like Chromium normally contains events belonging to several separate user action executions. To distinguish the abnormal ones among all the executions, we need to mark the beginning and end of each user action. Note also that we want to include all the factors that can influence the total duration of the executions in our analysis. Therefore, we must be able to track all the dependencies between execution threads as well as the interactions with the operating system. Because of these needs, we use a multi-level critical path analysis to identify the performance bottlenecks of Chromium either at user or kernel level. Looking at the critical path of any execution, we can precisely detect the root cause of performance degradation happening in kernel or user space.

## 1.3 Research objectives

The work presented in this thesis aims to answer the following research question:

Is it possible to design and implement a solution that speeds up the diagnosis of performance issues by automatically extracting information and identifying the root cause of problems at both kernel and user level?

To answer this question, we will try to achieve the following specific objectives:

1. Develop an efficient technique for collecting information, linking the low-level events of the operating system and the code of Chromium in user space.
2. Develop a method to distinguish the events of a trace belonging to different user action executions.
3. Adapt the critical path algorithm to both user and kernel level.
4. Validate our solution using execution traces, showing real performance problems (janks), in which the proposed method can quickly identify the root causes for these issues.

## 1.4 Outline of the research

Chapter 2 presents existing work in the areas of tracing and performance analysis of huge and complex multi-thread applications. Chapter 3 describes the methodology that allowed us to design and evaluate the solution proposed in this thesis. Chapter 4 contains the article "Performance evaluation of complex multi-thread applications through multi-level critical

path analysis" submitted to Software: Practice and Experience. This article details our proposed solution to automatically identify the causes of performance variations between multiple executions of the same task. It also explains how this solution allowed us to correct real performance problems in free and enterprise software. Chapter 5 presents complementary results and discusses contention analysis based on execution traces. Finally, Chapter 6 summarizes the work and proposes future improvements.

# CHAPTER 2 LITERATURE REVIEW

This chapter provides a survey of the state of the art about how to collect and analyze data on the performance of complex multi-thread applications. We describe different tracing tools, alongside their specifications, which can be used to extract and analyze data. We then introduce different profilers which can be utilized for performance analysis. We also talk about trace analysis and how we can build a meaningful analysis from a trace file with the Trace Compass analysis tool. We introduce the critical path as an efficient method to discover performance bottlenecks. At the end, we discuss lock contention analysis as a challenging problem in multi-thread applications. We propose the critical path analysis as a solution which can help developers to discover the code segment resulting in contention. The knowledge presented in this chapter is the basis for the solution which will be proposed in this research.

## 2.1 Observation tools for multi-thread applications

The behavior of multi-thread applications is difficult to observe. In fact, any slightest disturbance caused by an observation tool can vastly change the behavior of the application and make the gathered data invalid. This section presents some tools allowing to observe and track the behavior of multi-thread applications with a minimal impact. As part of our work, we will use these tools to collect data and analyze it at multiple layers.

### 2.1.1 Tracer

A tracer is a tool which is able to record events occurring during the execution of an application. Popular tracers are able to record all the events occurring either in the operating system such as a context change, writing on disk, or receiving a network packet, or in user space applications such as entering a function or receiving an HTTP request [8].

#### Linux Trace Toolkit next generation (LTTng)

LTTng is a tool for tracing the kernel and user spaces on the Linux operating system. The main objective of LTTng is to minimize the overhead of instrumentation on the observed system [17]. Each event can be individually activated. Tremendous efforts were made to ensure that the impact on the performance of the tracepoints is almost negligible.

When a system is instrumented with LTTng, the operating system and applications write events in memory buffers. To ensure scalability to multiple cores, different buffers are associated with each core, eliminating the need for locks. When a buffer is full, a signal is sent to the session daemon. The daemon then copies the buffer to a file or sends it over the network.

LTTng also offers a *flight recorder* mode in which, instead of consuming the buffers automatically, the new events continuously overwrite the older ones. A copy to disk of the buffers can be triggered when needed, for example when an error occurs or too much latency is detected. A trace of the past few seconds before the incident becomes available for analysis. The advantages of the flight recorder mode are to reduce the disturbances of the system when there is no problem and to avoid managing the storage of large traces. A user can write a custom program to monitor the conditions leading to saving the buffers content. However, it is easier and more efficient to use the latency_tracker module [18] if we need to detect a latency within the operating system. This module provides an Application Programming Interface (API) to signal the beginning and the end of arbitrary events. At the begin event, a time stamp is saved in a hash table *read-copy update* (RCU). Then, at the corresponding end event, the total duration is calculated. If this exceeds a predefined threshold, a callback function is executed. The extra cost of this module on the application **hackbench** [19] is 0.3% when all events are generated by the same core and 22% with 32 cores.

The kernel events captured by LTTng are generated by the TRACE_EVENT macro or by the dynamic instrumentation mechanism kprobes. The TRACE_EVENT macro is a simple and universal way to add static instrumentation in the Linux kernel [20] Any tracer can interface with it. The kprobes mechanism is more flexible, as it allows adding instrumentation to arbitrary locations without recompilation. However, it is more expensive in terms of overhead.

A user space application can also generate events. To do that, the developer has to insert a macro specific to LTTng in his C or C++ code and link his application with the **liblttng-ust** library. An event is generated each time the macro is encountered. There is also a similar procedure for Java. An advantage of tracing with LTTng in user space is that it does not require any system call, which allows it to be very efficient [21]. Events are written by apps in buffers in shared memory. A single daemon is responsible for collecting events generated by all the instrumented applications of the system. It is also worth mentioning that the timestamps of events in user space are consistent with those of system level events.

It is also possible to automate the instrumentation of an application. For instance, we can compile source files with the **-finstrument-functions** option of the GCC and LLVM compilers and link the application with the **liblttng-ust-cyg-profile** library to get an event at the input and output of each function. It has been shown that the GCC compiler slows down

by a factor of 3.22 when instrumented [17]. Instrumenting every function call is indeed very costly and typically a subset of functions will be instrumented for time sensitive applications.

The "context" feature of LTTng allows to dynamically add fields to events once setting up a trace session. These fields can provide information about the process that generated an event (pid, priority, name) or on the value of various performance counters at the time the event occurred (number of executed instructions, number of cache errors). Recent developments also allow to add an application call stack in both user and kernel spaces. This is very useful for understanding the sequence of function calls once an event, either in kernel or user space, is generated.

However, one limitation of LTTng can be the lack of links between the kernel events and the application inner behavior. Hence, the user has to either add manual instrumentation, which requires human intervention and may not be exhaustive, or use the **-finstrument-functions** option, which requires access to the source code and considerably slows down the application, to make this link [8].

### Event Tracing for Windows (ETW)

ETW is a tracer built into the Windows operating system. It allows tracing kernel events provided by Microsoft. Developers can generate events from their applications as well. Some popular applications like Internet Explorer and Node.js [22] already contain ETW instrumentation. A profiler is integrated with ETW [23]. When it is activated, events containing the call stack of running applications are periodically generated. These events allow to determine, for example, which functions are running in a thread when another thread is waiting for a resource. There is not much details about the implementation of ETW. However, we know that events are written in circular buffers in memory [24]. Buffers can be automatically copied to a file once they are full, or the copy can be made on request (similar to the LTTng flight recorder mode). One limitation of ETW is that it is not possible to create new kernel events according to our needs because of its proprietary license [8].

### DTrace

DTrace is a tracer for Solaris, introduced in 2004, for troubleshooting kernel and application issues on production systems in real time [25]. It allows, like LTTng, to trace the kernel and user spaces and aims for minimal impact on performance. The ability of associating D-language scripts with instrumentation points distinguishes DTrace from the other tracers. This feature allows to dynamically specify actions to perform once an event occurs. These

actions can be used to record information in the trace (for example, the execution stack of a process), filter events to be traced (thus reducing the size of the trace) or even online aggregations. Scripts can use variables to hold data from one event to another. A limitation of DTrace is that it is slower than LTTng, 6.42 times for the kernel [26] and 10 times for user space [27] [8].

### 2.1.2 Profilers

The profilers provide statistics describing the execution of a system for a given time interval [28]. For example, a processor profiler indicates processor time consumed, by instruction, function or call stack, in the analyzed time interval. As another example, a memory profiler indicates the amount of memory allocated per thread, call stack or type of object.

Unlike tracers, profilers do not account for the scheduling of events. Hence, with a tracer, we can determine the order in which different threads have acquired a lock. With a profiler, the only information extracted is the total waiting time for locks by each thread.

An advantage of profilers over tracers is that they generally produce less data. Indeed, they calculate global statistics rather than keeping the details of each event. This allows them to have less impact on the analyzed system [8].

#### Perf

Perf was developed to simplify the access to processor performance counters [29]. Performance counters are special registers which are incremented when hardware events occur (cache faults, processor cycles, ...). When perf is executed in its simplest mode, it produces a report indicating the number of times that various counters have been incremented during a given time range (see Figure 2.1).

Perf can also indicate which instructions of a program caused the events associated with performance counters. As most of these events happen at a very high frequency, it would be unreasonable to record the running instruction at each occurrence, without having a significant impact on performance. For this reason, perf issues an interrupt for a fraction of the occurrences. The instruction causing the event is captured during this trap. Statistically, each statement should represent a similar proportion in the samples captured and in the entire given time. Perf is also able to add dynamic kernel instrumentation with kprobes.

Perf can be used to periodically capture call stacks for user and kernel spaces. This mode is an effective way to identify the functions which consume the most CPU time. Perf saves 2 memory pages in the trace in order to extract the return addresses, later during the analysis

```
majid@majid-pc:~/ perf stat tar -xvf test.tar.xz

        64.515777      task-clock (msec)         #    1.025 CPUs utilized
              143      context-switches          #    0.002 M/sec
                0      cpu-migrations            #    0.000 K/sec
              587      page-faults               #    0.009 M/sec
      257,328,426      cycles                    #    3.989 Ghz
      267,689,431      instructions              #    1.04 insn per cycle
       33,581,568      branches                  #    520,517 M/sec
        5,602,166      branch-misses             #    16.68 of all branches

      0,062965435 seconds time elapsed
```

Figure 2.1 Execution of command perf stat

phase [30]. This quickly increases the size of the trace.

Perf is not designed to monitor systems in production [31]. Oprofile offers features similar to perf. However, it affects more the performance of the system [32] [8].

#### Java VisualVM

Java VisualVM can profile the processor time and memory consumption of a Java application [33]. This profiler instruments the program so that it generates an event at each function entry and exit. This has a higher cost than interrupting the program at regular intervals. However, this solution has the advantage of giving the exact number of calls to each function [8].

### 2.1.3 Debuggers

Debuggers enable source code analysis during the run time and help in understand program execution. This is done by dynamically adding breakpoints at arbitrary locations in the source code, like when the program crashes. One of the well-known debuggers is GDB. It starts as a separate server process and executes the user commands. GDB also supports several languages, such as C and C++. Debuggers stop the execution in the inserted breakpoints and analyze concomitantly the source code of the application. Hence, they increase the overhead of an application considerably due to the interruption in the program execution. Thus, this kind of analysis requires the addition of breakpoints in the source code. Also, as they present a snapshot of the current execution of the application, they often are not able to find complex

component interactions issues [34].

## 2.2 Chromium Tracer

In order to find critical hot spots in a complex shared memory application like Chromium, we need to have an informative graphical view of profiling data. Chromium comes with an inline profiling subsystem which is accessible by browsing to chrome://tracing. The great advantage of using this tool is that it allows to capture detailed profiling data about what Chromium is doing. We can properly adjust the Javascript execution, or optimize the asset loading. Chromium tracer offers an intimate view of the browser performance through recording all activities of Chromium across each thread, tab, and process [35] as shown in Figure 2.2 [1].

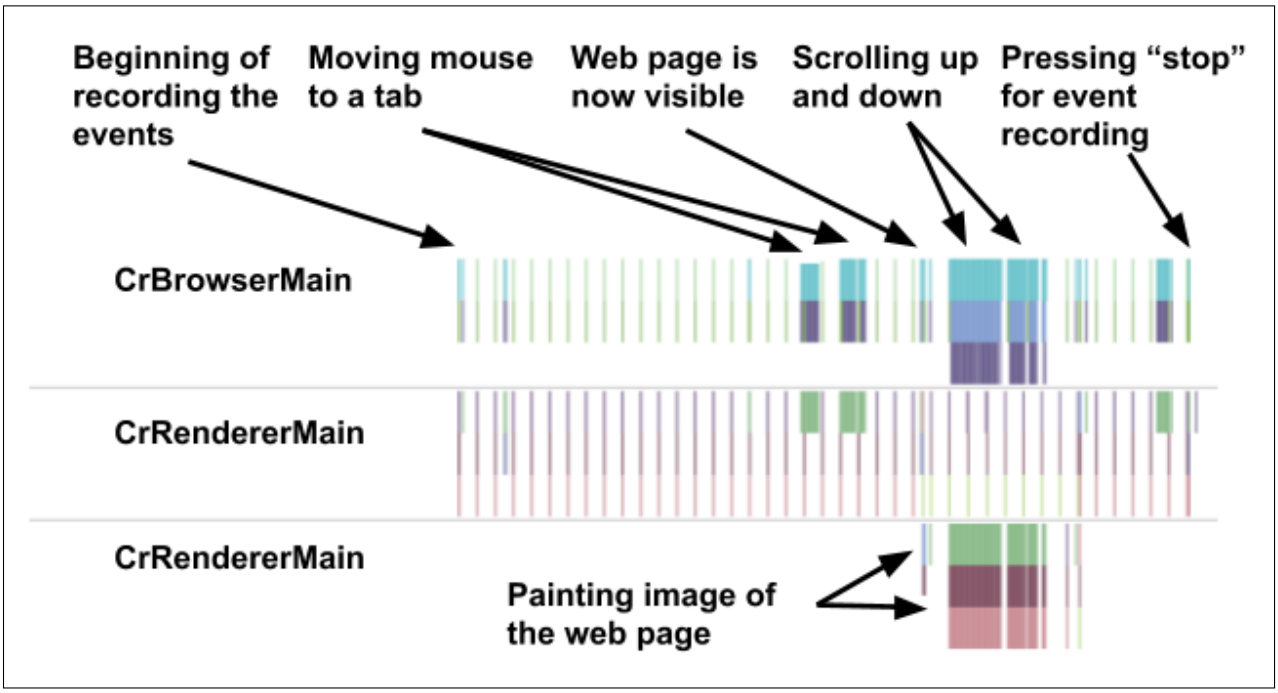


Figure 2.2 Graphical view of Chromium tracer [1]

The tracing tool was built with two main features in mind. Firstly, to provide a view of all activities of Chromium under the hood, and secondly, to profile Javascript code by inserting custom begin and end segments into the stream. The Javascript APIs of Chromium, version 19 and later, provides the console.time("eventname") and console.timeEnd("eventname") events, allowing to directly profile any custom Javascript code alongside the rest of what Chromium is executing [35].

The tracing data format is a non-optimized JSON file which contains a sequence of event blocks. Each block includes name/value pairs as follows [35, 36]:

- 'cat' – the category for this event. This is a comma separated list of categories for the event. The categories can be used to hide events in the trace viewer UI
- 'name' – the name of this event, as displayed in the trace viewer
- 'pid' – the processor ID that generated this event
- 'tid' – the thread ID that generated this event
- 'ts' – the processor time stamp at the time this event was generated. The time stamps are provided with microsecond granularity
- 'tts' – The thread clock time stamp of the event. This item is optional and is provided at microsecond granularity
- 'ph' – the phase or type of this event. This is a single character which changes depending on the type of event being output. The valid values are listed in Table 2.1. The full description of each phase is discussed below
- 'args' – any programmatic metadata attached to this event. Some of the event types have required argument fields. Otherwise, any information can be put there. The arguments are displayed in the trace viewer once a user views an event in the analysis section
- 'cname' – A fixed color name to associate with the event. This item is optional. If provided, cname must be one of the names which are listed in trace-viewer's base color scheme's reserved color names list

Table 2.1 lists all the event types and their associated phases as follows [36]:

Table 2.1 The list of event types with their associated phases

| Event type | Event phases |
|---|---|
| Duration events | B (begin), E (end) |
| Complete Events | X |
| Instant Events | i, I (deprecated) |
| Counter Events | C |
| Async Events | b (nestable start), n (nestable instant), e (nestable end), Deprecated: S (start), T (step into), p (step past), F (end) |
| Flow Events | s (start), t (step), f (end) |
| Sample Events | P |
| Object Events | N (created), O (snapshot), D (destroyed) |
| Metadata Events | M |
| Memory Dump Events | V (global), v (process) |
| Mark Events | R |
| Clock Sync Events | c |
| Context Events | (, ) |

A JSON file with a single begin/end sample is listed below. It is worth mentioning that an event includes the descriptions of its category as well as its name. Some events occurring in pairs are marked by complementary phases (B/E or S/F) [36].

```
[
"cat": "MY_SUBSYSTEM", //catagory
"pid": 4260, //process ID
"tid": 4776, //thread ID
"ts": 2168627922668, //time-stamp of this event
"ph": "B", // Begin sample
"name": "doSomethingCostly", //name of this event
"args":   //arguments associated with this event.
,
]
```

## 2.3 Analysis of Execution Trace

Execution traces encompass a large volume of raw information. In order to use it as part of a performance, reliability or security analysis, we need to use some automated tools. These tools are able to provide a view of the highest level of information in the trace. Using this representation, the user can effectively navigate through the whole trace and obtain an understanding of the overall behavior of the system. If necessary, he can also access the raw events, corresponding to a region of interest, to analyze the data in details [8].

As part of our work, the high level view of information is used to detect any kind of performance bug or problem. In fact, several executions can be put together to detect issues through comparisons between normal and abnormal executions. We also use the critical path analysis to identify the root cause of problems which need special attention. Since different threads compete to take the CPU in multi-thread applications, the critical path analysis can provides a useful insight into the performance bottlenecks of the execution.

### 2.3.1 Building a state history

The history of states in Trace Compass views are generated from the low-level events in execution traces [10]. Tracers usually provide special events to build the initial state of a system, for example lttng_statedump_* for LTTng or DCStart for ETW. Once the initial state of system is known (list and status of existing processes and threads at trace start time), Trace Compass uses state machines to compute state changes and generate a state history. State machines receive events from one or several traces of execution sequentially. Once an event meets the conditions of a transition from the current state, a state change is generated and written into the state history. Not only is it possible to follow the state of low-level resources, but it is also feasible to design state machines in order to follow the state of the whole system [37] [8]. The set of kernel and user level events, alongside their related fields required to build the state history, are listed in Tables 2.2 [7] and 2.3, respectively.

## 2.4 Visualization of the System State

The Trace Compass analysis tool [38] provides several "Gantt diagram" type views (Figure 2.3). In these views, the vertical axis represents system resources (threads, processors, interrupt vectors), while the horizontal axis describes the state of these resources as a function of time. It is possible to change the position and zoom level of the horizontal axis interactively. TraceCompass also offers views of "XY graph" type to display the use of a resource (CPU, memory) as a function of time. All the views are synchronized along the time axis.

Table 2.2 Required kernel tracepoints to build state history

| Events | Fields |
|---|---|
| sched_switch | prev_tid, prev_state, next_tid |
| sched_wakeup | tid, target_cpu |
| irq_handler_entry | irq, name |
| irq_handler_exit | irq, ret |
| hrtimer_expire_entry | hrtimer, now, function |
| hrtimer_expire_exit | hrtimer |
| softirq_entry | vec |
| softirq_exit | vec |

Table 2.3 User space tracepoints to build userl level states

| Events | Fields |
|---|---|
| DidStartNavigation | This event is fired if the user clicks or enters a URL |
| DidStopNavigation | This event is fired if the navigation is completely done |
| ViewHostMsg_ClosePage | This event generates if the user clicks on close icon of a tab |
| OnMouseEvent | A set of events can be generated under this category for various actions such as Pressing mouse, Releasing mouse and Dragging mouse |
| OnKeyEvent | A set of events can be generated under this category for various actions such as Pressing or Releasing a key |

Trace Compass supports several features, including the critical path computation and a call graph analysis. We deployed these features to highlight several aspects of traces by applying algorithms for comparison and analysis. However, this tool still requires a specific strategy as well as a deep knowledge of the system to fix an issue [8].

### 2.5 Storing state history

Because of its fine granularity, the state histories shown in Trace Compass views can easily require several gigabytes of data for a few minutes of trace. Obviously, such an amount of information cannot be stored in memory. Therefore, Trace Compass uses a data structure named State History Tree (SHT) [39] which is optimized for hard disk storage of values associated with time intervals. Basically, hard discs are more optimized for sequential operations rather than random accesses. For this reason, the state history tree does not resort to rebalancing operations, such as those found in AVL trees. After writing a node in the state

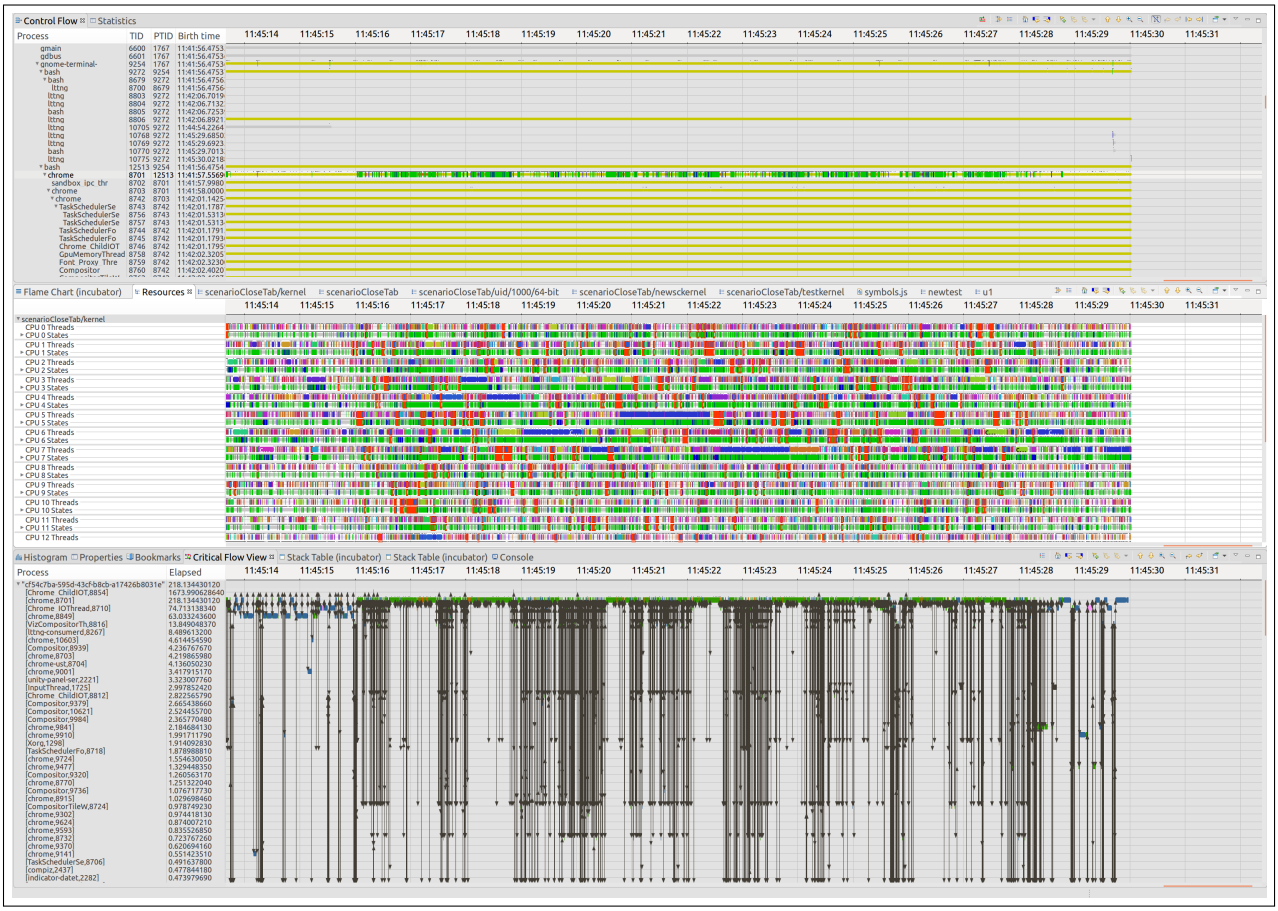


Figure 2.3 Trace Compass analysis tool

history tree, it cannot be modified [8]. There are two constraints on the insertion of intervals:

- The intervals must be inserted in increasing order of upper bounds
- The intervals should be short compared to the extent of all intervals of the tree. They must also be uniformly distributed over time

For many analysis, the size of the state history tree is close to the source trace. Duplicating data can be problematic as well. In this case, we can resort to a partial history which is in fact a state history tree that contains only the intervals crossing saving points. Missing intervals can be regenerated upon request. This strategy increases slightly the access time but reduces very significantly the storage requirements.

## 2.6 Tracking the execution of a task

Executing a task in a complex multi-thread application often requires the collaboration between several processes or threads on the machine running that application. In many cases, we need to investigate tracing information at different levels, in the complete hardware and software stack, in order to insure that all the needed information has been extracted. Chromium is a perfect example of such applications, with a tasking model handled by a large thread pool, and execution scheduling happening both at kernel and user levels. The source code of Chromium is written with the low-level functions in C and the layer above in JavaScript. Chromium is more complex than most open source applications. If we want to understand why the execution of a task requires more time than desired, it must be possible to insert different trace points in several parts of the source code to perform this multi-level analysis. It then becomes possible to measure the latency and resource consumption of each segment of a given task execution. In the context of our work, we want to investigate the performance issues during different user actions. In fact, we aim to find the root cause of performance problems, for different user actions which are taking more time than expected, in both kernel and user levels. This can be achieved by calculating the critical path of that specific execution and going deep in the detailed events to find execution anomalies. To do this, we will need to mark the beginning and the end of each user action, to figure out which tasks are associated to that action. This will be possible by the use of the techniques described in section 2.3.

## 2.7 Critical Path in Trace Compass

Critical path analysis is another methodology which can be used to analyze huge traces of a multi-thread application. In fact, a specific task alongside its dependencies is analyzed

through multiple related thread executions. The critical path algorithm is a methodology to analyze the tasks that rely on multiple threads. F. Gilardeau [7] proposed an algorithm which is able to find all the threads that contribute to the total time of an execution. This algorithm retrieves efficiently all execution segments contributing to the latency of a task. Using some event properties (sched_wakeup) of the Linux kernel, it is feasible to detect the multiple threads related to the specific task. However, this tool does not link this information to user space functions, which makes it difficult to use for analyzing application code [40].

Trace Compass proposes the critical path analysis to accurately identify where the time was spent during the execution of different tasks [7]. The critical path analysis enables to see in which process the run time was spent, but it does not identify where in the application the delays are associated with waiting for the network, disk or locks. This analysis is based only on Linux operating system events and does not require any additional instrumentation in the analyzed application [8].

## 2.8 Lock Contention Analysis

Since contention on shared resources can severely affect the performance of a multi-thread program, identifying its root cause is noteworthy. Most multi-thread applications use locking to control the access to shared data in parallel processing. Since lock contention can be one of the most challenging issues in multi-thread applications, having a platform to detect and analyze the root cause of lock contention is a vital task.

Pusukuri et al. [41] showed that shuffling can improve the performance of multi-thread applications with high-lock contention. Shuffling aims to reduce the variance in the lock arrival times of the threads which are scheduled on the same socket. They defined the lock arrival times as the time at which threads arrive at the critical section, just before successfully acquiring the lock. By scheduling threads whose arrival times are clustered in a small time interval, they can all get the lock without losing the lock to another thread on the same socket. Thus, shuffling ensures that once a thread releases the lock, most likely another thread on the same socket will successfully acquire the lock, and LLC misses will be avoided. As a result, it reduces the lock acquisition time and speeds up the execution of shared data accesses in critical sections which results in reducing the execution time of the application. Since it does not change the application source code or the kernel, it can be an efficient solution. However, it shows a slight performance improvement in systems running several multi-thread applications with high lock-contention. They implemented shuffling on a 64-core Supermicro server running Oracle Solaris 11™ and evaluated it using a wide variety of 20 multi-thread programs which had high lock contention. Their experiments showed that

shuffling leads to an average reduction of 15% in execution time [41].

Hence, Bin Nisar et al. [42] proposed a novel-scheduling technique as an improvement to earlier work called shuffling algorithm [41]. They proposed per-application thread grouping to reduce the lock contention due to imbalanced thread mapping in parallel applications. The authors addressed performance degradation of multi-thread applications due to the high penalty cost of cache misses. They group a set of threads sharing the same lock-object onto a processor where they can utilise the LLC. Hence, the number of cache misses is reduced, improving the program performance. They also proposed an algorithm considering both user and kernel mode lock times instead of only user mode, which the earlier shuffling algorithm ignores [42].

Tallent et al. [43] proposed and evaluated three approaches for obtaining information about performance losses due to lock contention. Their approaches moved from blaming lock contention on victims, then to suspects, and finally to perpetrators [19]. In the first approach, they used a simple strategy based on call stack profiling to attribute idle time and show that it cannot yield insight into lock contention. In the second approach, they considered a strategy that builds on a procedure used to analyze idleness in work-stealing computations. They showed that this strategy does not yield insight into lock contention either. Finally, they proposed a novel technique for the measurement and analysis of lock contention. It uses data associated with locks to blame lock holders for the idleness of spinning threads [43].

In another work, Pusukuri et al. [44] presented ADAPT, a scheduling framework which continuously monitors the resource usage of multi-thread programs. They developed ADAPT as a platform for effective scheduling of multi-thread programs on multi-cores machines [44]. ADAPT predicts the effects of interference between programs on their performance, using supervised learning techniques. It co-schedules programs that interfere the least with the performance of each other. Using simple performance monitoring tools, available on a modern operating system, it allocates cores adaptively and assigns appropriate memory allocation and scheduling policies, on the basis of the resource usage characteristics of multi-thread programs. They implemented ADAPT on a 64-core machine running Solaris 11 and evaluated it using 26 programs. The experiments showed that ADAPT leads to 44% improvement in the turnaround time and throughput of TATP, and 23.7% and 18.4% for JBB, compared to the default Solaris 11 scheduler. The overhead of ADAPT is insignificant and it requires no changes in the application source code or the OS kernel. Since ADAPT learns dynamically the appropriate contention factors and their impact on the program performance, on any given architecture, it may evolve automatically with changes in processor architecture and computing environment [44].

Lozi et al. [45] proposed a new locking algorithm, Remote Core Locking (RCL), which aims to improve the performance of critical sections in legacy applications on multi-core architectures. RCL replaces lock acquisitions by optimized remote procedure calls to a dedicated core [45]. RCL accommodates blocking within critical sections and nested critical sections as well. The design of RCL addresses both access contention and locality. RCL prevents the performance collapse observed with other locking algorithms, once several threads are trying to acquire a lock concurrently. It also eliminates the need for transferring lock-protected shared data to the core acquiring the lock, because such data can usually remain in the server processor core cache. They developed a profiler identifying the locks which are the bottlenecks in multi-thread applications. Hence, it can benefit from RCL, and tools that transforms POSIX locks into RCL locks. They evaluated their approach on 18 applications using RCL locks and got performance improvements of up to 2.6 times, with respect to POSIX locks on Memcached [45].

## 2.9 Critical Path Analysis

The Critical Path Analysis (CPA) or the Critical Path Method (CPM) is a technique which describes the longest execution sequence without wait states in a parallel program by identifying the activities that determine the overall program run time [46]. A multi-thread application is a set of threads running on multiple cores, and implemented in various programming languages [47]. The task of performance debugging can be more challenging due to the hidden nature of the processing and the incompatibilities between run time environments. Combining knowledge of the critical path with traditional tracing tools, we can identify the root cause of multi-thread applications. The advantage of the critical path analysis, compared to metrics that simply add values for individual threads, is that it provides a global view of the performance of a parallel computation and the interactions between the different threads.

The developers of multi-thread programs need to have knowledge of those parts of the program that benefit from optimisations. Broberg et al. [48] define the critical path as all the executed code segments that, reduced by an $\epsilon$, will also shorten the total execution time of the program. As a result, all sequential parts of the program are considered as part of the critical path, since shortening any part of a sequential program leads to a reduction in the total execution time of the program [48].

In the case of multi-thread programs running on multiprocessors, not all of code segments are included in the critical path. Unfortunately, most existing performance analysis tools do not consider the critical path of execution. They only show the most time-consuming functions and suggest them for optimization. Therefore, the need for tools which compute the critical

path is undeniable, in order to correctly optimize multi-thread programs on multi-processors.

Currently, many multi-thread applications are limited not only by processor core performance but also by interactions among the cores and threads, the memory subsystem, I/O devices, and the complex software layers that link them together. Programmers and developers of such applications encounter challenges to identify performance bottlenecks during the execution of the program. The reason of these challenges is that, in any concurrent system, the overhead in one component may be invisible due to overlapping with other operations. These overlaps occur in the user/kernel and software/hardware levels, making traditional debugging tools inadequate. Common software profiling techniques cannot help for hardware bottlenecks, in which software and hardware operations are overlapped [49].

Miller et al. [50] presented an algorithm for calculating the critical path of a distributed application based on message passing. The Program Activity Graph (PAG) is defined as a directed cyclic graph, in which the nodes are the events marking the beginning and end of an activity, and their arcs represent the processor time. The longest path is the critical path whose length is the sum of the weights of its segments. The study compares a centralized algorithm to a distributed algorithm for calculating offline the critical path. The focus is therefore on the acceleration of the offline calculation of the critical path by a parallel algorithm [50].

Hollingsworth et al. [51] proposed a method for the online calculation of the critical path of activity graphs. The presented technique avoids the construction of the complete graph and storing the events. The technique consists of adding the current value of the critical path to a sent message. When receiving such a message, this value is copied to a local variable of the process. The processing time of the process is added and the updated value is transmitted with the answer. When receiving a message, the highest value between the local one and the one received is kept [51].

The critical path computation of Miller and Hollingsworth is effective for applications whose performance is limited by processor time. However, this type of analysis is insufficient to determine the critical path of a system that would be bounded by the entry-exit. If the waiting time and communication time are included in the weight of the arcs, then all graph paths have the same weight, which makes the existing critical path computation algorithms inoperative. Also, dependencies between local processes using other communication mechanisms, such as tubes and signals, are not supported, which restricts the number of programs that can be effectively observed [7]. LTTV, a kernel trace viewer developed in the DORSAL laboratory, contains several modules, including an histogram of the number of events by time, a view of system resources and a module for displaying the status of the processes. The analyzer

restores the state of the system from the trace. Regarding critical path analysis, the LTTV dependency analysis module [52] uses kernel tracing to determine the causes for waiting time in a program. The analysis focuses on the blockages occurring during system calls. The emitter of the wake up event indicates the cause of the blockage. The algorithm assumes a tree waiting structure, which is not adequate for a distributed application, due to the messages passing that must generally be represented as a graph [7].

## 2.10 Detection of bottlenecks

Saidi et al. [49] presented a method to extract the critical path of execution to identify bottlenecks, whether software or hardware. The method consists of modeling as an automaton the components to be analyzed, whose events are the transitions. The links between the automatas are defined by the user and form the control flow. The states of the automata correspond to the arcs of the dependency graph. These states are annotated with the duration of the associated intervals. The measurement system has been implemented with a hardware simulator. The low-level hardware analysis is done by instrumenting the simulator with callback functions. Software analysis uses the capacity of the simulator to instrument the entry and exit of functions. The automatas are annotated with the function labels. The critical path is calculated without rebuilding the complete graph, with the method described by Hollingsworth et al. The visualization method of the proposed graph consists of grouping the isomorphic graphs and annotating the arcs by the number of executions, the total time elapsed and the proportion of time on the critical path. The analysis specifically targets the detection of bottlenecks in the source code and hardware, while we are interested in the general case of the execution graph at system level. In addition, the analysis requires knowledge of the application source code, kernel and hardware model, which is not possible for black box analysis [7].

The analysis of Layered Queuing Networks (LQN) has shown its efficiency in determining the capacity and bottlenecks of distributed systems [53]. In particular, this analysis has been successfully applied to determine the bottlenecks of a flexible real-time IP telephony system [54]. This is a generalized model of the queuing theory for distributed systems. The average waiting time in queue and the average length of the tail are examples of performance measures that this analysis can produce. For some simple models, there is an analytical solution. In other cases, the results can be obtained by discrete simulation of the model. The difficulty of using LQN relates to the generation of the performance model in the system. Automation of this approach is proposed to facilitate the construction of the model [55]. Analysis of the statistical behavior of network packets was used to determine the presence of

a bottleneck, without knowing the maximum capacity of the system in advance [56]. Based on the distribution of communication latency, compiled for a time window, the dissymmetry factor of the distribution is calculated. A positive dissymmetry indicates that the distribution is compressed to the limit of the system, and thus reaching a bottleneck [7].

## 2.11 Conclusion of the Literature Review

Tracers, profilers and debuggers can be used for the dynamic analysis of an application with their own specific advantages and disadvantages. Typical profilers are not able to reason about performance problems in multi-thread applications, as they do not dig into the collaborations between threads, and threads scheduling. They also add extra overhead and may not be helpful to find performance issues which occur between groups of several executions of an application. Debuggers that work by stopping the world can not also be of much help, because the delay imposed by stopping the program can hide many race conditions and time-related problems. Tracing, instead of stopping the program, works by collecting information while the program is running. Indeed, by using a powerful tracing tool, the behavior of several layers of the system, including application, CPU, memory can be observed. The collected run time information can then be used to detect many performance problems and identify their root causes. In the next chapter, we present the methodology used for our proposed solution.

# CHAPTER 3 METHODOLOGY

The design of a performance problem-solving tool should be supported by a rigorous methodology to ensure that it is adapted to the industry needs. The tool must address real-life problems, encountered by high performance applications developers, while respecting the constraints of this field. This chapter details the procedure followed to conduct our study.

## 3.1 Problem definition

We visited several companies developing high performance software and we scanned the bug directories of several open source software to understand the context in which our solution could be applied. This approach allowed us to identify causes of performance issues that our tool should be able to tackle. These have already been introduced in section 1.2. We present here features of the field that will guide our design choices.

### 3.1.1 The systems analyzed are unknown to developers

High performance multi-thread applications often have a huge code base, which is not necessarily understood as a whole by each of its developers. It would not be appropriate to design a performance analysis tool requiring comprehensive manual addition of instrumentation in the functions to be monitored. Developers cannot realistically in most cases scan the whole source code in search of strategic places to insert trace points. Developers prefer an approach in which a smart tool monitors the entire system and highlights the faulty code itself when a problem is detected.

Several existing tools rely only on events from the operating system core to analyze the performance of applications. They require no instrumentation in user space, which leads to their easy application. However, this makes it more difficult to identify the source code that caused the detected problems.

### 3.1.2 Complex multi-thread applications involve interactions between multiple components, processes and threads

Developers have effective tools to evaluate the performance of a single function. As they may face problems involving interactions between multiple threads, with the operating system or with the hardware, they deploy more advanced tools. Therefore, it is crucial that we be

able to demonstrate these multi-level interactions in our analysis tool. To our knowledge, there is no tool that can trace all sources of latencies in a large and complex application, and associate them with the relevant source code.

### 3.1.3 Reproducing the performance problems is difficult

The difficult problems, that remain unsolved for a long time, are typically those that are rare, for instance that only reproduce under very specific conditions. Tracers can be used at any time, in production systems, to capture such problems, thanks to their low overhead. Therefore, it is possible to extract the sequence of events leading to the problematic behavior, and analyze it in more details in a trace visualizer.

However, developers cannot manually spend too much time to reproduce each performance issue. Moreover, they do not know when the problem happens, in order to enable tracing when the problem occurs. Therefore, we need to have a flexible platform to automatically trace the system without any limitation, until the problem appears. Then, we can analyze the gathered data to see what has gone wrong.

### 3.1.4 Developers have little time for performance analysis

Companies value greatly the development of new features. Application performance is an important requirement but developers are often overloaded with their other tasks and have very little time to devote to it. Any tool that reduces the human time spent diagnosing performance issues is therefore highly beneficial.

Another goal of this study is to develop a performance analysis tool requiring a minimum of configuration or changes in Chromium. Developers like to have a tool being able to reveal why their application does not have the expected performance, without determining what types of events should be recorded.

## 3.2 Solution design

In this thesis, we propose a methodology to identify the root cause of performance degradation in a complex multi-thread application like Chromium using multi-level data collection and data analysis. We propose the following architecture for our work, as shown in Figure 3.1.

The proposed architecture consists of three distinct sections. These sections include Data Collection, Critical Path Analysis and Performance Evaluation. The section on Data Collection explains the challenges faced to extract multi-level data from a complex application like

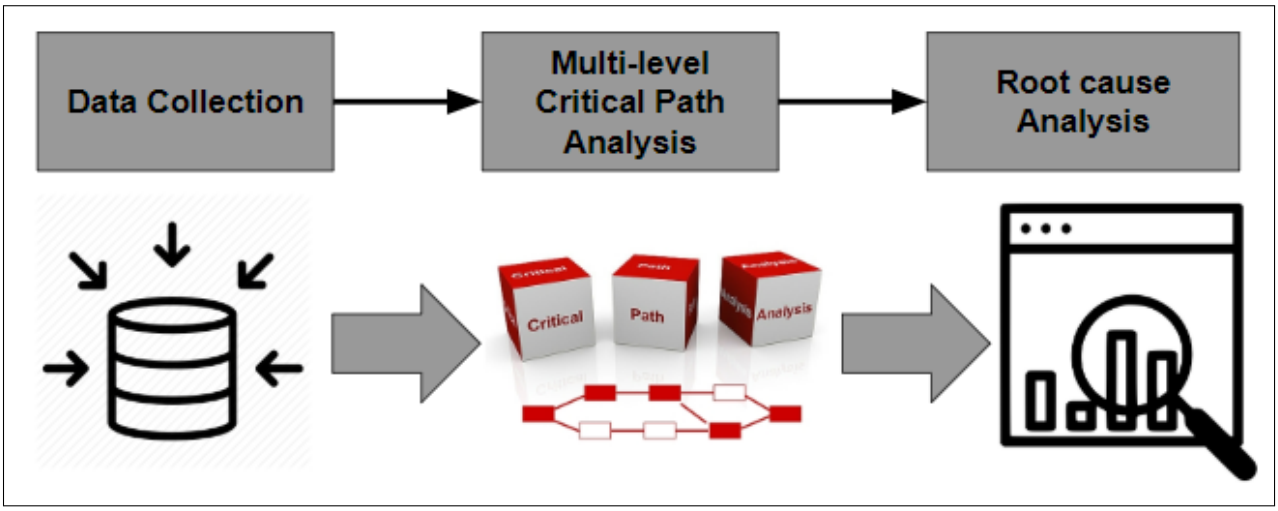


Figure 3.1 Proposed architecture in three main sections

Chromium. Furthermore, we propose a solution to collect unified, synchronized and multi-level data from Chromium. Then, we add the user level trace data to the system level trace data to build a multi-level critical path analysis platform. Finally, we analyze the gathered data to understand what went wrong for any execution, wether the problem is at kernel or user level.

The Chromium source code encompasses different trace points types, to report the operations performed by the multiple threads of the browser at run time. However, user level tracing is not sufficient, particularly in Chromium with its multiple threads and thread pools, when it comes to detect and analyze the root cause at low level.

The Distributed Open Reliable Systems Analysis Laboratory (DORSAL) at Polytechnique has developed numerous tools to efficiently capture and analyze traces for performance diagnosis. These tools are available as Open Source. We will rely on these implementations to accelerate the prototyping of the different parts of our contribution. First, we will use the LTTng tracer to capture execution traces. This tracer can capture, with low overhead, events from the operating system and also from user space, which suits our objective of detecting problems occurring at multiple levels, and associating them with the faulty application code. However, to obtain user space application events, instrumentation is required. Hence, we have to find an effective solution to instrument Chromium such that we are able to gather user space data as needed.

As mentioned above, we used LTTng to enable kernel tracing but kernel and user level events are not synchronized. LTTng produces events with nanosecond resolution but the Chromium

tracer produces events in milliseconds. Hence, the user and kernel level events could not be synchronized. Therefore, we came up with a better solution which can provide unlimited Chromium tracing, without time inconsistencies. We developed an API to redirect all user level events from the Chromium trace points to LTTng. This method is similar to the technique used on other operating systems. Indeed, on Windows these events are recorded using Event Tracing for Windows (ETW). User space applications such as Chromium can also emit events to ETW. We applied this idea to Chromium in order to export all Chromium user level events to LTTng, so that we can gather multi-level tracing data with nanosecond precision for both kernel and user level events.

We will reuse the critical path computation algorithm proposed by [7], and integrated into Trace Compass, to identify all the threads that affect the completion time of a task. This algorithm has been shown to produce accurate results in almost all cases. Finally, we will analyze the obtained data to detect the root cause of janks in Chromium.

The code for all the tools developed will be released under a free license, so that they may be reused by other researchers in the future. In chapter 4, we present the article "Performance evaluation of complex multi-thread applications through multi-level critical path analysis" which describes all the details of our solution. The article also discusses the effectiveness of our solution, to diagnose real performance problems, and its overhead.

# CHAPTER 4 ARTICLE 1: PERFORMANCE EVALUATION OF COMPLEX MULTI-THREAD APPLICATIONS THROUGH MULTI-LEVEL CRITICAL PATH ANALYSIS

**Authors**

Majid Rezazadeh, Naser Ezzati-Jivan, Seyed Vahid Azhari, Michel R. Dagenais

Polytechnique Montreal
Montreal, Quebec H3T 1J4
{majid.rezazadeh, n.ezzati, seyed-vahid.azhari, michel.dagenais}@polymtl.ca

**Index Terms -** performance analysis, multi-level analysis, complex multi-thread applications, Chromium analysis

**Submitted to** Software: Practice and Experience

## 4.1 abstract

The sporadic latency problems of multi-thread applications remain a major concern for developers. A performance issue may be caused by internal factors like an error in the code or an inadequate database design, or external factors such as threads imbalance, resource contention or system overload. In complex applications such as the Chromium browser, which is our focus in this paper, being able to collect precise unified information from several layers of the application/system is the first challenge. There exists a built-in tracer in Chromium, however it provides a limited view of the application execution. The correlated analysis of the data collected from the multiple execution levels of the application is another challenge. In this paper, after explaining our tracing method, for collecting data from the application and the operating system kernel simultaneously, we translate the tracing data, collected from several layers of the system, into a unified model. We then use this model to perform kernel-level control flow analysis for user-level browser actions, to detect latency issues and identify their main causes. We show the effectiveness of the proposed method by detecting three Chromium performance bugs and finding their root-causes. Although our use-cases are based on the Chromium browser, the proposed method and lessons learned are directly applicable to other complex multi-thread applications.

## 4.2 Introduction

Performance problems are typically difficult to analyse, especially when they occur rarely in the system and are not easily reproducible. This is even worse in the case of multi-thread applications, where the task executions are the result of the collaboration of several threads, and understanding the performance problems requires deep knowledge about CPU scheduling and lock contention.

Typical profilers are often not able to reason about performance problems in multi-thread applications, as they do not dig into the collaboration between threads and thread scheduling. Debuggers that work by stopping the world can not help much because the delay imposed by stopping the program can hide many race conditions and time-related problems. Tracing, instead of stopping the program, works by collecting information while the program is running. Indeed, by using a powerful tracing tool, several layers of the system, including application, CPU and memory can be observed and the collected run time information can be used to detect many performance problems and identify their root causes.

In this paper, our focus is on investigating latency performance problems in complex multi-thread and multi-process applications like Google Chromium. Although tracing itself is considered as an easy-to-achieve method in various research work, it is a challenge in Google Chromium where security mechanisms, like sandboxing, which are built-in to the system, prevent outside programs from being able to read internal data of the browser/file system, making tracing very difficult.

The Chromium browser is shipped with an elegant built-in tracing module, which enables application-level tracing of Chromium and provides visibility on function level executions. However, the time precision provided is in milliseconds which is not sufficient for many latency problems, where we indeed need to reason in nanoseconds. Moreover, it does not give insights into CPU scheduling or kernel-based lock contention, which are necessary for understanding many threaded applications [57].

In this paper, we propose a new method for instrumenting the Chromium browser, despite its defense layers and the existence of several security mechanisms. Our method supports multi-level trace collection for the Chromium browser. This includes tracing the application space (i.e., function calls, high-level actions like tab opening/closing, mouse click, scrolling, etc.) and kernel space (i.e., thread scheduling, memory accesses, lock contentions, etc.). The collected information is fed into a multi-level analysis module, to extract the critical flow of user-level tasks, used for detecting latency problems and reasoning about their root causes.

Our contributions, therefore, are twofold: **First**, we propose a method to trace the user

space and kernel space execution of the Chromium browser in a unified way, providing nano precision time-based data. **Second**, we propose our method for critical flow extraction of browser user actions, which is used directly in latency problem detection and analysis.

The remainder of the paper is organized as follows: After discussing the related work in Section 2, we present the architecture of our trace collection method. We elaborate on the proposed critical flow extraction method, and apply it to the latency problem detection and root cause analysis in Section 4. Three use-cases, followed by the threats to validity discussion, are presented to show the usability, benefits and limitations of the proposed approach. The conclusion and future work are then finally drafted.

### 4.3 Related Work

Most previous work propose solutions for one level of data, either in kernel space or user space, with a limited view of that level, preventing the obtention of an overall detailed view of whole system. However, our solution provides a unified multi-level view with all details in both kernel and user level, enabling us to discover the root cause of any performance issue stemming from either inside the application code or operating system kernel.

To analyse the performance of a complex multi-thread application like Chromium, we can use three kinds of tools such as profilers, debuggers or tracers. Profilers allows to verify where the program spent its time as well as the hierarchy of function calls during the execution. This information can be used to detect the contribution of each function to the total execution time. However, it does not necessarily reveal infrequent performance problems, nor the interactions between the different threads [58].

Debuggers enable source code analysis during the execution, to understand the program behavior. This is achieved by dynamically adding breakpoints at arbitrary locations in the code, such as where the program crashes. Debuggers, however, increases remarkably the latency of an application, because it needs to stop the execution of the application when a certain condition occurs. Thus, many timing related latency problems may disappear from the developer view. In addition, as they present a snapshot of the application execution, they often are not able to find complex component interaction issues [59].

Tracing is a low overhead technique to record the low-level events of the system during its execution. This record can then be used by other analysis mechanisms. Tracing can be used to measure several application properties, such as its performance. Unlike logging which deals with high level records of the system, tracing can collect multi-level data from kernel and user spaces [58].

We review some previous work which use tracing for performance analysis of Chromium browser, multi-thread applications and low-level software tracing.

### Chromium tracing

Different web browser architectures present different implementation details. Since there is no unique standard web browser architecture, each browser represents a different use case. However, they face several of the same challenges like sandboxing, accessing asynchronously a large number of network resources, and efficiently but safely executing complex scripts. In this paper, we study Chromium, the open source version of Google Chrome, as one of the best-known web browsers [60].

Chromium tracing is a technique to collect information about the Chromium browser execution. This information can be used to gain a better understanding of Chromium as well as to detect and diagnose some of its performance, stability and security problems. The Chromium source code encompasses different types of trace points to report the operations performed by the multiple threads of the browser at run time. On the desktop version of Chromium, the trace points can be enabled through chrome://tracing. The content of the recorded trace is saved as a JSON file and investigated using a web-based timeline view. This integrated tracing tool is portable and easy-to-use. It offers a neat solution to solve many user level problems encountered by Chromium and web developers [1].

However, user level tracing is not sufficient when it comes to detecting and analyzing the root cause of performance degradations at low level. Chromium creates one separate process per tab and each tab consists of several threads interacting with each other to accomplish a specific user action. In many cases, different threads are competing to get a specific resource from the operating system, resulting in performance degradation. Therefore, we need to know what is happening at the operating system kernel level. In addition, Chromium has a limited buffer size, resulting in limited tracing time, especially when several event categories are enabled. This problem can be accentuated when it comes to tracing a time consuming performance issue under a heavy workload, like loading various web pages in different tabs. This limitation leads to overflowing the buffer, causing the loss of numerous events which might be vital to detect the root cause of the problem. Hence, reliable root cause detection of performance problems cannot be realized through the Chromium tracer.

Our focus now shifts to trace analysis methods.

### Low-level software tracing

Many techniques, adapted specifically for multi-thread applications, focus on context propagation and tracing at scale. They can only collect high-level events, which limits developers for performing detailed root cause analysis. At a lower level, LTTng can be used as an efficient kernel and user space tracer for Linux with an emphasis on performance, with a measured impact of 2% for heavy workloads [17]. Since it is the most performant kernel tracer for Linux [61], a minimum impact is imposed on the system performance. Unlike SystemTap [62, 63], ftrace [64] and eBPF [65], LTTng is not subject to a substantial increase in latency when it is used on multi-core systems [66]. LTTng stores trace data on disk and relies on the Common Trace Format (CTF) [67], a binary trace format designed for high trace event throughput. It also offers a flight recorder mode that reduces the overhead by flushing the trace chunk to disk only once the user requests a snapshot [68]. This feature facilitates the latency analysis of huge applications like Chromium [69].

Low-level software tracing enables offline analysis after event collection. The collected traces may be analyzed to discover important metrics like CPU usage, with an arbitrary precision and without resorting to periodic monitoring and fixed sampling rates [70]. Trace Compass [38] is a trace analysis tool exploiting an efficient and queriable state system structure [7]. Its aim is to compute and visualize the results of complex trace analyses like the thread critical path analysis [71].

### Performance analysis of multi-thread applications

Understanding the performance characteristics of a multi-thread application, using profiling and analysis tools, is not straightforward. It requires detailed knowledge about the processors themselves to analyze the data, which usually encompasses absolute values collected from the hardware performance counters of the processors [72]. Marowka et al. [73] present a case study where the performance of a task-based programming model was compared to a thread-based programming model. Their high-level analysis shows that the former outperforms the latter while the low-level analysis reveals efficient code as the reason.

Kessis et al. [74] present Pajé, a visualization framework providing an interactive and scalable behavioral visualization of parallel multi-level applications, to facilitate the performance debugging of parallel programs by capturing the dynamics of their executions. They monitored some key performance elements of two application servers and identified some critical bottlenecks (contention regions, initialization problems, etc.) during their experiments. However, they limit the number of observed events in order to minimize the perturbation of their

results. This can result in missing some crucial events.

Trümper et al. [75] presented a dynamic analysis technique for visualization, that supports developers for understanding the behavior of multi-thread applications. This technique selectively instruments binaries of the analyzed C/C++ application to record execution traces at run time. It only collects data from the user level, without taking kernel events into consideration, even though it may be a determining factor in the analysis. Furthermore, this approach is not applicable to small size or time critical systems.

Fonseca et al. [76] presented the PIKE tool, for testing concurrent applications. PIKE can find both semantic and latent bugs. Semantic bugs stem from subtle deviations from the expected application behavior, while latent bugs slightly corrupt internal data structures. The authors applied PIKE to MySQL to find concurrency bugs and found several semantic and latent concurrency bugs in a stable version of MySQL. Like the previous work, their technique works only at user level and does not benefit from kernel level information. In addition, it does not investigate the interaction of different threads, which can be helpful to discover the root cause of many problems.

In summary, the above techniques cannot fulfill all the requirements for analyzing a complex application like Chromium. In the rest of this paper, we address these drawbacks and propose our technique which is compatible with this analysis.

### 4.4 Chromium Architecture

In this section, we present the Chromium architecture, a prime use case being a huge and complex multi-thread application. First, we explain some basic concepts used in the architecture. Chromium has a multi-process architecture. The *browser* process is the main orchestrator while other *renderer* processes are created for the individual tabs. Each process is heavily multi-threaded. Every Chromium process has a main thread, an I/O thread, a few special-purpose threads and a pool of general-purpose threads to serve task queues.

The main thread in the browser process is called the UI thread. It is in charge of keeping the main interactive user window alive and responsive. The main thread in renderer processes is responsible for reading the message sent by the browser process. The I/O thread in the browser process handles IPCs (Inter Process Communications) and network requests, while in renderer processes it handles IPCs. All the processes have interactions with each other through IPCs which are issued by the Chrome_IOThread in the browser process or in the Chrome_ChildIOThread in the other processes. Most threads simply loop to get and run tasks from a queue. The queue might be shared between multiple threads [77].

Chromium executes tasks as the smallest unit of work to be processed. In fact, a task is considered as an object having a closure, traits and a post timestamp. Tasks are placed in sequences for asynchronous execution. The order in which different tasks are added to a given sequence determines the order in which they will be executed. Chromium can execute a group of tasks in one of the following ways [77]:

1. **Parallel:** There is no task execution ordering, all tasks will be executed possibly at once on any thread.
2. **Sequenced:** Tasks will be executed in posting order, one at a time, on any thread.
3. **Single Threaded:** Tasks will be executed in posting order, one at a time, on a single thread.
4. **COM Single Threaded:** A variant of single threaded with COM initialized (COM helps prevent deadlocks in calls between objects through providing special functions [78]).

By scheduling sequentially in this way tasks working on the same resources, this removes the need for locking. Indeed, the usage of locks is discouraged in Chromium. Chromium synchronizes the execution of tasks in the following three main ways:

- Once a task is added to a sequence, previous tasks are completed before the subsequent ones in the same sequence.
- It is possible to specify a callback (new task) upon the completion of a task. The new task thus depends on the completion of the first one.
- The callback mechanism has been used through "barrier closures" to wait for the completion of several tasks, for example k tasks. Each of the k tasks has the same callback with a shared counter. When a task completes, the callback increments the counter and checks if it reached k. If so, the barrier is completed and some new task, which depended on the completion of all k tasks, is added.

There are different tasks types such as default_tq, control_tq, frame_loading_tq, input_tq and etc. Depending on each task type, it can be run by a thread in the thread pool. Thus, different threads may interact with each other to accomplish a user action involving several tasks [4].

Since rendering performance is a broad topic of great interest to the Chromium developers, and the web platform is essentially an application development framework, we need to ensure

that rendering has a satisfactory performance to handle input, and to paint the screen at a speed that keeps up with the display refresh rate [5]. Performance is a highly-cited problem area for web developers. They spend a lot of time and effort to understand the root cause of problems so that they can improve the performance of the platform by eliminating janks [5]. They use the term "jank" for an unexpected latency or performance degradation which directly influences the user interface [5]. To identify the symptoms and potential root causes of a jank, we have to gain a basic understanding of the Chromium rendering pipeline.

At the top level of the Chromium architecture, the browser main process coordinates with other processes which take care of different parts of the application. However, for the renderer process, multiple processes can be assigned to each tab. In very recent versions of Chromium, each tab has only one process whenever possible. Figure 4.1 illustrates the multi-process architecture of Chromium in which a subset of processes are interacting with each other, although there are even more processes like the Extension process and utility processes [60].

As can be seen in Figure 4.1, the browser process handles everything outside of a tab. Inside the browser process, there are some important threads performing distinct tasks. For instance, the UI thread draws buttons and input fields of the browser, the network thread manages the network stack to receive data from the network, the storage thread controls access to the files. Once the user types a URL into the omnibox, the input is handled by the browser process UI thread. Since in Chromium the omnibox can be also a search input field, the UI thread has to check the input field to decide whether to send the request to a search engine, or to the site which the user requested. The UI thread then generates a network call to get the site content through appropriate protocols like DNS lookup and establishing a TLS Connection for the request. At this moment, the network thread may receive a server redirect header like HTTP 301. Therefore, the network thread sends a message to the UI thread announcing that the server is redirecting the request, and hence another URL request will be initiated. When the response payload starts to arrive, the network thread must determine the data type according to the response content-type header. If the response is an HTML file, the data is passed to the renderer process. If it is a zip file or any other file type, then it signifies a download request, so the data should be passed to download manager.

However, if the network thread finds the data malicious, the data will not be sent to renderer process. In fact, this is where safe browsing checks happen. After all of the checks ensuring the site security, the browser navigates to the requested site. Therefore, the network thread tells the UI thread that the data is ready, and then the UI thread finds a renderer process to render the web page. Finally, once the data and the renderer process are ready, the browser process sends an IPC message as well as the data stream to the renderer process to commit

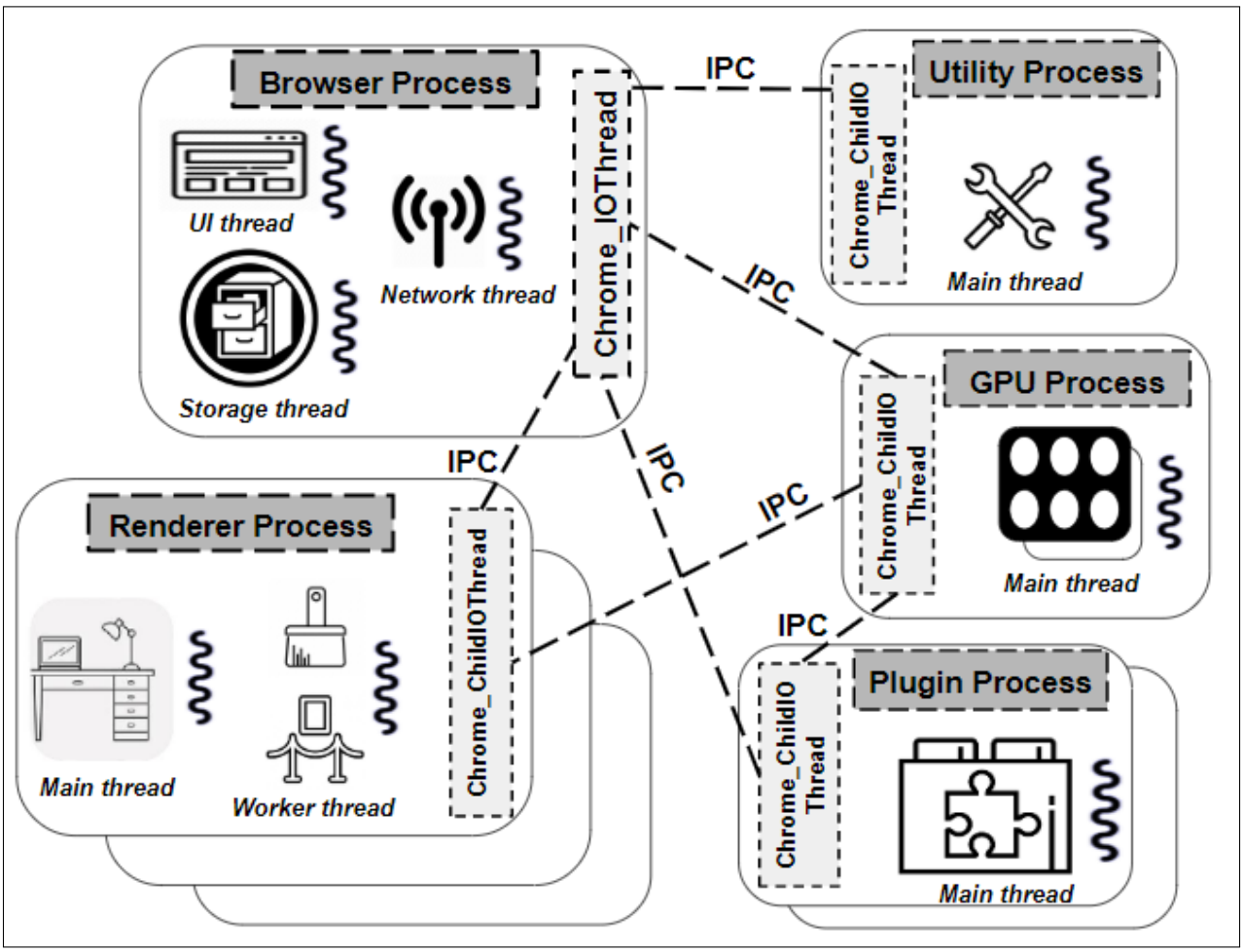


Figure 4.1 Chromium multi-process architecture. Multiple processes might be assigned to each tab

the navigation. When the browser process receives confirmation about committing in the renderer process, the navigation phase is complete and the document loading phase begins. In this step, the address bar is updated and the security indicator and site settings UI reflect the site information for the page. After finishing the rendering, the renderer process sends an IPC back to the browser process. At this point, the UI thread stops the loading spinner on the tab and the web page is ready for the user, although client side JavaScript can still load additional resources and render new views after this point [79]. Table 4.1 describes each Chromium process as well as its responsibility [60].

Table 4.1 Description of different Chromium processes alongside their responsibility

| Process | Responsibility |
|---|---|
| Browser | Controls the main parts of Chromium including the address bar, bookmarks, back and forward buttons. It also handles the privileged parts of Chromium such as network requests and file accesses |
| Renderer | Manages anything inside of a tab where a website is displayed |
| Plugin | Manages any plugins used by the web page like Flash |
| GPU | Handles GPU tasks separately from other processes. It is separated into different processes because the GPU manages requests from multiple apps and draws them on the same surface |

**The benefit of the multi-process architecture**

As mentioned earlier, Chromium uses multiple renderer processes. In most simple cases, with a normal workload, each tab has its own renderer process. Suppose that we have 4 tabs open and each tab is run by an independent renderer process. If one tab crashes, the unresponsive tab can be closed while keeping the other tabs alive. If all tabs are running in the same process, once one tab crashes, all the tabs are unresponsive [60]. Figure 4.2 illustrates this fact.

Another benefit of distributing different tasks to multiple processes is security and sandboxing. As operating systems propose a technique to restrict the privileges of processes, the browser can control certain features of certain processes. For example, Chromium restricts arbitrary file access for those processes handling arbitrary user input like the renderer processes [60].

Since each process has its own private memory space, it often contains copies of common infrastructure. In order to save memory, Chromium sets a limit on how many processes it can handle. The limit differs depending on the memory size and CPU power of the host

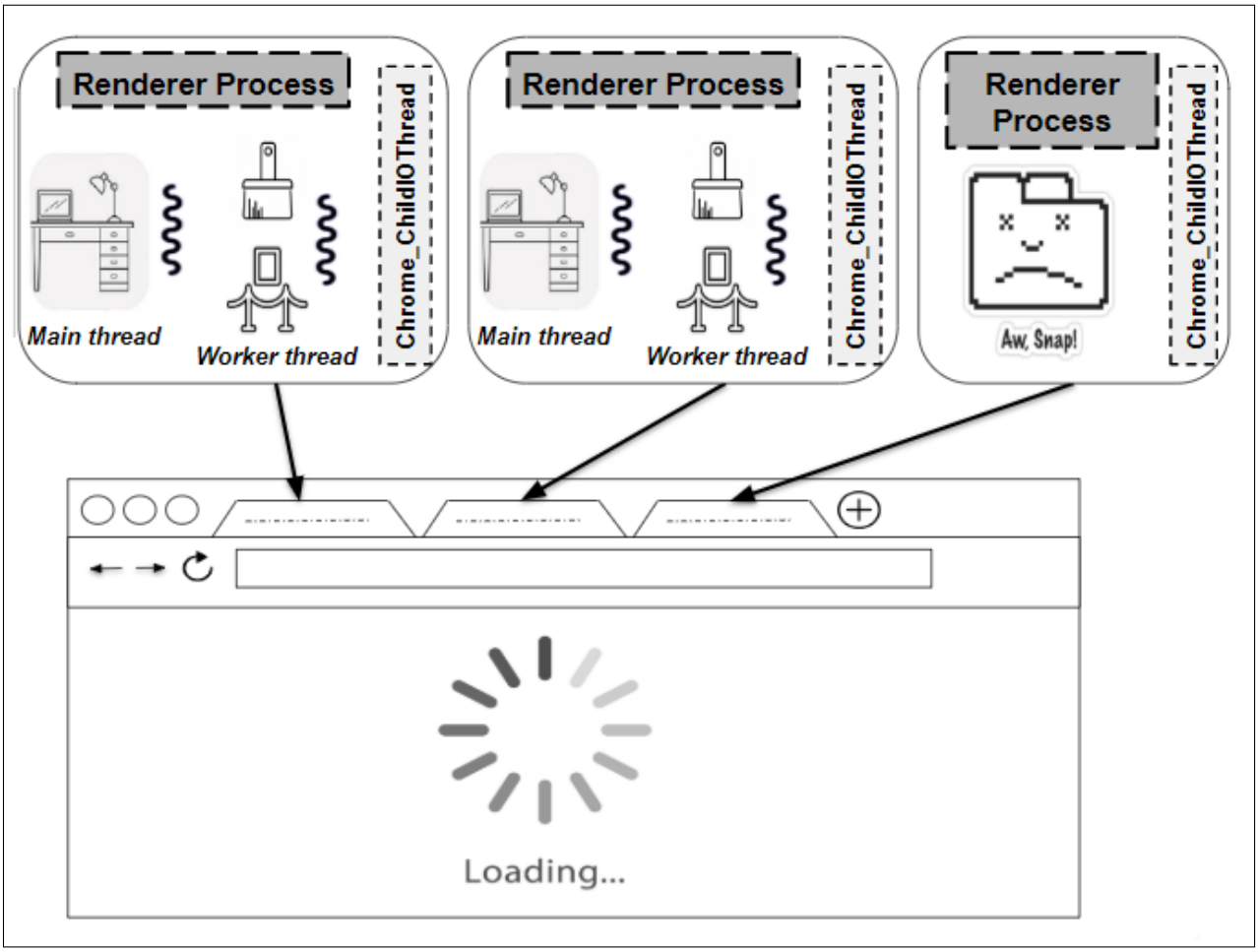


Figure 4.2 Multiple processes each running one separate tab

machine. Chromium starts to run multiple tabs from the same site in one process once it hits the limit [60].

## 4.5 Multi-level Analysis Approach

### 4.5.1 Architecture

Users of huge multi-thread applications like Chromium sometimes perceive unexpected run time latencies and performance issues within various browsing actions. Chromium has an interesting tracing tool which provides a user-level callstack view of functions. Although this view can aid us in collecting information about the cause of the issue, it may not be sufficient to identify the root cause of the problem. Hence, we need to extract more information to precisely discover the root cause.

In this paper, we propose a methodology to identify the root cause of performance degradations in Chromium using multi-level data collection and data analysis. We propose the architecture shown in Figure 4.3.

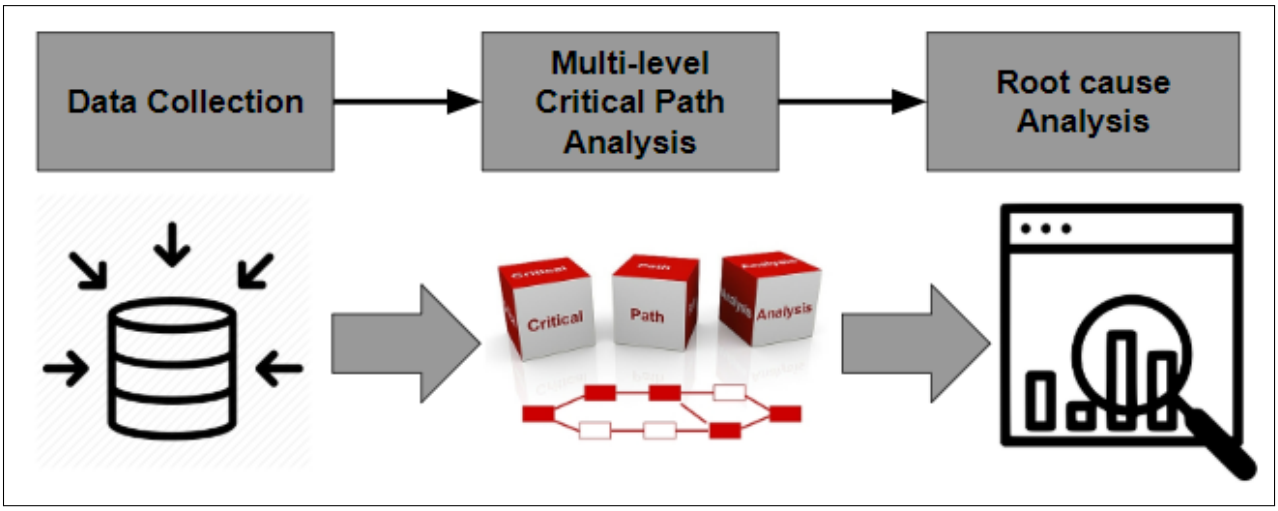


Figure 4.3 Architecture of our work in three main sections

As can be seen, our model architecture consists of three distinct parts: data collection, critical path analysis and root cause analysis. The Data Collection section explains the challenges tackled to extract multi-level data from a complex application like Chromium. Furthermore, we propose a solution to collect unified, synchronized and multi-level data from Chromium.

Our study showed that single thread analysis is not sufficient in Chromium and we need to find out the interactions between threads to identify the root cause of problems. This fact

motivated the analysis of all the threads involved in a user action, such as opening a tab, or loading a web page, instead of analyzing them separately. The term user action denotes when the user is interacting with the page through some input mechanism (typing, clicking on the mouse etc.) [3]. In other words, a user action is an interaction between the user and Chromium that involves potentially multiple nested function calls resulting in the generation of several tasks sequences.

Although the critical path analysis can provide useful information, it is not sufficient by itself to find out the root cause of performance issues. In this paper, we propose an algorithm which combines both critical path data and user level information, and also compares both normal and faulty executions, in order to precisely detect the root cause of performance bugs. This is explained further in the following sections.

### 4.5.2 Data Collection

In this section, we present our solution to collect multi-level tracing data from Chromium. We propose a solution in which we can have unlimited buffer size with arbitrarily long traces in order to collect both kernel and user level events. Using the LTTng-tools module [17], we are able to regroup three components which allow the large scale deployment of any application like Chromium, which is instrumented with the LTTng user space tracer alongside with the kernel tracer. Figure 4.4 illustrates the global architecture model as well as the three components which are LTTng CLI, consumer daemons and the session daemon [2].

Among the available Linux tracers, we choose the Linux Trace Toolkit Next Generation (LTTng) [17], a lightweight tracing tool, because of its low overhead kernel and user space tracing facilities. We also use Trace Compass [38], an Open source tool for analyzing traces and logs, providing an extensible and flexible framework to extract metrics and to build views and graphs.

We used LTTng to enable kernel tracing, but kernel and user level events were not synchronized. As the time resolution of Chromium tracing is millisecond and LTTng provides nanosecond precision for kernel events, the user and kernel level events could not be synchronized with Chromium tracing. Thus, we looked for a unified tracing technique to solve this problem. We instrumented the Chromium source code using the LTTng-UST library [17] so that we could get synchronized multi-level traces. Unfortunately, the LTTng-UST library creates a thread very early to accept tracepoint activation requests, while Chromium actively kills any unknown thread for security purposes.

Consequently, we had to modify the approach and reuse the mechanism already available to

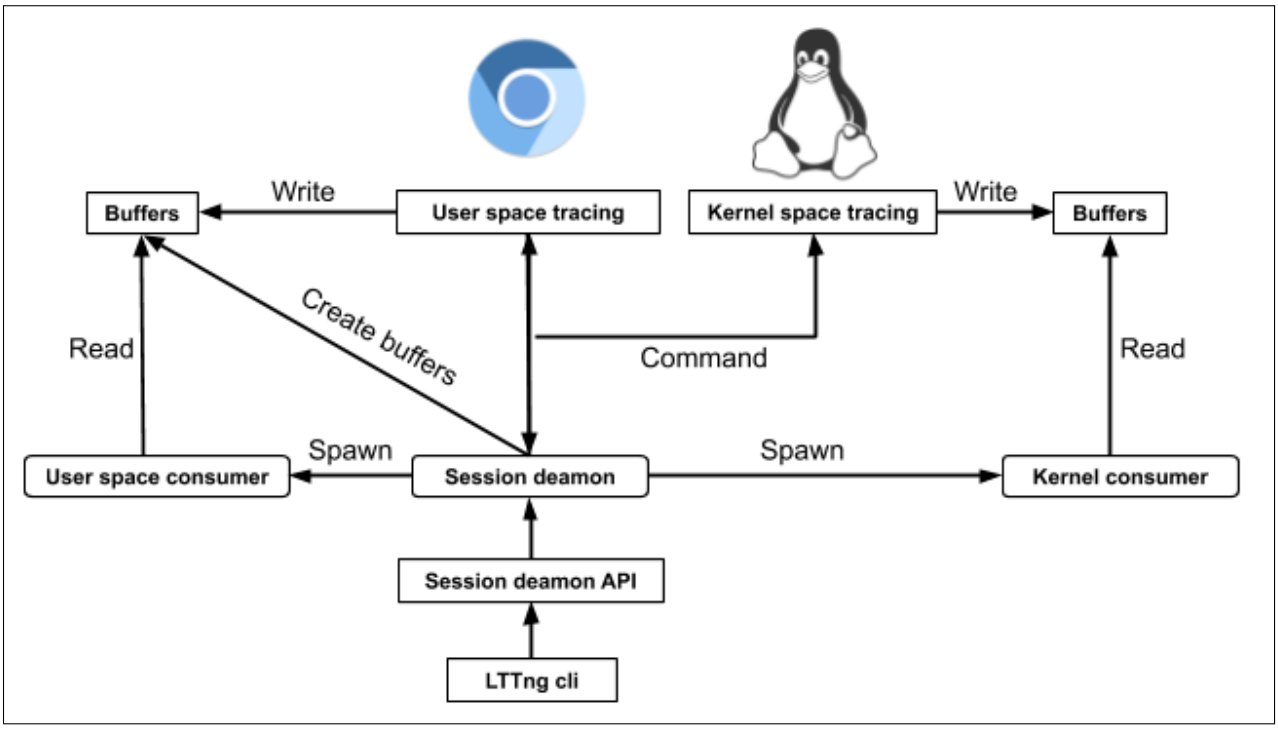


Figure 4.4 Global architecture of three modules to collect multi-level data [2]

redirect Chromium events to the Event Tracing for Windows (ETW) framework. Chromium events are then redirected using this mechanism, but LTTng-UST is called underneath instead of ETW. Redirecting Chromium events to LTTng has the following benefits:

- *Recording arbitrarily long traces:* LTTng is not limited to a fixed-length buffer. It can record traces that exceed the size of the system memory using some features such as buffer snapshots and trace file rotation.
- *Correlating events from multiple sources:* Using LTTng, events can be recorded with unique time stamps across the system (application and operating system). Analysis tools can then correlate events from multiple sources, for example to inspect the time spent waiting for system calls in specific Chromium tasks.
- *Tracing a crash:* With LTTng, no events are lost if a browser process crashes. Recording an LTTng trace with a fuzzer would help to diagnose crashes which are hard to reproduce.
- *Using existing tools:* Developers are used to deploy existing tools such as Trace Compass which enables many analyses and views on LTTng traces.

### LTTng Exporter Module

We Added the method TraceEventLTTngExport::AddLttngEvent() that produces an LTTng event. The method is no-op when TraceEventLTTngExport::IsLttngExportEnabled() returns false. The format of the produced event is listed in Tables 4.2 and 4.3.

Table 4.2 Header of the event produced by the added method

| | |
|---|---|
| Class.Type | Encodes the Chromium event type (TRACE_EVENT_PHASE_COMPLETE, TRACE_EVENT_PHASE_INSTANT, ...) |

Table 4.3 Payload of the event produced by LTTng exporter

| | |
|---|---|
| string | The event name |
| uint64 | An id that helps identifying related events.It corresponds to the id provided to the tracing macro of async and counter events. |
| string | The event category |
| uint32 | The number of arguments |
| | For each argument |
| string | The argument name |
| string | The argument value |
| uint32 | The number of frames in the stack back trace |
| | For each stack frame: |
| uint32 | A pointer from the current stack |

Argument "string" represents an ASCII null-terminated string. A static method TraceEventLTTngProvider::TraceWithArgs was also added to obtain the singleton instance of TraceEventLTTngProvider.

### Export all Chromium events to LTTng

The TraceEventLTTngProvider::TraceWithArgs method is called from TraceLog::AddTraceEventWithThreadIdAndTimestamp so that all Chromium events are emitted to LTTng once there is an active consumer. TRACE_EVENT_PHASE_COMPLETE events will be logged as TRACE_EVENT_PHASE_BEGIN events.

### Export duration of Chromium “complete” events to LTTng

The TRACE_EVENT macros in Chromium produce “complete” events, a span start and end events are combined into a "complete" event with a duration. They invoke the TraceLog::AddTraceEventWithThreadIdAndTimestamp method at the location at which they appear in the code. They also create a local trace_event_internal::ScopedTracer object which calls TraceLog::UpdateTraceEventDuration in its destructor, in order to update the duration of the “complete” event in the tracing buffer. This is fine if everything is traced into memory buffers and tracing stops when buffers are full.

LTTng events cannot be updated, since they may have been flushed to disk. A call to TraceEventLTTngProvider::TraceWithArgs with an event type TRACE_EVENT_PHASE_END will be added to TraceLog::UpdateTraceEventDuration. It will then be possible to retrieve the duration of a “complete” event at analysis time by computing the difference between matching BEGIN/END LTTng events.

### Add an enum value to enable tracing event categories to LTTng

An ENABLED_FOR_LTTng value was added to the TraceLog::CategoryGroupEnabledFlags enum.

An ENABLED_FOR_LTTng flag was added to the INTERNAL_TRACE_EVENT_CATEGORY_GROUP_ENABLED_FOR_RECORDING_MODE macro so that the tracing functions are invoked when LTTng tracing is enabled.

### Enable generating LTTng events when there is an active consumer

In TraceLog::UpdateCategoryGroupEnabledFlag, code was added to enable the ENABLED_FOR_LTTng bit of a category when there is an active LTTng consumer (i.e. TraceEventLTTngProvider::IsTracing() returns true). The disabled-by-default categories are only enabled if requested by the consumer.

The OnEventsEnabled() and OnEventsDisabled() methods were overloaded in TraceEventLTTngProvider in order to be notified when a consumer is enabled/disabled. The implementation calls TraceLog::UpdateCategoryGroupEnabledFlags() in order to update the status of all known tracing categories.

Using this solution, arbitrarily large traces can be extracted, free from the Chromium buffer size limitations.

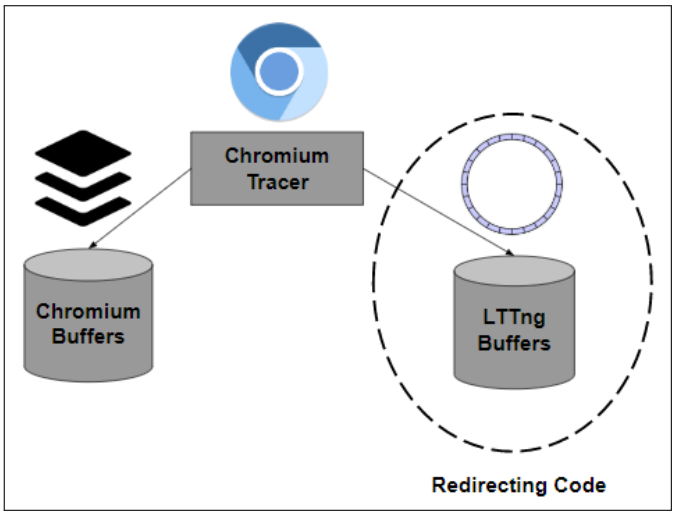


Figure 4.5 Exporting events to LTTng

### 4.5.3 Critical Path Analysis

The Critical Path Analysis investigates the dependencies between different tasks in a project [80] and is widely used to identify sections to optimize. Similarly, in Software Engineering, critical path analysis is often used to assess performance bottlenecks. We must be able to detect the code segments which cause performance degradation. Shortening any segment on the critical path by a small amount will shorten the total execution time by the same amount. As mentioned earlier, Chromium runs tasks in a loop. These tasks are posted to the Chromium task scheduler by different functions, and are executed in order to accomplish user actions. Each task, alongside its type and ID, identifies a unique state at user level. This information was not available in the critical path before the work proposed here. We have instrumented the functions where tasks are queued to run in the future (WillQueueTask) or when a previously queued task is run (RunTask). We also have some other events marking the start and end of each task execution (OnTaskStarted, OnTaskCompleted). We added some additional events, depending on the use cases studied, to detect different user level states. These states can be added to the existing critical path, in order to provide a multi-level critical path computation for Chromium analysis.

Trace Compass provides a critical path graphical view, proposed by [7], to follow the execution path of an arbitrary thread. This view can only demonstrate performance bottlenecks at kernel level. The output of the algorithm is a list of segments which belong to the critical path of the execution, each characterized by a thread id, a thread status, a start and an end timestamp. The thread status takes the value of running, preempted, interrupted or waiting. The waiting reason can be waiting for another thread or waiting for a resource at

the operating system level (block device, network, timer or user input). Thus, when a thread is executing at user level, shown in green, no further information was available.

Since we collect multi-level data in Chromium, we propose a multi-level critical path analysis algorithm which supports dependencies at both kernel and user levels. To do that, using an XML view analysis in Trace Compass, we parse the user space events and fill in further details about the user level execution of threads along the execution graph. The execution graph data structure would be a two-dimensional doubly linked list, where the horizontal links are the start and end of different states within each thread, and the vertical edges indicate wake up signals from one thread to another. We compute the critical path of an execution using the Chromium Execution Graph Construction (CEGC) Algorithm. The input to Algorithm 1 is the trace events and the output is the execution graph G.

In the existing version of the execution graph algorithm, we have only one state for "running at user level" and one reason (event) to change to that state. Therefore, we need to add user space events which can lead to further details about the state while "running at user level". Thus, we have defined two sets (Set1 and Set2) which indicate dependencies in both kernel and user levels. For example, OnTaskStarted marks the beginning of the execution of a task which ends with the occurrence of OnTaskCompleted. The events in Set2 have a context value which indicates whether or not the event comes from another thread. The events which can change the context are defined in Set3. The TASKS set includes the tasks in the system once tracing begins. The array CPU stores the running task for each CPU. The main procedure processes each event according to its type. The events in Set1 (lines 11-14) create one vertex for each task involved. The task prev_tid was running and the edge label is set accordingly. The state of next_tid was preempted because it was in the running queue but not running. In the case of events in Set2 (lines 15-23), a new vertex is appended to the target task. If the events in Set2 occur inside an interrupt, then the edge label is assigned according to the interrupt type. If no interrupt is executing, it means that the event is related to the current task while the target task was blocked. A vertex is appended to the current task and its state becomes running. A vertical edge is created from the tail of the current task to the tail of target task.

**Algorithm 1** Chromium Execution Graph Construction (CEGC) Algorithm, for the Chromium multi-level analysis where further details are provided about each thread when running at user level .

**Input** Trace T, Threads = {t1, t2, ..., tn},
Set1 = {sched_switch, TakeTask, OnTaskStarted, StartNavigation, OnMouseEvent, OnKeyEvent},
Set2 = {sched_wakeup, RunTask, OnTaskCompleted, StopNavigation, OnMouseEvent_End, OnKeyEvent_End},
Set3 = {hrtimer_expire_entry, hrtimer_expire_exit, irq_handler_entry, irq_handler_exit, softirq_entry, softirq_exit, inet_sock_local_in, inet_sock_local_out}
CTX = {Soft_IRQ_1, Soft_IRQ_2, Soft_IRQ_3, Soft_IRQ_4, Soft_IRQ_5, IRQ_19, IRQ_23, High_Resolution_Timer}
**Output** execution graph G

1: $TASKS \leftarrow$ initial set of tasks active in the Chromium scheduler [Declarations]
2 : $CPU \leftarrow \emptyset$
3 : $CTX \leftarrow \emptyset$
4 : **for all** task t ∈ TASKS **do** [Initialization]
5 : **if** t.state is running **then**
6 : CPU[t.cpu] ← t
7 : **end if**
8 : Create initial vertex of task t with timestamp t.begin
9 : **end for**
10 : **for all** event e ∈ T **do** [Main procedure]
11 : **if** e.type ∈ Set1 **then**
12 : new_h_edge(e.prev_tid, e.ts, preempted)
13 : new_h_edge(e.next_tid, e.ts, running)
14 : CPU[e.cpu] ← e.next_tid
15 : **if** e.type ∈ Set2 **then**
16 : int_var ← Peek CTX[e.cpu]
17 : **if** int_var is not null **then**
18 : new_h_edge(e.tid, t.ts, labelof(interrupt))
19 : **else**
20 : v1 ← new_h_edge(e.tid, e.ts, blocked)
21 : v2 ← new_h_edge(CPU[e.cpu], e.ts, running)
22 : new_v_edge(v1, vimages2)

```
23 :        end if
24 :     if e.type ∈ Set3 then
25 :        Change CTX // Update Context
26 :     end if
27 : end for
28 : function new_h_edge(task, ts, l)
29 :    tail←last vertex of task from G
30 :    Create vertex v with timestamp ts
31 :    Create edge tail[right] → v[left] with label l
32 :     return v
33 : end function
34 : function new_v_edge(from, to, l)
35 :    Create edge from[up] → to[down] with label l
36 : end function
```

---

Many user actions result in several interactions between different threads in Chromium. Chromium spawns a large thread pool in which each thread is responsible to accomplish a specific task. For example, the main thread updates the UI in the main browser process, while the I/O thread handles IPCs and network requests in the same process. These threads have different roles in the renderer processes. In addition, there is a pool of general-purpose as well as special-purpose threads which can take tasks from a shared queue. In this paper, we propose a multi-level critical path analysis algorithm, to identify the contribution of each thread in the latency encountered for any user action. This can be especially helpful for cases with several nested user actions, when Chromium is overloaded, and where different threads are interacting with each other. In this state, we have to know the beginning and end of each user action, and the related kernel events, to exactly identify the root cause of issues, whether at user or kernel level. The goal is to improve application responsiveness by identifying performance bottlenecks affecting the total execution time. Hence, we need to have all the information about the threads involved in any user action, to understand what is going wrong.

We have extended the existing critical path algorithm to include all user level blocking states as well. For example, a user action like closing a tab leads to nested function calls, and each function can post one or several tasks sequences to the Chromium task scheduler. Figure 4.6

illustrates the dependency types obtained when performing a user action, leading to several function calls and consequently posting tasks sequences.

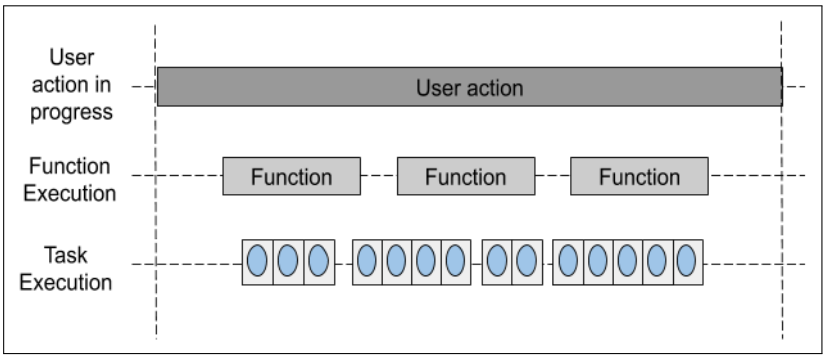


Figure 4.6 Dependency type of user actions, functions and tasks

We can then investigate kernel events to detect the potential root cause at kernel level. Adding the new dependency types, extracted from user level in Chromium, to the existing critical path, we compute the multi-level critical path for any type of performance problem in Chromium. This multi-level critical path can demonstrate the interactions between Chromium threads and also operating system threads.

Trace Compass contains an analysis module which supports user-defined custom analysis in the form of a finite state machine, expressed as XML, to run over the trace data. This flexible mechanism may be fine-tuned to show exactly the desired view to system analysts. Using this feature, we developed an analysis to detect long user actions. We specify user space events which mark the beginning and end of each user action. Once the corresponding intervals are found, we combine the critical path information from user and kernel level. To do that, we defined a Finite State Machine (FSM) in which we build the initial state once the first relevant event for a specific user action is hit. We then build the other user level states happening within that user action on different threads. We then replace the portions from the system level critical path, where we only know that the program is executing, with that more detailed information. The result is a critical path displaying all kernel and user level events during a specific user action.

As mentioned earlier, Trace Compass provides kernel level critical path analysis. For any thread of interest, the analysis calculates its longest path of waiting for resources. It uses low-level events such as context switches or futex system calls to infer the cause of threads being blocked. The minimal set of events and related fields required for critical path analysis are listed in Table 4.4 [7]. Since we aim to extend the critical path analysis to user level events, it is crucial to understand how the critical paths of different threads intersect, so that

we can figure out how the critical paths of different user level requests intersect as well. In fact, it is possible for two different threads to share a state in their respective critical paths. One possibility, for example, is that two threads are waiting on the same mutex, held by a third thread, leading to contention. In this case, the critical paths of all three threads can represent the contention by having a shared state.

Table 4.4 Required kernel tracepoints for critical path analysis

| Events | Fields |
|---|---|
| sched_switch | prev_tid, prev_state, next_tid |
| sched_wakeup | tid, target_cpu |
| irq_handler_entry | irq, name |
| irq_handler_exit | irq, ret |
| hrtimer_expire_entry | hrtimer, now, function |
| hrtimer_expire_exit | hrtimer |
| softirq_entry | vec |
| softirq_exit | vec |

We implemented our event analyzers as separate modules in Trace Compass. Trace Compass is an open source tool to analyze traces. It provides an extensible platform to extract metrics, and to build views and graphs.

Trace Compass produces the critical path as a doubly linked list in which each node can have either an interval with starting time, duration, state and an associated tid, or an arrow with starting time, duration, state, a source tid and a target tid. The status can be running (at a specific level of execution), preempted (in different levels) or waiting for operating system resources (block device, network or timer). We propose an algorithm which produces a multi-level critical path for arbitrary user actions such as opening a new tab, closing a tab, rendering a web page in Chromium. The aim is to analyze the collected trace data, using multi-level critical paths, to identify the root cause of any performance bug or issue, either at user or kernel level.

### 4.5.4 Root Cause Analysis

In previous sections, our global architecture was presented. We explained how to collect multi-level data from Chromium and then how to analyze the gathered data efficiently, so that we can precisely identify the root cause of performance problems. In this section, we demonstrate how to deploy these facilities to come up with the desired result.

First, we explain our setup to generate janks and to identify the actions to investigate.

The Chromium tracer consists of two important parts: a system for collecting performance-relevant data from the browser, and a tool for analyzing and visualizing that data. Users can specify the event categories related to a specific performance issue. Then, the user performs actions to eventually trigger severe janks, while the tracer is gathering events. Depending on the performance bug, we may need to trace a comprehensive set of events, about different layers and threads, for a long time. Hence, we use a flexible approach for multi-level data collection. We use the same triggers, in Chromium, to decide when to collect the system level traces. We initiate continuous system level tracing, using LTTng rotation mode, when starting the Chromium execution. We collect all the user space events categories, as well as an important set of kernel events.

Using an extension developed by Google developers, we then put Chromium under a heavy workload where the browser process is opening back-to-back tabs, and loading a random demanding web page in each tab. Then, several user actions are performed in each tab, such as page scrolling, zooming in and out, reloading the web page and etc. Having the whole trace, it is more flexible to decide what to look for, after the occurrence of performance issues. A python script was developed to go through the trace file and identify all the slow user actions lasting more than a pre-defined threshold. We can thereafter investigate the trace file during the intervals where user actions encountered an unexpected latency. If necessary, we can also examine the trace before and after those intervals to understand the impact of other user actions, functions and tasks.

As defined in section 4.5.1, each user action is an interaction with Chromium resulting in calls to several functions, generating potentially numerous tasks sequences. Depending on the tasks type, they may be run by different threads. Once all the tasks for the user action run, the user action is accomplished. We thus need to know the beginning and the end of each user action to identify their time interval at user level. We also need to track the different tasks, in order to detect which tasks cause problems. For this purpose, we have built two distinct views which provide the capability of tracking either a specific thread or task.

We have defined an XML analysis to read the data extracted from Chromium. Since each task has a unique identifier in Chromium, we can read data either by task ID or thread ID. We can therefore produce two distinct views, either to track a specific task or to follow a thread to understand the entire behaviour of Chromium. Using the XML analysis, we can easily mark the beginning and end of any user action using the corresponding user level events. To do this, we need to determine the user level events marking the beginning and the end of each user action. Table 4.5 shows some of the user level events, and the corresponding user actions, used in our study.

Table 4.5 User actions alongside their user level events

| Events | Fields |
|---|---|
| DidStartNavigation | This event fires if the user clicks or enters a URL |
| DidStopNavigation | This event is fired if the navigation is completely done |
| ViewHostMsg_ClosePage | This event is generated if the user clicks to close a tab |
| OnMouseEvent | A set of events can be generated under this category for various actions such as Pressing or Releasing mouse buttons and Dragging the mouse |
| OnKeyEvent | A set of events can be generated under this category for various actions like pressing or releasing a key |

Depending on the targeted analysis, we can read the trace based on task ID or thread ID. For example, we can investigate the behaviour of the task scheduler based on a specific task types such as default_tq, control_tq, frame_loading_tq, input_tq and etc. This analysis may be more useful for identifying a performance problem occurring in a specific task type in the task scheduler. Figure 4.7 illustrates this kind of analysis.

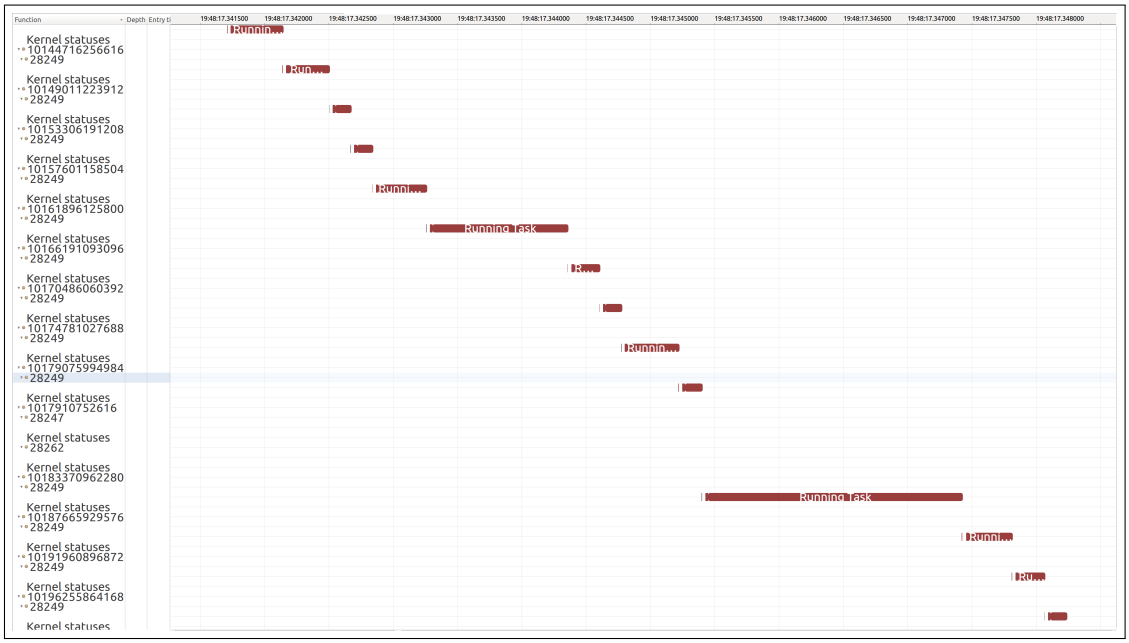


Figure 4.7 Task-based view of execution for different tasks

We can also take a closer look at a specific thread causing a performance issue. For instance, some threads like the main browser thread, or renderer threads, are under more pressure when loading a web page with numerous animations and frames. Thus, we can filter the analysis based on specific threads, as shown in Figure 4.8.

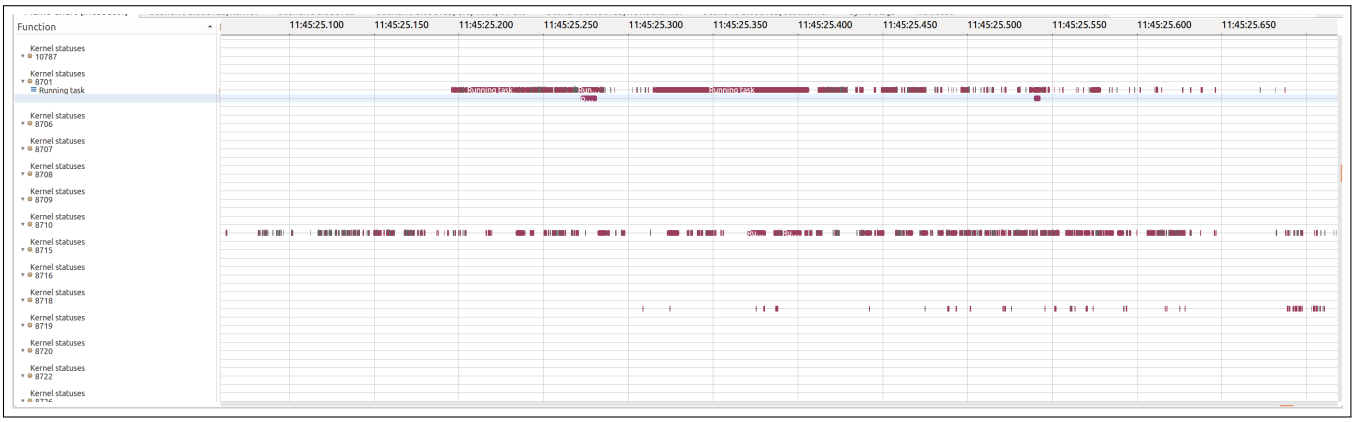


Figure 4.8 Thread-based view of execution for different tasks

Using these views, we can extract the user level part of the critical path, and consequently we can understand what is happening for each user action execution, either at kernel or user level.

## 4.6 Use Cases

We demonstrate the usefulness of our architecture for two existing janks reported directly by Chrome users on the blog with the issue code. Since these use cases are representative of issues that thousands of people have encountered and reported, they can be the appropriate cases to detect and investigate the performance bugs in more details. We show how our implementation can help identify the root cause of them either in kernel or user space by combining both kernel tracing events and user events in our analyses. To do this, we utilized an extension developed by Google team to put Chromium under a very heavy workload where the browser process is opening back-to-back tabs, with a maximum of the arbitrary number of tabs, and loading a heavy website in a loop. After entirely loading the website contents, it closes the tabs one by one if the maximum number of opened tabs is met. As mentioned above, this scenario is repeating in an endless loop as long as we perceive several instances of our desired janks.

### 4.6.1 Use Case 1

As mentioned before, we apply our solution to the existing janks reported by Chrome user [6]. The first one is the jank issue 125264 blocking UI thread by GPU [81]. Based on the chromium bug report blog, GpuChannelHost::Send blocks UI thread to send IPC messages

using IPC::SyncChannel but the root cause is not clear enough [81]. We could reproduce this jank and apply our analysis to discover the root cause. Performing multi-level data analysis alongside critical path analysis for several executions, we could perceive a latency of 100 $\approx$ 400 millisecond for the "membarrier" system call on main thread causing a longer than usual waiting time on it. Figure 4.9 illustrates the critical path of this jank in which memory barrier system call causes a performance bottleneck.

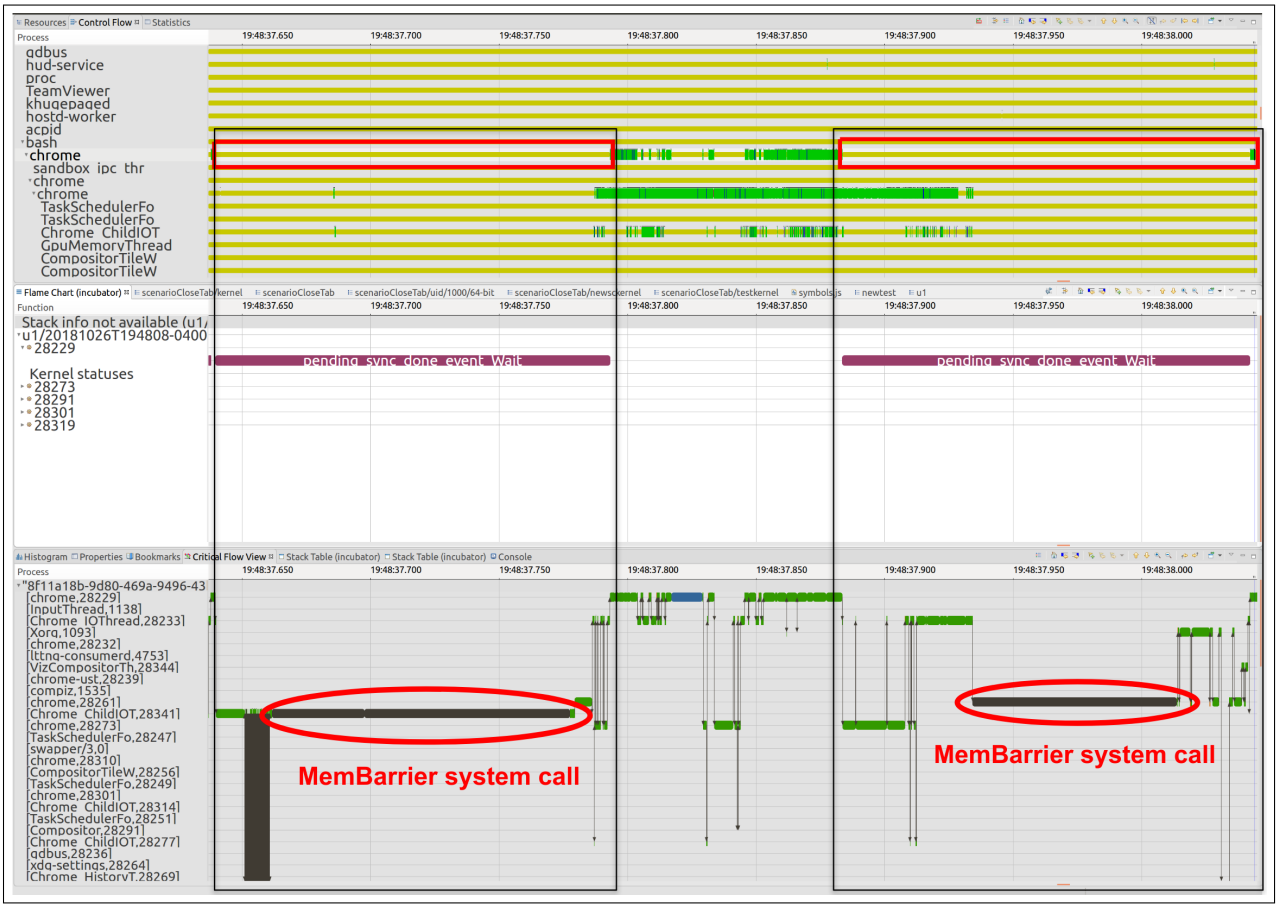


Figure 4.9 Memory barrier system call causing a jank for an IPC message

We also found out that this jank is a result of running the slow path on earlier versions of RCU library which is commonly used in previous versions of kernel (earlier than 4.19). Apparently, membarrier system call has several commands including the fast PRIVATE_EXPEDITED and the much slower global command. It turns out that older versions of liburcu (before 0.9.5) fallback on the slow path which is why the janks happen. Liburcu 0.9.5 introduces the fix. Repeating the scenario for kernel version 4.19 with the RCU library version 0.10 fixes the problem. Doing several executions, we did not receive any jank with the root cause of "membarrier".

### 4.6.2 Use Case 2

Since the main goal in Chromium is to keep the main thread (UI thread in the browser process) and IO thread (each process' thread for handling IPC) as responsive as possible, it should be tried to offload any blocking I/O or other expensive operations to the other threads asynchronous. Therefore, expensive work should not be performed on the main thread, the IO thread or any sequence that is expected to run tasks with a low latency.

In the same jank, we perceived, in many other executions, some long futex system calls on the main UI thread due to the lock contention with Chrome_IO thread. As can be seen in Figure 4.10, there is a wait-blocked state (futex system call) on UI thread (tid:28229). In addition, critical path shows that UI thread is blocked on Chrome_IO thread (tid:28233). As soon as Chrome_IO thread releases the lock, main UI thread continues running tasks. So, lock contention between UI thread and Chrome_IO thread can result in an unexpected latency on UI thread.

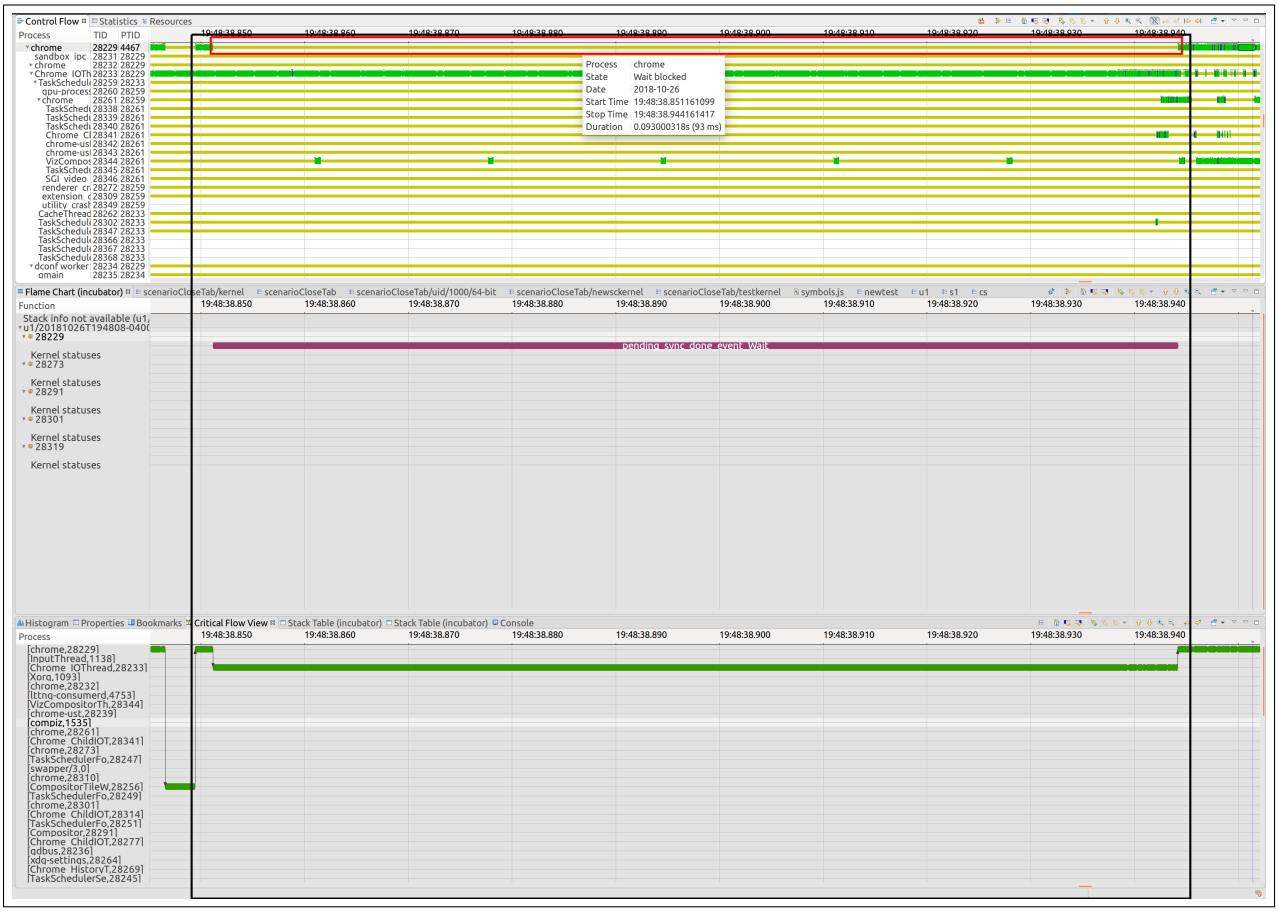


Figure 4.10 Lock contention of threads causing a jank for an IPC message

### 4.6.3 Use Case 3

The third use case is the jank issue 892747 in which processing an IPC message causes a long latency on main thread [82]. Based on the chromium bug report blog, many slow-reports show cases where processing the IPC message ViewHostMsg_ClosePage_ACK on the main thread is extremely slow [82]. In some cases the main thread is stalled even for 15 seconds but there is not enough context about the root cause [82]. We could reproduce this jank on ViewHostMsg::ClosePage in which the main thread is stalled for 911 millisecond. Performing multi-level data analysis alongside critical path analysis for several executions, we could perceive a latency of 911 millisecond in which numerous tasks (approximately 2700 tasks) from different sources are being sent and queued to the main thread while it is running a task from "OnMessageReceivedNoFilter" in src/ipc/ipc_channel_proxy.cc. The most frequent source functions for the posted tasks are "Notify" in "src/mojo/public/cpp/system/simple_watcher-.cc", "SendMessage" in "src/ipc/ipc_mojo_bootstrap.cc" and "FrameDeleted" in "src/content/browser/media/forwarding_audio_stream_factory.cc". Figure 4.11 demonstrates the critical path of this jank in which the main browser process is mostly busy in user space. However, this information cannot approach us to the real root cause and hence, we have to know what is happening in the kernel level. Looking at the kernel level events, we perceived numerous page faults between the beginning and the end of request for closing the tab. As mentioned before, as soon as changing a renderer process to another one, all the frames and views of the old one are swapped out. Once we put Chromium under stress, the browser process is interacting with many renderer processes causing numerous swaps. This can be critical when there is not enough memory and they must be transferred to the disk. Hence, this can lead to a jank once browser process has to access the disk to fetch all the old frames and views again. In the current use case, once Chromium is about to close a tab (renderer process), the browser process has to read the disk to destroy all old frames and views which are swapped out long time ago. This access generates numerous page faults leading to a long latency on the main browser process.

## 4.7 Evaluation

In this section, we explain the evaluation of our method applied to performance analysis of Chromium. Our approach is independent of the operating system and can work on different architectures using Intel or AMD processors. The events are collected by LTTng tracer in both kernel and user levels, and then the events are sent to the trace analyzer (Trace Compass). Our experimental setup is described in Table 4.6. We evaluate our solution in two distinct perspectives: the overhead of our proposed data collection approach and also its

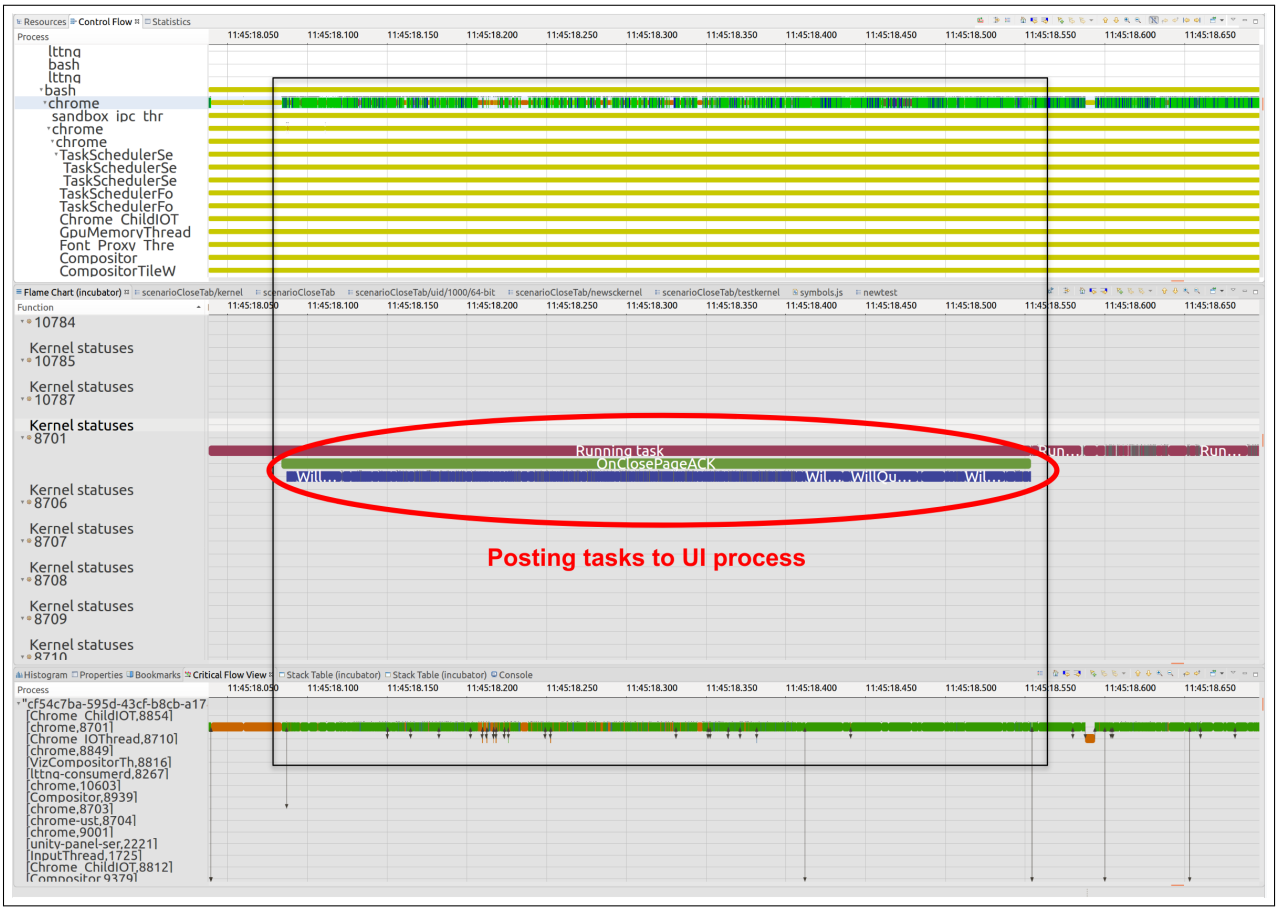


Figure 4.11 Numerous page faults causing a long jank when closing a tab

usefulness. We then state the limitations of our study.

Table 4.6 Experimental Environment of our analysis

| | System specification |
|---|---|
| CPU | 16x Intel(R) Core i7-7820X CPU @3.60GHz |
| Memory | DDR4 2400 MHz,32GB |
| OS | Ubuntu 16.04.6 (Kernel 4.15) |
| LTTng | v2.11 |
| Chromium | 73.0.3672.0 (Developer Build) (64-bit) |

**Data Collection Evaluation**

In this section, we measure the overhead of our trace collection solution, as well as the time required for running our analyses. We show that our solution adds a reasonable overhead to only collecting Chromium traces in user level. We measure the average execution time of opening a tab for 50, 100, 150 and 200 times in the five following scenarios:

**A. No tracing** Both Chromium tracer and LTTng are turned off: no traces are collected.

**B. User space tracing with LTTng** We enabled the user space tracing in LTTng while running Chromium. Since the frequency of user space events are remarkably less than kernel ones, the observed overhead is really negligible.

**C. Kernel space tracing with LTTng** We enabled kernel space tracing in LTTng while running Chromium. It should be noted that we only enabled the events which were crucial to our study such as those needed for critical path analysis (look at Table 4.5.3) as well as for our use cases. Since LTTng is a low-impact tracer which uses about 2% of CPU time on a heavy workload [17], the overhead of kernel space tracing would be reasonable.

**D. Multi-level tracing with LTTng** This is the first configuration for our study. Chromium traces are collected using the technique explained in section 4.5.2. LTTng is also running to collect synchronization events, and a subset of kernel events required for our analyses in Trace Compass. LTTng is running in rotation mode [83], meaning that the current trace chunk is periodically stored on the disk for arbitrary intervals. This is a production-friendly mode of LTTng allowing for collecting huge traces divided into different trace files so that the size of the trace files can be controlled more easily. In this scenario, we are able to investigate the traces before and after a jank if the anomaly is related to another dependent user action.

**E. Tracing with Chromium's tracer** In this scenario, we trace Chromium only with its own tracer without any interference from LTTng.

The average execution time would be an important metric because it represents the performance of Chromium under different tracing methods. For such a huge application, it is vital that the trace collection does not reduce significantly the real-time performance. The average execution time of opening tab action is summarized in Figure 4.12 for 50, 100, 150 and 200 executions, respectively. As shown in Figure 4.12, Scenario A has the minimum average execution time as it only runs the executions without tracing. When we trace Chromium at kernel and user levels under Scenario D, the maximum execution time is obtained. Since the kernel events are more frequent than user level ones, we encounter a higher average execution time in Scenario C as compared to Scenario B. Scenario B and E have relatively similar average execution times as they only perform user level tracing in all executions.

### Usefulness

As mentioned earlier, Chromium tracer has a limited buffer size which is working only in user level and consequently, we cannot trace Chromium for a long time notably under a heavy workload. Using the solution proposed for data collection, we could eliminate the limitations of Chromium tracer. We have a tracing platform specifically designed for Chromium in which we can have any arbitrary trace size in both kernel and user levels.

In addition, the collected data can be used to detect any performance bug. Since performance issues rarely occur, reproducing them again can be a challenging task which makes root cause analysis difficult. We have proposed a tracing platform to detect any kind of jank automatically in run time execution with its multi-level tracing data to investigate the root cause in further details.

We proposed our approach for Chromium browser while it can be used for any kind of complex multi-thread application.

### Limitations

Our solution can identify any performance bug in Chromium. However, there are other kinds of issues like functionality problems. Since we focus on jank detection that stems from unexpected latency in run time, other kinds of problems which do not necessarily affect the performance cannot be detected by our technique. For example, any unwanted response (like pop-up windows) cannot be detected.

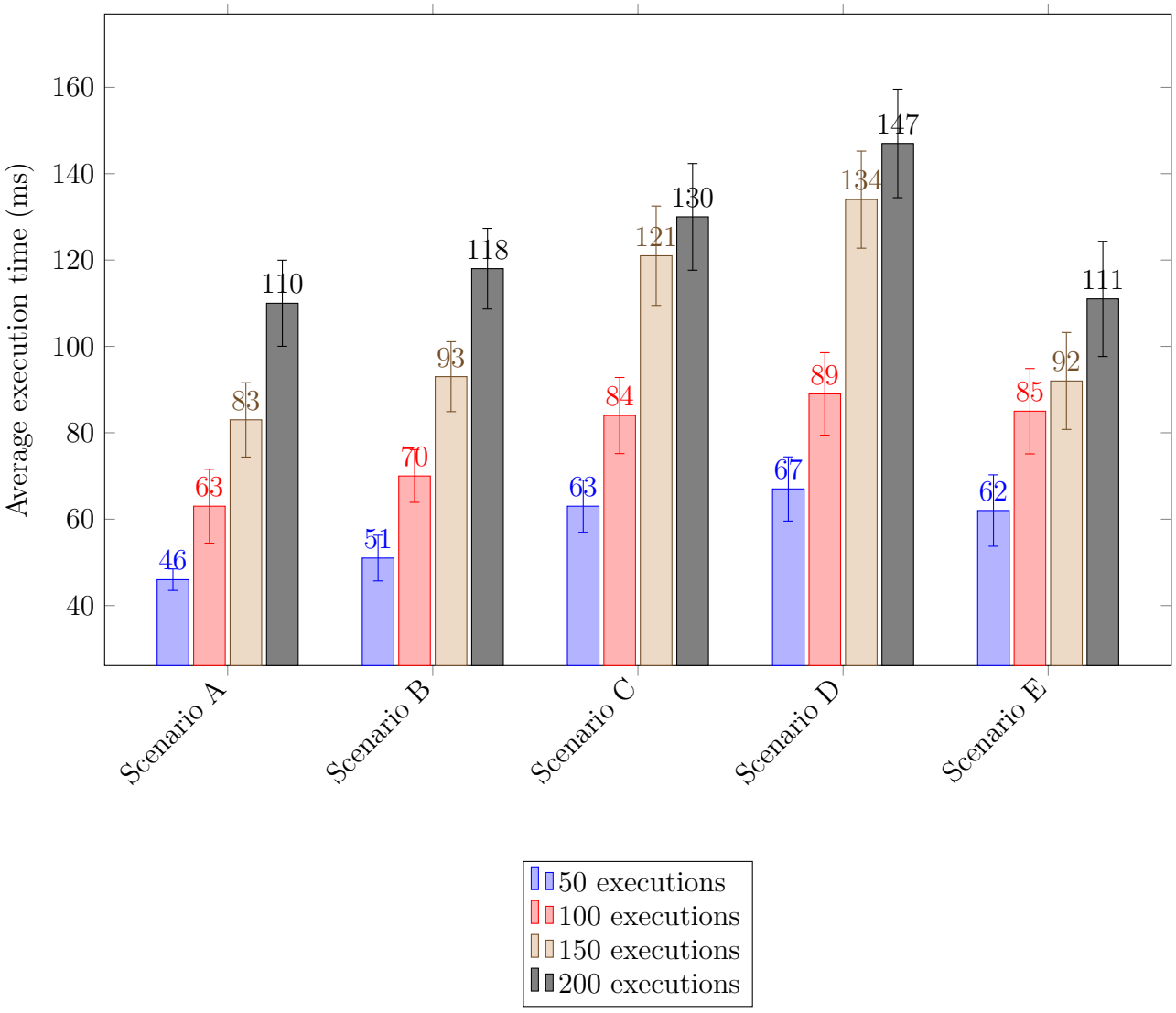


Figure 4.12 Average execution time of several user actions

## 4.8 Conclusion and Future Work

Complex multi-thread applications like Chromium suffer from latency performance issues which directly affect user interface. Developers make tremendous effort to mitigate these impacts. However, they do not have appropriate tools to find out the root cause of problems. In this paper, we proposed a unified solution for collecting and synchronizing the trace events in both kernel and user levels with nano precision time-base. We also proposed our method for critical flow extraction of user actions which is used directly in latency problem detection and analysis. We also investigated the root cause of three real use cases (janks) using our solution and perceived some long system calls like MemBarrier or other kernel events like page fault can cause unexpected latency in user space which is hidden from developers' perspective. We calculated the overhead of our solution and showed that it is reasonable for the immense tracing data of Chromium.

Future work can focus on integrating machine learning algorithms into our tool to automatically distinguish groups of similar janks. It will be then possible to compare the call stacks of a group of similar janks to normal executions to understand the other factors causing the janks.

**Acknowledgement**

We would like to thank Ciena, the Natural Sciences and Engineering Research Council of Canada (NSERCC), EfficiOS, Ericsson and Google for their financial support. We also thank Geneviève Bastien and Adel Belkhiri for their help and comments.

# CHAPTER 5 GENERAL DISCUSSION

The previous chapter focused on the multi-level critical path analysis, for large multi-thread applications like Chromium, when the source code is available and may be instrumented. In this context, the proposed approach can efficiently gather information about the user and system level and combine this information. The developer can then get a clear picture of where the time is spent, with the necessary level of details, whether the problem is in user or kernel space. In our earlier work, other applications were studied, for which the source code was unavailable and user level instrumentation not possible. For such cases, similar techniques were proposed but with a much greater emphasis on the study of threads interactions, especially around locking primitives. The main focus of this chapter is this additional work and associated results.

In this chapter, we use a dynamic trace analysis to discover resource contentions in multi-thread programs, without knowledge of their source code or changing the operating system kernel. We propose a unified multi-level solution, providing different views to diagnose the specific parts of the program leading to lock contention and performance degradation. In these views, the sequence of critical sections running, or waiting to enter the critical section, is shown for each thread separately. More importantly, we extend the critical path of the execution to user and kernel levels to show exactly which thread holds the lock and which other threads wait for that lock. The proposed tool and views can help programmers to get more insight about the real execution of their application, and can guide them trough the possible problems that can occur and point them to what they can do to get them sorted.

Monitoring and analyzing multi-thread applications has always been a big challenge for developers, in terms of debugging and troubleshooting. This challenge becomes more critical, when there is performance degradation or unexpected latencies in the application, because detecting the root cause can be difficult. Dynamic tracing is a mechanism which allows to provide multiple statistics sources, as well as the means to describe exactly what data to gather, how to manipulate the data, and how to present that data to the user. Using dynamic analysis, we can obtain information about program actions and events during the execution time. This technique may be used effectively in order to diagnose performance issues in multi-thread applications, without having any knowledge about the source code of the application. Since the dynamic analysis (through execution traces) can provide a comprehensive insight into various system levels, from kernel level [37, 84, 85] to user level [86], it consequently leads to the detection of many performance bottlenecks and misbehaviors. It

is broadly used among developers and analysts [87–89].

The source of many of these problems stems from the contention on shared resources. Understanding contention issues can be a difficult task using the existing tools and debuggers. Most of these tools are based on knowledge about the source code, which is often not the case for large multi-thread applications.

In this chapter, we propose an efficient approach for multi-level contention analysis. Using this method, we can precisely detect the start and end times of locking, method calls to shared objects, waiting and blocking due to running critical sections, and critical section executions as well. To do that, we have extended the existing critical path analysis to both kernel and user spaces to deploy our analysis for different types of locking methods. We have also limited the proposed critical path to lock contention analysis, so that we could distinguish the dependencies between the different threads in multiple levels, causing the lock contention and consequently performance degradation of the program.

We have instrumented the kernel space and user space using LTTng, a lightweight open source tracing tool. Using this tool, we can analyze different locking methods at a detailed level. The lock methods studied in this chapter cover the main locking primitives: mutex, spinlock and semaphore. We will demonstrate that our approach can be helpful in order to understand performance issues in multi-thread programs. Moreover, it can be applied to large multi-thread applications for detecting bugs and root causes arising from resources contention. Therefore, in this chapter we first propose a lightweight and efficient method for contention analysis in user space and kernel space layers, without any change in the program source code. Therefore, we can determine whether existing resource contention in the program is efficient or not. Second, we present tools and views which show contention from different aspects. Third, more importantly, we propose a multi-level critical flow view of the threads contenting for shared resources, to exactly show which thread holds the lock and which others are in the waiting queue to get the lock, at any moment during the execution.

## 5.1 Contention Analysis

### 5.1.1 Motivation

Since the performance troubleshooting and analysis of large multi-thread application can be a crucial challenge, it is necessary to study contention analysis in order to obtain a comprehensive view. The problem is that the performance counters (in their current form) toss away a considerable useful context. For example, the performance counters in a large multi-thread applications can be used to examine the total wait time over a fixed period of

time, across all mutex instances, but they cannot tell us which threads were affected most, or which specific mutex instances were to blame for large wait times. Nonetheless, this is very useful information for debugging and performance analysis. We analyzed two releases of Trace Compass, a large multi-thread application for viewing and analyzing different types of logs or traces, for performance diagnosis. We detected an unexpected contention in the new version. Looking at the source code, we could find the contention bottlenecks and solve the performance issue in new release. This motivated us to propose an unified method to handle contention analysis using dynamic tracing.

Furthermore, programmers and developers may not be aware of what is happening in their multi-thread programs when it comes to a performance degradation and latency. This is particularly the case when we encounter some difficulties in large applications like Chromium. Hence, we will show how the program runs critical sections and any contention, in the shape of visual graphs and charts, to get a deep insight into multi-thread applications. This can help the developers to understand how the threads run in the program.

### 5.1.2 Data Gathering

Some locking methods (mutex, semaphore) are implemented in kernel space (futex-syscall), in order to block and wake up the critical section. However, some others (spinlock) rely solely on user space, without any interaction with the kernel. Therefore, we decided to record traces at kernel and user space level. We have investigated the futex system call, in order to know the details when it is called. The results are reported in Table 5.1.

Table 5.1 The events of a futex system call

| TP provider name | TP name | Description | Instrumented function |
|---|---|---|---|
| lttng_ust_pthread | pthread_mutex_lock_req | The thread requests to get mutex object | pthread_mutex_lock |
| lttng_ust_pthread | pthread_mutex_lock_acq | The thread acquires the mutex object | pthread_mutex_lock |
| lttng_ust_pthread | pthread_mutex_unlock | The thread unlocks the mutex object | pthread_mutex_unlock |

We instrumented the pthreads library using the LD_PRELOAD technique. LD_PRELOAD is an optional environment variable which contains one or more paths to shared libraries or shared objects, that the loader will add before any other shared library, including the C runtime library (libc.so) [90]. As a result, program behavior can be non-invasively modified, i.e. without recompilation. When we preload the lttng-pthread-wrapper shared object, it replaces the functions listed in the Table 5.1 with wrappers containing tracepoints and a call to the replaced functions.

### 5.1.3 Data Model

In this section, we explain how events are converted to states. Each state can be defined as the time duration between two events. We store all the states in the state system. The state system is an efficient tree-based structure that stores the state of different attributes over time. The general idea of our model is to extract and record the intervals of different state values, computed from the trace events for system resources (processes, CPUs, disks, etc.).

A. Monplaisir et al. [10] showed that the state system contains two important parts: the current state and the state history tree. The current state handles the ongoing state values for the current trace time, whereas the state history tree includes the past state values of the system attributes. Using the state provider, we can compute the effects of an event on the current state. Indeed, the state provider converts the trace events into state changes. It checks the type and contents of all events in the trace and updates the state system. As a case in point, we define that the state of a futex is locked, between two events "syscall-entry-futex" and "syscall-exit-futex". It is also possible that some events make several state changes, or no change at all. Therefore, the state provider module needs to know the types, names and contents of the events of interest which will be present in the trace. The state provider is a crucial part of the system to model the state of the traced system, which is the basis for further analysis.

The term “attribute” represents the fundamental element of our state system. Each attribute describes a modeled aspect of a component which can store a single state value at any given time within the execution. An attribute can represent anything, this is determined by the model to be built for a traced system. Each attribute can represent a system resource (e.g. a futex), but a resource may have several attributes to characterize its different aspects (e.g. different attributed of a file). To manage the huge number of different but somehow related attributes, they are organized hierarchically as a tree, called the attribute tree. The attribute tree is an in memory data structure of the attributes of the system resources. For complex systems, as the number of attributes increases, putting them in a tree leads to a better structured organization and, depending on the type of queries, better performance. In the attribute tree, there is only one access path for each attribute from the root node. To have a better understanding of the attribute tree, we compare this structure to the files and directories in a filesystem. Directories resemble the attribute access paths, and filenames correspond to the attribute names. Finally, the content of the file represents the state value of that attribute [10].

When the state provider reports a state change, a transient state record is created for the new attribute and stored in memory, in the current state system. Each transient record involves

an attribute name, a time and a state value. If an entry exists for this attribute, the present value is replaced by the new one. However, the previous one is kept and stored in the state history tree to enable interactively navigating through the trace and showing timelines for the state of different resources (e.g. the state of each thread over time). Thus, we first set one interval record from the available information, which include an attribute (from the attribute tree), the old state value, the old time value and the new time value (interval start and end times). The completed interval can now enter into the state history tree. This process is continued until the end of the trace [10].

We apply the above-mentioned architecture to a multi-thread program with 3 threads, in order to query and visualize the state of threads in each time interval. We use the thread ID (TID) as an attribute and define different states for threads, including running, waiting, blocked, futex, gettid and etc. We store the states in the state history tree in order to facilitate finding the thread status as a function of time. Figure 5.1 depicts the state system structure when a thread obtains a mutex (lock) to execute a critical section. We can define two different states, named "wait for lock" and "running critical section", for each thread, between events "pthread-mutex-lock-req", "pthread-mutex-lock-acq", and "pthread-mutex-unlock", respectively.

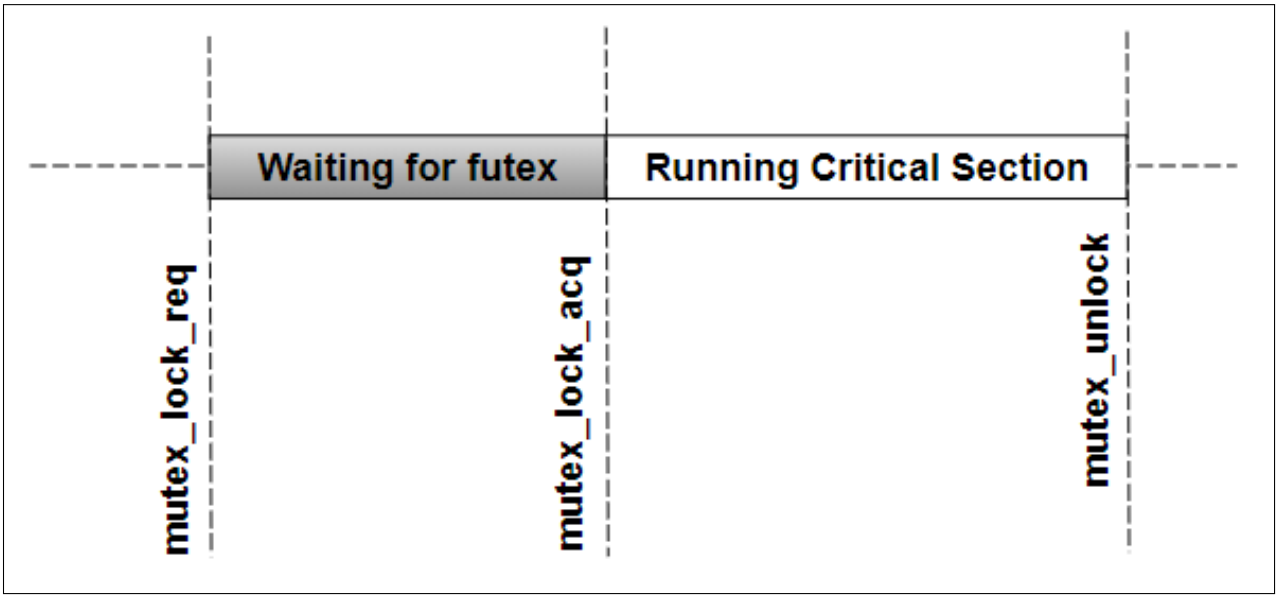


Figure 5.1 State system structure for a thread which is obtaining mutex

### 5.1.4 Analysis

We use the structure introduced in section 5.1.3 in order to extract useful information for the performance analysis of multi-thread programs. All the views and graphs which are presented in this chapter are related to a multi-thread program calculating PI. The first analysis of this section is the wait-block analysis. We use the wait-block analysis in order to compute the amount of waiting and blocking time for each thread. Indeed, we can detect which threads are waiting when a thread holds a mutex. This view can help programmers to know the percentage of execution time during which each thread runs a critical section or waits before the execution of a critical section. This view clearly shows the waiting and execution time for each thread and for each critical section. If there are performance bugs in the program source code, the waiting time of some threads increases and this leads to latency when running the program. Hence, the programmer can improve the multi-thread effectiveness of the program and easily find the time-consuming threads, and consequently the slow part of the program, in order to diagnose the performance issues.

The wait-block analysis finds the tasks and devices causing waits. The wait cause detection is recursive, encompasses all running tasks in the system, and acts across communicating computer nodes using packet-based trace synchronization [91]. We run a program with 3 threads to demonstrate the efficiency of our analysis. Figure 5.2 displays the wait-block timeline for a program which is run with 3 threads. As we can see, when a thread enters the critical section (shown in purple), the other threads are blocked until the mutex is free. Apart from that, we can calculate the exact time for running the critical section, running in user space, or waiting for a futex, a CPU core, I/O devices, or blocked for other causes, for each thread separately.

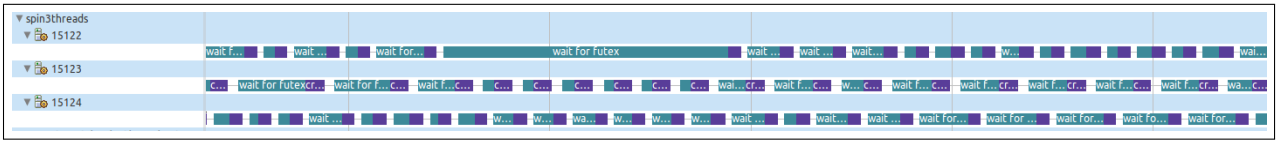


Figure 5.2 Wait-block analysis view for a multi-thread program with 3 threads

Another analysis which is proposed is the flame graph, an aggregated view of the function calls from the call stack view [38]. Each entry in the flame graph shows a combination of all the calls to a function, in a specified depth of the call stack, with the same caller. This enables the user to easily distinguish the most executed code path. Figure 5.3 shows the flame graph view in which are shown the total waiting time to acquire the lock as well as the critical section execution of all threads. Using this view, the programmer can study

the performance of each thread during the execution. This is useful when the programmer expects to keep a specific thread, like the main thread of a browser, as responsive as possible. If the total waiting time is more than a specific threshold, the developer can be notified.

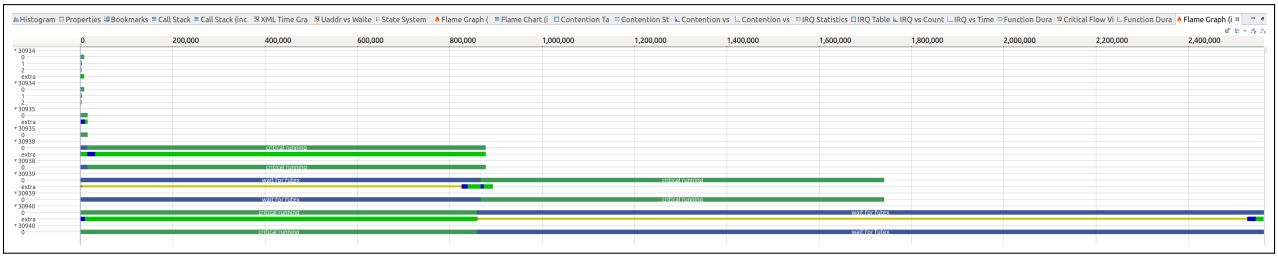

Figure 5.3 Flame graph view for a multi-thread program

The critical flow view is the last one explained in this chapter. Once the programmer finds a suspicious or problematic execution, the goal is to analyze it in more details. The first step is to use the critical path analysis to provide interesting information about the significant dependencies of a thread. The critical flow view shows the dependency chains for a given process, based on kernel traces. This gives us the possibility to detect if a higher priority process was blocked by a lower priority process. Another mode of this view helps us to see all the threads which interact with the execution thread. This can be important to understand the system behavior without looking at all the threads that are not related. In addition, we can also show the critical path of different executions and complementary information. For instance, we can see that the execution was preempted because another thread had a higher priority. It is a configuration problem, because the programmer did not realize that this other thread would have a higher priority in that circumstance. Once the problem and its origin are discovered by the tool, the remedy is often simple to devise for the programmer. Figure 5.4 shows the critical flow view for a program with 3 threads.

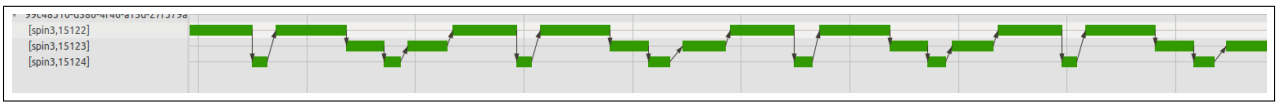


Figure 5.4 critical flow view for a multi-thread program with 3 threads

# CHAPTER 6 CONCLUSION

In this section, we conclude our work by summarizing the contributions we made to the field of performance analysis using Chromium as a demonstration vehicle of complex multi-process and multi-thread application. We also present the limitations of our solution and recommend future improvements.

## 6.1 Summary of Work

This research project aimed to provide a multi-level tracing platform in order to automatically diagnose performance bugs and issues in complex multi-thread applications like Chromium. To do this, we first tackled several problems to collect the necessary data from Chromium and then analyze it in Trace Compass. We proposed a new algorithm to automatically compute the multi-level critical path in order to identify the root cause of various janks, either at kernel or user level. Indeed, we investigated two case studies in which Trace Compass allowed a rapid and accurate diagnosis of performance issues. We have also achieved our 5 specific goals. (1) We have developed a solution to capture all the user and kernel events from Chromium, making the link between low level events and the application logic. (2) We eliminated the limitations of the Chromium tracer in terms of tracer buffer size as well as gathering low level events. (3) We built the call stack for events to detect nested function calls and facilitate the root cause analysis. (4) We have proposed an algorithm for calculating the critical path, allowing to find all the events affecting the completion time of a given user action execution. (5) We presented three case studies demonstrating the effectiveness of our solution.

## 6.2 Limitations

A limitation of this research work is that we conducted our study only on Chromium. While it may be one of the most used applications in the world, each application has its own security mechanisms as well as its own architecture, data collection and can represent a challenging and tedious task. Since many open source multi-thread applications have event-triggered embedded mechanisms for ETW, SystemTap or etc., it is possible to deploy a similar methodology to trace them. However, it would be interesting to put our solution in the hands of many external users to get more feedback. Since Chromium is an open source software, and since our group collaborates with several Open Source communities and industrial partners, we expect to receive valuable feedback in the coming months. It would

be particularly interesting to know how our solution can identify some more complicated janks in a variety of complex applications.

Another limitation of our work is that our solution focuses on the identification of performance bugs. However, there are other kinds of issues like functionality problems. Since we focus on jank detection, that is unexpected latencies at run time, other kinds of problems which do not necessarily affect the performance cannot be detected with the proposed technique. For example, unwanted response (like pop-up windows) cannot be detected. In addition, some janks which are independent of user actions cannot be detected at user level. For instance, memory leakage in Chromium, if it happens while the user is idle, cannot be detected via user level events marking a specific action. Hence, in such a case, further trace points should be inserted for each specific issue.

## 6.3 Future Research

As mentioned in 6.2, it would be interesting to investigate some complicated use cases causing very long latencies which directly affect the main UI process. Apart from that, our tracing platform requires an effort on the part of the user to specify groups of janks, in order to analyze them in a more flexible and automated way. It would be interesting to integrate Machine Learning algorithms into our tool to automatically distinguish groups of similar janks. It will be then possible to compare the call stacks of different similar janks to normal executions to understand which functions causes the latency.

Using Machine Learning algorithms, it will be possible to identify the relation between the task types and janks. It is also possible to count the frequency of each task type resulting in different janks. This can be useful to find out the overhead of each task type at run time. The same analysis can be applied to the critical path analysis.

# REFERENCES

[1] C. McAnlis, "In-depth: Using Chrome://tracing to view your inline profiling data," https://www.gamasutra.com/view/news/176420/Indepth_Using_Chrometracing_to_view_your_inline_profiling_data.php, 2012, [Online; accessed August 24, 2018].

[2] D. Goulet, "Unified kernel/user-space efficient linux tracing architecture," Master's thesis, École Polytechnique de Montréal, 2012.

[3] M. Ahmed, "User Activation v2 (UAv2)," https://mustaqahmed.github.io/user-activation-v2/, 2017, [Online; accessed 2017].

[4] F. Doray, "Browser I/O Scheduler," https://docs.google.com/document/d/1S2AAeoo1xa_vsLbDYBsDHCqhrkfiMgoIPlyRi6kxa5k/edit#,, 2018, [Online; accessed September 2018].

[5] "Anatomy of Jank," https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/anatomy-of-jank,, 2018, [Online; accessed 2018-09-10].

[6] "Chromium bug report," https://bugs.chromium.org/p/chromium/issues/list,, 2018, [Online; accessed 2018-09-01].

[7] F. Giraldeau, "Analyse de performance de systèmes distribués et hétérogènes à l'aide de traçage noyau," Ph.D. dissertation, École Polytechnique de Montréal, 2015.

[8] F. P. Doray, "Analyse de variations de performance par comparaison de traces d'exécution," Master's thesis, École Polytechnique de Montréal, 2015.

[9] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 8:1–8:13. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670987

[10] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, "Efficient model to query and visualize the system states extracted from trace data," in *Runtime Verification*, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 219–234.

[11] M. B. P. S. M. P. D. B. S. J. e. C. S. B. Sigelman, L. Barroso, ""dapper, a large-scale distributed systems tracing infrastructure," in *Google research*, 2010.

[12] I. Ceaparu, J. Lazar, K. Bessière, J. P. Robinson, and B. Shneiderman, "Determining causes and severity of end-user frustration," *Int. J. Hum. Comput. Interaction*, vol. 17, pp. 333–356, 2004.

[13] "How much more traffic should you actually be getting," https://neilpatel.com/blog/loading-time/,, [Online; accessed 2019-09-01].

[14] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 171–187. [Online]. Available: https://doi.org/10.1109/FOSE.2007.32

[15] "Freebsd flame graphs," http://www.brendangregg.com/blog/2015-03-10/,, [Online; accessed 2019-09-01].

[16] "Node.js in flames," http://techblog.netflix.com/2014/11/nodejs-in-flames.html, [Online; accessed 2019-09-01].

[17] M. Desnoyers and M. R Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," *OLS (Ottawa Linux Symposium)*, 01 2006.

[18] "Full stack system call latency profiling," https://lttng.org/blog/2015/03/18/full-stack-latencies/, [Online; accessed 2017-09-01].

[19] "linux-test-project/ltp," https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/sched/cfs-scheduler/hackbench.c, [Online; accessed 2019-09-07].

[20] "Using the trace_event() macro," http://lwn.net/Articles/379903/, [Online; accessed 2017-09-01].

[21] e. M. R. D. P.-M. Fournier, M. Desnoyers, "Combined tracing of the kernel and applications with lttng," *OLS (Ottawa Linux Symposium)*, 2009.

[22] "Ms open tech contributes support for windows etw and perf counters to node.js,," http://blogs.msdn.com/b/interoperability/archive/2012/12/03/ms-opentech-contributes-support-for-windows-etw-and-perf-counters-to-node-js.aspx, [Online; accessed 2019-07-22].

[23] "Introduction to profiling with event tracing for windows," https://www.wintellectnow.com/Videos/Watch/introduction-to-profiling-withevent-tracing-for-windows?videoId=introduction-to-profiling-with-event-tracingfor-windows, [Online; accessed 2019-08-12].

[24] "Improve debugging and performance tuning with etw," https://msdn.microsoft.com/magazine/cc163437.aspx, [Online; accessed 2019-08-14].

[25] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247415.1247417

[26] M. Desnoyers, ""low-impact operating system tracing," Ph.D. dissertation, École Polytechnique de Montréal, 2009.

[27] "A userspace tracing comparison : Dtrace vs lttng ust," http://www.dorsal.polymtl.ca/fr/blog/yannick-brosseau/userspace-tracing-comparisondtrace-vs-lttng-ust, [Online; accessed 2019-05-08].

[28] "Profiling vs. tracing," http://ipm-hpc.sourceforge.net/profilingvstracing.html, [Online; accessed 2018-11-08].

[29] "Performance tools developments," http://indico.cern.ch/event/141309/session/4/contribution/20/material/slides/0.pdf, [Online; accessed 2017-11-15].

[30] "perf : Add backtrace post dwarf unwind," http://lwn.net/Articles/499116/, [Online; accessed 2017-12-09].

[31] "Sytemtap dtrace comparison," https://sourceware.org/systemtap/wiki/SystemtapDtraceComparison, [Online; accessed 2018-03-25].

[32] "perf : the good, the bad, the ugly,," http://rhaas.blogspot.ca/2012/06/perfgood-bad-ugly.html, [Online; accessed 2018-03-25].

[33] "Visualvm : Profiling applications,," http://visualvm.java.net/profiler.html, [Online; accessed 2018-10-12].

[34] A. Spear, M. Levy, and M. Desnoyers, "Using tracing to solve the multicore system debug problem," *Computer*, vol. 45, pp. 60–64, 12 2012.

[35] "In-depth: Using Chrome://tracing to view your inline profiling data," https://www.gamasutra.com/view/news/176420/Indepth_Using_Chrometracing_to_view_your_inline_profiling_data.php, [Online; accessed 2018-01-23].

[36] "Trace Event Format," https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview#, [Online; accessed 2018-03-07].

[37] N. Ezzati-Jivan and M. R. Dagenais, "A stateful approach to generate synthetic events from kernel traces," *Adv. Soft. Eng.*, vol. 2012, pp. 6:6–6:6, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/140368

[38] "Trace Compass," https://projects.eclipse.org/projects/tools.tracecompass,, [Online; accessed 2017-09-12].

[39] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais, "State history tree: An incremental disk-based data structure for very large interval data," in *2013 International Conference on Social Computing*, Sep. 2013, pp. 716–724.

[40] I. De Melo Junior, "Software tracing comparison using data mining techniques," Master's thesis, École Polytechnique de Montréal, 2017.

[41] K. Kumar, P. Rajiv, G. Laxmi, and N. Bhuyan, "Shuffling: A framework for lock contention aware thread scheduling for multicore multiprocessor systems," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014, pp. 289–300.

[42] U. B. Nisar, M. Aleem, M. A. Iqbal, and N. Vo, "Jumbler: A lock-contention aware thread scheduler for multi-core parallel machines," in *2017 International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom)*, Jan 2017, pp. 77–81.

[43] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," *SIGPLAN Not.*, vol. 45, no. 5, pp. 269–280, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693489

[44] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan, "Adapt: A framework for coscheduling multithreaded programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 45:1–45:24, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400704

[45] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *Proceedings of the 2012 USENIX Conference on Annual Technical*

*Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342827

[46] "Critical path method," https://en.wikipedia.org/wiki/Critical_path_method, [Online; accessed 2018-04-21].

[47] "Multithreading (computer architecture)," https://en.wikipedia.org/wiki/Multithreading_(computer_architecture), [Online; accessed 2018-05-12].

[48] M. Broberg, L. Lundberg, and H. Grahn, "Performance optimization using extended critical path analysis in multithreaded programs on multiprocessors," *J. Parallel Distrib. Comput.*, vol. 61, no. 1, pp. 115–136, Jan. 2001. [Online]. Available: http://dx.doi.org/10.1006/jpdc.2000.1667

[49] A. Saidi, N. Binkert, S. Reinhardt, and T. Mudge, "Full-system critical path analysis," 04 2008, pp. 63–74.

[50] C. . Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *[1988] Proceedings. The 8th International Conference on Distributed*, June 1988, pp. 366–373.

[51] J. K. Hollingsworth, "An online computation of critical path profiling," in *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ser. SPDT '96. New York, NY, USA: ACM, 1996, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/238020.238024

[52] P.-M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 77–87, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773932

[53] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, March 2009.

[54] G. Franks, D. Lau, and C. Hrischuk, "Performance measurements and modeling of a java-based session initiation protocol (sip) application server," 01 2011, pp. 63–72.

[55] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside, "Automatic generation of layered queuing software performance models from commonly available traces," in *Proceedings of the 5th International Workshop on Software and Performance*, ser.

WOSP '05. New York, NY, USA: ACM, 2005, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1071021.1071037

[56] F. Ricciato and W. Fleischer, "Bottleneck detection via aggregate rate analysis : A real case in a 3g network," 01 2006.

[57] "Understanding about:tracing results," https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/trace-event-reading, [Online; accessed 2017-12-25].

[58] "Profiling vs. Tracing," http://ipm-hpc.sourceforge.net/profilingvstracing.html, 2017, [Online; accessed 2018-03-05].

[59] A. Spear, M. Levy, and M. Desnoyers, "Using tracing to solve the multicore system debug problem," *Computer*, vol. 45, pp. 60–64, 12 2012.

[60] M. Kosaka, "Inside look at modern web browser," https://developers.google.com/web/updates/2018/09/inside-browser-part1#browser-architecture,, 2018, [Online; accessed 2019-08-15].

[61] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 26:1–26:33, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3158644

[62] F. E. M. H. J. K. V. Prasad, W. Cohen and B. Chen, "Locating system problems using dynamic instrumentation," *Ottawa Linux Symposium*, pp. 49–64, 07 2005.

[63] F. C. Eigler, "Problem solving with systemtap," *Ottawa Linux Symposium*, pp. 261–268, 2006.

[64] J. Kilbury, K. Bontcheva, and Y. Samih, "Ftrace: A tool for finite-state morphology," in *Proceedings of the 9th International Workshop on Finite State Methods and Natural Language Processing*, ser. FSMNLP '11. Stroudsburg, PA, USA: Association for Computational Linguistics, 2011, pp. 88–92. [Online]. Available: http://dl.acm.org/citation.cfm?id=2206310.2206322

[65] "A thorough introduction to ebpf," https://lwn.net/Articles/740157/,, 2017, [Online; accessed 2017-12-02].

[66] M. Desnoyers and M. R. Dagenais, "Lockless multi-core high-throughput buffering scheme for kernel tracing," *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 3, pp. 65–81, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2421648.2421659

[67] "Common trace format, a flexible, high-performance binary trace format," https://diamon.org/ctf/, 2019, [Online; accessed 2019-04-02].

[68] J. Desfossez, M. Desnoyers, and M. R. Dagenais, "Runtime latency detection and analysis," *Softw. Pract. Exper.*, vol. 46, no. 10, pp. 1397–1409, Oct. 2016. [Online]. Available: https://doi.org/10.1002/spe.2389

[69] "The chromium projects," https://www.chromium.org/Home, 2019, [Online; accessed 2017-12-02].

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

[71] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais, "State history tree: An incremental disk-based data structure for very large interval data," in *2013 International Conference on Social Computing*, Sep. 2013, pp. 716–724.

[72] M. C. Rinard, "Analysis of multithreaded programs," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01. London, UK, UK: Springer-Verlag, 2001, pp. 1–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=647170.718289

[73] A. Marowka, "On performance analysis of a multithreaded application parallelized by different programming models using intel vtune," in *Proceedings of the 11th International Conference on Parallel Computing Technologies*, ser. PaCT'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 317–331. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041072.2041103

[74] M. Kessis and J.-M. Vincent, "Performance monitoring and visualization of large-sized and multi-threaded applications with the paje framework," 09 2006, pp. 34 – 34.

[75] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," 01 2010, pp. 133–142.

[76] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 215–228. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966465

[77] "Common trace format, a flexible, high-performance binary trace format," https://chromium.googlesource.com/chromium/src/+/master/docs/threading_and_tasks.md, 2018, [Online; accessed 2018-02-07].

[78] "Processes, Threads, and Apartments," https://docs.microsoft.com/en-us/windows/win32/com/processes--threads--and-apartments, 2019, [Online; accessed 2019-10-20].

[79] M. Kosaka, "Inside look at modern web browser2," https://developers.google.com/web/updates/2018/09/inside-browser-part2,, 2018, [Online; accessed 2019-08-15].

[80] R. Willis, "Critical path analysis and resource constrained project scheduling — theory and practice," *European Journal of Operational Research*, vol. 21, no. 2, pp. 149 – 155, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0377221785900268

[81] "Issue 125264: GpuChannelHost::Send shouldn't block UI thread," https://bugs.chromium.org/p/chromium/issues/detail?id=125264,, 2018, [Online; accessed 2018-04-26].

[82] "Issue 892747: Jank detected in ViewHostMsg::ClosePage," https://bugs.chromium.org/p/chromium/issues/detail?id=892747,, 2018, [Online; accessed 2018-10-05].

[83] T. E. team, "LTTng session rotation: the practical way to do continuous tracing," https://lttng.org/blog/2018/03/21/lttng-session-rotation/,, 2018, [Online; accessed 2018-03-21].

[84] W. Fadel, "Techniques for the abstraction of system call traces to facilitate the understanding of the behavioural aspects of the linux kernel," Master's thesis, Concordia University, November 2010. [Online]. Available: https://spectrum.library.concordia.ca/7075/

[85] H. Waly, "A complete framework for kernel trace analysis," Master's thesis, Laval University, 2011.

[86] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering architectures from running systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 454–466, July 2006.

[87] C. LaRosa, L. Xiong, and K. Mandelberg, "Frequent pattern mining for kernel trace data," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC 08. New York, NY, USA: ACM, 2008, pp. 880–885.

[88] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *2009 Ninth IEEE International Conference on Data Mining*, Dec 2009, pp. 588–597.

[89] M. Meyer and L. Wendehals, "Selective tracing for dynamic analyses," in *Proc. of the 1st Workshop on Program Comprehension through Dynamic Analysis (PCODA), co-located with the 12th WCRE, Pittsburgh, Pennsylvania, USA*, 2005.

[90] (2012) All about ld_preload. [Online]. Available: https://blog.fpmurphy.com/2012/09/all-about-ld_preload.html

[91] F. Reumont-Locke, N. Ezzati-Jivan, and M. R. Dagenais, "Efficient methods for trace analysis parallelization," *International Journal of Parallel Programming*, Feb 2019. [Online]. Available: https://doi.org/10.1007/s10766-019-00631-4