Titre: Title:	Multi-Level Trace Abstraction, Linking and Display
Auteur: Author:	Naser Ezzati Jivan
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ezzati Jivan, N. (2014). Multi-Level Trace Abstraction, Linking and Display [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1402/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1402/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

MULTI-LEVEL TRACE ABSTRACTION, LINKING AND DISPLAY

NASER EZZATI JIVAN
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

MULTI-LEVEL TRACE ABSTRACTION, LINKING AND DISPLAY

présentée par : EZZATI JIVAN Naser
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. ANTONIOL Giuliano, Ph.D., président
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. GAGNON Michel, Ph.D., membre
M. KHENDEK Ferhat, Ph.D., membre

I would like to dedicate this thesis to my wife Leila, for all of her constant love, support and inspiration.

ACKNOWLEDGEMENTS

I would like to thank all those who have supported me during my doctoral studies. I am particularly grateful to my supervisor, Professor Michel Dagenais, not only for his sincere support during my studies but also for his continuous encouragement and understanding. It was a great honor to work with such a wonderful supervisor. Without his precious advice, guidance and persistent help, this thesis would not have been possible.

My time at Ecole Polytechnique de Montreal has been an enjoyable, memorable and erudite journey. I have learned so much here that will help me in my future life. I would like to praise the Ecole Polytechnique de Montreal and the department of Computer and Software Engineering for giving me the opportunity of studying and working here.

I also acknowledge all my committee members : Professor Giuliano Antoniol, Professor Michel Gagnon, and Professor Ferhat Khendek for sparing their precious time in order to review and evaluate my research work.

Also I would like to thank my colleagues at DORSAL laboratory in the department of Computer and Software Engineering, for their thoughtful and interesting comments on my research. Their kindness helped make this journey so much more enjoyable and unforgettable.

Last but not the least ; I want to thank my parents, for providing me the opportunity to follow the road that lead me to my scientific studies and also for all of their years of encouragement, support, patience and love.

And finally, my deepest and warmest thanks to Leila, my beloved wife, for dreaming with me, for being a constant source of encouragement and inspiration, for being tolerating and patient when I was busy and not available for her, and for providing me love, support and inspiration. I can't even in imagination pay the immense debt of gratitude I owe her.

RÉSUMÉ

Certains types de problèmes logiciels et de bogues ne peuvent être identifiés et résolus que lors de l'analyse de l'exécution des applications. L'analyse d'exécution (et le débogage) des systèmes parallèles et distribués est très difficile en utilisant uniquement le code source et les autres artefacts logiciels statiques. L'analyse dynamique par trace d'exécution est de plus en plus utilisée pour étudier le comportement d'un système. Les traces d'exécution contiennent généralement une grande quantité d'information sur l'exécution du système, par exemple quel processus/module interagit avec quels autres processus/modules, ou encore quel fichier est touché par celui-ci, et ainsi de suite. Les traces au niveau du système d'exploitation sont un des types de données des plus utiles et efficaces qui peuvent être utilisés pour détecter des problèmes d'exécution complexes. En effet, ils contiennent généralement des informations détaillées sur la communication inter-processus, sur l'utilisation de la mémoire, le système de fichiers, les appels système, la couche réseau, les blocs de disque, etc. Cette information peut être utilisée pour raisonner sur le comportement d'exécution du système et investiguer les bogues ainsi que les problèmes d'exécution.

D'un autre côté, les traces d'exécution peuvent rapidement avoir une très grande taille en peu de temps à cause de la grande quantité d'information qu'elles contiennent. De plus, les traces contiennent généralement des données de bas niveau (appels système, interruptions, etc) pour lesquelles l'analyse et la compréhension du contexte requièrent des connaissances poussées dans le domaine des systèmes d'exploitation. Très souvent, les administrateurs système et analystes préfèrent des données de plus haut niveau pour avoir une idée plus générale du comportement du système, contrairement aux traces noyau dont le niveau d'abstraction est très bas. Pour pouvoir générer efficacement des analyses de plus haut niveau, il est nécessaire de développer des algorithmes et des outils efficaces pour analyser les traces noyau et mettre en évidence les évènements les plus pertinents.

Le caractère expressif des événements de trace permet aux analystes de raisonner sur l'exécution du système à des niveaux plus élevés, pour découvrir le sens de l'exécution en différents endroits, et détecter les comportements problématiques et inattendus. Toutefois, pour permettre une telle analyse, un outil de visualisation supplémentaire est nécessaire pour afficher les événements abstraits à de plus hauts niveaux d'abstraction. Cet outil peut permettre aux utilisateurs de voir une liste des problèmes détectés, de les suivre dans les couches de plus bas niveau (ex. directement dans la trace détaillée) et éventuellement de découvrir les raisons des problèmes détectés.

Dans cette thèse, un cadre d'application est présenté pour relever ces défis : réduire la taille d'une trace ainsi que sa complexité, générer plusieurs niveaux d'événements abstraits pour les organiser d'une façon hiérarchique et les visualiser à de multiples niveaux, et enfin permettre aux utilisateurs d'effectuer une analyse verticale et à plusieurs niveaux d'abstraction, conduisant à une meilleure connaissance et compréhension de l'exécution du système.

Le cadre d'application proposé est étudié en deux grandes parties : d'abord plusieurs niveaux d'abstraction de trace, et ensuite l'organisation de la trace à plusieurs niveaux et sa visualisation. La première partie traite des techniques utilisées pour les données de trace abstraites en utilisant soit les traces des événements contenus, un ensemble prédéfini de paramètres et de mesures, ou une structure basée l'abstraction pour extraire les ressources impliquées dans le système. La deuxième partie, en revanche, indique l'organisation hiérarchique des événements abstraits générés, l'établissement de liens entre les événements connexes, et enfin la visualisation en utilisant une vue de la chronologie avec échelle ajustable. Cette vue affiche les événements à différents niveaux de granularité, et permet une navigation hiérarchique à travers différentes couches d'événements de trace, en soutenant la mise à l'échelle sémantique. Grâce à cet outil, les utilisateurs peuvent tout d'abord avoir un aperçu de l'exécution, contenant un ensemble de comportements de haut niveau, puis peuvent se déplacer dans la vue et se concentrer sur une zone d'intérêt pour obtenir plus de détails sur celle-ci.

L'outil proposé synchronise et coordonne les différents niveaux de la vue en établissant des liens entre les données, structurellement ou sémantiquement. La liaison structurelle utilise la délimitation par estampilles de temps des événements pour lier les données, tandis que le second utilise une pré-analyse des événements de trace pour trouver la pertinence entre eux ainsi que pour les lier. Lier les événements relie les informations de différentes couches qui appartiennent théoriquement à la même procédure ou spécifient le même comportement. Avec l'utilisation de la liaison de la correspondance, des événements dans une couche peuvent être analysés par rapport aux événements et aux informations disponibles dans d'autres couches, ce qui conduit à une analyse à plusieurs niveaux et la compréhension de l'exécution du système sous-jacent.

Les exemples et les résultats expérimentaux des techniques d'abstraction et de visualisation proposés sont présentés dans cette thèse qui prouve l'efficacité de l'approche. Dans ce projet, toutes les évaluations et les expériences ont été menées sur la base des événements de trace au niveau du système d'exploitation recueillies par le traceur Linux Trace Toolkit Next Generation (LTTng). LTTng est un outil libre, léger et à faible impact qui fournit des informations d'exécution détaillées à partir des différents modules du système sous-jacent, tant au niveau du noyau Linux que de l'espace utilisateur. LTTng fonctionne en instrumentant le noyau et les applications utilisateur en insérant quelques points de traces à des endroits différents.

ABSTRACT

Some problems and bugs can only be identified and resolved using runtime application behavior analysis. Runtime analysis of multi-threaded and distributed systems is very difficult, almost impossible, by only analyzing the source code and other static software artifacts. Therefore, dynamic analysis through execution traces is increasingly used to study system runtime behavior. Execution traces usually contain large amounts of valuable information about the system execution, e.g., which process/module interacts with which other processes/modules, which file is touched by which process/module, which function is called by which process/module/function and so on. Operating system level traces are among the most useful and effective information sources that can be used to detect complex bugs and problems. Indeed, they contain detailed information about inter-process communication, memory usage, file system, system calls, networking, disk blocks, etc. This information can be used to understand the system runtime behavior, and to identify a large class of bugs, problems, and misbehavior.

However, execution traces may become large, even within a few seconds or minutes of execution, making the analysis difficult. Moreover, traces are often filled with low-level data (system calls, interrupts, etc.) so that people need a complete understanding of the domain knowledge to analyze these data.

It is often preferable for analysts to look at relatively abstract and high-level events, which are more readable and representative than the original trace data, and reveal the same behavior but at higher levels of granularity. However, to achieve such high-level data, effective algorithms and tools must be developed to process trace events, remove less important ones, highlight only necessary data, generalize trace events, and finally aggregate and group similar and related events.

The expressive nature of the synthetic events allows analysts to reason about system execution at higher levels, to uncover execution behavior in different areas, and detect its problematic and unexpected aspects. However, to allow such analysis, an additional visualization tool may be required to display abstract events at different levels and explore them easily. This tool may enable users to see a list of detected problems, follow the problems in the detailed levels (e.g., within the raw trace events), analyze, and possibly discover the reasons for the detected problems.

In this thesis, a framework is presented to address those challenges: to reduce the execution trace size and complexity, to generate several levels of abstract events, to organize the data in a hierarchy, to visualize them at multiple levels, and finally to enable users to perform a top-down and multiscale analysis over trace data, leading to a better understanding and comprehension of underlying system execution.

The proposed framework is studied in two major parts: multi-level trace abstraction, and multi-level trace organization and visualization. The first part discusses the techniques used to abstract out trace data using either the trace events content, a predefined set of metrics and measures, or structure-based abstraction to extract the resources involved in the system execution. The second part determines the hierarchical organization of the generated abstract events, establishes links between the related events, and finally visualizes events using a zoomable timeline view. This view displays the events at different granularity levels, and enables a hierarchical navigation through different layers of trace events by supporting the semantic zooming. Using this tool, users can first see an overview of the execution, and then can pan around the view, and focus and zoom on any area of interest for more details and insight.

The proposed view synchronizes and coordinates the different view levels by establishing links between data, structurally or semantically. The structural linking uses bounding timestamps of the events to link the data, while the latter uses a pre-analysis of the trace events to find their relevance, and to link them together. Establishing Links connects the information that are conceptually related together (e.g., events belong to the same process or specify the same behavior), so that events in one layer can be analyzed with respect to events and information in other layers, leading to a multi-level analysis and a better comprehension of the underlying system execution.

In this project, all evaluations and experiments were conducted on operating system level traces obtained with the Linux Trace Toolkit Next Generation (LTTng). LTTng is a lightweight and low-impact open source tracing tool that provides valuable runtime information from the various modules of the underlying system at both the Linux kernel and user-space levels. LTTng works by instrumenting the (kernel and user-space) applications, with statically inserted tracepoints, and by generating log entries at runtime each time a tracepoint is hit.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SIGNS AND ABBREVIATIONS

FSM	Finite State Machine (FSM)
IDS	Intrusion Detection System
LTTng	Linux Trace Toolkit Next Generation
LTTV	Linux Trace Toolkit Viewer
TMF	Trace Monitoring Framework
OS	Operating System
SHD	State History Database
UST	User-Space Tracer
DNS	Domain Name System
DoS	Denial of Service
OLAP	Online Analytical Processing
CPN	Colored Petri Nets
DCA	Directed Acyclic Graphs
GD	Granularity Degree
TBSAM	Tree-based Statistics Access Method

CHAPTER 1

INTRODUCTION

1.1 Introduction

Software debugging and profiling require an understanding of how programs behave at runtime. Program comprehension is an important step for both software forward and reverse engineering, which can facilitate software optimization, maintenance, bug fixing, and performance analysis [28]. However, program analysis and comprehension has become a difficult and challenging task in recent years, due to the appearance of complex multi-threaded and distributed systems and applications.

Static properties (e.g., source code, documentation and other artifacts) of a software can be used to predict its runtime behavior and comprehend its execution. However, they may be of limited use, for instance in distributed systems, or operating system level applications, where a set of modules is running concurrently and interacting through message passing. For a simple example, consider the late/dynamic binding in C++ or JAVA, in which the real execution flow is decided in runtime. It then becomes difficult to predict the execution flow of complex systems by only inspecting the source code.

Dynamic analysis, on the other hand, is better suited to analyze system runtime behavior [15, 86]. Dynamic analysis is the inspection of the program's actions and messages, typically in logs gathered by program tracing. Program tracing is the process that instruments some parts of a program binary or source code to insert hooks and generate trace logs while the program is running [28]. Tracing can produce precise and detailed information from various system levels, from the hardware and operating system level [37] to high architectural levels [125].

Although dynamic analysis through trace data can be used to analyze a program's runtime behavior and comprehend its execution, this can be challenging. Because, tracing an operating system or a software module at a detailed level, even for a short execution period, may generate very large traces. Therefore, the main challenge of dynamic analysis is dealing with the huge amount of data and finding solutions to analyze huge volumes of execution trace data to understand and comprehend the runtime behavior of software programs with minimal performance impact.

Several techniques are proposed in the related literature to cope with this challenge : e.g., reducing the trace size and abstracting the data by compressing the trace events [67, 78], filtering out useless information [56], generalizing the data [120], aggregating trace events and generating compound events [56, 99, 142], and finally mining trace data and extracting some interesting patterns (e.g., suspicious activities, resource overuses, etc.) [86]. Visualization is another approach, used in combination with these techniques, to facilitate the dynamic analysis and runtime behavior comprehension [8, 34, 122].

Even with trace abstraction techniques and interactive visualization, the resulting information may still be too large for effective trace analysis and system comprehension. An interesting technique to alleviate this problem is to organize, visualize, and analyze the information at multiple levels of detail[127, 136]. This way, multi-level visualization can enable a top-down/bottom-up analysis by displaying an overview of the data first and letting users go back and forth, as well as up and down, for any areas of interest, to focus, dig into and get more detailed information [107, 126].

Of the many different trace abstraction and visualization techniques proposed in the literature, only a few of them support operating system (kernel) level trace data. Kernel traces provide valuable information about system execution at different levels and from different points of view, which can be very helpful to analyze system execution, debug, and find a large class of system problems (e.g., excessive process/thread migration, non-optimal cache utilization, repeated writes of small data to disk, brute force requests, security problems and so on). Nonetheless, these traces have some specific features and need additional processing. For example, there is a discontinuity between the events from a single process, because of the CPU scheduling policies. Another difference is that kernel traces usually contain events from different modules (disk blocks, memory, file system, processes, interrupts, etc.), which may complicate the analysis.

The main focus of this thesis is on the efforts to facilitate the analysis of operating system execution traces, and to enhance the comprehension of application execution. This includes the development of a scalable multi-level trace abstraction and visualization framework, with support for various analysis techniques targeting trace size and complexity reduction, and for visualization of the reduced traces at multiple levels of detail.

1.2 Definition of Execution Trace and Abstract Events

We define an execution trace as a set of punctual events e_i. Each event contains a timestamp t_i which indicates the time when the event occurred, CPU number, channel name (group to which the event belongs : kernel, file system, memory management, disk block, etc.), event name, corresponding thread name or identification, parent process name or identification, execution mode in which the event was executed (user mode, system call, interrupt, etc.), and finally the event parameters including file name, network ip and port, etc.

An execution trace may contain several threads : each trace is a mixed set of events from different threads, in which there is a partially ordered set of events for each thread (based on the event timestamp) : if $t_i < t_j$ then $e_{t_i} < e_{t_j}$.

Abstract events are in turn defined as events generated by applying one or more trace analysis (i.e., abstraction) techniques over the original execution trace. Each abstract event contains a name, key (a running process or a combination of processes), boundary information (start t_s and end t_e), and possible event parameters.

Among the different tracing tools used to instrument the applications at user and operating system (kernel) levels (LTTng [38], Dtrace [16], SystemTap [46]), our thesis focuses mainly on the Linux kernel-level trace data gathered by the Linux Trace Toolkit Next Generation (LTTng) [38] tracer. LTTng [38] is a low impact [133], lightweight, open source Linux tracing tool that provides useful information about different kernel operations :

- Running processes, their execution name, ID, parent and children ;
- CPU states and scheduling events, used to reason about scheduling algorithms and CPU utilization ;
- File operations like open, read, write, seek, and close, used to reason about file system operations and IO throughput ;
- Disk level operations, used to gather statistics about disk access latencies ;
- Network operations and the details of network packets, used to analyze network IO throughput and network faults and attacks ;
- Memory management information like allocating or releasing a page, used to analyze memory usage.

Kernel traces have specific features that make trace analysis a difficult and challenging task.

Challenges of Kernel Trace Analysis

The first issue facing kernel trace analysis is the discontinuous nature of the operating system execution. Sometimes, a process is preempted in the middle of execution and another task is executed instead. This process may be resumed at a later time and continued using the same or a different CPU, but different separate sets of logs are generated for each CPU [48].

A second challenge is that only analyzing the events of a particular process may not be enough. For instance, when an attacker gains control of the Apache web server and creates a remote shell, all events of the remote shell will be recorded as a new process (e.g., /ls/bash) instead of the Apache process. Therefore, analyzing only the Apache process is not sufficient to detect the attack or to determine what the attacker is doing. In essence, the control may be passed from one process to another, and this information is required to reach an extensive comprehension [48, 63].

The third challenge is the existence of very detailed low-level data (i.e, interrupts, memory page, page faults, scheduling, timers, disk block operations, etc.) in the kernel trace. This complicates the analysis because understanding such data requires a deep knowledge of the operating system. It may also makes the analysis techniques and tools dependent upon a specific version of the operating system or trace format [49, 63].

The fourth and final challenge, not only with kernel trace data, is the size of trace data [56]. The trace logs can grow very rapidly, complicating the analysis, and threatening the scalability of the tools. Some techniques should be applied to reduce the trace size before conducting further analysis and visualization.

1.3 Objective

Tracing complete systems provides detailed data at several levels : operating system, virtual machine, and user space. This data is in the form of raw traced events and contains low-level information about the execution of different system modules. The first difficulty arises in how to deduce several synthetic events (e.g., high-level problems) from a set of raw trace events that represents the underlying system execution.

Many elaborate analysis modules may provide different levels of synthetic events. However, the relationship between the new synthetic events and the underlying detailed events may not be direct. For instance, a high concentration of network requests may indicate a cyber-attack ; no isolated network request is necessarily part of the attack, but together they form a strong symptom of a possible attack. Considering these modules that can generate different abstract events at different levels of granularity, a common underlying recurring need can be identified : the user must be able to navigate through the base raw events, as well as the new information generated by the analysis modules, through linked and coordinated views.

In summary, the challenges can be defined at three levels : conceptual (how to generate different levels of abstract events, modeling them and their links in a uniform way, covering different dimensions and levels) ; algorithmic (how to efficiently store and access these events and links, considering the large number of events involved) ; and ergonomic (how to present a simple and effective visualization of these events and links).

1.4 Contributions

Based on the architectural view shown in Figure 1.1, the contributions of this thesis belong in the two following main categories : (1) multi-level trace abstraction (2) multi-level trace visualization.

1.4.1 Multi-level Trace Abstraction

In this part, we propose different trace abstraction techniques, including data-driven trace abstraction, metric-driven trace abstraction, and structure-based trace abstraction. The main contributions in this category can be summarized as follows :

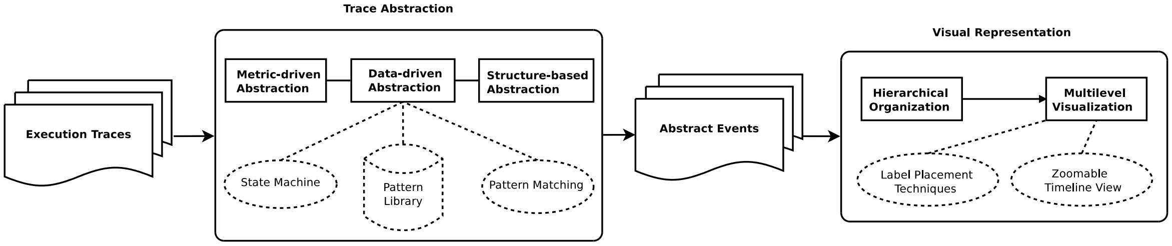


Figure 1.1 Architectural view of the proposed work.

- Presenting a stateful data-driven trace abstraction technique that uses different approaches like generalization, aggregation and filtering to construct different levels of high-level compound events. A pattern library, including the execution model of various low-level and high-level Linux operations (e.g., file, socket, process operations, fork bomb attack, frequent port scanning, etc.) is developed and used in this technique to extract and recognize specific system behaviors through pattern matching [48, 49] (Chapter 3).
- Proposing a metric-driven trace abstraction technique to extract statistical information from the traces of the underlying program execution. This technique aggregates the trace events based on a given set of metrics and measures the requested metrics and parameters. It also discusses the models and methods to organize, store and access the generated statistical information [50] (Chapter 4).
- Proposing a structure-based trace abstraction method that extracts and organizes the system resources (e.g., processes, files, etc.) involved in the execution of the underlying program. This organization is used later in the visualization phase to present (abstract) trace events in a proper way [50, 51] (Chapter 4).
- Introducing an architecture as well as corresponding data structures and algorithms to summarize trace streams and organize them hierarchically at different levels of granula-

rity, to conduct a multi-level and multi-dimensional analysis, similar to OLAP (Online Analytical Processing) analysis. The proposed solution aggregates the input streams and records them in a compact form at multiple granularity levels, enabling efficient access to data for any arbitrary time interval. It identifies and solves the problem of dealing with large volumes of timestamped trace data, primarily by modifying the granularity of stored data as it ages. Unlike previous contributions used to analyze offline trace data, this approach works efficiently with streaming traces (live trace events), traces of theoretically unlimited size [51] (Chapter 5).

1.4.2 Multi-level Trace Visualization

This part mainly discusses the techniques that organize and visualize execution traces at different levels. Establishing links between trace events is also discussed in the following contributions :

- Proposing a collection of heuristics to place labels along multiple parallel lines with designated points or line-segments. This label placement technique is used in the visualization phase to place readable and legible labels on only a subsection of the selected visible events (e.g., those events that have specific features, and thus are important). Unlike cartography and graph labeling, the problem here is notably different because the two dimensions of the plane are used asymmetrically : label offsets along the x axis are more acceptable than label offsets that cross a lane boundary (in Y). This difference distinguishes this work from most prior research, and makes the needed algorithms distinct from graph and cartographic labeling [52, 55](Chapter 6).
- Introducing a zoomable timeline view to visualize the trace data at multiple levels of granularity. The view integrates different abstraction levels in a single display and provides useful navigation mechanisms (i.e., drill-down and standard and semantic zooming) to explore the trace data at various granularity levels. The proposed method uses a set of heuristics to increase the speed of labeling, the quantity of labeled items and the quality of the labels [47, 53] (Chapter 7).
- Presenting a data model to hierarchically organize both the abstract trace events at different levels of granularity and also the relationships between them. This is used as the background data store for the zoomable timeline view. The hierarchical organization of the trace data already describes a hierarchical relation. However, it is sometimes required to model and display other (non-trivial) links and relations between data elements, to enable following and digging into a special high-level behavior of the underlying system. This data organization technique enables a top-down navigation of trace events, resulting in a better comprehension of the underlying system [53] (Chapter 7).

1.5 Thesis Outline

In addition to the survey of the background work in Chapter 2, other parts of this thesis are presented as five journal publications (research papers) which are included in Chapters 3, 4, 5, 6 and 7. The chapters are organized as follows.

Chapter 2 presents a detailed review of the related literature and research, and is organized as a survey paper [54]. It provides a taxonomy of different trace abstraction techniques and their applications. This chapter also presents various hierarchical organizations as well as multi-level visualization techniques for large trace data. A detailed discussion and a comparison of the methods are also included in this chapter.

Chapters 3, 4, and 5 describe different trace abstraction methods. Chapter 3 discusses several ways to generate different levels of abstract events using data-driven techniques. It also describes a pattern library constructed for this method, containing a set of low-level patterns of ordinary Linux operations (e.g., file operations, socket operations, etc.) as well as some high-level behavioral scenarios (e.g., brute force attacks). The proposed abstraction techniques exploit a stateful approach to share the common states and information among the concurrent patterns, resulting in a better computation time and efficient memory usage.

Chapter 4 illustrates a metric-driven trace abstraction approach. It describes the way to extract statistical information from raw trace data, using a predefined mapping table or a pattern library. It then discusses an organization of the extracted statistical information to support efficient point/range queries, for any arbitrary time interval, in the trace analysis phase. This chapter also proposes a structure (resource) based trace abstraction technique that focuses mainly on extracting and organizing system resources involved in a trace file. The resource organization may help later to easily query and access(e.g., filter out) the events of a specific resource or to compare utilization of the different resources.

Chapter 5 proposes a solution for the same problems in chapter 4, but in the context of live trace streams that are of unlimited size. This chapter illustrates how to store intermediate snapshots of the statistical information in a tree-based data cube structure to support OLAP (Online Analytical Processing) style analysis of the trace data. The proposed solution supports static and dynamic time window queries to extract the aggregated information of the system for any duration in the past, or for any duration between a time in the past and the current time. This technique can be used to monitor and control critical systems and applications.

Chapters 6 and 7 focus on the techniques required for the visualization of the generated abstract events. Chapter 6 discusses an efficient way to place labels on the trace events represented in the different lanes on the screen. It discusses they ways to efficiently select a set of master events, from the initial candidate events in order to place and display readable and unambiguous labels in the display screen.

Chapter 7 introduces a new visualization technique, called zoomable timeline view, to visualize trace events at multiple levels of granularity. This view first displays an overview of the trace (i.e., the highest abstract level of events) and enables users to move around in the trace, zoom or focus on any interesting area. It supports both standard zooming to enlarge the visible objects of the display, and semantic zooming to display more events and information (another level of trace data) when zooming on any selected area. This technique includes two linking mechanisms between objects at different levels to enable drilling down/rolling up operations : boundary-based links using event timestamps to synchronize the different level views, and content-driven links using preset relationships among data. Establishing links has direct applications in root-cause analysis to follow and dig into a selected problem within the related individual trace events and data.

Finally, chapters 8 and 9 discuss the techniques proposed in previous chapters, the general results, and possible avenues for future work.

1.6 Publications

Many chapters outlined above are based on the published/submitted papers mentioned in this section. The publications are divided into two sections : papers containing the main contributions and included in the thesis, and secondary papers mostly presented in conferences, which provide more further viewpoints on the work and sometimes involve other researchers as co-authors. The survey paper presented in Chapter 2 is also listed in the secondary papers.

1.6.1 Main Journal Papers

1- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2012) ”A Stateful Approach to Generate Synthetic Events from Kernel Traces,” Advances in Software Engineering, vol. 2012, Article ID 140368, 12 pages, 2012. DOI=10.1155/2012/140368.

Presented in chapter 3 (Published).

2- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2013) ”A framework to compute statistics of system parameters from very large trace files,”. ACM SIGOPS Operating Systems Review 47, 1 (January 2013), 43-54. DOI=10.1145/2433140.2433151.

Presented in chapter 4 (Published).

3- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). ”Cube data model for multilevel statistics computation of live execution traces,” accepted in Concurrency and Computation :

Practice and Experience.

Presented in chapter 5 (Accepted).

4- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). ”Fast label placement algorithm for multilevel visualization of execution traces,” submitted to Computing.

Presented in chapter 6 (Submitted).

5- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). ”Multilevel visualization of large execution traces,” submitted to Journal of Visual Languages and Computing.

Presented in chapter 7 (Submitted).

1.6.2 Secondary Papers

1- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). ”Multiscale abstraction and visualization of large trace data : A survey,” submitted to The VLDB Journal.

Presented in chapter 2 (Submitted).

2- EZZATI-JIVAN, N. and DAGENAIS, M. R. (2012). ”An efficient analysis approach for multi-core system tracing data,” Proceedings of the 16th IASTED International Conference on Software Engineering and Applications (SEA 2012).

Contain additional information for chapter 3 (Published).

3- EZZATI-JIVAN, N. and ShAMELl-SENDI, A. and DAGENAIS, M. R. (2013) ”Multi-level label placement for execution trace events,”, 26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2013), vol., no., pp.1,6, 5-8 May 2013.

Contain additional information for chapter 6 (Published).

4- EZZATI-JIVAN, N. and DAGENAIS, M. (2014). ”Multiscale navigation in large trace data,” 27th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2014) .

Contain additional information for chapter 7 (Accepted).

CHAPTER 2

Paper 1 : Multilevel Abstraction and Visualization of Large Trace Data : A Survey

Naser Ezzati-Jivan and Michel Dagenais

2.1 Abstract

Dynamic analysis through execution traces is frequently used to analyze the runtime behaviour of software systems. However, the large size of traces makes the analysis, and understanding of systems, difficult and complex. Trace abstraction and visualization are potential solutions to alleviate this problem. Many analyses start with an inspection of an overview of the trace, before digging deeper and studying more focused and detailed data. These techniques are common and well supported in geographical information systems, automatically adjusting the level of details depending on the scale. However, most trace visualization tools operate at a single level of representation, which is not adequate to support multi-level analysis. Sophisticated techniques and heuristics are needed to address this problem. Multi-scale (multi-level) visualization with support for zoom and focus operations is an effective way to enable this kind of analysis. Considerable research and several surveys are proposed in the literature in the field of trace visualization. However, multi-scale visualization has yet received little attention. In this paper, we provide a survey of techniques aiming at multi-scale visualization of trace data, and discuss the requirements and challenges faced in order to meet evolving user demands.

2.2 Introduction

Software comprehension is the process of understanding of how a software program behaves. It is an important step for both software forward and reverse engineering, which facilitates software development, optimization, maintenance, bug fixing, as well as software performance analysis [28]. Software comprehension is usually achieved by using static or dynamic analysis [27].

Static analysis refers to the use of program source code and other software artifacts to understand the meaning and function of software modules and their interactions [137]. Although the software source codes and documents can be useful to understand the meaning of a program, there are situations where they are not very helpful. For instance, when the documents are outdated, they may not be very useful. Similarly, a rarely occurring timing related bug in a distributed system may be very difficult to diagnose by only examining the software source code and documents.

Dynamic analysis, on the other hand, is a runtime analysis solution emphasizing dynamic data (instead of static data) gathered from program execution. Dynamic analysis records and examines the program's actions, logs, messages, trace events, while it is being executed. Dynamic analysis is based on the program runtime behavior. This information is obtained by instrumenting the program's binary or source code and putting hooks at different places (e.g., entry and exit points of each function) [28, 67].

The use of dynamic analysis (through execution traces) to study system behavior is increasing among system administrators and analysts [86, 100, 145]. Tracing can produce precise and comprehensive information from various system levels, from (kernel) system calls [49, 56, 142] to high-level architectural levels [125], leading to the detection of more faults, (performance) problems, bugs and malwares than static analysis [44].

Although dynamic analysis is a useful method to analyze the runtime behavior of systems, this brings some formidable challenges. The first challenge is the size of trace logs. They can quickly become very large and make analysis difficult [68]. Tracing a software module or an operating system may generate very large trace logs (thousands of megabytes), even when run for only a few seconds. The second challenge is the low-level and system-dependent specificity of the trace data. Their comprehension thus requires a deep knowledge of the domain and system related tools [49].

In the literature, there are many techniques to cope with these problems : to reduce the trace size [27, 142], compress trace data [67, 78], decrease its complexity [99], filter out the useless and unwanted information [56] and generate high level generic information [48]. Visualization is another mechanism that can be used in combination with those techniques to reduce the complexity of the data, to facilitate analysis and thus to help for software understanding, debugging and profiling, performance analysis, attack detection, and highlighting misbehavior while associating it to specific software sub-modules (or source code) [8, 34, 122].

Even using trace abstraction and visualization techniques, the resulting information may still be large, and its analysis complex and difficult. An efficient technique to alleviate this problem is to organize and display information at different levels of detail and enable some hierarchical analysis and navigation mechanism to easily explore and investigate the data [127, 136]. This way, multi-scale (multi-level) visualization, can enable a top-down/bottom-up analysis by displaying an overview of the data first, and letting users go back and forth, as well as up and down (focus and zoom) in any area of interest, to dig deeper and get more detailed information [107, 126].

Of the many different trace visualization tools and techniques discussed in the literature [34, 137, 141, 150] and among those few interesting surveys on trace abstraction and visualization techniques [28, 68, 122], only a small fraction discusses and supports multi-level visualization. This motivated the current survey, to discuss and summarize the techniques used in multi-level visualization tools and interfaces (whether used for tracing, spatial tools, online maps, etc.) and the way to adapt those solutions to execution trace analysis tools.

Indeed, constructing an interactive scalable multi-level visualization tool, capable of analyzing and visualizing large traces and facilitating their comprehension, is a difficult and challenging task. It needs to address several issues. How to generate a hierarchy of abstract trace events ? How to organize them in a hierarchical manner, helping to understand the underlying system ? How to visualize and relate these events in various levels with support of appropriate LOD (Level of Details) techniques ?

Although the technique discussed here are rather generic and applicable for any tracing data, our focus will be more on operating system (kernel) level trace data, when this distinction is relevant. Kernel traces have some specific features that differentiate them from application (user) level traces. For example, unlike in user level tracing, there is a discontinuity between the execution of events for a process, because of preemption and CPU scheduling policies. Another example is that, unlike other (user-level) tracers that monitor only one specific module or process, kernel traces usually contain events from different modules (disk blocks, memory, file system, processes, interrupts, etc.), which may complicate the analysis.

The paper is structured as following. First, we discuss the techniques to generate multiple levels of trace events from the original logs, focusing on kernel trace data. Secondly, we present a taxonomy for multi-scale visualization methods targeting hierarchical data, looking at the existing trace visualization tools. Then, we study various solutions to model the hierarchical data. Finally, this paper will conclude with a summary and outline for future work. Figure 2.1 depicts the topics investigated in this paper.

2.3 Multi-level Trace Abstraction Techniques

As mentioned earlier, execution traces can be used to analyze system runtime data to understand its behavior and detect system bottlenecks, problems and misbehaviors [86, 145]. However, trace files can grow quickly to a huge size which makes the analysis difficult and cause a scalability problem. Therefore, special techniques are required to reduce the trace size and its complexity, and extract meaningful and useful information from original trace logs.

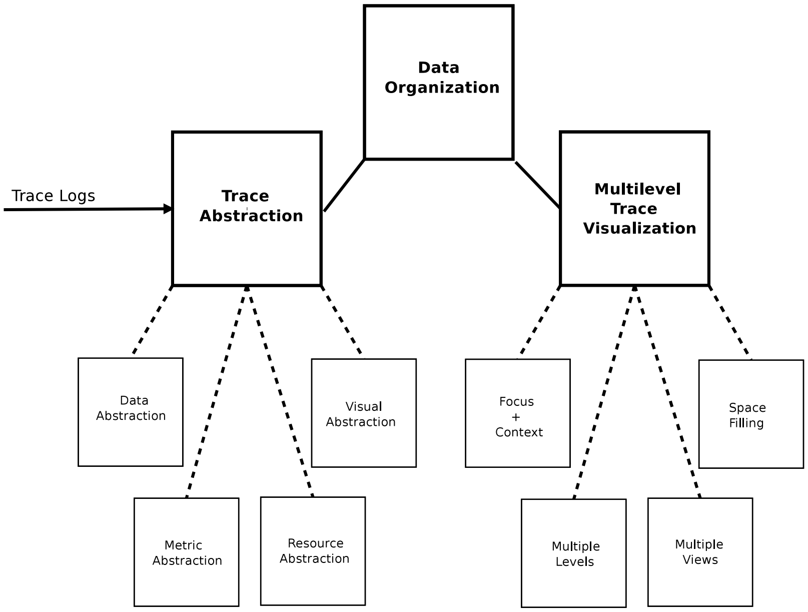


Figure 2.1 Taxonomy of topics discussed in this paper.

In the literature, various trace abstraction techniques are surveyed, including two recent systematic surveys [27, 28]. Here, we present a different taxonomy of trace abstraction techniques, based on their possible usages in a multi-level visualization tool.

We categorize trace abstraction techniques into three major categories : 1- Content-based (data-based) abstraction techniques, those techniques based on the content of events. 2- Visual abstraction, the techniques mostly used in the visualization steps. 3- Resource abstraction techniques, those techniques about extracting and organizing the resources involved within trace data.

2.3.1 Content-based (data-based) Abstraction

Using the events content to abstract out the trace data is called *content-based abstraction* or *data-based abstraction.* Data-based abstraction can be used to reduce the trace size and its complexity, generalize the data representation, group similar or related events to generate larger compound events, and aggregate traces based on some (predefined) metrics. In the following, we study all of these content-based abstraction techniques.

Trace Size Reduction

In the literature, various techniques have been developed to deal with the trace size problem : selective tracing, sampling, filtering, compression, generalization and aggregation.

Selective tracing [100] refers to tracing only some selected modules/processes of the system, instead of the whole system. Abstract execution [87] is one of the selective tracing techniques. It stores only a small set of execution trace events for later analysis. Once needed, for any area of interests, this technique re-generates a full trace data by re-executing the selected program module. Shimba [137], a trace visualization tool, also uses a selective tracing method.

Instead of processing all events, trace sampling selects and inspects only a variety of events from the trace data. Trace sampling is used in [19], [43], [59] and [117] and also in AVID visualization tool [141]. Since sampling filters trace events in an arbitrary manner, it may lose information and not preserve the actual system behavior.

Trace filtering is the removal of all redundant and unnecessary events from the trace events, highlighting the events that have some pre-specified importance. Filtering can be done based on various criteria such as event timestamp, event type, event arguments, function name, process name, class or package name, and also the priority and importance of events [29, 96]. Fadel et al. [56] use filtering in the context of kernel traces to remove uninteresting data (memory management events) and noise (page faults) from original data.

Trace compression is another technique used to reduce the trace size. It works by storing the trace events in a compact form by finding similarities and removing redundancies between them [67]. Compression has two common forms : lossy compression that may discard some parts of the source data (e.g., used in video and audio compression) and lossless compression that retains the exact source data (e.g., used in ZIP file format). Kaplan et al. [78] studied both lossy and lossless compression techniques for memory reference traces and proposed two methods to reduce the trace size by discarding useless information [78]. The drawback is that the compression technique cannot be used much for trace analysis purposes. Indeed, compression is more a storage reduction technique rather than an analysis simplification technique. Moreover, since trace compression is usually applied after generating trace events and storing them in memory or disk, it is not applicable for online trace analysis, when there is no real storage for all live trace events.

Event Generalization

Being dependent on a specific version of the operating system or particular version of tracer tool can be a weakness for the trace analysis tools. Generalization is one solution to this problem : the process of extracting common features from two or more events, and combining them into a generalized event [120]. Generalization can be used to convert the system related events into more generalized events. Especially in Linux, many system calls may have overlapping functionality. For example, the read, readv and pread64 system calls in

Linux may be used to read a file. However, from a higher level perspective, all of them can be seen as a single read operation. Another example is generalizing all file open/read/write/close events to a general ”file operation” event in the higher levels [49].

Event Grouping and Aggregation

Trace aggregation is one of the critical steps for enabling multi-scale analysis of trace events, because it can provide a high-level model of a program execution and ease its comprehension and debugging [92]. In the literature, it is a broadly used method to reduce the size and complexity of the input trace, and generate several levels of high level events [13, 48, 56, 58, 92, 142]. In essence, trace aggregation integrates sets of related events, participating in an operation, to form a set of compound and larger events, using pattern matching, pattern mining, pattern recognition and other techniques [58].

Using a set of successive aggregation functions, it is possible to create a hierarchy of abstract events, in which the highest level reveals more general behaviors, whereas the lowest level reveals more detailed information. The highest level can be built in a way that represents an overview of the whole trace. To generate such high level synthetic events, it is required to develop efficient tools and methods to read trace events, look for the sequences of similar and related events, group them, and generate high level expressive synthetic events [55, 58].

Fadel et al. [56] used pattern matching to aggregate kernel traces gathered by the LTTng Linux kernel tracer[1]. Since trace events usually contain entry and exit events (for function calls, system calls, or interrupts), it is then possible to find and match these events and group them to make aggregated events, using pattern matching techniques. For example, they form a ”file read” event by grouping the ”read system call” entry and exit events [56], and some possible file system events between these two (Figure 2.2). A similar technique is exploited by [44] to group function calls into logically related sets, but in user-space level.

Wally et al. [142] used trace grouping and aggregation techniques to detect system faults and anomalies from kernel traces. However, their focus was on creating a language for describing the aggregation patterns (i.e., attack scenarios). Both proposals [56, 142], although useful for many examples of trace aggregation and for applications to the fault identification field, do not offer the needed scalability to meet the demands of large trace sizes. Since they use disjoint patterns for aggregating the events, for large traces and for a large number of patterns, it will be a time-consuming task to take care all of patterns separately. Further optimization work is required in order to support large trace sizes and a more efficient pattern processing engine [48].

[1] http ://lttng.org/

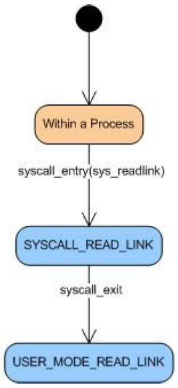


Figure 2.2 Recovering a ”file read” event from trace events.

Source [56]

Matni et al. [99] used an automata-based approach to detect faults like ”escaping a chroot jail” and ”SYN flood attacks”. They used a state machine language to describe the attack patterns. They initially defined their patterns in the SM language and using the SMC compiler [2]. They were able to compile and convert the outlined patterns to C language. The problem with their work is that the analyzer is not optimized and does not consider common information sharing between patterns. Since their patterns mostly examine events belonging to a small set of system processes and resources, it would be possible to share internal states between different but related patterns. Without common shared states, patterns simply attempt to recreate and recompute those shared states and information, leading to reduced overall performance. Also, since preemptive scheduling in the operating system mixes events from different processes, the aforementioned solutions cannot be used directly to detect complex patterns, because of the time multiplexing brought by the scheduler. It needs to first split the events sequences for the execution of each process and then apply those pattern matching and fault identification techniques.

These problems were addressed in [48, 49]. They proposed a stateful synthetic event generator in the context of operating system kernel traces. Their trace aggregation method is designed for kernel traces considering all kernel specific features (extracting execution path, considering scheduling events, etc). They also show that sharing the common information simplifies the patterns, reduces the storage space required to retain and manage the patterns, increases the overall computation efficiency, and finally reduces the complexity of the trace analysis (Figure 2.3).

Pattern mining techniques are also used to aggregate traces and extract high level in-

[2] http ://smc.sourceforge.net

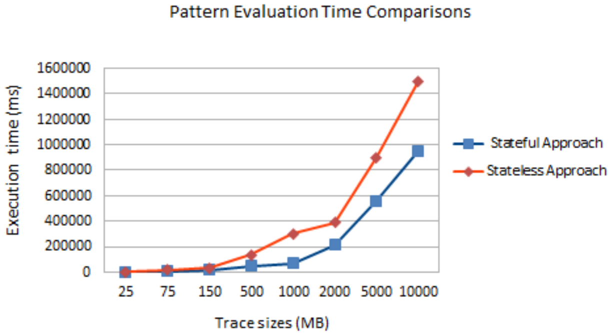


Figure 2.3 Trace aggregation time comparisons for stateful and stateless approaches [48].

formation from trace logs [5]. It is used to find the patterns (e.g., system problems) that are frequently occurring in the system. Several pattern mining techniques have been studied in [20]. Han et al. [70] classified the pattern mining techniques into correlation mining, structured pattern mining, sequential pattern mining, frequent pattern-based clustering, and associative classification ; they described several applications for each category. They reported that frequent pattern mining can lead to the discovery of interesting associations between various items, and can be used to capture the underlying semantics in input data.

Pattern mining may also be used to find system bugs and problems through mining live operating system trace logs as applied by Xu et al. [145] and LaRosa et al. [86]. LaRosa et al. [86] developed a kernel trace mining framework to detect excessive inter-process communication from kernel traces gathered by LTT and dTrace kernel tracers. They used a frequent itemset mining algorithm by dividing up the kernel trace events into various window slices to find maximal frequent itemsets. Using this technique, they were able to detect the excessive inter-process interaction patterns affecting the system's overall performance [86]. Similarly, they also could find the denial of service attacks by finding processes which use more system resources, and have more impact on the system performance. However, their solution cannot be used to find (critical bug) patterns that occur infrequently in the input trace.

Although pattern based approaches are used widely in the literature, they face some challenges and have some limitations, especially for use in the kernel trace context. Efficient evaluation of patterns has been studied in several experiments [4, 151]. Productive evaluation of the specified patterns is closely related to multiple-query optimization in database systems [151] that identifies common joins or filters between different queries. These studies are based on identification of sub-queries that can be shared between distinct concurrent queries to improve the computational efficiency. The idea of sharing the joint information and states has also been deployed by Agrawal et al. [4]. They proposed an automaton model titled

"NFA^b" for processing the event streams. They use the concept of sharing execution states and storage space, among all the possible concurrent matching processes, to gain efficiency. This technique is also used in the context of kernel trace data to share the storage and computation between different concurrent patterns [48].

2.3.2 Metric-based Abstraction

Besides the grouping of trace events and generation of compound events, used to reduce the trace size and its complexity, another method is to extract some measures and aggregated values from trace events, based on some predefined metrics. These measures (e.g., CPU load, IO throughput, failed and succeeded network connections, number of attack attempts, etc.) may present an overview of the trace and can be used to get an insight into what is really happening in the (particular portion of) trace, in order to find possible underlying problems.

Bligh et al. [15], for instance, show how to use the statistics of system parameters to dig into the system behavior and find real problems. They use kernel traces to debug and discover intermittent system bugs like inefficient cache utilization and poor latency problems. Trace statistics, to analyze and find system problems, have been also used in [24, 25, 145]. Xu et al. [145] believe that system level performance indicators can denote the high-level application problems. Cohen et al. [24] firstly established a large number of metrics such as CPU utilization, I/O request, average response times and application layer metrics. They then related these metrics to the problematic intervals (e.g., periods with a high average response time) to find a list of metrics that are good indicators for these problems. These relations can be used to describe each problem type in terms of atypical values for a set of metrics [25]. They actually show how to use statistics of system metrics to diagnose system problems ; However, they do not consider scalability issues, where the traces are too large, and storing and retrieving statistics is a key challenge.

Ezzati et al. [50] proposed a framework to extract the important system statistics by aggregating kernel traces events. The following are examples of statistics that can be extracted from a kernel trace [36, 50] :

- CPU used by each process, proportion of busy or idle state of a process.
- Number of bytes read or written for each/all file and network operation(s), number of different accesses to a file.
- Number of fork operations done by each process, which application/user/process uses more resources.
- Which (area of a) disk is mostly used, what is the latency of disk operations, and what is the distribution of seek distances.
- What is the network IO throughput, what is the number of failed connections.

- What is the memory usage of a process, which number of (proportion of) memory pages are (mostly) used.

To perform metric-based abstraction, a pattern of events is registered for each metric (i.e., a mapping table). Each pattern contains a set of events and an aggregation function identifying how to compute the metric values from the matching trace events. For example, the pattern of a metric like process IO throughput includes all corresponding file read and write events as well as a SUM function (as the aggregation function). The trace abstraction module uses these patterns to inspect the events and aggregate them for all predefined metrics.

Most visualization tools support metric-based abstraction, TuningFork [8], Vampir [107], Jumpshot [150], etc. In these tools, the statistics gathered by metric-based abstraction are displayed as histograms or counts (or average, min, max, etc.), and usually rendered as the highest level of the data hierarchy to show an overview of the underling trace.

2.3.3 Visual Abstraction

Data abstraction, as explained in the previous section, mostly deals with the data content and does not usually have a sense of the visualization environment. When displaying a large trace set, it may not be possible to visualize all abstract trace events together, because of the limited screen display area. It is sometimes required to perform separately a visual abstraction of trace data, to aggregate the data and display a small set of events, enabling a simpler and more usable view. Applying different types of visual abstraction, like filtering some unimportant items, grouping and aggregating related events (related rectangles), displacement, simplification, exaggeration, or reducing or enlarging the size and shape of events, may reduce the visual complexity of the trace, increase its readability and clarity, and sometime improve the performance of graphical rendering of the trace items [72].

While data abstraction is based on analyzing and manipulating the trace content, visual abstraction is more about manipulating the trace representation (and not the data itself) [136]. Many visualization tools use colors and shapes as the elementary elements to represent the trace events. TMF and LTTV[3] use colors to differentiate the various states (waiting, system call, user space, etc.) extracted from trace events. Rectangles are usually used to represent the abstract events and states. In the same way, arrows are used to represent the communication and message passing between different modules (processes). In the LTTV and TMF visualization tools, for example, when there is more than one trace event in an area smaller than a pixel, a black dot is shown instead of those events.

Annotation is another way to visually abstract the trace items. Annotation can be used to describe a group of events, to display user comments about a trace section, or the content

[3] http ://lttng.org/

of a message passed between different resources [124]. Labels, as a specific type of annotation, is also used in different visualization tools. Ovation [114], the Google chrome tracing tool[4] and TMF[5] use labels to represent function names, system calls, return values, etc. Labels can be also filtered, shortened, enlarged or aggregated when there is more or less display area available [55].

2.3.4 Resource Abstraction

Trace events are usually multi-dimensional in nature and include interactions of different resources (dimensions). There may be a large number of resources (processes, files, cpus, memory pages, ...) about which a tracer gather information. For instance, a "file open" trace event may contain information from the running process, the file that has been opened, the current scheduled CPU for this operation and the return value (i.e., file descriptor (fd) of the file). Therefore, the abstraction of the resources, and their representation, can also be important to reduce the complexity of the trace display. Grouping resources [13, 126], filtering of uninteresting resources [81], or hierarchical organization of resources [50, 51, 103] are the related techniques used in the reviewed literature.

Montplaisir et al. [103] exploited a tree representation (called attribute tree) to organize resources extracted from trace events. One important feature of their work is extracting the resources dynamically from traces. It may also be possible to statically define and organize all existing system resources (all classes, all packages, all processes, all files, etc.). However, since the trace data may contain events and information for only a small number of resources, defining the resources statically and in advance will be a waste of time and display space (Figure 2.4).

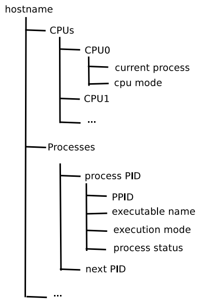


Figure 2.4 Hierarchical abstraction of resources extracted from trace events.

Another representation of resources extracted from trace events is presented in [50, 51].

[4] http ://dev.chromium.org/developers/how-tos/trace-event-profiling-tool

[5] http ://lttng.org/

In their work, a hierarchy is defined for each resource type (also called domain), e.g., one hierarchy for processes, one for files, etc., and the metrics are defined between these domains. They present solutions for constructing cubes of trace data to be used later in trace OLAP (OnLine Analytical Processing) analysis.

Schnorr et al. [126] group the threads based on their associated processes and then by machine name, cluster and so on. Automatic clustering of system resources is performed in [13]. Two approaches were used : automatic clustering of processes based on their run-time inter-process communication intensity, and combining the inter-process communication information and the information of processes extracted from static source code [13].

2.3.5 Applications of Trace Abstraction Techniques

As explained, trace abstraction techniques are used to reduce the size and complexity of trace logs to make their analysis simple and more straightforward. One direct application, for trace reduction and simplification, is system behavior understanding and comprehension [28]. Other applications are security problems detection (e.g., network attacks), system problems investigation (e.g., performance degradation), comparisons of system execution, monitoring, etc.

Beaucamps et al. [9] propose a method for malware detection, using abstraction of program traces. They detect malware by comparing the aggregated events to a reference set of malicious behaviors. Uppuluri [120] uses a pattern matching approach to diagnose system problems. In [42], [83] and [90] similar techniques are also used to detect system problems and attacks. They mainly differ in describing and representing the patterns. STATL [42] models use signatures in the form of state machines, while in [83] signatures are expressed as colored petri nets (CPNs), and in MuSigs [90] directed acyclic graphs (DCA) are used to represent the security specifications.

Trace abstraction can also be used to detect system and network problems, and attacks, by looking for fault and attack patterns and scenarios in execution traces. Using this method, users can discover problems like inefficient CPU scheduling, network attack attempts, slow disk accesses, lock contention, inappropriate latency, non-optimal memory and cache utilization, and uncontrolled sensitive file modifications [15, 99, 145]. The techniques used in pattern based fault identification tools resemble the attack discovy techniques in intrusion detection systems (IDS). Intrusion detection systems analyze network packets and look at their payload to find and detect attack signatures. Similar to IDS systems, trace analysis tools, upon detecting such a problematic pattern, may generate an alarm or even trigger an automatic response (e.g., killing a process, rebooting the system, etc.) [131].

One of the other techniques to reduce trace complexity and improve understanding is visualization. A proper visualization, specially multi-scale visualization, can significantly help to alleviate big data analysis problems, as investigated in the next section.

2.4 Multi-level Trace Visualization

After creating a hierarchy of events, using various abstraction techniques, it is important to have a proper visualization model to store and display the hierarchy of events, including a proper navigation and exploration mechanism. Without such a visualization model, displaying large traces may become overly complex, leading to information overload [111]. In this section, we focus on multi-level visualization techniques and tools, and investigate the different techniques for displaying and visually organizing the hierarchical trace data.

2.4.1 Hierarchy Visualization Techniques

In the literature, there are many techniques to visualize large hierarchical structures. We categorized them into four main groups : space filling, context+focus, multiple levels, and multiple views. In the following, we explain each technique in more detail.

Space Filling Technique

Space filling is a visualization technique used for hierarchical structures which uses almost all available screen space. In this technique, intermediate and leaf nodes are both displayed as rectangles (or polygons), for which sizes are computed based on their importance or property values. One space filling technique is the radial method [41], in which items are drawn radially, the higher levels at the center of the display and lower layers away from the center. Treemap [132] is another popular space filling technique. It allocates the rectangle space of the visible screen to the root node and then splits it among its children based on their properties. In this technique, the rectangle corresponding to each node is labeled with an attribute of that node (size, resource usage, importance, etc). The technique was first used to visualize the directory structure of the file system of a 80MB hard disk [132]. It was also used to visualize the trace data in Triva visualization tools [126], as well as in Gammatella [110] and LogView [94]. Figure 2.5 shows the visualization of one million items using the Treemap technique.

The main focus of space filling techniques is on the leaf nodes, whereas the non-leaf nodes are not clearly shown. Thus, it can be a candidate for the cases where the leaf-level nodes are of interest and important for analysis (e.g., for presenting unusual patterns at the leaf level). Furthermore, users may loose the focus of the whole system, since it is more difficult to focus on one part, and at the same time keep a global overview. Another problem with this technique is that the small areas (rectangles or polygons) may be difficult to distinguish.

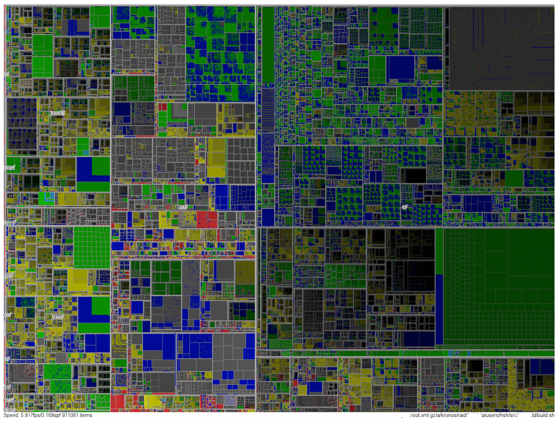

jazz/
Speed: 5.917fps/0.169spf 971061 items
root.xml.gz/a/kronos/raid/ a/users/hsh/src/ ld/build.sh

Figure 2.5 Visualization of one million items using treemap.

Focus+Context Technique

One problem with the space-filling techniques is that users may loose the position of the visible items, due to the lack of a global overview. Focus+Context techniques solve this problem by simultaneously displaying an overview of the data while also zooming on a small part of the view. In other words, users can zoom and focus on any part of the view, while they see the overall context [84]. One example of this technique is using a magnifier over a text document. You can zoom on any part of the text and enlarge the content by moving the magnifier, while retaining an overview of the data (i.e., the entire page). Tree based visualization techniques are another example of this method, e.g., in a tree based window explorer, you can expand a node to see its content and details, while you see the overview of the data simultaneously.

Hyperbolic browser/tree, originally introduced by Lamping et al. [84], exploits a Focus+Context technique. Hyperbolic browser/tree displays the hierarchical data on the hyperbolic space rather than Euclidean space and then maps it to the unit disk, making the entire tree visible at once. In this tree, the focused node is displayed larger, and the degree of interest of other nodes is calculated automatically based on their distances from the selected node. Since the degree of interest for each node is changing with respect to the focused node, the cost of tree redrawing leads to speed concerns. The other problem with this approach is that it cannot display non-hierarchical relationships. Mizoguchi [101] has used the Hyperbolic tree and also machine learning techniques to find the tracks of an intruder in his anomaly detection system. Figure 2.6 depicts a view of the Hyperbolic technique.

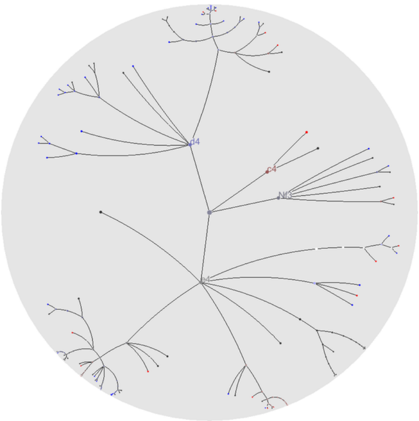


Figure 2.6 An example of a hyperbolic browser.

Multiple-levels Technique

Focus+Context techniques are typically used for displaying and following data sets having clear hierarchies and categories. These techniques display both the context and focus within the same screen, but sometimes, when there is more detailed data for each selected area, it might not be possible to display both the details and the overview on the same display at the same time [23]. One solution for this problem is using various views for displaying the different levels, as often used in online geographical maps.

Two techniques are usually supported in this method. First, semantic zooming [136], in which another semantic level of objects is displayed when changing the zoom level. In a non-semantic zooming approach, as used in many visualization tools, (TuningFork [8], Jumpshot [150], TMF and Chrome tracing environment), the objects are displayed in larger size as the available display area becomes larger.

The multiple-levels technique usually supports (different types of) linking between the different data layers. To do so, a direct link is kept for different objects (referring), or objects are organized in such a way that relationships can easily be extracted later (matching). Establishing links between different layers enables multi-scale analysis, because it makes possible following one data item within other hierarchy levels.

In the multiple-level techniques, various levels are used to display the different data resolutions, overview on top, and details at the bottom. Users typically see a top-level view and then can zoom and focus on any selected part to get more details [23]. The difference with the Focus+Context approach is that here the views appear separately, not simultaneously. In other words, more information can be displayed in this method, because the whole display is used to visualize a view, and there will be no wasted space to show the overview or other data levels. In this method, the trace data is divided, and visualized in various spatial layers. When users zoom, the current view becomes hidden and a completely new view with more detailed data appears (more details and more labels are shown). Supporting semantic zooming in this method enables showing more/less information about objects, when users zoom in/out. For example, in the highest overview level, only labels of important objects are shown, while in the lowest level, more labels are shown (like in Google maps) [55].

There is a special case of this method called overview + details, used in many visualization tools (Jumpshot [150], Vampir [107], TMF, etc.), in which the overview and detailed views are shown next to each other (Figure 2.7). The overview can be the statistics of the important system parameters or only a simple time navigator. Users can browse the overview pane and click on interesting parts to see in another view that section in more details. This method allows a simultaneous display of the views. However, it splits the screen and limits the available space for each view [23].

This method shows different data resolutions at different levels and lets the user understand and detect the relationships between the different data resolutions intuitively. Although space-usage efficient, and widely used in visualization tools and applications to display large data sets, this method is not exploited much in trace visualization tools. It deserves more consideration, because of its remarkable features.

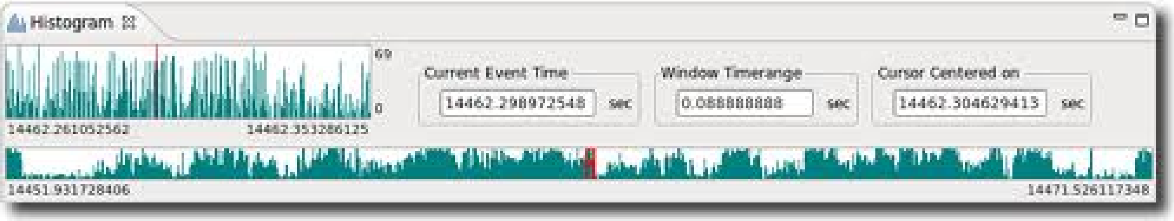
Multiple Views Technique

One popular visualization technique, widely used in trace visualization tools, is multiple-views. This technique exploits a flat visualization technique and shows sequentially the different aspects of the trace in different (coordinated) views [143].

Using this method, displaying hierarchical data is also possible. It is often implemented as separate views, in which each view displays a separate level. A Primary view is used to show the mainline of the system or an overview of the execution, and a set of auxiliary views display other aspects (or levels), such that selecting an item in one view leads to highlighting corresponding areas in other views [109].

This technique helps users to get a better comprehension through different views. Many trace visualization tools like Zinsight [34], TraceVis [124], Vampir [107], LTTV, TMF, etc. use coordinated views to display trace elements. The problem with this technique is that users need to simultaneously follow various related views to analyze the system execution, that can be difficult for some users.

The views are usually coordinated. Timestamps of events or system resources (e.g., a process name) are used to coordinate the views together. For example, when an area is selected in a statistics histogram, all events whose timestamps are in the range of the selected



Current Event Time 14462.298972548 sec

Window Timerange 0.088888888 sec

Cursor Centered on 14462.304629413 sec

Figure 2.7 Overivew + Detail method (TMF LTTng viewer).

area will be displayed or highlighted in the events view. Also, when a process is selected in the control flow diagram, all events belonging to this process will be shown in the events view.

In the same way, to show hierarchical relationships between events at different levels, one possible solution is using implicit links between elements at different levels. For example, when an item is selected in one level (view), related events in another level (view) can be highlighted. Trace Visualization tools typically use event timestamps to link the related data together in the hierarchy. When a high level event is selected in one view, all related events in other views with the same timestamp are highlighted and shown.

Data Correspondence

One important issue in multi-level visualization is deciding about managing and representing the correspondence between data. Techniques are required to extract, organize and visualize the links between related data at different levels. Supporting data correspondence and multi-level navigation are important features of visualization tools to provide. For instance, when users want to follow and dig into an interesting high level entity (e.g., an event displaying a performance problem or a network attack), these features can be used.

In the focus+context and other tree-based methods, the relations between entities at different levels are normally displayed using a tree-based view (expand/collapse). Such relations are also fairly explicit in space filling techniques. However, in multiple-level views, the detection of relations between data at different levels is less explicit, left to the user intuition. In this technique, it is important to present data in such a way that users can seamlessly navigate and understand the relationships between relevant data.

To link events in the multiple-view technique, as explained earlier, two general methods may be used : 1- structure-based linking 2- content-based linking. In the former, the events bounding information (e.g., timestamps) or the system resources (e.g., process, file, etc.) are used to relate events together. In the latter, data semantics are used to link events from

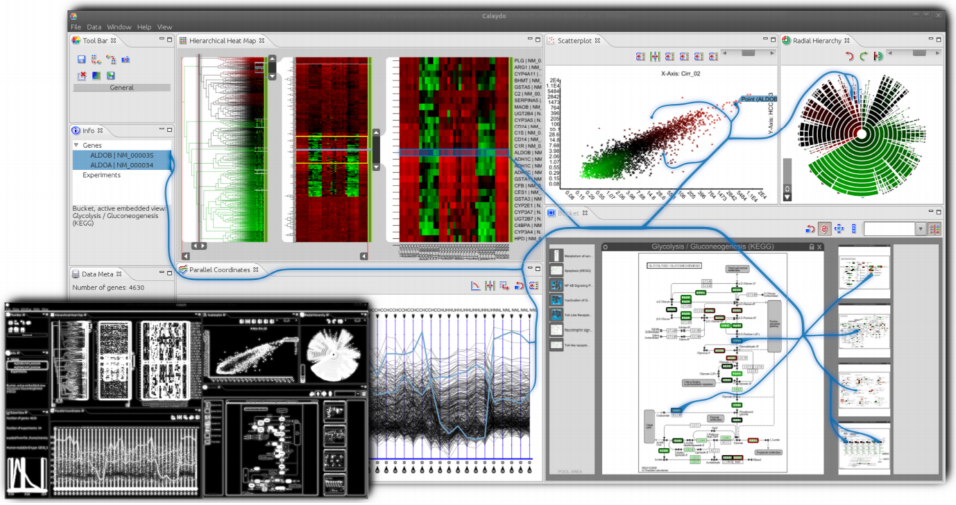


Figure 2.8 Context-preserving visualization of related items and links.

different layers (views), e.g., selecting a ”HTTP connection” abstract event in a view results in displaying all related events (e.g., socket create, socket call, socket send, socket receive, socket close and so on) in another detailed view.

The details of the links between events, regardless of their type (e.g., structure or content based), can be extracted statically (in the process of aggregation), or dynamically using matching techniques. In the former, event links are generated in advance and stored in a type of index to be used later. In the latter, no linking information is stored in advance, the related events are instead matched dynamically in the visualization phase.

Matching is commonly used in spatial applications using a geometric matching algorithm [144]. This algorithm assumes that some attributes of the related objects should be matched. In this technique, semantic attributes, metrics, geometrical and structural information from objects are used, for matching similar objects at different scales [80].

To display the links, besides highlighting the related events, it is sometimes useful to also show lines and arrows between relevant views and items, as investigated by Collins et al. [26], Waldner et al. [140] and Steinberger et al. [135].

Collins and Carpendale [26] introduced VisLink, a visualization environment that reorganizes various 2D elements and displays links between them. VisLink is generalized for different visualization techniques like Treemap, Hyperbolic, etc. In VisLink, each graph is represented on a separate plane, and the relationships between these planes are displayed using edge propagation from one visualization to another. The same approach is also used in [88] to effectively present all relevant gene expressions and the relations between them and between multiple pathways. Waldner et al. [140] extend visual links to graphically present links between related pieces of items across desktop application windows. Figure 2.9 shows an example of the VizLink visualization.

Steinberger et al. [135] believe that using this links visualization technique may lead to hiding valuable information. For solving this problem, they introduced context-preserving visual links [135]. In this approach, the items and the connecting lines are highlighted, helping to find and discover the related items, but special care is taken to avoid hiding valuable content. Indeed, the links are routed along the window borders to avoid obstructing the view of items. This technique is attractive and has clear advantages over traditional visual links. However, it may lead to possible visual interferences. Moreover, the issue of dealing with scalable data sets was not addressed in their work and is likely problematic. In the case of particularly large traces, visualizing the links between events may cost too much, and should be studied carefully. Figure 2.8 depicts context preserving links visualization.

In this section, different ways to visualize hierarchical data, popular in trace visualization tools, were investigated. In the next section, we study some existing trace visualization tools, explain the abstraction techniques used, and investigate their support for multi-level visualization.

2.4.2 Visualization Tools

By using an appropriate multi-level visual representation, users can view an overview of the trace, then focus and zoom into areas of interest to get more details. Several studies have been conducted in the area of scalable visualization [34, 45, 68, 127]. In the following, we study various trace visualization tools, from operating system level to application level tracing, and for each we review the abstraction method used, and discuss if they support multi-level visualization and the way they handle visualization scalability.

Vampir [107] is a trace analysis tool used for performance analysis and visualization of MPI (Message Passing Interface) programs. Vampir uses the multiple views technique and contains different synchronized views such as global timeline view, process timeline view and communication statistics view. Global timeline is the default view that gets activated automatically when opening a trace. It displays a fine-grained view of the execution behavior. Analysts can then select any individual section of the timeline and zoom to get more detailed information for that part. In addition, users can view statistics (metric based abstraction) for the displayed time interval [64]. Hierarchical visualization is supported using a multi-level space-time (timeline) diagram. In this view, the highest level shows the statistics and there is the possibility of focusing and zooming, where users can explore data for different levels of resources such as cluster, machine, or process. However, the metrics used for representing

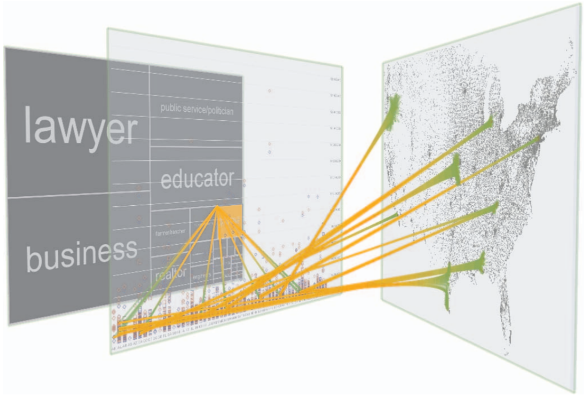


Figure 2.9 Visualization of related items and the links between them.

the system behavior are too low level, and analysts may therefore not understand easily what improvements are possible. A summary of this tool (in conjunction with other tools) is displayed in Table 2.1.

Similar to Vampir, Jumpshot [150] is a visualization tool for understanding the performance of parallel programs. Jumpshot first displays an overview of the whole trace, and, once the user selects an interval, it displays a detailed view of that interval. The length of time intervals is fixed at each level, and is selected statically or dynamically. Jumpshot uses the SLOG2 trace format [18], a hierarchical file format to handle a large number of events and states in a scalable way, even for large-scale applications. Jumpshot abstracts out the trace events to generate ready-to-display hierarchical preview drawable objects : preview_state, preview_arrow and preview_event. At each detail level, it uses a separate pack degree that shows how many underlying events are used to create the preview objects of the current level. Users can zoom-in and narrow the view to see large intervals and states that were too small in the previous views, but large enough for the current view. Jumpshot supports also resource abstraction, displaying an aggregated set of processes in a special view called ”mountain range”. The link between views is established based on timestamps. There is no other mechanism to directly link the different views and their associated events.

AVID (Architecture Visualization of Dynamics in Java System) [141] is an Ecplise based offline visualization tool to summarize and visualize the object-oriented system's execution at the architectural level. The diagrams generated by AVID demonstrate the system execution by visualizing the objects and the interactions between them. The system execution is represented by animating the sequences of cells. Each cell denotes the aggregated summary information of the system execution up to that point. A trace sampling method is used to select only a portion of the trace data related to a specific analysis scenario, instead of the whole trace. Even using a sampling method, AVID has scalability issues, and scenarios dealing with large traces cannot be efficiently visualized. As mentioned, AVID can be used to understand the system behavior at the architecture level, but it needs the architecture model of the system to be available. This may not always be feasible for all systems. Moreover, it needs a high level model, and the mapping between classes and the architectural entities, to be manually defined by the analyst. However, the analyst is not expected to be familiar with the system architecture [68].

Shimba [137] integrates both static and dynamic information to analyze the behavior of manually selected artifacts of Java applications. Its purpose is not to display a general overview of the system. Using reverse engineering, the static information including various software artifacts (classes, methods, variables...) are extracted and displayed. Users may then select any interesting component to be traced by Shimba. The resulting trace extracts the interactions between selected components and visualizes them using UML sequence diagrams. Shimba uses several abstraction techniques to cope with scalability problems. It generates several levels of abstraction from both static and dynamic information. An example of the former is grouping related software components to construct higher level entities. An example of the later is abstracting out the dynamic information, such as sequence diagrams, to higher level diagrams. A pattern matching technique is used to generalize the sequence diagrams [137]. Unlike other tools that use system-level tracing and then reduce the data, Shimba uses component-level tracing, resulting in a small trace data size. It relies on users to select the component that implements a specific feature. However, this is not always possible in a complex system. One problem with this tool is that diagrams are sometimes very large, and thus harder to understand.

Jinsight [113], is an Eclipse based tool for visualization and performance analysis of Java programs. It provides several dynamic views depicting object populations, method calls, thread activities and memory usage. Using an automatic pattern recognition technique, Jinsight can detect repeated behavior patterns from trace data. It uses these patterns to produce an abstract view of the program execution. Jinsight uses different views to illustrate different aspects of the investigated software behavior. Jinsight supports information filtering, enabling analysts to filter out uninteresting behaviors. De Pauw et al. [113] showed that the tool was successful in detecting various problems, such as memory leaks in industrial applications. However, it does not scale well to large traces. Jinsight supports trace aggregation over time axis in some views.

Ovation [114] is another tool that uses a tree based view called ”execution patterns” to visualize execution traces. Similar to Jinsight, it uses automatic pattern recognition to extract and identify patterns of repeated executions. The execution pattern view displays multiple levels of data to users : a high level view first and more details on demand. Ovation firstly shows a generalized view of the system behavior using execution patterns. It supports drill-down, roll-up, zoom and pan operations. Users can zoom and focus on any area of interest to see and find more details in an enlarged view. De Pauw et al. [114] reported that Ovation has been used successfully to detect unexpected execution patterns and to improve performance of large systems. However, Ovation reveals relatively detailed low level (i.e. method-level) information about the application execution, and assumes that the analyst has enough knowledge of the system to query this information. Furthermore, it works better for single program comprehension, rather than general multi-module systems (like operating system).

Zinsight introduced by De Pauw et al. [34], is a visualization tool that facilitates system comprehension and performance problems detection. It is used to analyze special mainframe software and hardware by examining operating system level trace events. Zinsight provides facilities for creating high level abstract views from the low level execution traces. It uses pattern recognition techniques to discover and extract execution flow patterns from raw events. It supports different coordinated views like event flow view, sequence context view, statistics view. These linked views help analysts to see the system execution at a higher level of abstraction, and make them able to navigate through different levels. Although Zinsight supports trace abstraction and linked views, it has scalability problems and is not able to visualize traces unless they completely fin in main memory [33].

TuningFork [8] was introduced to analyze and visualize traces gathered from the execution of very large realtime systems. They vertically integrate data from multiple levels of abstractions, from hardware and operating system levels to the application level, in order to analyze the system. TuningFork also stores coarse-grained summaries of data, enabling navigation across different time scales, from hours to milliseconds, and exploration in multiple levels of granularity.

Tracevis [124] is used to visualize Java programs. It uses multiple views to display different aspects of program execution, including timeline and dynamic call graph. Tracevis supports zooming in any selected area to display detailed trace data. Tracevis also supports correlating trace instructions with both the assembly and c code. A multi-level statistics view is supported to show different levels of abstraction. Tracevis uses metrics based abstraction and displays the statistics for any selected area by means of annotations.

Triva [126] is another tool that aims to visualize the trace events of large parallel applications in a hierarchical manner, supporting visualization scalability. It supports the dynamic hierarchical organization of trace data and uses the Treemap space filling technique to visualize trace data. Triva also supports a hierarchical organization of resources, in which the threads of a process are grouped together, and processes are then grouped by machine, cluster and then grid. In this way, Triva can simultaneously visualize more monitoring entities than other traditional techniques (e.g., space-time), in different time granularity (from microseconds to days) [126].

LTTV (Linux Trace Toolkit Viewer) [6], developed at Dorsal lab, is another visualization tool that shows events generated by the LTTng kernel tracer [38]. It provides a statistics view, control flow view and event view. The statistics view presents metrics for the different event types. The control flow view displays an overall view of the system execution, aggregated by processes and threads. It supports physical zooming, scrolling and filtering. However, it does not support multi-level trace abstraction and linked events. The same limitation applies to TMF (Tracing and Monitoring Framework) [7] a full-featured Eclipse based LTTng trace viewer, with more views and features and supporting different visual abstraction mechanisms (e.g., labels, etc).

2.5 Hierarchical Organization of Trace Data

A single pass over the trace data is normally sufficient to aggregate the trace and generate high level events to reduce its size, and possibly detect some problems. In multi-level trace visualization, users want to access any arbitrary area of a trace with support for interactive exploration, panning, searching (e.g., interval search, etc.), focusing and zooming. A proper hierarchical modeling of the trace data then becomes very important for large traces.

Behind the scene of the aforementioned visualization tools and techniques, are the supporting data structures. The first step for building a multi-level visualization tool, and linking events at different levels, is modeling and storing the generated abstract events in such a way that they can easily be fetched and extracted on demand. For efficient management of various abstract events, it is required to exploit an optimized data organization in terms of compactness and access efficiency. The data organization should enable both horizontal and vertical analysis in large traces. Horizontal analysis means going back and forth in the trace, inspecting events and processing the data for understanding the trace. Vertical analysis refers to hierarchical and multi-level analysis. In the following, we review different data structures used to model and manage trace events, abstract events and states, and also establish links between them.

Every time users query the system and ask to view the events and information (e.g., statistics of some system parameters), at arbitrary places in the trace, it would be possible but impractically long to reprocess the trace from the starting point, apply the requested

[6] http ://lttng.org/tracingwiki/index.php

[7] http ://lttng.org/tracingwiki/index.php

Table 2.1 Trace visualization tools and their features.

Trave Visualization Tool	Multiple Levels	Abstraction Type				Visualization Type			
		data abstraction	metric-based abstraction	visual abstraction	resource abstraction	space filling technique	multiple levels	multiple views	context + focus
Vampir	✔	filtering + aggregation	✔	✔	✔		✔	✔	
AVID	manual	sampling		animation	manual			✔	
Jumpshot	✔	✔	✔	✔	mountain range		✔	✔	
Shimba	static analysis	selective tracing and abstraction		aggregation of sequence diagram					
Jinsight	✔	repetition detection	✔	✔			✔	✔	
Ovation	✔	repetition detection	✔	✔				✔	execution patterns
Zinsight	✔	repetition detection	✔	annotation	✔		✔	event flow	event type statistics
TuningFork	✔		✔	color				✔	
Tracevis	✔	filtering	✔	annotation			✔	✔	
Triva	✔		✔	color	✔	treemap			
LTTV			✔	color	✔		✔	✔	

trace analysis (abstraction, reduction, etc.) techniques and get the requested information. The cost of reading and processing the whole trace for each single query, would indeed be too costly for anything but the smallest traces. Indeed, the horizontal analysis aims to support the efficient and scalable analysis of large traces for arbitrary points (and intervals). Users want to jump efficiently, without any undue or apparent delay, to any place in the trace to analyze the data to get an insight into the program behavior at that point. The challenge is similar for the vertical analysis.

Some projects use periodic snapshots (at checkpoints) to collect and store the intermediate analysis data. This method splits the input trace into equal (or possibly different) durations (e.g., a checkpoint for each 100K events) and stores aggregated data at each checkpoint. Later, at interactive analysis time, instead of reading the whole trace, the viewer reads and reruns the trace from the nearest previous checkpoint to the given query point and regenerates the desired information. The checkpoint method is used in LTTV (the LTTng trace viewer) and was used initially in TMF (Tracing and Monitoring Framework) [8].

Related to the checkpoint method, LTTngTop, developed by Desfossez et al. [36] at Dorsal lab [9], is an efficient command-line tool to index LTTng kernel traces and extract various statistics on the system parameters. It is, to a certain extent, a more detailed and efficient version of the Linux TOP command. It dynamically displays an ongoing state of the system, for continuous time periods, using operating system trace data. LTTngTop aggregates trace events and stores them in different checkpoints to enable navigating through the time axis and producing statistics for any time period, for the current time or any time in the past.

Although the checkpoint method is useful to avoid reading the whole trace for each query, it has some limitations. For instance, it always requires accessing the original trace data. Then, when the original trace data is not available, or if there is not enough space for storing a whole streaming trace, this method cannot be used. Moreover, the snapshots have a fairly constant size while their number increases with the trace duration. There may indeed be a lot of redundancy between the content of snapshots at consecutive checkpoints. Some values change frequently while others rarely vary and remain constant from one checkpoint to the next. This does not scale very well for large trace sizes and long durations. In some tools, the snapshots were stored in main memory, forcing a limit on the trace size that they could handle.

To solve this problem, Ezzati et al. [50] proposed a dynamic checkpoint method (different checkpoint intervals for different metrics) to analyze trace events. In their work, each metric uses its own checkpoint interval, based on its precision, degree of granularity and importance.

[8] http ://lttng.org/tracingwiki/index.php

[9] http ://www.dorsal.polymtl.ca

They also avoid storing the whole trace to answer the analysis queries. Queries are served using only a compact trace index, created at trace pre-processing time. Although the solution presented in [50] works well with offline traces and was tested for very large traces, the data structure behind their work grows linearly with the trace size and can become unexpectedly large for long streaming traces. The cube data model [51] was proposed to address this problem for live trace streams of arbitrary size. It enables efficient multi-level and multi-dimensional trace analysis, similar to OLAP analysis in database applications.

Montplaisir et al. [102, 104] introduced a tree-based structure, called ”state history tree” to store and query the incrementally arriving interval data. They used this data structure to manage and model the system state value changes [103]. Since a tree-based structure is used, they can answer the queries quickly and directly by only traversing the related branches and nodes, without requiring access to the original trace. The problem with their proposed structure is the large size of the generated tree, which was addressed by the partial history tree [102]. A partial history tree avoids storing all intermediate data in the tree, but requires access to the original trace data during the analysis phase. The partial state history reduces the state history tree size by more than a hundredfold for a small increase in query time. In their approach [103, 104], the proposed history tree essentially stores intervals, each interval represents a state value and contains a key, a value and a time interval. Since (raw or abstract) trace events usually have the same content, they can also be modeled and stored as interval data, as is the case for the SLOG file format [18].

Chan et al. [18] proposed a file format called SLOG (and improved later as SLOG2), a hierarchical trace format used to store trace events (called drawable objects in their work) in a R-tree like structure for efficient interactive display. The SLOG file format contains intervals, representing the drawable objects, and a tree index schema to provide direct access to any address within the file. Each interval is described by bounding times and a thread number. The index tree is like a binary tree that stores the elements in different-size buckets. At each level of the tree, the duration of the buckets is the same. In this tree, the root node represents the whole trace length (interval 0 to T). Each node in the second level represents time intervals of size $\frac{0+T}{2}$. The next level shows intervals of size $\frac{0+T}{4}$. In the same way, leaf nodes show intervals that are smaller than or equal to a ΔT_{min}. Objects are stored in the smallest containing node, in both leaf and non-leaf nodes.

At display time, for any given time t, only the intervals intersecting with t will be read and displayed. Using this file format, Chan et al. [18] reported that they could achieve nearly constant access time for different time intervals. The main problem with their work is that they use the same time durations to store the drawable items. For the cases where trace events are not uniformly distributed along the time axis, some nodes will be full and others almost empty. This may affect the access time and give different access times for different locations in the trace. This solution models the different drawable objects (events, states and so on) but does not consider the links and the relations between objects. Their file format has been used in Jumpshot [150]. Modeling trace events as interval data, and using interval management structures to store the trace events, appears to be a good solution for very large traces.

There are many other interval management structures that could be used as storage for trace events. Segment-tree is a basic data structure used for storing the line segments. It is a balanced binary tree where each node is defined by its bounding box. In this tree, leaf nodes represent the elementary segments in an ordered way, and non-leaf nodes correspond to the intervals that are union of the underlying children's intervals. The interval duration of each node is approximately half the interval of its parent node. Figure 2.10 shows an example of a segment tree. Searching a segment tree with n intervals and retrieving k intervals intersecting a query point p, take O(logn + k) time. For retrieving the segments that intersect a point p, the search starts at the root node and follows only the branch whose nodes contain point p, and returns only the intervals that contain the given point p. A segment tree with n intervals also needs O(n logn) storage size and can be built in O(n logn) time. Segment tree is an ideal solution for storing intervals in main memory. However, for very large traces, when the tree size would exceed main memory, such a binary tree (each node has two children) would be difficult to store efficiently on disk, given its small node size and important depth.

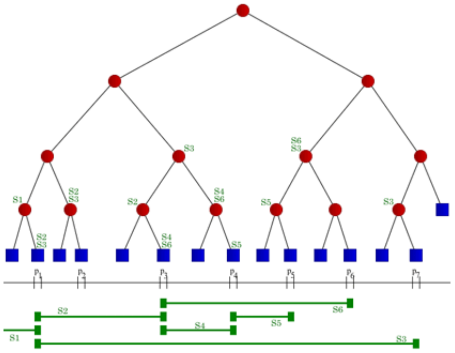


Figure 2.10 Examples of the segment and interval tree.

(source : wikipedia.org)

Very similar to the segment tree is the interval tree. The intervals in this tree are defined as the projection of the segments (in a segment tree) on the x-axis (Figure 2.11). In other words, the difference is that an interval tree performs stabbing queries in a single dimension. Since the intervals are not decomposed like for the segment tree, there will be no redundancy, and therefore the construction time complexity will be better (O(n) instead of O(nlogn)).

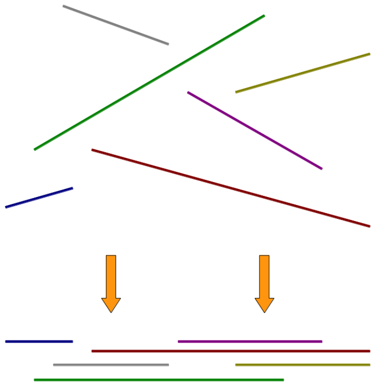


Figure 2.11 Segments versus intervals.

The Relational Interval Tree (RI-tree) proposed by Kriegel et al. [82], uses built-in indexes on top of relational databases to optimize the interval queries. It does not require any interface below the SQL level and just adds an in-memory index for interval data to the existing RDBMS. It can be used to answer efficiently the intersection queries. The main idea here is to use the built-in relational index structures, instead of accessing directly the disk blocks, to manage the object relationships. It is an interesting idea, since it can easily integrate with relational database systems, without having to re-implement their existing features. However, it would remain restrictive for tracing tools to use only a specific database tool. Montplaisir et al. [102] experimented with storing traces in databases but incurred a considerable overhead in both storage space and access time.

R-tree [66] is one of the most common tree data structures for indexing multi-dimensional information. The R-tree and its several variants are commonly used to store spatial information, usually in two or three dimensions. The data structure groups nearby objects and represents them with their minimum bounding box (MBB) in the higher levels. Nodes at the leaf level represent a single object by keeping track of pointers to that object. However, the nodes in non-leaf levels describe a coarse aggregation of groups of low level objects. These objects may overlap or may be contained in several higher level nodes. In this case, the object is associated with only one tree node. Thus, for answering some queries, it may require examining several nodes for figuring out the presence or absence of a specific object. R-tree has several extensions such as the R+-tree, R*-tree, SS-tree, SR-tree and many other variants that aim to increase the efficiency of the original R-tree method [10, 129].

In general, the R-tree and its extensions do not work well for sequences of long intervals with significant overlap [123]. Indeed, splitting and merging (re-balancing) the nodes cause

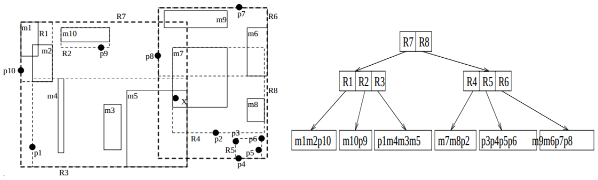


Figure 2.12 An example of the R-tree.

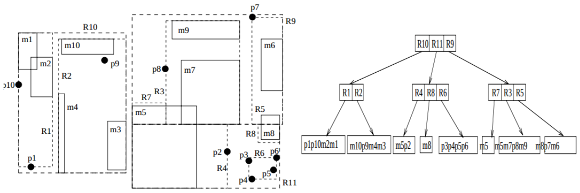


Figure 2.13 An example of the R+-tree.

many updates to the data structures, and as a result induce severe performance degradation.

In the same way, other access methods and spatial index structures have been surveyed in [61], reviewing the B-tree, Hb-tree, R-tree (and its variants : R+-tree and R*-tree), Quad-tree, and other data structures used for managing intervals and their hierarchical organization. Although most of these techniques have some interesting characteristics than can be used to organize trace events, they are generally not used directly in the trace tools. A custom built file storage and indexing format is typically used e.g., SLOG [18] and State History Tree [104].

Please note that this review of data structures and access methods focused on abstract trace events in interval form. Moreover, the access methods of interest were for ordinary operations like explore, zoom and pan. Obviously the assumptions may differ for other use-cases, for example to implement a call-graph view and to focus on the caller-callee relations, where a graph data structure may be more appropriate.

2.6 Discussion

The discussion of the surveyed methods is presented through the following three subsections : trace abstraction, trace visualization and data model.

Trace Abstraction

Trace tools can help users to understand the runtime behavior of systems. One difficulty with trace tools is that they have to deal with large data sets. Multi-level visualization is

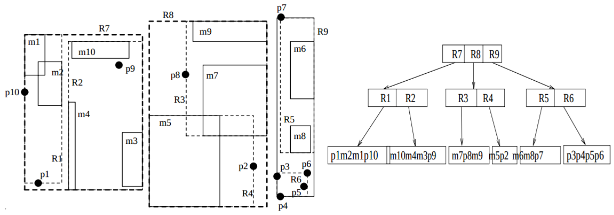


Figure 2.14 An example of the R*-tree.

one technique than can help trace tools to cope better with huge traces. However, multi-level visualization tools need a good support for data abstraction techniques, to reduce the original trace size and generate multiple levels of high-level events.

In this paper, a taxonomy of trace abstraction techniques is presented. The studied trace techniques are i) the content-based method, mostly using the content of the trace data to reduce its size, ii) the metric-based method, used to generate statistics for important system parameters, iii) visual abstraction, focusing on the simplification of the trace events display and finally iv) resource-based abstraction, centred around the organization of system resources to better display the underlying data.

Among the above abstraction methods, the metric-based technique is widely used in trace visualization tools. It helps to get a better overview of the underlying system execution. The output of the metric-based abstraction is typically displayed by means of histograms or other similar charts. These views are usually displayed in the highest level view in trace visualization tools.

A combination of these methods is often used by trace tools to reduce the trace size and complexity, remove the noise and unimportant data, and generate high level information for the different levels of granularity. This is later used for program comprehension, problem detection, monitoring, visualization, etc.

Trace Visualization

The main goal of multi-level visualization is enabling a multi-level analysis of trace data to get a better comprehension of the underlying data. Indeed, displaying the data at multiple levels of details can enable top-down and bottom-up analysis, which matches the human cognitive model.

Existing trace visualization tools usually display the behavior of underlying systems using timelines, sequence diagrams, control flow diagrams, etc, which are different types of space-time (or Gantt charts) data representations. In these views, one axis is used to display the software modules, system resources or processes, and another axis to display the time. The main problem with space-time diagrams is scalability. The number of resources, and also the number of time line elements (i.e., executions) are limited to the visible screen size and resolution. It is not usually possible to display more than a few elements on the screen.

This problem is alleviated using multiple synchronized views. In this model, in addition to the main space-time view, another view is used, for example to display an overview (high level view) of the trace. Users can then explore the overview and go back and forth, selecting an area to get the details in the main space-time diagram. This overview usually shows the statistical information of one aspect of the execution (e.g., IO throughout, CPU usage, number of calls, etc.) to provide an overview of the system execution. Adding other views (e.g., views for other types of statistics) can be useful. However, increasing the number of views, and forcing users to rely on multiple different views, may raise the complexity of the tool and the analysis process.

Some tools solve this problem by hierarchical organization of the trace data, exploiting inherent hierarchical visualization techniques like treemaps [126] node-link, tree-representations [114], multiple levels, etc. However, the multiple levels technique supporting semantic zooming is seldom used in the context of trace data, even though it proved very valuable in spatial applications (e.g., Google maps). It should be noted that in the multiple levels view, the data organization is an important factor to achieve scalability, and thus should be considered very carefully. Also, avoiding big jumps between zooming views, and providing smooth and seamless transitions, is another important factor.

Among the different techniques presented to display hierarchical trace data, it is difficult to determine which one achieves a globally better performance. It strongly depends on the requirements and use-cases. For example, to display a cube of trace data and to provide analysis based on system statistics, a space filling technique is generally more suited. To visualize the operations of a process at different levels, or to relate user space function calls to the kernel system calls, a multiple levels view is particularly appropriate. To study a trace

Table 2.2 Comparison of hierarchy visualization techniques

Technique	Main feature	Weakness	Use-case
multiple views	display different aspects of the trace in different views	it could be difficult to follow all views together	displaying different and unrelated aspects of the trace : statistics view, execution states, function calls, individual events and so on
multiple levels	display a related set of behaviors at different levels and enable zooming between levels	users have to conform and relate the different zoom states together	displaying system functionalities and operations at several levels (user space level to kernel level)
focus+ context	display the whole view at once and enable users to focus on a selected area	not applicable for very large data as well as for semi or unstructured data	displaying the tree-based hierarchies : list of active system processes and the parent/child relations between them
space filling	splits the screen between all nodes based on their properties (size, importance, etc.)	it is difficult to follow the global and overall relationship	displaying a set of statistical data and their proportions together : CPU usages for different processes

from different (unrelated) aspects, a multiple view technique is suggested. To browse a tree hierarchy, focus+context methods may perform better. In some use-cases, a combination of methods can also be used. For example, a multiple levels treemap method is proposed in the literature [14] to explore very large hierarchical data (e.g., 700k nodes amongst 13 levels). A summary is shown in Table 2.2.

Another feature, sometimes neglected in trace visualization tools, is establishing links between related events in different levels and their visual representation. If trace visualization tools want to be more useful in general, they should propose models to easily address the links between corresponding events and items, where linking enables a proper hierarchical navigation to follow a high level issue. For example, when a problem is displayed in a high level view, users may want to dig into that and find more relevant detailed information. Establishing links between the corresponding events, and proper visualization of links, can help users to better understand the situation by following the links to the relevant entities and actions, from among the thousands or millions of trace events.

Data Model

In trace visualization tools, the design of the underlying data model should enable compact and afficient storage, good interactive query response time, and excellent scalability. Different techniques are described in this paper to model the trace data. Trace data can be hierarchically organized using the either of the studied techniques : R-tree, Quadtree, custom methods (SLOG, State History Tree), etc.

Although the internal data structures were not detailed for the presented techniques, they can all be used to index the (abstract) trace data at multiple levels. However, this is different from managing the links and connections between data items. Indeed, although some of those techniques can be used to manage the hierarchy, they only support one form of hierarchy (e.g., time based hierarchy or aggregation hierarchy). However, the trace data may have separate hierarchies for different attributes, which could be addressed by a proper linking method. Also, the non-hierarchical relationships (e.g., the relation between file operations and disk block operations) should be covered as well. Therefore, the design of the data structure should clearly take into account the relations between events in different layers, and the way they are stored, extracted and visualized.

An example of the links between different elements can be found in a file system, where there are hierarchies of disk blocks, files and directories. For a better presentation, and navigation through this hierarchy, the links should be modeled in the infrastructure. For example, while they use a similar file tree hierarchy, Linux uses I-nodes to implement the links between a directory and its files and a file and its blocks, and Windows may use a File Allocation

Table (FAT), a chain of addresses, to manage the links between directories, files and disk blocks. Links between elements can also be found in SOLAP (Spatial OnLine Analytical Processing) systems [93, 95, 115]. However, they mostly use existing database systems to store links between elements, and their internal structure is not clearly known. Most existing trace analysis and visualization tools do not consider the linking between events in different levels. It should be an important focus for future work, for the trace visualization tools.

2.7 Conclusion

Different trace abstraction methods, visualizations of multiple-level data, and related data models are discussed in this paper. The Focus+context method is one approach to display the data hierarchy at different levels. It is more frequently used to display data that has a clear hierarchical structure and fits nicely in a graph or tree. The Space filling technique efficiently uses the entire display screen. It splits the display screen between the items of the hierarchy based on their importance. The focus of this method is more on the leaf nodes, while detecting and studying the intermediate nodes remains possible. Multiple levels supporting semantic zooming is another important method to display a data hierarchy. This technique is exploited in many spatial applications (e.g., online maps) and proved to have a good performance to increase the comprehension of the underlying data. However it is not much used in trace tools and deserves to be investigated in future work. Displaying different data levels in separate views is the essence of the fourth method and is used in most trace visualization tools.

Trace analysis tools often generate events at different levels of abstraction. It is possible to study the behavior of the system under consideration using each level separately. However, due to the predefined degree of details for each level, it is difficult to interpret and understand all aspects of the system by analyzing each level separately. Thus, it can be extremely useful to be able to extract on demand more details for an area of interest. However, this requires an exploration mechanism through different levels of events. In the literature, the related research mostly describes the abstraction method, as well as several interactive techniques to represent and visualize the results. However, many of these lack a proper linking and link navigation mechanism between views. This impedes the interpretation and exploration of trace events. Mechanisms to drill down to the lower levels, and extract more detailed information, are seldom discussed. Nonetheless, such a linkage between extracted abstract events and the underlying data can support better runtime behavior navigation and comprehension. An effective visualization tool should support such links. Trace abstraction and hierarchy visualization techniques are other interesting aspects presented in this paper which should be considered in future work.

Acknowledgement

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research, and Defence Research and Development Canada (DRDC) is gratefully acknowledged.

CHAPTER 3

Paper 2 : A Stateful Approach to Generate Synthetic Events From Kernel Traces

Naser Ezzati-Jivan and Michel Dagenais

3.1 Abstract

In this paper we propose a generic synthetic event generator from kernel trace events. The proposed method makes use of patterns of system states and environment-independent semantic events rather than platform-specific raw events. This method can be applied to different kernel and user level trace formats. We use a state model to store intermediate states and events. This stateful method supports partial trace abstraction and enables users to seek and navigate through the trace events and to abstract out the desired part. Since it uses the current and previous values of the system states and has more knowledge of the underlying system execution, it can generate a wide range of synthetic events. One of the obvious applications of this method is the identification of system faults and problems that will appear in the Illustrative Examples section. We will discuss the architecture of the method, its implementation and the performance results.

Keywords : Trace abstraction ; Synthetic event ; System state ; Intrusion detection

3.2 Introduction

Tracing complete systems provides information on several system levels. The use of execution traces as a method to analyze system behavior is increasing among system administrators and analysts. By examining the trace events, experts can detect the system problems and misbehaviors caused by program errors, application misconfigurations and also attackers. Linux Trace Toolkit next generation (LTTng), a low-impact and precise Linux tracing tool, provides a detailed execution trace of system calls, operating system operations and user space applications [38]. The resulting trace files can be used to analyze the traced system at kernel and user space levels. However, these trace files can grow to a large number of events very quickly and make analysis difficult. Moreover, this data contains too many low-level system calls that often complicate the reading and comprehension. Thus the need arises to somehow reduce the size of huge trace files. In addition, it is better to have relatively abstract and high-level events that are more readable than raw events and at the same time reflect the similar system behavior. Trace abstraction technique reduces the size of original trace by grouping the events and generating high-level compound synthetic events. Since synthetic events reveal more high-level information of the underling system execution, they can be used to easily analyze and discuss the system at higher levels.

To generate such synthetic events, it is required to develop efficient tools and methods to read trace events, detect similar sections and behaviors and convert them to meaningful coarse-grained events. Most of the trace abstraction tools are based on pattern matching techniques in which patterns of events are used to detect and group similar events or sequences of related events into compound events. For instance, Fadel et al. [56] use pattern matching technique to abstract out the traces gathered by LTTng kernel tracer [38]. They have also created a pattern library that contains patterns of Linux file, socket and process operations. Wally et al. [142] use the same technique to find system faults and anomalies. They have also designed a language for defining fault patterns and attack scenarios.

Although defining patterns over trace events is a useful mechanism for abstracting the trace events and finding the system faults, there are other types of faults and synthetic events that are difficult to find with these techniques and need more information of the system resources. In this way, modeling the state values of a system and using them may help much for finding those complex kinds of system problems. Indeed, without a proper state model, many patterns will simply attempt to recreate and recompute some of that repetitive information in a time and performance consuming manner.

This paper mainly describes the architecture of a stateful trace abstractor. Using a state database to store system state values enables us to have more information about the system at any given point. Indeed, after reading the trace files and making the state database, we can jump to a specific point and abstract out the trace at that point. For example, suppose we see there is a high load or a system problem at a certain time. In this case, we can load the system states at that point, reread, abstract out and analyze only the desired part to discover the main reason of the given problem.

The main goal of this paper is to explain how to use the system state information to generate synthetic events as well as to detect complex system faults and anomalies. In this paper, we first explain how to convert the raw trace events to semantic events. Secondly, using a predefined mapping table, we describe how to extract system metric values from trace and create a database of the system state values. Finally, we investigate using pattern matching technique over semantic events and system state values to generate synthetic events as well as to detect system faults and misbehaviors.

The next section discusses related work. It is followed by a section explaining the proposed techniques and also the architecture of the model. Subsequently, we discuss our method in detail and provide some illustrative examples to show how it can be adopted to generate a wide range of synthetic events. Finally, our paper concludes by identifying the main features of the proposed method and possibilities for future research.

3.3 Related Work

The related work can be divided into two main categories : trace abstraction techniques, and their usage in Intrusion Detection Systems (IDS). Trace abstraction combines groups of similar raw events into compound events and by means of eliminating the detailed and unwanted events, reduces the complexity of the trace [13]. Furthermore, abstraction provides a high-level model of a program and makes understanding and debugging the source code easier [92]. Several studies have been conducted on analyzing, visualizing and abstracting out large trace files [56],[98], [142]. Trace visualization is another way to show abstractions of trace events [60]. It uses visual and graphical elements to reveal the trace events. Through their work in [68], Hamou-Lhadj et al. carry out a survey on the several trace exploration and visualization tools and techniques. Some important tools that make use of these techniques are : Jinsight [35], which is an Eclipse based tool for visually exploring a program's run-time behavior ; Program Explorer [85], a trace visualization and abstraction tool that has several views and provides facilities like merging, pruning and slicing of the trace files, and ISV [74], that uses automatic pattern detection to summarize similar trace patterns and thereupon reduces the trace size. Other tools include AVID [19], Jive [121], and Shimba [137].

Besides visualization, pattern matching technique has been widely used for trace abstraction [42], [98]. Most of the aforementioned tools use this technique to detect repeated contiguous parts of trace events and to generate abstract and compound events [68]. Fadel [56], Waly [142], and Matni [98] use pattern matching technique to generate abstract events from the LTTng kernel trace events. Pattern matching can also be used in intrusion detection systems [120]. For example, STATL [42] models misuse signatures in the form of state machines while in [83] signatures are expressed as colored petri nets (CPNs), and in MuSigs [90] directed acyclic graphs (DCA) are used to extract security specifications. Beaucamps et al. [9] present an approach for malware detection using the abstraction of program traces. They detect malwares by comparing abstract events to reference malicious behaviors. Lin et al [90] and Uppuluri [120] use the same technique to detect system problems and misuses.

Almost all of these pattern matching techniques have defined their patterns over trace events and did not consider using the system state information. Our work is different as, unlike many of those previous techniques, it considers the system states information and provides a generic abstraction framework. Our proposed method converts raw events to platform-dependent semantic events and extracts the system state value, and sends them as inputs to the pattern matching algorithms.

3.4 Overview

First, here are some terms that will be referred to throughout this text :

– ”Raw event” is used to identify the event that are directly generated by the operating system tracer. Raw events show various steps of the operating system running, such as a system call entry/exit, an interrupt, disk block read and etc.

– ”Semantic events” to show the events resulting from conversion of platform-specific raw events to environment-independent events. As will be discussed later, there is a mapping table between raw events and environment-independent semantic events.

– Also the term ”synthetic events” is used to identify events that are the result of trace abstraction and fault identification analysis modules and depict high-level behavior of the system execution. In other words, synthetic events are generated from raw and semantic events to explain the system behavior at various higher levels. ”sequential file read”, ”attempt to write to a closed file”, ”DNS request”, and ”half-opened TCP connection” are examples of synthetic events. Figure 3.1 shows the relations of these three event types.

In order to support different versions of trace formats, we propose a generic synthetic event generator that uses a set of semantic events rather than versioned raw events. Indeed, having different versions of kernel tracers as well as an evolutionary Linux kernel that leads to different trace formats, make it difficult to have a stable version of an analyzer module. It means that, for each new release of kernel or the tracer and also for any change in the trace events format, the abstraction module will have to be updated. However, by designing a generic tool, independent of kernel versions and trace formats, we can achieve a generic abstraction module that will not be dependent on certain kernel or tracer version. Semantic events help in this way : we define a semantic ”Open File” event instead of events of both the sys_open and sys_dup Linux system calls. In the Linux kernel, there is often no unique way to implement user-level operations. Thus by grouping the events of similar and overlapping functionalities, we reach a new set of semantic events that will be used in the synthetic event generator. Examples of such semantic events have been outlined in table 3.1.

Later, for the trace formats, one must simply update the mapping table for converting the raw events to semantic events. With this table, the synthetic event generator will work without the need to be updated for the new trace formats, being independent of the specific format and version.

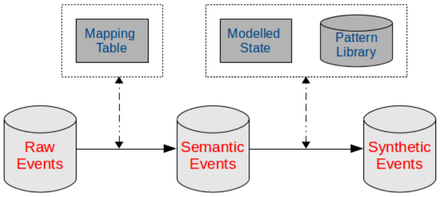


Figure 3.1 raw, semantic and synthetic events and conversion between them

Table 3.1 Raw Events To Semantic Events

Raw Events	Semantic Events
sys_open event sys_dup event sys_create event	file open
sys_read event sys_pread64 sys_readv event	file read
sys_write event sys_pwrite64 event sys_writev event	file write
sys_kill event sys_tkill event sys_tgkill event	process kill

Another technique to make the synthetic event generator more generic and powerful is to use the system state values. In this work, we use system state values besides the trace events for extracting the high-level information. As discussed in the related work, most of the abstraction techniques use patterns over raw and high-level events to generate abstract events. However, generating some complex types of abstract events can be very time consuming, and can affect system performance. It is also difficult to generate some complex types of synthetic events that deal with several system resources or under different user identities and at different levels. Thus, using patterns of trace events is somehow not enough and there is a need to extract and use more system information. To do so, we model state values of important system resources in a database named ”modeled state”. This database contains current and historic state values of the system resources, keeping track of information about running processes, the execution status of a process (running, blocked, waiting), file descriptors, disks, memory, locks, and other system metrics. The modeled state can then be used to show users the current system states. For example, each scheduling event which sets the current running process in the modeled state will be readily available for the upcoming events. Similarly, file open events can associate filenames to file descriptors in the modeled state. These values can then be used to retrieve the filename of the given file descriptor in the context of the upcoming file operation events.

3.5 Architecture

In this section, we explain the architecture of the proposed solution which is shown in Figure 3.2. It consists of various modules : event mapper, modeled state and synthetic event generator. In the following, we will explain how each module works.

3.5.1 Mapper

In the architecture, the event mapper is used to convert the trace raw events to environment-independent semantic events and also to extract the state values. The mapper actually has two steps : converting the raw events to semantic events and converting the semantic events to state changes. For each step, it uses a different mapping table. The first table is used for converting the raw events to the semantic events and the second one has been used to convert the semantic events to corresponding state changes. It makes use of two mapping tables that contain a list of conversion entries for each event type. There is not necessarily a one-to-one relationship between the raw events and corresponding outputs. A raw event or a group of raw events can be converted into one or more semantic events. For example, a Linux_sched_schedule(p1 , p2) event may be mapped into two semantic events : process_stop(p1) and process_running(p2). Thus, when there is a one-to-many relationship, for one input event two or more outputs or state changes could be generated. Table 3.2 lists examples of these mapping entries. The mapper in Figure 3.2 includes both the mapping tables 1 and 2. In other words, when events are processed in the mapper, corresponding semantic events are generated, and changes in the modeled state are also occurred. Most of the changes take place by directly changing a state value. However, some events may have a more complex effects on the modeled state. In this case, a set of changes may be queued to be performed on the state values.

Table 3.2 Semantic Events To State Changes

Semantic Events	Corresponding State Change
file open(fd)	changes the state of the input fd to opened
file read(fd, count)	changes the state of the input fd to read
file close (fd)	changes the state of the input fd to closed
kill process (p1)	changes the state of the input p1 to killed

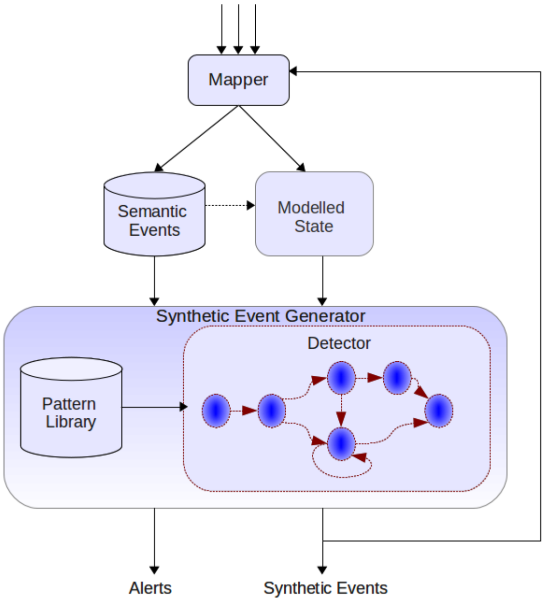


Figure 3.2 architectural view of the stateful synthetic event generator

3.5.2 Modeled State

The modeled state stores the system metrics and different state values of them. The system metrics (e.g. process name, file name, etc) are stored in a tree based data structure called ”attribute tree” [102] or ”metric tree”. In this tree, each metric has an address starting from its machine name. In this tree, the hierarchy of the resources (e.g., system –> virtual machines –> processes [or files or network nodes] –> states) is defined statically but its real instance is generated dynamically, in the trace reading phase. The content of the attribute tree is similar to the tree shown in Figure 3.3, where names starting with $ identify the variables and can have many values. For example $pid can be process1 , process2 , ...

The tree shown in Figure 3.3, acts like a lookup table in which various system resources and attributes are defined in a file system like path. Besides that, there is another tree based interval database to store the different values of those resources and attributes during the system execution. We store all of the extracted state values in this database that enables us to retrieve the state values at any later given time. Alexandre et al. [102] proposed a Java implementation of the modeled state that is used in this project to store and retrieve the state values.

Different types of data may be stored in the modeled state : the state of a process, the current running process on a CPU, state of a file descriptor and so on. System resources statistics are another types of information that can be stored in the modeled state. Examples of these statistics include : number of bytes read and written for a specific file, for a process or

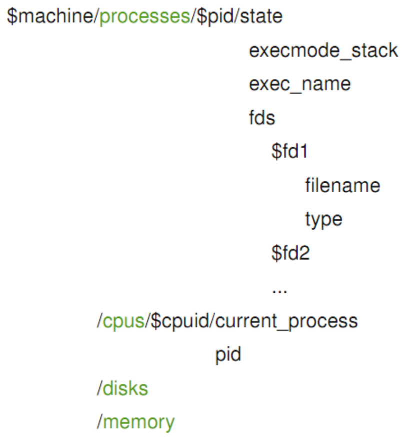


Figure 3.3 typical organization of the modeled state elements

for the whole trace, total CPU usage per process or per trace or for a specific time period, the CPU waiting time, number of TCP connections in the last 2 seconds, the duration for which a disk was busy, large data transfers, etc. Statistics extracted from trace events are similar to the extraction of the state values from events. By processing trace events, one can interpret the event contents, gather relevant statistics, and aggregate them by machine, user, process or CPU for the whole trace or for a time duration. For that, we identify the event types and event arguments used to count, aggregate and compute the statistics of the system metrics. In other words, when one defines a mapping between events and states, in the same way he or she can also define a relation between the trace events and the corresponding statistics values.

The statistics can then be used to generate synthetic events to detect system faults and problems. In this project, we use a threshold detection mechanism [42, 128] to detect system problems. In this approach, the occurrences of specific events and statistics values of important system metrics are stored, updated, and compared to predefined threshold values. If the values cross the thresholds, or in the case of a quick rise in a short period, an alarm is raised or a log record is generated [42]. With this approach, some hook methods may be registered and invoked in the case of unusual growth or the reaching of certain threshold. For some predefined semantic events, we store and update the statistics values of the important metrics (e.g. quantity of I/O throughput, number of forks, number of half-opened TCP connections, CPU usage, number of file deletions, etc) in the modeled state.

It is important to mention that system state is a broad term and it could be too resource and performance consuming to care of all system resources. For trace abstraction, we only need to store a subset of that information required to represent the synthetic event patterns and associated initial, intermediate and final states. In other words, the amount of data stored in the modeled state will depend on the patterns and will be extracted from them. Therefore, by having created a pattern library in advance or by importing them during analysis, it would be possible to determine the required set of system attributes and metrics that should be kept track of, in the modeled state.

3.5.3 Synthetic Event Generator

Synthetic event generator is another module used to generate high level events from trace events. The synthetic event generator may use either the semantic events, modeled state values, or both to generate the synthetic events. It makes use of a pattern library that contains various patterns for reducing the trace size and also for detecting the system faults and attacks.

In this paper, we use finite state machine to define the patterns. In this way, we have created a set of state machines based on the semantic events and state values to abstract out the Linux kernel execution events. In addition, it contains patterns to detect Syn flood attack, fork bomb attack and port scanning. Each state machine represents a set of states and a sequence of actions and transitions. A transition is triggered by reading an associated semantic event or a state value change. Reaching a particular state targets a synthetic event that can reveal either a high-level system behavior or a system fault or a misbehavior. In this model, we use the modeled state to store both the states of system resources as well as the state machines' intermediate states. Based on this idea, common methods are used for storing, retrieving, exploring the states.

As an example, suppose we want to list all ”write to a closed file” synthetic events. In this case, by comparing the filename of each write event to the open files list kept in the modeled state, we can generate and list all of these synthetic events. Here, we actually have a pattern of ”write to file” semantic events and a specific path in the modeled state (open files list). Another example of such a pattern is when we want to detect all sequential file reading operations. In this case, for each read event, we keep track of the last read position as well as whether all previous read operations were sequential or not. Then, upon closing a file, if the state values for all previous read operations were sequential, and the last read operation has read the file to the end, then a ”sequential file read” synthetic event can be generated. In the same way, a pattern is defined over ”file read” semantic events and a relevant modeled state values. Please note that, all mentioned state definitions are in fact the different possible status values of files or processes or other system resources and can generally fit in our previously defined modeled state system. In other words, the modeled state is defined in a way that it can model and store all required state values of various system resources.

The resulting synthetic events are again passed to the mapper. The Mapper may have mapping entries for synthetic events as well, which means that, in the same way that semantic events change the state values, synthetic events can affect them. For instance, a "TCP Connection" synthetic event can change the state of a socket to connecting, established, closing, and closed. Previously generated synthetic events may also be passed again through the synthetic event generator, which means that they can be used to generate complex higher-level events. For example, one can generate "Library files read" synthetic event from the consecutive "read */lib/* files" synthetic events. Figure 3.4 depicts this issue.

Another result is that the stateful approach enables the analyst to seek and go back and forth in the trace, select an area, and abstract out only the selected area. In other words, it supports partial trace abstraction. For instance, suppose we see there is a high load at a specific point, we can jump to the starting point of the selected area, load the stored system information and run abstraction process to get meaningful events and to achieve a high-level understanding of the system execution.

3.6 Illustrative Examples

Using patterns of semantic events and system states help to develop a generic synthetic events generator. This way, we are able to generate more meaningful high-level events and to detect more system faults and problems. Here we provide a few examples that outline aspects of synthetic events we can generate using the proposed method.

3.6.1 System load and performance

By keeping track of the system load and usage (e.g. CPU usage, I/O throughputs, memory usage, etc), and aggregating them per process, per user, per machine and per different time intervals, it becomes easy to check the resource load values against predefined threshold values. Thus, by processing the trace events and having defined standard patterns, we can compute the system load and store in the modeled state. For example, for each file open semantic event, we increment the number of opened files for that process and also for the whole system. Likewise, we decrement the number of opened files for each file close semantic event. In the same way, for each schedule in and out event, we add the time duration to the CPU usage of that process. We perform the same processing for memory usage, disk blocks operations, bandwidth usage and so on, and update the corresponding values in the modeled state.

The detector module then compares the stored values against predefined thresholds and detects whether an overload exists in the system. In case of overload, a "system overload" synthetic event would be generated and an alarm would be raised to the system administrator

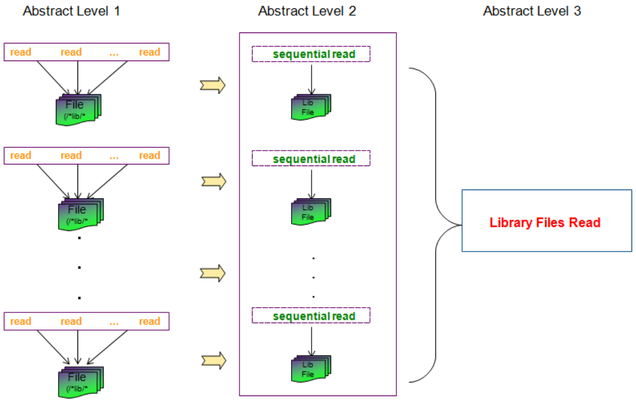


Figure 3.4 generating several levels of synthetic events

or any predefined monitoring systems. The Administrator or the monitoring system, in turn, responds appropriately to the problem. This solution can be extended to all types of system load and performance problems.

3.6.2 Denial of service attacks

The proposed stateful method can also be used to detect denial of service (DOS) attacks. For instance, a ”Fork Bomb” is a form of denial of service attack which clones a large number of child processes in a short time, in order to saturate the system resources. To detect this attack, one can store the number of fork operations for each process in the modeled state. In this case, upon forking a process, value of a corresponding counter is incremented and upon killing a process, the value is decremented. Each time the value changes, it is compared to the predetermined threshold value and in case the threshold value is reached, it will generate a synthetic event and will send an alarm to the system monitoring module. Figure 3.5 outlines the state machine used for detecting these kinds of attacks.

The same technique may be used for detecting the ”Syn flood attack” and for other similar DOS attacks as well. For each connection (even half-opened or established) we keep track of a counter in the modeled state and the attack (or a possible attack attempt) is detected by comparing the value of the counter to a predefined value. The predefined value can be defined by an expert or by an automated learning technique. These values can be adjusted for different servers and applications.

3.6.3 Network scan detection

Network scanning -especially port scanning- is the process in which a number of packets are sent to a target computer to find weaknesses and open doors to break into that machine.

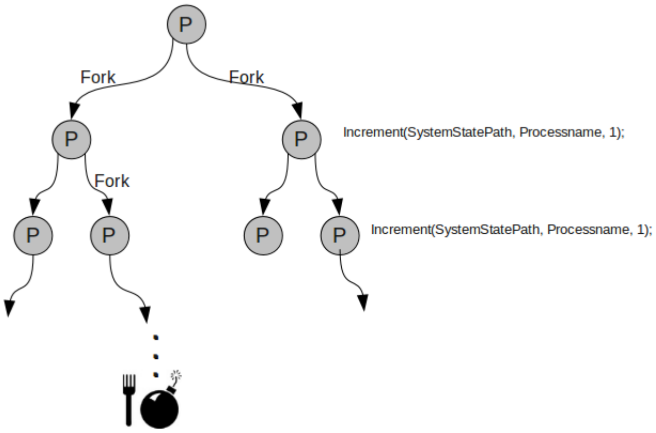

P
Fork
Fork
P
P
Increment(SystemStatePath, Processname, 1);
Fork
P
P
P
P
Increment(SystemStatePath, Processname, 1);

Figure 3.5 state transition for detecting the fork bomb attack

The scanner sends a connection request to targeted ports and, depending on the scanning method, receives a list of open ports, version of operating system, and running services on the other end. Port scanning has legitimate uses in computer networks and is usually considered one of the early steps in most network attacks. Nmap [1] is the most widely used tool for such network scanning. We used Nmap to generate the relevant kernel traces of high level port scanning. There is actually no way and no reason to stop a user from scanning a computer. However, by detecting port scanning, one can be alerted because of a potential attack.

There are many ways to perform and accordingly detect a port scanning from network packets. A common method, which is implemented in Nmap, involves establishing a typical TCP connection and immediately closing it by sending a RST or a FIN packet and repeating it to different ports at defined time intervals. Another hard to detect port scanning method is sending a dead packet (a typical TCP SYN or TCP FIN instead of a regular TCP connection). According to RFC 793 [2] an open port must ignore this packet and conversely, a closed one must reply with a RST packet. Consequently, any answer from the other end will determine the status of the port : whether it is open, closed or filtered.

The LTTng kernel tracer traces the packets in both the IP and TCP layers. However, the proposed prototype uses socket-related events (TCP layer) to detect ”port scanning” synthetic events. Simplifying the prototype, we consider every TCP connection pattern -even established or half opened- followed by a disconnection, a single port scan. Upon detecting these kinds of events, we update an associated entry in the modeled state. The scanned port number, pattern used, source address, source port, and timestamps of the first packet are stored in that entry. There is a registered method in the detector module which monitors the associated modeled state entry and, in case of detecting a successive port scanning, generates a ”ports scanning” synthetic event and raises an alarm to the system monitoring

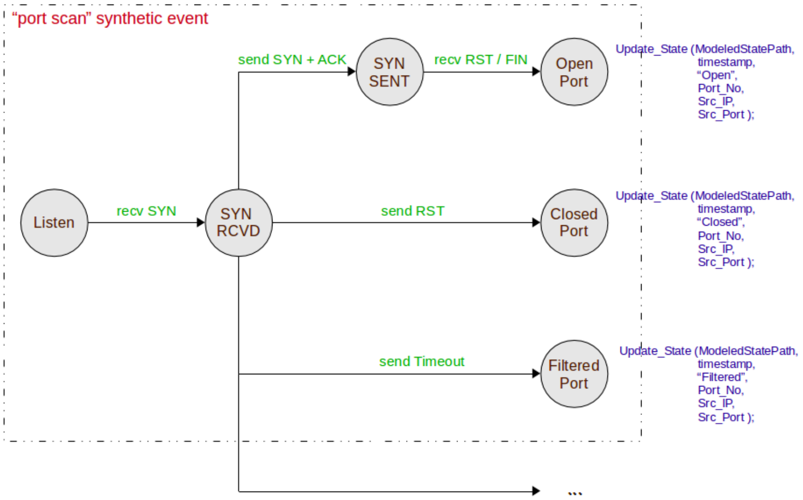

"port scan" synthetic event
Listen
recv SYN
SYN RCVD
send SYN + ACK
SYN SENT
recv RST / FIN
Open Port
Update_State (ModeledStatePath, timestamp, "Open", Port_No, Src_IP, Src_Port);
send RST
Closed Port
Update_State (ModeledStatePath, timestamp, "Closed", Port_No, Src_IP, Src_Port);
send Timeout
Filtered Port
Update_State (ModeledStatePath, timestamp, "Filtered", Port_No, Src_IP, Src_Port);
...

Figure 3.6 state transition for detecting the port scanning

module. Due to the completeness of the defined rules and patterns, this technique can detect distributed port scanning even with different timings between packets. Figure 3.6 depicts the state machine used to detect ”port scanning” synthetic events.

3.7 Implementation

We have prototyped a Java tool in Linux, that takes the LTTng trace as input and applies the proposed techniques to create the several levels of abstract events. We use the Java library provided by Alexander et al. [102] to manage the modeled state and other shared information. Our synthetic event generator stores all the system states, as well as the states required for state machines in this tree.

We also created a prototype pattern library that covers common Linux file, network and process operations. Using this pattern library and the proposed method, we can generate various levels of file operations (e.g. file simple open and close operation, sequential and non sequential file read and write, etc) network connections (e.g. TCP connection, Port scanning, etc), and process operations (Fork process, Kill process, etc). The pattern library also includes patterns of some system problems and attacks.

In this implementation, we have defined the patterns in XML format. Also we hardcoded the pre-actions and post-actions for each state transition. Wally [142] has developed a language for defining patterns and attack scenarios over LTTng events. Efforts are needed to extend this language to support the patterns of system states. Generally, the required language should be a declarative language with which one can declare the patterns and scenarios and mapping between events and outputs. It can, however, also be a programming-like language with which one can define state variables, pre and post transition actions, conditions, output formats and etc. Designing such a language is a future work of this project.

We have used a C helper program that simply calls the ”wget WWW.YAHOO.COM” in Linux to generate the corresponding kernel traces. Figure 3.7 shows the highest level of generated synthetic events for this command. The count of generated raw events for this command is 3622. The synthetic event generator has converted these 3622 raw events into less than 10 synthetic events including : ”Library File Read” and ”Configuration Files Read”, ”TCP Connection to WWW.YAHOO.COM” and sequentially writing the fetched data to file index.html.1, followed by some other events.

3.8 Performance

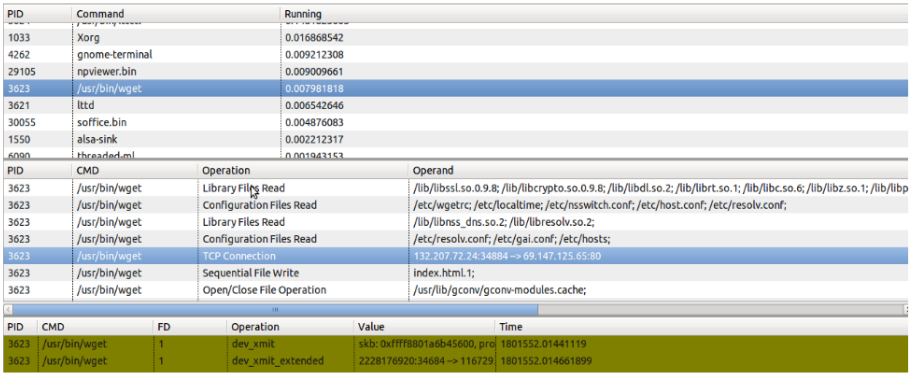
For the performance testing, the Linux kernel version 2.6.38.6 is instrumented using LTTng and the tests are performed on a 2.8 GHz with 6 GB ram machine. We will show the results from different points of view. As discussed in the Implementation section, we have generated two other levels of abstraction in addition to the semantic events. For the first abstract level, we have used 120 patterns and, for the second level, we have used 30 patterns. Following tables and figures show the results for traces with different sizes from 25 MB to 10000 MB. There is no predefined database of trace events and they should be generated based on the use-case and application. To generate these trace files, we have used ”wget -r -l0 URL”, ”ls -R” and also ”grep x . -r” commands consecutively.

3.8.1 Reduction ratio

As discussed earlier, one goal of the abstraction technique is reducing the trace size. We have measured the reduction ratio of the proposed method. Figure 3.8 and table 3.3 show the reduction of the number of events in the different levels of abstraction.

Table 3.3 Number of events in different abstraction levels

Size (MB)	Number of Events		
	Original trace	Abstract Level 1	Abstract Level 2
25	2279766	335362	4954
75	5420727	710052	5466
150	8872888	976672	86697
500	37328387	7668926	178426
1000	68961889	20186771	192788
2000	140507496	33924846	328430
5000	328868336	130293720	2099836
10000	621132167	159023500	2247225



PID	Command	Running
1033	Xorg	0.016868542
4262	gnome-terminal	0.009212308
29105	npviewer.bin	0.009009661
3623	/usr/bin/wget	0.007981818
3621	lttd	0.006542646
30055	soffice.bin	0.004876083
1550	alsa-sink	0.002212317
6090	threaded-ml	0.001943153

PID	CMD	Operation	Operand
3623	/usr/bin/wget	Library Files Read	/lib/libssl.so.0.9.8; /lib/libcrypto.so.0.9.8; /lib/libdl.so.2; /lib/librt.so.1; /lib/libc.so.6; /lib/libz.so.1; /lib/libp
3623	/usr/bin/wget	Configuration Files Read	/etc/wgetrc; /etc/localtime; /etc/nsswitch.conf; /etc/host.conf; /etc/resolv.conf;
3623	/usr/bin/wget	Library Files Read	/lib/libnss_dns.so.2; /lib/libresolv.so.2;
3623	/usr/bin/wget	Configuration Files Read	/etc/resolv.conf; /etc/gai.conf; /etc/hosts;
3623	/usr/bin/wget	TCP Connection	132.207.72.24:34884 --> 69.147.125.65:80
3623	/usr/bin/wget	Sequential File Write	index.html.1;
3623	/usr/bin/wget	Open/Close File Operation	/usr/lib/gconv/gconv-modules.cache;

PID	CMD	FD	Operation	Value	Time
3623	/usr/bin/wget	1	dev_xmit	skb: 0xffff8801a6b45600, pro	1801552.01441119
3623	/usr/bin/wget	1	dev_xmit_extended	2228176920:34684 --> 116729	1801552.014661899

Figure 3.7 a view of the implemented stateful synthetic event generator

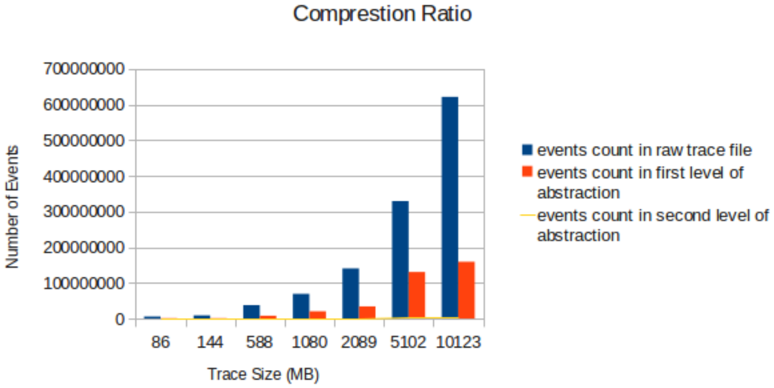


Figure 3.8 reduction ratio

Table 3.3 shows the numbers of events at each abstract level. For instance, by applying the synthetic event generator over a 10 GB trace file, we could reduce it to a trace with 25 % of the original size but still with mostly the same meaning. It is important to note that the content of trace files at different levels should yield the same interpretation of the system execution. In the Implementation section, we showed that the same meaning can be extracted from different levels of abstraction. Using same pattern matching technique, the reduction ratio is related to the number of patterns and also the number of containing events in each pattern. For that, the reduction ratio of the different examined traces are not the same.

3.8.2 Patterns and containing events

Table 3.4 shows the number of selected synthetic events of various trace sizes for the first abstract level. In the proposed pattern library, a set of patterns has been defined for each of these synthetic events so that by applying them over trace events and by retrieving the current system state, the engine is able to detect those events. We have defined 120 patterns for abstracting out the trace events and generating the first level of abstraction. These patterns cover important aspects of file, socket, process and also system wide operations. The patterns are stored in a XML file and the system is defined in such a way that administrators can easily add new patterns to the system.

As the second step, Table 3.5 shows some important synthetic events for the second abstract level. To generate these events, a set of patterns were defined over the events of the first level of abstraction and also the relevant modeled state values. For instance, "Check File" synthetic events contain a sequence of an open file, check file status or check file permission and a close file. Also "Sequentially File Read (Write)" refers to a set of subsequent file operations that consists of a file open, read from (write to) a file sequentially and finally close it. A "HTTP (DNS) Connection" is detected through a pattern of socket create, socket connect to a 80/53 port with transport protocol equal to TCP (UDP), zero or more send/receive data, and finally socket close events. However, there are situations that the engine can not detect the type of the connection, therefore a general "Network Connection" synthetic event is generated. This event means that there is a connection (sequence of socket create - socket connect, send/receive data, close socket), but the engine could not find the type of the connection. This happen because of probable missing events. It may also occur if the connection was already established at the starting time the trace. At this level, we have defined 30 different patterns for detecting such kind of synthetic events.

It would be interesting to know the number of events participating to form a synthetic event on one upper level. Table 3.6 shows the average number of events from one level below contained in each synthetic event.

As shown in Table 3.6, "Check File" synthetic events always contain three events : open a file, check a file attribute and close that file. It is important to note that the numbers here only show the containing events from one level below. On the other hand, in this example, each of these three events can in turn contain several events from a lower level. As another example, the average number of 48 events obtained from a 5 GB trace means that each "Sequentially File Write" synthetic event contains 48 events on average ; one for open file, one

Table 3.4 Count of different event types in first level of abstraction

Events Count	Size (MB)	Number of synthetic events in first level of abstraction								
		File Operations				Network Operations				
		File Open	File Read	File Write	File Close	Socket Create	Socket Connect	Socket Receive	Socket Send	Socket Close
2279766	25	2727	8401	12913	2327	150	112	4583	7911	150
5420727	75	2474	18563	30251	2122	370	718	21523	26570	611
8872888	150	86780	59108	20913	86536	154	106	7574	10767	161
37328387	500	87484	158484	1025703	88143	673	979	60880	70651	1070
68961889	1000	98583	218789	6507052	96226	159	73	57965	62003	168
140507496	2000	161458	562239	5980178	161577	2140	2500	166420	245718	2638
328868336	5000	1045710	612821	23001032	1044562	4592	5154	137733	296566	5356
621132167	10000	755647	3541833	23214444	720016	27676	20741	735271	1288181	29059

Table 3.5 Count of different event types in second level of abstraction

Events Count	Size (MB)	Number of synthetic events in second level of abstraction						
		File Operations				Network Operations		
		Check File	Sequentially File Read	Sequentially File Write	Read Write File	Network Connection	HTTP Connection	DNS Connection
2279766	25	928	673	496	230	43	103	4
5420727	75	1117	742	253	10	372	193	46
8872888	150	66141	19742	440	213	58	103	0
37328387	500	53371	32093	1791	888	688	274	108
68961889	1000	81607	14343	207	69	139	29	0
140507496	2000	61647	96921	2505	504	1486	867	285
328868336	5000	948436	87095	8047	984	1921	3116	319
621132167	10000	395067	303452	20267	1230	11740	16668	651

Table 3.6 Average number of containing events for generating the upper level synthetic events

		Average number of containing events						
Events Count	Size (MB)	Check File	Sequentially File Read	Sequentially File Write	Read Write File	Network Connection	HTTP Connection	DNS Connection
2279766	25	3	3	10	9	1	38	5
5420727	75	3	3	25	87	3	37	5
8872888	150	3	4	11	9	2	38	0
37328387	500	3	4	10	26	2	44	5
68961889	1000	3	3	20	67	4	82	0
140507496	2000	3	4	61	16	5	92	5
328868336	5000	3	6	48	62	3	34	5
621132167	10000	3	4	34	120	4	42	5

for close file and the rest for write events. The null value for the DNS connection is because there is no such connection for those traces.

3.8.3 Execution time

As discussed earlier, one of the important feature of proposed method is its execution efficiency. In this section, execution times for different trace sizes will be shown. Table 3.7 shows the time spent to read the relevant events, to look for patterns and to generate the first abstraction level.

The numbers in table 3.7 show the time spent for reading the events, looking for the patterns and generating the abstract events. The execution time for checking each pattern has a relatively minor impact when checking all patterns simultaneously. Reading the trace events and finding relevant events for each pattern takes most of the analyzing time.

Figure 3.9 shows the differences between execution times spent for different levels of abstraction.

Table 3.7 execution time for generating the first abstraction level

Events Count	Size (MB)	All(ms)	Time spent (ms) for analysing and generating the first level of abstract events								
			File Operations				Network Operations				
			File Open	File Read	File Write	File Close	Socket Create	Socket Connect	Socket Receive	Socket Send	Socket Close
2279766	25	4748	3963	4024	3995	4038	4151	3952	3871	3937	3777
5420727	75	9176	5900	6195	6394	6218	6394	6257	5884	6014	5957
8872888	150	12735	10403	10308	10183	10282	9975	10025	97125	10125	10424
37328387	500	58477	50251	50855	51928	50424	49366	47896	48962	49163	48104
68961889	1000	148284	128252	129157	138988	128186	127912	128312	127191	126915	128061
140507496	2000	226569	177913	180017	187366	178640	177389	174385	179372	178509	179097
328868336	5000	650228	513230	514012	546782	512659	510139	507873	511201	510139	512771
621132167	10000	1105124	902245	916292	941668	909893	907177	901127	899976	904227	874227

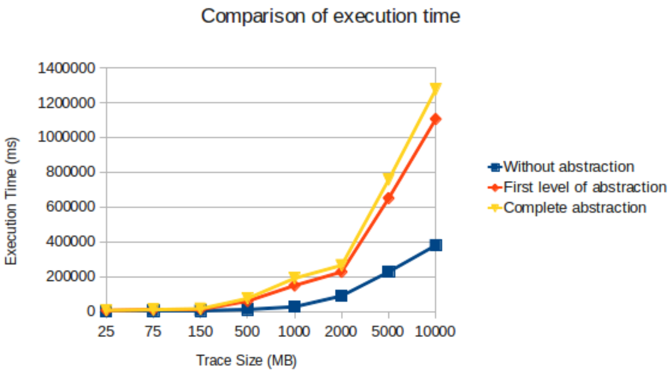


Figure 3.9 execution time comparisons

In Figure 3.9, the blue line shows the execution time needed for reading the trace events. For this step, no pattern was specified and the diagram only shows the time needed for reading all trace events. In the same way, the orange line shows the execution time for generating the first level of abstract events. The execution time, consists of reading events, looking up the patterns and generating the abstract events. The yellow line depicts the time needed for analyzing and generating both levels of abstract events. Differences between the orange and yellow lines with the blue line show that the time needed for analyzing and generating the first level of abstraction is more than the time needed for the second level. This is explained by the fact that the number of patterns in the first level is much greater than the number of patterns in the second level. The results show that the execution time is relatively linear with the trace size and also the number of relevant events. However, there exists other factors that affect the results. The complexity of patterns, and also the number of containing events for each pattern may lead to different execution times for different synthetic events. For example, in Table 3.7, we see different values for different synthetic events for the same trace file and the reason is that they have different scenarios with different complexities and different numbers of containing events.

Another important factor is the number of coexisting patterns. As shown in Figure 3.9, the number of patterns and the number of coexisting patterns affects the execution times for the two different abstraction levels. Since the first level deals with file and socket operations, they need to be called for each file/socket access, thus having a large impact on the performance and execution time of the analyzer. By contrast, in the second abstraction level, the analyzer works with fewer coexisting patterns, less often called during a process lifetime ; thus, less time is needed for analyzing and generating the second level of abstract events. The execution time is therefore related to the size of the trace files, the number of relevant events, the number of coexisting patterns and also the complexity of patterns.

3.9 Conclusion and Further Work

In this paper, we proposed an abstraction method that uses a set of patterns over semantic trace events and system state values to generate synthetic events. Using the notion of semantic events (the events of a generic type that replace platform-dependent events) can help decouple the synthetic event generator from the system and tracer-dependent events. Using semantic events as well as system state values makes the abstraction process more generic to supports different versions of trace formats, operating system kernels and also to support both the kernel and user level tracing. As shown in the Illustrative Examples section, using proposed techniques we efficiently generate a wide range of synthetic events that reveals more of the system behaviors and can also be used to detect a larger range of system problems.

Although most of the synthetic events can be defined by patterns and state machines, we do not support the synthetic events that are not representable using state transitions. For example, a dependency analysis between different processes and resources, leading to the computation of the critical path for a request, cannot be defined as a simple pattern. Another possible limitation is related to the output completeness of the tracers. Because not every state modification is logged with a tracer, this may somehow limit the proposed technique.

Using the proposed method and having defined a complete pattern library can also lead to an efficient host-based intrusion detection system. In our project, expanding the rule base and pattern library is an important future work. We have implemented a prototype pattern library that works over semantic events and system states. However, more work is needed to complete it and to support more aspects of system behavior. Examples of the patterns we should extend include memory usage and inter-process communications.

As mentioned in the Implementation section, there is a need to further develop the supporting language. This language can be declarative or similar to a programing language and should support the requirements needed to implement the proposed method. It could be used for defining the mapping table between raw events and semantic events, as well as the mapping between events and state changes. We will focus on extending the language defined in [142] in a future investigation.

CHAPTER 4

Paper 3 : A Framework to Compute Statistics of System Parameters from Very Large Trace Files

Naser Ezzati-Jivan and Michel Dagenais

4.1 Abstract

In this paper, we present a framework to compute, store and retrieve statistics of various system metrics from large traces in an efficient way. The proposed framework allows for rapid interactive queries about system metrics values for any given time interval. In the proposed framework, efficient data structures and algorithms are designed to achieve a reasonable query time while utilizing less disk space. A parameter termed granularity degree (GD) is defined to determine the threshold of how often it is required to store the precomputed statistics on disk. The solution supports the hierarchy of system resources and also different granularities of time ranges. We explain the architecture of the framework and show how it can be used to efficiently compute and extract the CPU usage and other system metrics. The importance of the framework and its different applications are shown and evaluated in this paper.

Keywords : trace, statistics, system metrics, trace abstraction, Linux kernel

4.2 Introduction and Problem Statement

The use of execution traces is increasing among system administrators and analysts to recognize and detect problems that are difficult to reproduce. In a real system, containing a large number of running processes, cores and memory modules, when a runtime problem occurs(e.g. a performance degradation), it may be difficult to detect the problem and discover its root causes. In such a case, by instrumenting the operating system and applications, tracer tools can provide valuable clues for diagnosing these failures.

Although trace information provides specifics on the system's runtime behavior, its size can quickly grow to a large number of events, which makes analysis difficult. Thus, to facilitate analysis, the size of large traces must somehow be reduced, and high level behavior representation must be extracted. Trace abstraction techniques [49, 56, 142] are used to reduce the size of large traces by combining and replacing the raw trace events with high level compound events. By using trace abstraction, it is possible to create a hierarchy of high level events, in which the highest level reveals general behaviors, whereas the lowest level reveals more detailed information. The highest level can start with the statistics view. This statistics view can be used to generate measurements of various system metrics, and create the general overview of a system execution in the form of histograms or aggregated counts. It is also possible to use other mechanisms (e.g. navigation and linking) to focus on a selected area, go deeper, and obtain more details and information.

Statistics play a significant role in every field of system analysis, especially in fault and attack detections. The overview of different system parameters, such as the CPU load, number of interrupts, IO throughput, failed and successful network connections, and number of attack attempts can be used in various applications that range from optimization, resource utilization, bottleneck, fault and attack detection to even benchmarking and system comparisons. In short, inspecting the statistics of several system metrics may be used in the earlier steps of any trace based system analysis.

Since the statistics computation may be used interactively and frequently for large traces, it is worth having efficient data structures and algorithms to compute system metrics statistics and parameters. These data structures and algorithms must be optimized in terms of construction time, access time and required storage space.

The simplistic solution for providing the required statistics is to take a trace, read the events of the given interval, and compute the desired statistics. However, it is not recommended to have the trace events read each time, especially when the trace size is large, as reading trace events of large intervals is inefficient, and may waste valuable analysis time. The problem of statistics computation in general faces two main challenges : first, the difficulty of efficiently computing the system metrics statistics without having to reread the trace events ; and second, finding a way to support large traces. In other words, on one hand, the challenge lies in finding a way to compute -in a constant time- the statistics of various system metrics for any arbitrary time interval, without rereading the events of that interval (e.g. computing the system metrics statistics of a 20 GB interval in a 100 GB trace short of rereading that particular 20 GB of the trace). On the other hand, the next challenge then becomes providing a scalable architecture to support different trace sizes (from a few megabytes to over a terabyte of trace events), and at the same time, different types of statistics and hierarchical operations.

In this paper, we propose a framework for incrementally building a scalable metrics history database to store and manage the precomputed system metrics values, used to rapidly compute the statistics values of any arbitrary intervals.

The framework is designed according to the following criteria :

1. performance, in terms of efficiency in statistics computation and query answering algorithms,

2. compactness, in terms of space efficiency of the data structures and finally,
3. flexibility, in terms of supporting different system metrics (e.g. IO throughput, CPU utilization, etc.), and hierarchy operations for different time scales (i.e. millisecond, second, minute and hour) and system resources (e.g. a process, a group of processes, a virtual machine, or the whole system).

To test the proposed data structures and algorithms, we use kernel traces generated by Linux Trace Toolkit next generation (LTTng) [38]. LTTng is a low impact, precise and open source Linux tracing tool that provides detailed execution tracing of operating system operations, system calls, and user space applications [38]. By evaluating the resulting trace events, this method automatically draws an overview of the underlying system execution, based on a set of predefined metrics (e.g. number of bytes read and written), which can then be used to detect system problems and misbehaviors caused by program errors, an application's misconfiguration or even attacks. Further investigations may lead to opportunity to apply some administrative responses to solve those detected problems [130].

The remainder of the paper is organized as follows : first, we present the architecture of the statistics framework, and the details of its different modules. Secondly, we present an example highlighting the use of the proposed framework, and subsequent data structures and algorithms. Then, we discuss our experiences and evaluation of the proposed method. Finally, we conclude by outlining specific areas of investigation for future enhancements.

4.3 Related Work

Bligh et al. [15] use kernel trace data to debug and discover intermittent system problems and bugs. They discuss the methods involved in debugging the Linux kernel bugs to find real system problems like inefficient cache utilization and poor latency problems. The interesting part is that for almost all investigated problems, inspecting the statistics of system metrics is the starting point of their analysis. Using trace data to detect and analyze the system problems also mentioned in [24, 25, 145]. Xu et al. [145] believe that system level performance indicators can denote high level application problems. To address such problems, Cohen et al. [24] first established a large number of metrics such as CPU utilization, I/O requests, average response times and application layer metrics. They then related this metrics to the problematic durations (e.g. durations with a high average response time) to find a list of metrics, indicative of these problems. The relations that Cohen et al. discovered can be used to describe each problem type in terms of a set of metrics statistics [25]. They denoted how to use statistics of system metrics to diagnose system problems. However, they do not consider scalability issues, where the traces are too large and where the storing and retrieving of statistics are key challenges.

Some research uses the checkpoint or periodic snapshot method to collect and manage the system statistics [36, 77]. This method splits the input trace into equal parts (e.g. a checkpoint for each 100 K events), and stores the aggregated information from each checkpoint in a memory based database. After reading a trace and creating the checkpoint database, for computing statistics of any given time point, the method accesses the previous checkpoint, rereads and reruns the trace from the previous checkpoint up to the given point, and computes the desired statistics. Kandula et al. [77] store snapshots of the system configurations in order to analyze how each component depends on the network system, and to determine the main cause of each system fault. LTTV (a LTTng viewer) and TMF (Tracing and Monitoring Framework) [1] use the checkpoint method for extracting system state values at any given time point.

Although the checkpoint method is considered a useful solution for managing the statistics, it requires rereading the trace and does not support the direct computation of statistics found between two checkpoints. Moreover, different metrics with varying incident frequencies (e.g. number of events vs number of multi step attacks) are treated in the same way. In other words, since the method uses equal size checkpoints for the metrics that have trifle value changes during a system execution, loading a checkpoint and rereading the trace to compute its statistics at different points may waste time, and not produce any values. In the same way, metrics with large incidents, demand more effort to recompute the required statistics. In this paper, we will show that creating variable length checkpoints for different metrics leads to better construction and access performance.

The checkpoint method uses memory-based data structures to store the checkpoint values. However, using a memory-based database imposes a strict limit on the trace size that can be supported. The same problem exists for other interval management data structures like the Tree-Based Statistics Access Method (TBSAM) [134], segment-tree, interval-tree, Hb-tree, R-tree (and its variants R+-tree and R*-tree), etc. [61]. Segment tree and interval tree work properly for static data sets, but do not work well for incrementally built intervals, because they lead to performance degradation. Likewise, the R-tree and its extensions do not work well for managing interval sequences that have long durations, and are highly overlapped, as indicated in [123]. Furthermore, the splitting and merging (rebalancing) of the nodes drive many changes in the pointers, inducing severe performance degradation.

In the DORSAL lab [2], co-workers Montplaisir et al. [102] introduced an external memory-based history tree for storing the intervals of system state values. In their history tree, system

[1] http ://lttng.org/eclipse

[2] http ://www.dorsal.polymtl.ca/

state values are modeled as intervals. Each interval in this tree contains a key, a value, a start and an end time. The key represents a system attribute whose state value is stored in this interval. The start and end times represent the starting and ending points of the given state value. In this tree each node corresponds to a time range and contains a bucket of intervals lying within the node's time range. Since they use a disk-based data structure to store the state information, the solution is scalable, and successfully tested for traces up to 1 TB, yet is optimized for interval queries, and has a fast query time. It takes O(log(n)) time (i.e. n equals the number of nodes) to perform a stabbing query to locate and extract a state value from the data store. The tree is created incrementally in one pass of the trace reading. The nodes have a predefined fixed size in the disk. Whenever a node becomes full, it is marked as "closed", and a new node is created. For that reason, it does not require readjustment as seen in the R-tree and its variants. Montplaisir et al. reported that they could achieve a much better query performance than the LTTV[3], TMF, R-tree and PostgreSQL database system [102].

They have designed that solution to store and manage the modeled state, but have not studied and optimized it for the statistics. Although the history tree, as they report, is an efficient solution for managing the state values, it can not be used directly in the statistics framework, as if it stores metrics as state, every increment will become a state change in an interval tree, which wastes much storage space. Our experiment shows that in this case the size of the interval tree is comparable to the original trace size, which is not reasonable. Although their partial history tree approach [102] works like a checkpoint mechanism, and solves the storage problems, the query time remains a problem, since it must access the original traces to reread and recompute the statistics for the points lying between two checkpoints ; that could be a time-consuming task for large checkpoints. Despite the same idea of handling the value changes as intervals behind both the history tree and the proposed framework, there are major differences that are explained in the following :

1. Granularity degree (GD) is introduced to make the data structure as compact as possible,
2. Different organization of the system resources and metrics is used to avoid duplication in the interval tree structure.
3. Rereading and reprocessing the trace events is avoided. Instead, the interpolation technique is used to calculate the half-way values.

[3] http ://lttng.org/lttv

4.4 General overview of the solution

Kernel tracing provides low-level information from the operating system execution that can be used to analyze system runtime behavior, and debug the execution faults and errors that occur in a productive environment. Some system runtime statistics can be extracted using system tools like prstat, vmtree, top and ps, however, these tools are not usually able to extract all the important information, necessary for a complete system analysis. For instance, they do not contain the operation timestamps, nor always the owning process information (e.g. which process has generated a packet), both of which are important in most system analysis. This information can be extracted from a kernel trace. Kernel traces usually contain information about [63] :

- CPU states and scheduling events, can be used to calculate the CPU utilization ;
- File operations like open, read, write, seek and close, can be used to reason about file system operations and extract IO throughput ;
- Running processes, their execution names, IDs, parent and children ;
- Disk level operations, can be used to gather statistics of disk access latencies ;
- Network operations and the details of network packets, can be used to reason about network IO throughput and network faults and attacks ;
- Memory management information like allocating or releasing a memory page, can be used to obtain and analyze the memory usages.

Since the kernel trace contains valuable information about the underlying system execution, having a mechanism to extract and render statistics of various system metrics, based on system resources and time data, can be helpful in finding system runtime problems and bugs. By providing such a statistics view, the trace analysis can start with an overview of the system, and continue by zooming in on the strange and abnormal behaviors (e.g. spikes in a histogram view) to gain more information and insight.

By processing the trace events, one can compute important system metrics statistics, and aggregate them per machine, user, process, or CPU for a whole trace or for a specific time range (e.g. for each second). The following are examples of statistics that can be extracted from a kernel trace :

- CPU time used by each process, proportion of busy or idle state of a process.
- Number of bytes read or written for each/all file and network operation(s), number of different accesses to a file.
- Number of fork operations done by each process, which application/user/process uses more resources.
- Which (area of a) disk is mostly used, what is the latency of disk operations, and what

is the distribution of seek distances.

- What is the network IO throughput, what is the number of failed connections.
- What is the memory usage of a process, which number of (proportion of) memory pages are (mostly) used.

Although each trace contains a wealth of information, it is not always easy to extract and use it. The first problem is the huge size of the trace. A large trace usually complicates the scalable reading and analysis. Another problem is that the statistics information is also not clearly displayed, and is hidden behind millions of events. This means the trace events have to be analyzed deeply to extract the desired statistics. Moreover, since the statistics computation will be used widely during system analysis, it must be fast and efficient enough to extract the desired analysis on demand. Therefore, tools and algorithms must be developed to deal with these problems.

In this paper, we propose a framework that efficiently provides statistical information to analysts. The framework works by incrementally building a tree-based metric data store in one pass of trace reading. The data store is then used at analysis time to extract and compute any system metrics statistics for any time points and intervals. Using a tree-based data store enables the extraction and computation of statistics values for of any time range directly, without going through the relevant parts in the original trace. Such a data store also provides an efficient way to generate statistics values from a trace, even if it encompasses billions of events. The architecture, algorithms and experimental results will be explained in the following sections.

This framework also supports hierarchical operations (e.g. drill down and roll up) among system resources (i.e. CPU, process, disk, file, etc.). It enables gathering statistics for a resource, and at the same time, for a group of resources (e.g. IO throughput for a specific process, for a group of processes, or for all processes). Furthermore, it supports different time scales, and it is possible to zoom in on the time axis to retrieve statistics for any time interval of interest.

The framework we propose also supports both online and offline tracing. In both cases, upon opening a trace, the framework starts to read and scan the trace events, precompute the predefined metric values, and store in the aforementioned interval history data store. Whenever an analysis is needed, it queries the data store, extracts the desired values and computes the statistics. With this system, users can go back to retrieve the statistics from any previous points of system execution.

4.5 Architecture

In this section, we propose the architecture of a framework for the live statistics computation of system metrics. The solution is based on incrementally building a metrics history database to be used for computing the statistics values of any arbitrary intervals in constant time. Constant time here means the independence of the computation time on the length of the interval. It works by reading trace events gathered by the LTTng kernel tracer and precalculating and storing values of the prespecified system metrics at different points in a tree-based data structure. Using tree-based data structures enables an efficient access time for large traces. Figure 4.1 depicts a general view of the framework architecture, covering its different modules.

As shown in Figure 4.1, this architecture contains different modules such as trace abstraction, data store and statistics generation, which are explained in the following sections :

Kernel Tracer : LTTng

We use the LTTng [38] kernel tracer to trace operating system execution. LTTng is a powerful, low impact and lightweight [133] open source Linux tracing tool, and provides precise and detailed information of underlying kernel and user space executions. LTTng contains different trace points in various modules of the operating system kernel, and once a predefined trace point is touched, it generates an event containing a time-stamp, CPU number and other information about the running process. Finally, it adds the event to an in-memory buffer to be stored later on disk [38].

Trace Abstractor

The large trace size makes difficult the analysis and understanding of the system execution. Most of the time another analysis tool is required to abstract out the raw events and represent them with higher-level events, reducing the data to analysis. Trace abstraction is typically required to compute statistics of complex system metrics that are not directly computable from the raw trace events. For instance, to compute synthetic metrics statistics like ”number of HTTP connections”, ”CPU usage”, and ”number of different types of system and network attacks”, raw events must be aggregated to generate high-level events ; then, the desired statistics must be extracted and computed. The details of the trace abstraction tool we use to generate such high level meaningful events from raw events, may be found in [49]. In the remainder of this text, the term event is used to refer to both raw and abstract events.

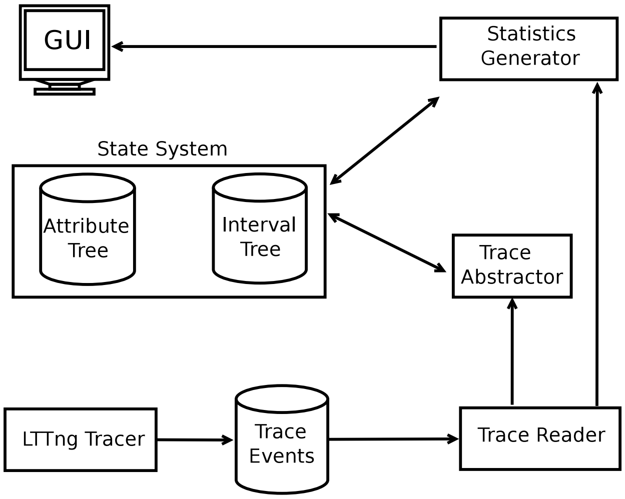


Figure 4.1 General architecture of the framework.

State System

The state system is a database used for managing the state values of a system at different points. Examples of state values are : execution status of a process (running, blocked, waiting), mode of a CPU (idle, busy), status of a file descriptor (opened, read, closed), disks, memory, locks, etc.

State values are extracted from trace events based on a predefined mapping table. In this table, there is an entry specifying how an event can affect the value of a resource state. For example, the state of a process (whether it is running or blocked) can be extracted using CPU scheduling events.

In the DORSAL lab[4], Montplaisir et al. [102] introduced a tree-based data structure, called state history tree, which stores and retrieves the system state values. In their history tree, system state values are modeled as intervals and each interval contains complete information about a state value change. For instance, when the state of a process is changing from ready to running, an interval is created in the history tree, specifying start and end points of the change, the process name and the state value.

The state system proposed in [102] uses two mechanisms for managing the state values :

- partial history tree, that makes use of a method similar to the checkpoint method, to store and manage the state values. In this system, for extracting state of any halfway points lying between two checkpoints, it is required to access the trace events to reread, rerun and extract the state value. However, another access and reread the trace, may

[4] http ://www.dorsal.polymtl.ca/

waste time. In our method, we read the trace once and will not refer to it again.

- complete history tree, that stores every change of state values and extracts directly any required state value. Although it extracts the state values directly without rereading the trace, the method needs lots of storage space. Experiment results show that in some occasions, related to the number of active system resources, it needs a storage space larger than the original trace size. For example, if each event causes a state change and wants to create a record in the state system, the resulting database can be larger than the original trace, considering the size of each record in the state system and the other overheads of the data structure.

  However, since the statistics may be used widely in the system, we need a compact data store, and at the same time, a faster access time.

Managing the statistics values of different system metrics can be implemented using the same mechanism as what is used for managing the states. Although the history tree introduced in [102] is an efficient solution for managing the state changes, it still needs some modifications to be used here in the statistics framework. In Montplaisir et al.'s work, any value change of the system state is stored in a separate interval. However, storing all statistics value changes in the interval tree will waste much storage space. As explained, our experiment shows that the size of the interval tree, in this case, will be comparable to the original trace size. Although using their partial history tree [102] that works like checkpoint mechanism, solves the storage problems, the query time remains a problem. What we look for here is a compact data store and efficient algorithms to directly compute the system metrics statistics, without having to reread and rerun the trace events.

4.6 Statistics Generator

The statistics generator, the main module of the framework, is responsible for computing, storing and retrieving the statistics values. Since in the kernel traces all data comes in the form of trace events, a mapping table is needed to extract quantitative values from the events. Similar to the event-state mapping, the statistics module uses an event-statistics mapping table that identifies how to compute statistics values from the trace events. In this table, there exists an entry that specifies which event types, and their subsequent payload are required for extracting metric statistics. For instance, for computing the number of ”disk IO throughput”, file read and write events are registered. In the same way, the ”HTTP connection” abstract events are counted to compute the number of failed/successful HTTP connections. The former is an example of a basic metric, computed using the raw events directly. However, the latter is an example of a synthetic metric, computed using outputs of the abstracter module.

In this framework, we store the statistics values in interval form. To do so, in the trace reading phase the duration of any value change in a system metric is considered an interval, and is stored in the data store. The time-stamp of the first event (registered to provide values of the metric) is considered the starting point of the interval. In the same way, the time-stamp of the next value change is considered the end point of that interval. Similarly, any other value change is kept in an other interval.

The parameter ”granularity degree” (GD) is defined to determine how often the computed statistics of a metric should be stored in the database. It does not however affect the computation frequency of a metric. Computation is accomplished any time a relevant event occurs, and is independent of the granularity degree. The granularity degree can be determined using the following different units :

- Counts of events (e.g. each 100 events).
- Counts of a specific event type (e.g. all scheduling events) ;
- Time interval (e.g. each second).

For instance, when one assigns a granularity degree , say k, to a metric he or she has already specified the frequency of updates in the database. In this case, for each k changes in the statistics value, an update will be accomplished in the database. There is a default value for the granularity degree but it can be adjusted separately for each metric. Figure 4.2 shows updates for a case in which the granularity degree is one, while Figure 4.3 shows the number of updates for a larger granularity degree.

Using the notion of granularity degree leads to a faster trace analysis, data store construction, and also a better query answering performance. The efficiency increases because with a large granularity degree, less information will be written to disk.

Although defining a proper granularity degree leads to a better construction and access time, it may require additional processing to answer queries for the halfway points, leading to search performance degradation, particularly when the granularity degree is coarser than the query interval range. In this case there are two solutions to compute the desired statistics : rereading the trace or using the interpolation technique.

The first solution is similar to the checkpoint method, which rereads the trace to computes the desired statistics. This technique is also used in the partial history tree proposed by Montplaisir et al. [102]. Although this solution works well for small traces, it is not a great idea to reread the trace and reextract the values, each time users query the system. Especially when the trace size is large, checkpoint distances are large, and the system load is high.

The second solution is to use the interpolation technique. Using linear interpolation, as shown in Figure 4.4, makes possible to find any halfway values within two extremes of the

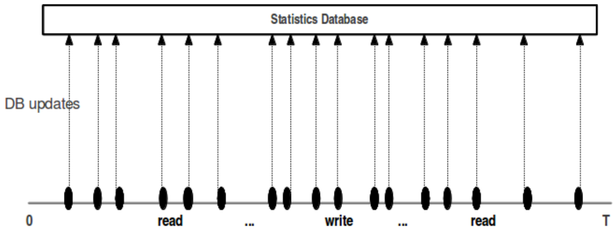


Figure 4.2 Database updates for granularity degree = 1.

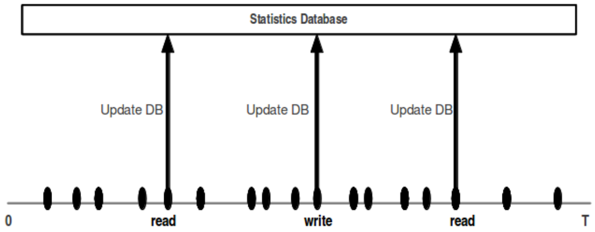


Figure 4.3 Database updates for granularity degree = 5.

granularity checkpoints, without rereading the trace events. The accuracy of the results, computed with the interpolation technique is directly related to the granularity degree of the metric.

Figure 4.5 shows an example of statistics computation using both the interpolation technique and the granularity degree parameter. In this example, the goal is to find the statistics of a particular metric between two points A, B in the trace. The bold points show the borders of the GD durations and there is one data structure update for each point.

$$Val_{AB} = Val_B - Val_A = x1 + x2 + x3 \tag{4.1}$$

The value of x2 can be computed using the subtraction of the two values in t2 and t3. However, since the x1 and x2 are half values between two updates (inside a GD), they can be computed using the interpolation technique, as denoted in the Formula 4.2. One last point is that the values Val_{t1}, Val_{t2} and Val_{t3} can be extracted directly from the interval tree data structure using stabbing queries (as will be explained in the next section).

$$\begin{aligned} x1 &= Val_{t1} + (A - t1)\frac{(Val_{t2} - Val_{t1})}{(t2 - t1)} \\ x2 &= Val_{t3} - Val_{t2} \\ x3 &= Val_{t3} + (B - t3)\frac{(Val_{t4} - Val_{t3})}{(t4 - t3)} \end{aligned} \tag{4.2}$$

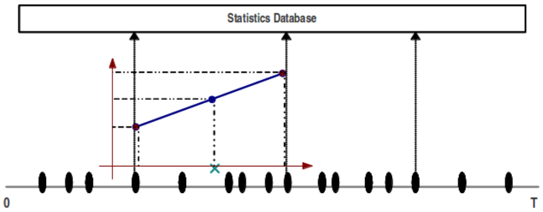


Figure 4.4 Using linear interpolation to find a halfway value.

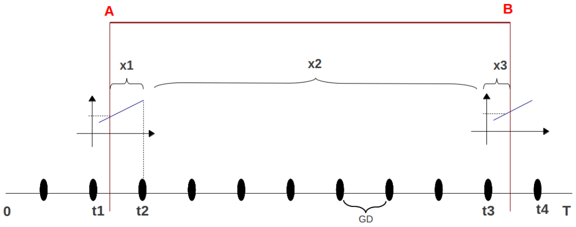


Figure 4.5 Example of using linear interpolation and granularity degree parameter.

Although the interpolation technique returns an estimation of the real value, there may be doubts about its precision. The precision of the result actually depends on the size of granularity degree. In other words, by carefully adjusting the granularity degrees for different metrics according to their importance, it is possible to estimate a fairly accurate results. For instance, by selecting small values for more precise metrics (e.g. CPU utilization for evaluating the affinity of a scheduling algorithm) and relatively large values for less precise metrics (e.g. number of trace events), one can achieve better results. These values are set by an expert or can be selected by evaluating the previous experiences.

We will continue explaining the architecture with an example in the next section.

Illustrative Example

In this section, we investigate an example to show how to use the proposed method for computing the statistics of system metrics in a large trace. The example shows how to compute the CPU utilization for different running processes, separately or in a group, during any arbitrary time intervals of the underlying system execution.

The first step is to specify how the statistics values are extracted from the kernel trace events. We use trace scheduling events to extract the CPU utilization. The scheduling event arguments show the CPU number, the process id that acquires the CPU as well as the process that releases the CPU. Utilization is computed by summing up the length of the durations that a CPU is used by a running process. Figure 4.6 shows a possible case of CPU scheduling for two processors and four processes. In the example shown in Figure 4.6 :

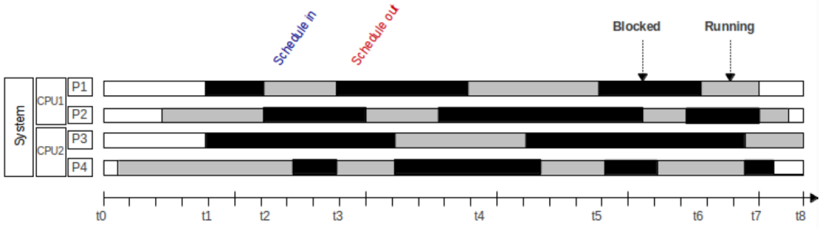


Figure 4.6 An example of the CPU scheduling.

$$\begin{aligned}&CPU_1\ utilization\ of\ running\ process\ P_1:\\&U(P_1)_{CPU1}=(t_3-t_2)+(t_5-t_4)+(t_7-t_6)\end{aligned}\tag{4.3}$$

$$\begin{aligned}&Utilization\ of\ CPU_1:\\&U(CPU_1)=\frac{\sum_{i=1}^{i=2}U(P_i)_{CPU_1}}{t_8-t_0}\end{aligned}\tag{4.4}$$

And in general :

$$\begin{aligned}&CPU_k\ utilization\ of\ each\ running\ process\ P_i:\\&U(P_i)_{CPU_k}=\sum_{m,n=t_s}^{m,n=t_e}(t_m-t_n)\end{aligned}\tag{4.5}$$

$$\begin{aligned}&Utilization\ of\ CPU_k\ \ :\\&U(CPU_k)=\frac{\sum_{i=0}^{i=n}U(P_i)_{CPU_k}}{t_e-t_s}\\&=\frac{\sum_i\sum_{m,n=t_s}^{m,n=t_e}(t_m-t_n)_i}{t_e-t_s}\end{aligned}\tag{4.6}$$

As explained earlier, one of the features of this framework is being able to perform hierarchical queries. To do this, we build a tree called the ”metric tree” containing a hierarchy of resources and metrics. The construction of such a hierarchy makes it possible to drill down and roll up between the resources, and to aggregate the statistics values for different granularities (e.g. for a process, group of processes, a virtual machine or even for whole system). Figure 4.7 models a typical organization of the metric tree.

As shown in Figure 4.7, there are hierarchies of system resources and metrics separately. In this tree the system resources (e.g. processes, files, cpus, network ports, etc.) are organized in the separate branches of the tree. Then, the metrics nodes that could be a tree as well, connect the resources together. For example, the metric ”cpu usage” connects two nodes, a process and a cpu, representing the cpu usage of that process. Each metric node is assigned a unique number that is used as a multivalue key for future references to the corresponding

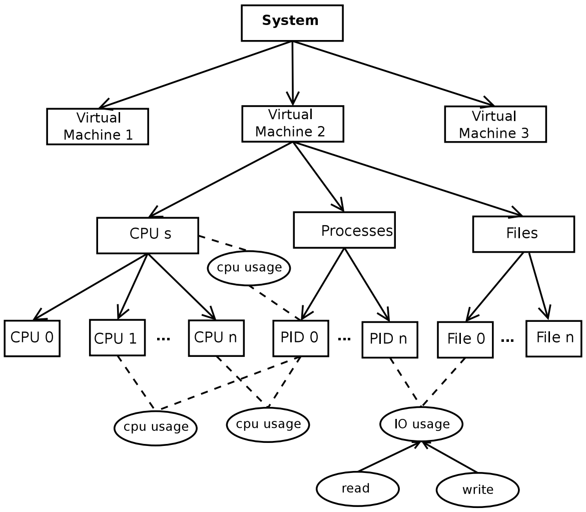


Figure 4.7 A general view of the metric tree.

statistics values in the interval tree. Metric nodes can be used to connect different resources together, individually or in a group. For instance, a IO usage metric may connect a process to a particular file or to the files of a folder or even to all files, showing respectively the bytes of this file read or written by that process, the bytes read or written in the files of a folder, or the whole IO of that process. For each resource, there is a path from the root node. For each metric node, there is at least one path from the root node as well, representing which resources this metric belongs to. In this tree the metrics and the resource hierarchies are known in advance, however, the tree is built dynamically. Organizing the resources and metrics in such a metric tree enables us to answer hierarchical queries like ”what is the CPU usage (a specific CPU or all of them) of a process (or a group of them) ?”, and ”what is the IO throughput of a system, or a group of running processes ?”.

This hierarchy of resources facilitates the computing of the hierarchy statistics. For any resources or group of resources for which statistics values should be kept, a metric node is created and connects them to the other resource (or group of resources) using a proper key value. For example, when cpu usage of a specific core of a specific process is important, we consider a cpu usage metric node between these two resources and assign a unique key value to that. As another example, in addition to the IO usage of the different processes, someone may be interested in monitoring the IO usage of an important file like /etc/passwd since it is accessible by all processes. In this case, it is just required to make a IO usage node, connect that node to that particular file and to the ”Processes node” in the tree, and assign another unique key value to that. This key value is used as a reference to the different statistics values in the interval tree.

The solution is based on reading trace events, extracting the statistics from them, and storing them in the corresponding interval database nodes, according to the corresponding granularity degrees.

After creating a hierarchy of resources and metrics, it is then time to decide how to store and manage the statistics values. As explained earlier, we model the statistics values in intervals and store them in a tree-based interval data store. In this data store, data is stored in both the leaf and non-leaf nodes. Each node of the tree contains a bucket of interval entries lying within the range of its time boundaries. A view of this tree and corresponding intervals are shown in Figure 4.8.

Each interval contains a start time, end time, key and value. The key refers to a metric node in the metric tree. The value shows a cumulative value, from the starting time of the trace to that point. Based on the start and end times, intervals are organized into the minimum containing tree nodes.

The created metric and statistics trees are used to extract the desired statistics in the analysis phase. In other words, computing the statistics of a metric is accomplished by performing a stabbing query at any given query point. The stabbing query returns all the tree nodes intersecting a given time point [30]. The query result is the statistics value of the metric at that given point. In the same way, computing the statistics of an interval of interest, instead of a single point, is answered by performing two stabbing queries : one for extracting the aggregated value of the interval start point of the interval, and one for the end point. For each metric, it is reasonable to retrieve at most one result per each stabbing query, as only one value for each point or interval has been stored during the event reading time. After performing the stabbing queries for the start and end point of a given interval, the desired result is the difference between these two query results. The algorithm is shown in Listing 1. The algorithm takes 2*O(log(n)) time to compute the statistics values of a metric and time range (log(n) for each stabbing query, n is the number of tree nodes). Detailed experiment results will be shown in the next chapter. For each stabbing query, a search is started from the root downward, exploring only the branch and nodes that possibly contain the given point. Within each node, it iterates through all the intervals and returns only the entry intersecting the given point. Since the intervals are disjoint, the stabbing query will return at most one value. However, it is possible to not find any stabbing interval for the given time point, which means there is no metric value for the given time point, and it will be considered zero.

To traverse the tree, the algorithm performs a binary search on the tree, and copies the resulting branch to the main memory. It then searches the nodes and its containing interval entries to find the statistics value of the given metric. In other words, by doing a stabbing query, statistics values of the other metrics which intersect the query point, will also be in

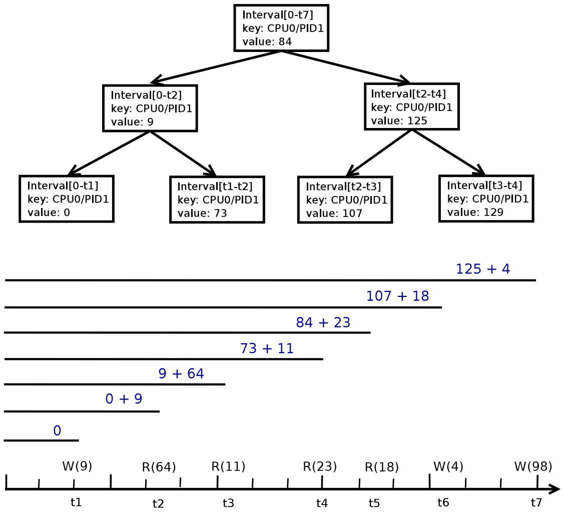


Figure 4.8 A view of the intervals and the corresponding nodes.

ALGORITHM 1: Complete interval query.

```
Require: a time range $[t1$ - $t2]$ and $v$ a set of metric keys.
set $upperBoundValue = 0$ ;
find all nodes of the tree that intersect t2 (stabbing query) ;
search within nodes' intervals and find all entries that contain at least one of the metric
keys ;
if found any then
    for each entry do
        set $upperBoundValue = upperBoundValue$ + entry.value ;
    end for
end if
set $lowerBoundValue = 0$ ;
find all nodes of the tree that intersect t1 (stabbing query) ;
search within nodes' intervals and find all entries that contain at least one of the metric
keys ;
if found any then
    for each entry do
        set $lowerBoundValue = lowerBoundValue$ + entry.value ;
    end for
end if
return $upperBoundValue$ - $lowerBoundValue$ ;
```

the main memory. We call this data set ”current statistics values”. The current statistics values can then be used for computing and extracting the statistics of other system metrics for the same query point. The important point here is that, since the current statistics values data set is in main memory rather than the external disk, performing subsequent queries to compute other metrics statistics for the same query point will be much faster than the base stabbing query.

ALGORITHM 2: Stabbing query.

```
Require: time point $t$ and the metric name.

search all intervals in the root node to see whether exists any containing interval for the given time point $t$.
if found then
  return the entry value
else {if not found and the node is not a leaf}
  find the corresponding child node regarding to the children intervals and given time point $t$.
end if
if exists any node then
  Perform a query in the subtree that this node is its root.
else
  return zero
end if
```

Hierarchy Operations

As mentioned earlier, the framework supports hierarchical queries between resources. For example, it is possible to compute the CPU utilization of one process, at the same time as a group of processes, or for the whole system.

To support these queries, there are two general approaches in the proposed framework. The first and obvious solution is to separately compute and store the temporary statistics values of each resource (e.g. process, file, CPU, etc.), and all possible groups of resources from the trace events. For instance, the count of CPU time for any group of processes, and for the whole system are accumulated separately when relevant trace events are received. The problem of this solution is that all groups of processes that will be queried later by an analyst must be known in advance. However, it is not always possible to predict which group of processes an analyst may be interested in. Also, the solution requires much space to store the duplicate values of all resource combinations. Moreover, at analysis time, all statistics computations must be answered by querying the external interval tree, which is too time consuming.

The second and better solution is to compute the hierarchy statistics values by summing up the children resource statistics. For instance, the count of CPU time for any group of processes is extracted by summing up the total CPU time of the group's children processes. Unlike the previous approach, it is not necessary to know in advance or even to predict the resource groups that will be inspected by analysts. In this solution, we use the aforementioned current statistics values data set to answer the hierarchical queries. Since all of intersecting nodes and intervals will be brought to the main memory upon performing a stabbing query, it is possible to integrate and sum up the statistics values of any system resources to quickly compute the statistics value of a group of resources. Since the current statistics values data set is in main memory, hierarchical queries can be performed quickly.

Using the second approach, for performing a hierarchical query it first find the values of children nodes using the aforementioned stabbing queries, and then, aggregate results to find a value of the desired high level node. Equations 4.5, 4.6, and 4.7 show relations between a high level statistics value and its containing nodes :

$$\begin{aligned} &Utilization\ of\ all\ processors\ :\\ &U(All\ CPUs) = \textstyle\sum_{j=0}^{j=n} U(CPU_j)\\ \\ &And\ totally: \\ \\ &= \textstyle\sum_{j=0}^{j=n} \frac{\sum_{i=0}^{i=m} U(P_i)_{CPU_j}}{t_e - t_s} \end{aligned} \tag{4.7}$$

The CPU utilization of the whole system can be computed by summing up the CPU utilization of each processor separately, which can be acquired in turn by computing the utilization of each process by doing two stabbing queries over the disk based interval tree, and consecutively a memory based linear search. In the same way, the same solution can be used for other metrics and resource hierarchies.

The above example shows the addition aggregate function. It is however possible to apply other aggregation functions as well, such as minimum, maximum, etc. Using these functions, makes possible to find special (e.g, abnormal) characteristics of a system execution : high throughput connection, most CPU (or other resources) consuming virtual machines, operations or processes, and average duration of read operations (for checking the cache utilization).

The required query time for computing the statistics value of a resource metric is O(log n) (i.e. n is the number of nodes in the interval tree). This query brings the current statistics values from the disk to the main memory. Other queries, for the same time point, will be answered by iterating through this data set rather than extracting the data from disk. For instance, the IO throughput of a system is computed by summing up the IO throughput of all running processes at that given time point. For the aforementioned reasons, the IO throughput of all running processes for the query time point will be available in the current statistics values data set, and can be easily used by going through its entries.

Altogether, the required processing time for summing up these values, and computing the statistics value for a group of resources (or any hierarchy of resources) is O(Log n + K). The time (Log n) specifies the query time to bring up the data from the disk resident interval tree, and fill the current statistics values data set. The time K shows the required time for iterating through and summing up the values of containing resource statistics in the current statistics data set. Since the external interval tree is usually large, and querying a disk-based data structure maybe a time consuming task, time (Log n) is generally considered to be larger than K. However, in a very busy system, or in busy parts of a system execution, with lots of running processes and IO operations, K may dominate (Log n), especially when the tree is fat and short (each tree node, encompasses many interval entries). The results of the hierarchy operation experiment using both solutions will be discussed in the Experiments section.

4.7 Experiments

We have prototyped the proposed framework in Java on the Linux operating system. Linux kernel version 2.6.38.6 is instrumented using LTTng, and the tests are performed on a 2.8 GHz machine with 6 GB RAM, using different trace sizes. This prototype will be contributed to TMF (Tracing and Monitoring Framework)[5]. In the prototype, the defined metrics are : CPU usage, IO throughout, number of network connections (for both the incoming and outgoing HTTP, FTP, DNS connections), and also counts of different types of events. Figure 4.9 shows the memory used by the framework to store the interval information. The graph shows different on-disk sizes for the different trace files. The trace files vary from 1 GB to 40 GB. The size of the resulting interval data store, where the granularity degree equals 1, is about 2.5 to 4.5 times the original trace size. As explained earlier, this is not a reasonable storage size. To solve this problem, larger granularity degrees (i.e. 100, 500, 1000) are used. Figure 4.9 depicts a comparison of the proposed method and the checkpoint method. Since the range of values for the case where the granularity degree is 1 is much higher than those with larger granularity degrees, the logarithmic scale is used for the Y axis.

[5] http ://lttng.org/eclipse

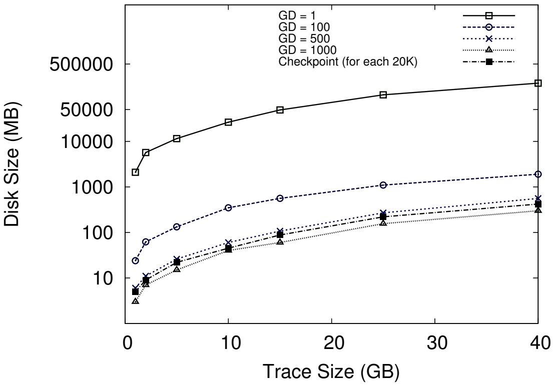


Figure 4.9 Disk size of the interval tree data structure.

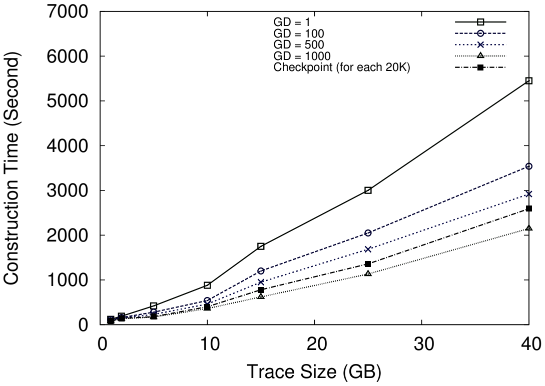


Figure 4.10 Construction time for different trace sizes.

A comparison of the tree construction time between different solutions is shown in Figure 4.10. The figure shows that the time used for tree construction considerably depends on the number of updates to the interval data store, and that the time decreases when less information is written to disk, thus underlining the importance of selecting coarse granularity degrees.

The query time analysis is shown in Figure 4.11. For each graph, we have tested 20 runs, in which we have used the aforementioned complete interval query (two stabbing queries for each request) for 100 randomly selected time intervals. As shown in Figure 4.11, the best case is the tree with a granularity degree of 1000. In this case (and the cases with a granularity degree of larger than 1), we have used linear interpolation to approximate the real values when querying a time within a checkpoint duration. The case with GD = 1 has the worst case, because the database in this case is larger and other cases. Therefore, more time is required to query the database and fetch the results. With the checkpoint method, we reread the trace and regenerate the values inside the checkpoints. Since this method reopens and

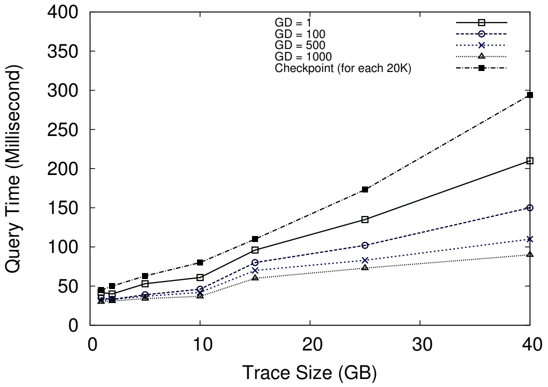


Figure 4.11 Query time for different trace sizes.

rereads the trace for each query, the query time is longer.

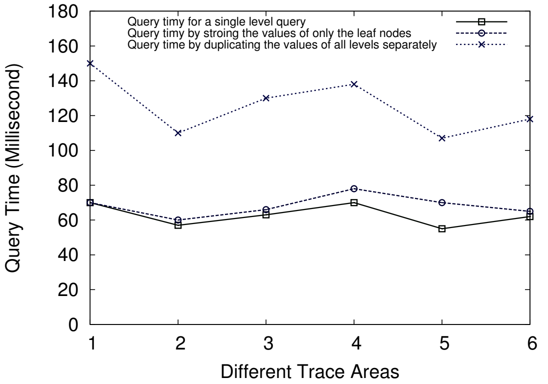


Figure 4.12 Comparison of different approaches for supporting the hierarchical operations.

A comparison of the two aforementioned approaches for performing the hierarchical operations is shown in Figure 4.12. In this graph, the X axis shows the different points in the trace, and the Y axis shows the query time for computing the IO throughput of a process separately, a process and the whole system together using the first and second aforementioned approaches. As explained earlier, computing the hierarchical statistics values, by summing up the containing values, is faster than storing them separately in the interval tree, and querying them by reading the disk data structure for each query. As shown in the figure, summing up the values of containing resources in the busy areas of the system execution (points 4,5) takes much more time than in the other trace points (points 1,2,6), in which the system is not too busy and encompasses less IO operations. The trace size 10 GB and granularity degree 100 are used for this comparison.

The analysis of the above results shows that one can achieve a better efficiency (both in disk size, construction time and query time) by using larger granularity degrees. However, this is not always true. The granularity degree is somehow related to the precision of a metric. Larger degrees may lead to less precision, and longer query times for the points lying within an interval. Thus, a trade-off is required between the precision, granularity degree and the evaluation context. However, more investigations are required to find proper values for the granularity degrees of different metrics, as will be studied in a future work.

4.8 Conclusion and Future Work

Early steps in most analysis methods start by looking at the general overview of a given trace. The analysis can be continued by narrowing the current view and digging into it to obtain more details and insight. The several previous studies that provide such a view are examined in the literature review. However, they are not able to compute system statistics in a relatively constant time for any given interval. We proposed a framework architecture to analyze large traces and generate and provide such a view. We also presented the performance results of this method.

The main effort was on creating a compact data structure that has reduced overhead, and a reasonable access and query time. The details of the proposed data structures and algorithms, with their subsequent evaluations for different cases have been analyzed. The framework models the system resources in a hierarchy to support hierarchical operations between different resources. To avoid a size explosion of precomputed statistics, a proper granularity degree should be chosen for each metric. Then, intermediate points are computed using linear interpolation. Granularity can be expressed in count of events or time units. We evaluated the proposed framework by assigning different granularity degrees for different metrics. The results denote that one can achieve a better efficiency and performance by determining proper granularity degrees for metrics. Constant access time (with respect to the time interval) for statistics computation is achieved by computing the final result from two values, at the start and end of the interval.

Possible future work is to analyze the effects of using the interpolation technique, as well as developing a formula to link the granularity degrees to metrics and trace sizes. We have prototyped the framework for LTTng Linux kernel tracer. Other future work includes extending the framework and related data structures to support other tracing systems as well as connecting the proposed framework to kernel-based fault and attack detection systems.

Although the proposed method can be used for online tracing as well, this was not investigated during this phase of research. The online construction of the interval tree will probably lead to new challenges and will be experimented as a future work.

CHAPTER 5

Paper 4 : Cube Data Model for Multilevel Statistics Computation of Live Execution Traces

Naser Ezzati-Jivan and Michel Dagenais

5.1 Abstract

Execution trace logs are used to analyze system runtime behavior and detect problems. Trace analysis tools usually read the input logs and gather either a detailed or brief summary of them to later process and inspect in the analysis steps. However, continuous and lengthy trace streams contained in the live tracing mode make it difficult to indefinitely record all events or even a detailed summary of the whole stream. This situation is further complicated when the system aims to compare different parts of the trace and provide a multilevel and multi-dimensional analysis.

This paper presents an architecture with corresponding data structures and algorithms to process stream events, generate an adequate summary -detailed enough for recent data and succinct enough for old data- and organize them to enable an efficient multilevel and multi-dimensional analysis, similar to OLAP analyses in the database applications. The proposed solution arranges data in a compact manner using interval forms and enables the range queries for any arbitrary time durations. Since this feature makes it possible to compare of different system parameters in different time areas it significantly influences the system's ability to provide a comprehensive trace analysis. Although the Linux operating system trace logs are used to evaluate the solution, we propose a generic architecture which can be used to summarize various types of stream data.

Keywords : stream processing, multi-level analysis, OLAP analysis, trace abstraction, Linux kernel.

5.2 Introduction

Many applications such as network monitoring, web log analysis and stock exchange analysis tools provide different statistics over data streams [7, 65, 69]. In these applications, a system administrator or an automated program monitors the statistics of different system parameters (e.g., the usage of different system resources) to detect any possible problems, patterns or attacks. For instance, monitoring and counting the number of half-open connections in a short duration and comparing it to a predefined threshold (or to the number of completed connections) may help to detect a denial of service attack.

In a previous work [50], we presented a framework to store a history of (offline) trace summary in the trace reading phase to compute and provide the different statistics of system parameters in the analysis phase. However, since the trace stream size, despite the offline trace, is considered unlimited, it is not possible to store a complete history of the stream. Thus, heuristics are needed to select and store only the parts of the input data that are enough to provide accurate statistics. In other words, a trade-off between the size of the summary and the accuracy of the query responses is necessary. But in general, to guarantee the scalability of the solution the size of data structures should be small, somewhat independent of the length of the input trace stream or at most poly-logarithmic to that.

The query response time is also an important factor in the aforementioned stream data analysis applications. The system should provide a fast response time, facilitate the interactive use of the system and satisfy the real-time constraints of the streaming applications. The other factor is the processing time of the input stream. Since a new event may arrive at any arbitrary time, per event data processing rate should be efficient so that the analysis system can operate without congestion or having to drop input events.

Another challenge is providing a multi-dimensional and multi-level analysis. Analysts usually wish to perform the multi-dimensional analysis of the input trace stream on an expressive abstract level, including some multi-level exploration operations like drill down or roll up to get more or less detailed information [69]. Trace events are multi-dimensional in nature and usually represent interactions of different dimensions. For instance, a ”file read” trace event may contain information from the running process, the file that has been read, the current scheduled CPU for this operation and the return value (i.e., the number of bytes read by the operation).

In this paper, we contribute data structures and corresponding algorithms to construct stream cubes and provide OLAP [1] style analysis over stream trace data. The solution incrementally constructs a compact and scalable data store from the input data, records it in the main memory (and possibly in the disk) and provides an efficient query mechanism for any flat or hierarchical queries over a system parameter or a group of them. Using this approach, users will be able to compute statistics of multiple system parameters, at different granularity levels, and for any arbitrary time ranges of the system execution.

Another contribution is that the proposed solution supports efficient range queries over the time dimension. Other approaches that support range queries usually work by storing

[1] Online Analytical Processing

a solid value of the data and counting or summing up the values for the queried range. However, this method could be a time-consuming task, especially when the selected range is relatively large. By storing the summary data as intervals, our solution provides an efficient query response time for range queries, regardless of the size and position of the given range.

The rest of the paper is organized as follows : first, after looking at the related work we present the architecture of the solution, the data structures and the techniques used. Second, we describe the different query types that the system supports. Then, we discuss the evaluation and experimental results of the proposed method. Finally, we conclude by outlining specific areas of investigation for future enhancements.

5.3 Related Work

Stream analysis has many applications in network monitoring, web logs and click stream analysis, call records analysis, stock exchange and bank transaction analysis, medical records monitoring, weather monitoring, etc [7, 65]. Several research studies have been conducted in the literature on the stream data management [7, 91], OLAP analysis over stream data [3, 69, 118] and data mining [62, 139]. These studies present interesting ideas and results on stream data analysis to extract changes, trends and detect problems.

Several data structures have been proposed to store a history of stream data. Using a modified version of H-Tree, a stream cube [69] is proposed to perform a multi-dimensional and multi-level OLAP analysis over data streams. They use different time granularities for recent and decent information and a tilted time frame [21] to compress the data over the time dimension. They avoid recording information of all levels and only store the information along the critical paths. With this technique, the information that is not stored directly requires on-the-fly processing to be extracted. Even though our method uses a time frame similar to the presented tilted time frame, it uses different data structures and organizations to manage the stream cube. In our method, all items of the same time points can be extracted synchronously with a single query. Moreover, in our method the range query over the time dimension is supported directly and efficiently.

Patroumpas et al. [112] propose a stream management approach that uses different window sizes. In their technique, different windows are filled simultaneously. The problem with this solution is that they duplicate data in the different windows for the same time. In our approach, however, we avoid duplicating data at different windows while we support different windows and time granularities. Users can retrieve information for the last n milliseconds, seconds, minutes or even for any coarser granularities.

The fixed and moving sliding window methods are used in the literature to analyze the stream data [6, 7, 65, 112]. In these techniques, a recent window of items is kept, processed and used for extracting the desired statistics. The fixed sliding method refers to the case where a window (fixed or variable size) is kept or monitored for fixed durations in the past (e.g., each 1 second, starting from 2 :00 PM yesterday), while the moving window refers to non-fixed start and end points that move with the time and is measured using the current time (e.g., every last 10 seconds) [65]. In our approach, we support both the variable-size fixed and moving sliding window queries over the stream cube. At each point, user can extract multi-dimensional statistics for the last n time units or can compute statistics for a fixed window in the past.

5.4 Problem Statement

In this section, we describe formal notations and definitions required to present the problem.

5.4.1 Preliminary Definitions

A dimension schema D is a tuple $\prec Name, L_D, \preceq \succ$ where : $Name$ represents a unique name for the schema, L_D denotes a set of levels, representing the multiple granularity levels of a dimension, and $\preceq$ represents a partial order between elements of L_D forming a graph or a hierarchy of dimension member items. Each level contains a set of members. A dimension instance d_i is defined as a set of members d_m from all levels. Figure 5.1 shows an example of four dimension schemas : Operation, Machine, Process and File.

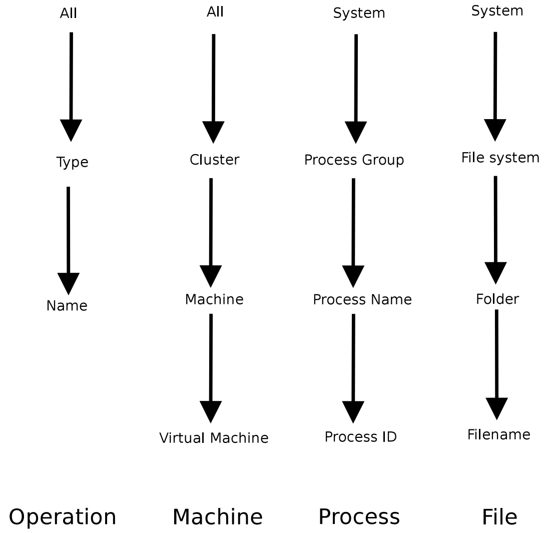


Figure 5.1 Examples of dimension schemas.

In the example shown in Figure 5.1, the dimension instance Process is defined as {(System $\preceq$ Process Group), (Process Group $\preceq$ Process Name), (Process Name $\preceq$ Process ID)}. Process Group 1, Apache, Firefox, etc. are also member items of the Process dimension, shown in Figure 5.2.

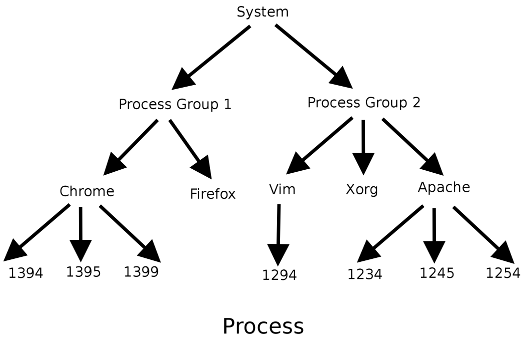


Figure 5.2 An instance of the Process dimension schema.

A trace stream is defined as a sequence of timestamped events $\ldots$, e_i, $\ldots$,e_n, in which e_n is considered the most recent event. Each event e = $(d_{m1}, ..., d_{mk}, r_1, ..., r_m)$ represents an interaction between a set of dimensions ($d_{m1}, ..., d_{mk}$, i.e., a set of system resources such as CPU, process ID, filename, disk block number) that results in one or more return numbers $(r_1, ..., r_m)$. For instance, a file open event (open, CPU1, process2, file0, 3) represents a file open operation that is performed by a process process2 run by CPU CPU1 to open a file file0. The result, number 3, shows the output of the operation : the assigned file descriptor value.

One mandatory member of each multi-dimensional event is the timestamp field $t_i \in T$ that is used to order the events. T, the time dimension, is the set of Natural numbers : T= $\{t | t \in \mathbb{N}\}$. In this domain, a time interval $[t_1,\ t_2]$ is defined as $\{t \in T | t_1 \leq t \leq t_2\}$. We also assume that the timestamp values are distinct and any two events e_i, e_i have different values, $t_i \neq t_j$. Figure 5.5 represents a set of trace stream events gathered by the LTTng kernel tracer. LTTng [38] is a low-impact and lightweight open source Linux tracing tool, and provides detailed execution logs of operating system and user space applications.

We define the term trace stream cube as a collection of cuboids constructed over a trace data stream. Each cuboid represents a possible group-by operation of a measure over a set of dimensions. The most specific cuboid, the base cuboid, contains items from all dimensions. It can be used to gather the statistics for any dimension combinations,e.g., return the IO throughput for a particular process over a text file in a given virtual machine. The most generalized cuboid that is also called an Apex cuboid, contains the total value of a measure for all dimensions, e.g., whole CPU utilization of the system. Exploring downward from the apex cuboid to the base cuboid is called drilling down and the opposite operation, going upward from the base cuboids to the apex cuboid is called rolling up.

Example. Having three dimensions D1, D2, D3 and a measure M, the apex cuboid is (*, *, *, M), while the base cuboid is (D1, D2, D3, M). The combination of all possible cuboid forms a cube : (*, *, *, M), (D1, *, *, M),(*, D2, *, M),(*, *, D3, M),(D1, D2, *, M),(D1, *, D3, M),(*, D2, D3, M),(D1, D2, D3, M).

The term metric(s) is used to represent quantities to compute, monitor, compare or evaluate the usage or performance of the different system parameters at different levels of abstractions. For instance, CPU utilization and network throughput are examples of these metrics. We refer to these quantities using both the terms ”metric(s)” and ”measure(s)” in the remainder of this text.

Having defined these terms, we seek to perform a multi-dimensional analysis : we will efficiently extract and compute the different system statistics from the input trace stream for not only the different single time points but also for any arbitrary time ranges at the different levels of granularity. For example, one might require the input/output (IO) throughput of the whole system, a specific virtual machine, process or a file, for a specific exact time, e.g., at 3 :36'. or for the last 30 minutes.

5.4.2 Statistics to Monitor

Different types of statistics are supported in our approach :

1- Statistics such as sum, count and average are supported for any combination of the defined dimensions. Typical examples are the IO throughput of all files in a particular folder, the count of specific event types, or the average usage of a specific CPU. Using these queries, it is possible to provide frequency counting, or top-k elements, for any time range. One obvious application of frequency counting is to detect whether the statistics values exceed the predefined threshold values. Similarly, top-k queries can be used to identify the users, processes or applications that consume the majority of the system resources.

2- Range queries (for the time dimension) are supported for different time points and periods. The selected time range could be a time duration completely in the past, e.g., retrieve the desired statistics between 2 PM on February 3 to 3 AM on February 4, or a range between a time in the past and now $[t - \tau, t]$, e.g., retrieve the desired statistics for the last 30 minutes. In other words, moving sliding windows [65] are supported, in addition to the fixed sliding window where the user seeks statistics per fixed time units, e.g., each 5 minutes.

3- Different time scales are supported in this method. The selected time range could vary from milliseconds to days, weeks or even months. For instance, users may ask queries like ”return the desired statistics for the last n milliseconds, seconds, minutes or any coarser granularity”. However, for the earlier times we use a larger time granularity to extract the finer grain statistics. In other words, for the most recent time, any time range from the millisecond scale is supported, while for time periods in the past, the precision is decreased and coarser grain times are supported (i.e., hours or days instead of seconds and minutes). This consideration is normal in many applications [21], as users usually seek highly precise data for the more recent times and possibly coarser time ranges for the more distant times. In short, the proposed solution guarantees that different time scales (larger than the base time granularity) are supported and users can extract the desired statistics for different time scales and ranges.

4- Hierarchical operations like drill down, roll up, slice and dice are supported in this design, similar to the offline OLAP systems. Users may query the system to get a higher level aggregate value or may ask for more detailed statistics values. For example, having total IO throughput of a virtual machine, one may wish to access the detailed throughput of its processes separately, or having the network traffic of each IP address separately, one might see the aggregated traffic for a range of network addresses in the past days.

For all of the above problems, the compactness of the data structures and the construction and query performance of the method are considered the main requirements.

5.5 Architecture

A high-level view of the architecture is shown in Figure 5.3. In this architecture, the trace reader reads and processes the input data, extracts the required summary and records in a data store named cube data model. Cube data model, in turn, contains different structures to organize and manage the data summary. Query engine is responsible to handle and respond queries received from the users. These modules will be explained in detail separately in the following sections.

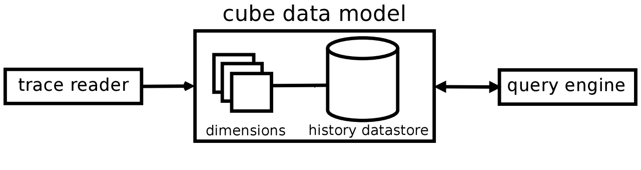


Figure 5.3 A high-level view of the architecture.

5.5.1 Trace Reader

The first step in trace analysis is to extract the required data from the input trace events. Trace events usually describe the system in a very low-level form (Figure 5.5), and useful and synthetic information (e.g., the statistics data we are looking for) are usually hidden behind these low-level events. Some data analysis steps are required to extract the desired high-level information from the original trace data. To do so, for each statistic metric, a set of events is registered and monitored. For instance, socket-based events like socket connect, socket send, socket receive, etc. are registered and monitored to collect statistics about the network traffic. When one of the registered events arrives from the input stream, it is analyzed and the desired information is extracted and stored for future use.

Since the observed trace events are too low-level, some high-level statistics measures are not obviously recognizable and may require a more sophisticated analysis of the input stream. For instance, calculating the number of file downloads (as a high-level measure) is not obvious, as it is necessary to integrate firstly some specific low-level events [49] to generate the higher-level data to be able to compute their high-level statistics.

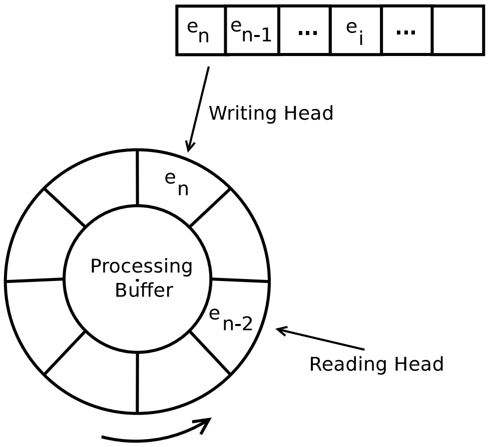


Figure 5.4 Circular buffer to read and process the stream events.

One issue in the stream events data extraction is that the distance of consecutive events could be less than the time required to process them. Indeed, this may happen when the events are too low-level (as shown in Figure 5.5) and must be abstracted out to higher levels before extracting the statistics information. In our implementation, we use a separate thread, rather than the thread used to read the stream, to process the incoming events. To do so, a circular buffer is initialized and preliminary information gathered from the events are copied into that buffer (as shown in Figure 5.4). The processing thread then operates on the buffer, gathers the statistics information, and stores it in the history data store.

```
kernel.syscall_entry: 509.147214537 (kernel_1), 1697, 1697, lttctl, 1, 0x0, syscall { 0x7f0b0eb96cbd, 0 [syscall 0] }
kernel.sched_schedule: 509.147214752 (kernel_7), 0, 0, swapper, 0, 0x0, syscall { 48, 0, 1 }
fs.read: 509.147220219 (fs_1), 1697, 1697, lttctl, 1, 0x0, syscall { 8176, 21 }
kernel.syscall_exit: 509.147220604 (kernel_1), 1697, 1697, lttctl, 1, 0x0, user_mode { 241 }
fs.write: 509.147227093 (fs_5), 568, 538, rs:main Q:Reg, 1, 0x0, syscall { 66, 4 }
kernel.syscall_exit: 509.147227571 (kernel_5), 568, 538, rs:main Q:Reg, 1, 0x0, user_mode { 66 }
kernel.syscall_entry: 509.147234027 (kernel_1), 1697, 1697, lttctl, 1, 0x0, syscall { 0x7f0b0eb96cbd, 0 [syscall 0] }
kernel.syscall_entry: 509.147235150 (kernel_5), 568, 538, rs:main Q:Reg, 1, 0x0, syscall { 0x7fcb30146c5d, 1 [syscall 1] }
net.socket_recvmsg: 509.147236434 (net_1), 1697, 1697, lttctl, 1, 0x0, syscall { 0xffff880188c90580, 0xffff880199fe1d70, 4096, 64,
fs.read: 509.147237217 (fs_1), 1697, 1697, lttctl, 1, 0x0, syscall { 4096, 3 }
kernel.syscall_exit: 509.147237564 (kernel_1), 1697, 1697, lttctl,1, 0x0, user_mode { -11 }
kernel.syscall_entry: 513.772101451 (kernel_4), 2334, 2334, /opt/google/chrome/chrome-sandbox, 2037, 0x0, syscall { 0x7fccc73b0007, 2 [sys_open+0x0/0x30] }
fs.open: 513.772106530 (fs_4), 2334, 2334, /opt/google/chrome/chrome-sandbox, 2037, 0x0, syscall { 3, ”/etc/ld.so.cache” }
kernel.syscall_exit: 513.772107125 (kernel_4), 2334, 2334, /opt/google/chrome/chrome-sandbox, 2037, 0x0, user_mode { 3 }
```

Figure 5.5 LTTng trace events for common files accesses.

However, for the time periods in which too many events are received, e.g., when the system is too busy and the tracer module generates many events, the buffer may overflow due to the slow computation. In this case, two following solutions may solve the problem. First, by increasing the gaps between the processing and database updating steps, the statistics can be computed and the database updated less frequently (e.g., update the database every 10 seconds instead of every 1 second). The other solution is to disregard the extra events and not process them until the system returns to its normal state. In the experimental results section, we discuss an evaluation of the distance between the incoming events and the time required to process them. However, it is outside the scope of this research to discuss all possible cases and the detailed solutions for each. We focus here on the data structures and the way we manage the processed data in a long, continuous stream data.

5.5.2 Cube Data Model

The proposed method works by extracting the preliminary and punctual statistics from trace events and storing them in a disk-based data structure. It incrementally builds an efficient history of data, so as to be readily retrieved when needed. The overall view of the cube data model is shown in Figure 5.6.

As shown in Figure 5.6, the cube data model contains two main structures : dimension tree and history data store. Dimension tree models the different system dimensions and parameters and acts as a set of key references for the history data store. History data store is the real storage of the data in which the summary of input trace is stored. This history data store contains different cubes for different time frames (Figure 5.6). The cubes in turn are implemented by tree structures. In other words, each cube corresponds to a time frame (range) and contains an interval tree that stores the information of that time frame. Each interval has a key from the dimension tree, a value that is gathered from trace, a start time and an end time. More details will be provided for each structure in the following subsections.

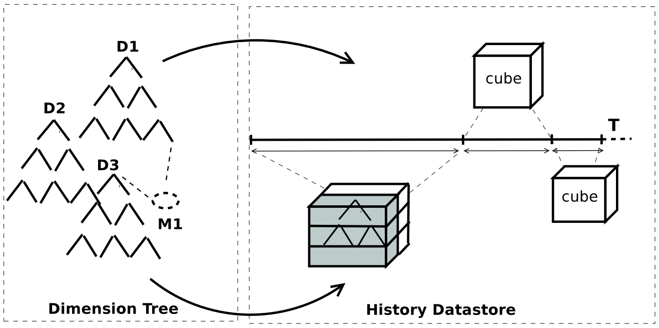


Figure 5.6 Two internal structures of the cube data model.

Dimension Modeling

One may wish to gather statistics about several types of system resources (e.g. memories, processes, files, devices, etc). In our model, each resource (i.e., dimension) is structured as a hierarchy and the metrics of interest are defined between these hierarchies. Figure 5.7 shows two dimensions -Process and File- and the IO usage metric that are defined between these two resources.

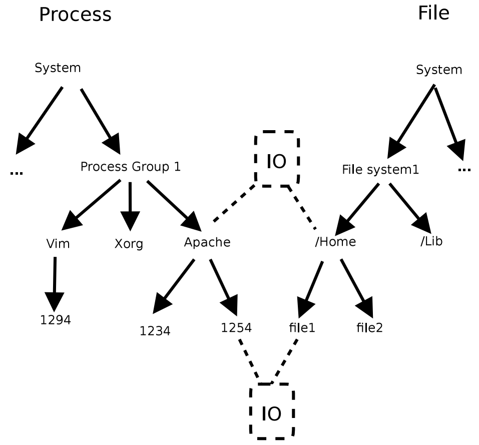


Figure 5.7 Dimension hierarchies and metrics.

In this organization, that is called dimension (metrics) tree, it is possible to define the metrics between any set of resources at any granularity. For example, as shown in Figure 5.7, one may define the metric node between the process and file dimensions to compute the IO throughput of a particular file/folder performed by a specific process, e.g., to place the desired metric between the “Chrome” and “/Home” nodes to compute the number of bytes read or written for all files in the “/Home” directory by the ”Chrome” process.

As explained, an important feature of this tree is its ability to define the metrics at any granularity levels. For instance, some users may only be interested in a specific high-level granularity (e.g. the IO throughput of a whole virtual machine, and not their containing processes), thus the system can avoid storing finer or coarser values, thus saving a significant amount of storage. We will further study the gains of this design in the Experimental Results.

In the aforementioned dimension (metrics) tree, the metric nodes (indicated by dotted rectangles) comprise a link to the corresponding value records in the history data store. They also keep track of pointers to the first and last occurrences of the metrics value in the stream, representing the operational scope of the metrics. These pointers increase the speed of queries for the points that are beyond the operational scope.

History Data Store

For any predefined metric, we keep a history of its values during its whole trace lifetime. To do so, when a registered event arrives and the corresponding statistic value is changed, we create and store a summary of that change in the history data store. We model this change as an interval record and store an interval instead of two single records. For each interval, the bounding points (start time and end time), a key and a value are all stored. The key is a combination of a set of dimensions and a measure referring to the metric nodes (dotted rectangles) in the dimension (metrics) tree. The value represents the statistics value for the chosen metric in this time interval. For example, suppose a ”file read” event is registered in advance for the IO throughput metric. Then when a file read event (e.g., process p1 read 400 bytes of file f1 in time t2.) is arrive, and changes the value of a corresponding metric (i.e., the IO throughput), an interval record is created and stored for this change : {IO throughput of process p1 and file f1, 400 bytes, t1, t2}. This record actually shows that the IO throughput of process p1 that is read/written from/on file f1 between time t1 and time t2 is 400 bytes. Here the time t2 is the current event timestamp and t1 is a time that a previous registered event (read event or write event or etc.) is seen and processed.

Using the above technique, we store statistic summary as interval values instead of single values which enables the range queries. Hence, in a general case, to store the interval values, any interval container, such as an R-tree [66], the SLOG2 file format [18] or a State History Tree [102] can be used.

Significant space is required to store the summary of a stream in this way, which means after a short while, both the main memory, and finally the disk will be filled. We use some heuristics to solve this problem. One heuristic uses a proper granularity degree (GD) [50], and stores a cumulative interval instead of each single interval. To do so, for any k (i.e., the GD value) consecutive intervals, we store a single record representing the statistics value between the time ranges of those intervals. In the example shown in Figure 5.8, for each 7 operations, only one data interval is created and stored (instead of 7 separate interval records). A small GD value increases the history storage space while a larger GD value reduces the precision of the statistics. Thus, a careful consideration of the proper GD value is important and can be achieved through balancing the importance of the metrics and the available storage space.

Another heuristic that can be used to alleviate this problem is using the so-called Tilted Time Frame technique [21] to compress the time dimension. The idea is to use a coarser granularity degree (GD) for the older history, yet a finer value for the most recent history.

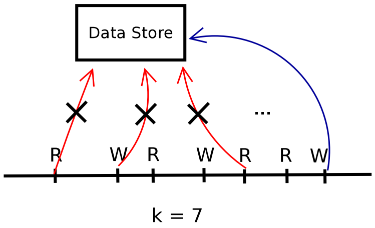


Figure 5.8 Efficient updating the history data store.

Time Frame

In the trace analysis applications, the most recent history is more interesting than older history [69]. Thus, for the long term history, we can use coarser granularity degrees. For example, as shown in Figure 5.9, for the last 5 minutes of the trace a minimal granularity degree (e.g., 1 second) is used, and for the last 24 hours range, a larger granularity degree (e.g., 5 minutes) is used. With this method, after reading any 5 minutes of the input stream, aggregate statistics of the last 5 minutes are calculated and passed through the cube in the other level. After 60 minutes, another aggregation is calculated and stored in another cube level.

A high-level view of the time frame is shown in Figure (5.10). In this method, for the duration of each time unit, a separate stream cube is constructed and materialized. The cube is actually implemented using the dimension (metrics) tree and the history data store. In other words, to manage and store the cuboids we do not use any external database applications, as is common in the OLAP applications, but instead we implement each cube using two tree-based structures, the one is used to mange the dimensions (dimension tree) and the interval tree that contains the real data for the current time frame (history data store) (Figures 5.6, 5.14).

Each cube is used to answer the queries inside the corresponding time duration. After passing this time yet before dropping the cube, an aggregate of this cube is computed and stored in the cube at one step coarser.

Using this method, one can store a history of the statistics values for a long time and utilize a small amount of storage space. We will now explore the storage space this method requires to keep track of the intervals.

Let us assume there exist n metric nodes in the dimension (metrics) tree. We also assume that all metrics have changed each second and a new interval must be created for each metric and each second. For the recent 5 minutes, $5 \times 60 \times n = 300n$ record spaces are required to store all the interval values for these n metrics. In the same way, the other

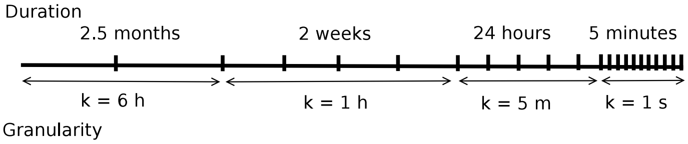


Figure 5.9 Different granularity degrees for different time durations.

24 hours require : $(24 \times 60)/(5 \times n) \approx 300n$. Similarly, for two weeks and 2.5 months, $(14{\times}24)/(1{\times}n) \approx 300n$ and $(75{\times}24)/(6{\times}n) \approx 300n$ respectively are required. Thus, for a 2.5-month period, $300n + 300n + 300n = 900n$ record spaces are required which is 1/10000 of the case that uses a uniform time frame (i.e., $900n/(75{\times}24{\times}60{\times}60{\times}n \approx 6480000n) \approx 1/10000$).

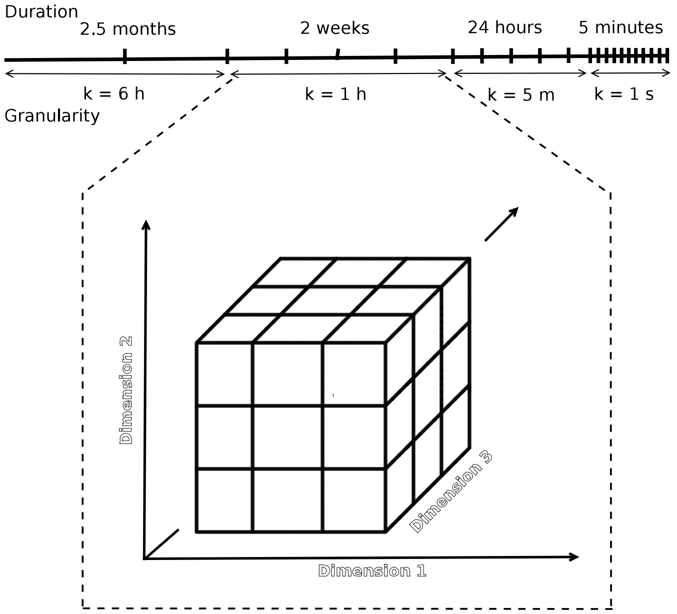


Figure 5.10 A separate sub-cube for each time unit.

Cubes Construction

Different trees (cubes) correspond to different time units are created, thanks to the tilted time frame technique. When the time unit is changed, a set of aggregated values must be passed to a granular tree (cube). In other words, after crossing the first tf_1 time units (e.g., the first 5 minutes), this method aggregates the statistics and inserts these aggregated values into the tree of another time unit, such as tf_2 (Figure 5.11). This also occurs after crossing any other tf_1 time units (e.g., each 5 minutes). In the same way, after crossing tf_2 time units (e.g., 24 hours), it must perform another aggregation of the values of tf_2 time units and insert into the coarser time unit tree. This process is repeated after passing each single time unit.

As shown in Figure 5.11, a separate cube corresponds to each time unit, one for the area with minimum time scale tf_0, one for the area with the minimum time scale tf_1 and so on. After passing the first tf_1 time, an aggregation process is called and the aggregated records are inserted into the other level tree. This process is repeated when it passes the tf_2 time. Let us now explore the costs of these aggregate updates.

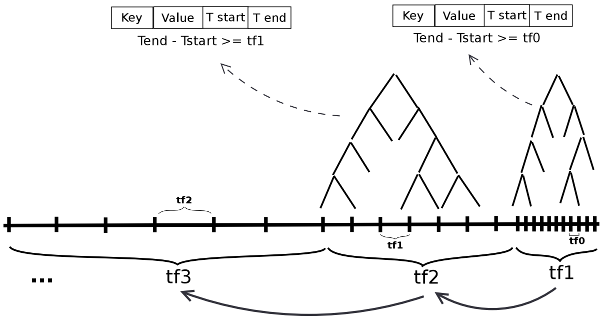


Figure 5.11 Moving the aggregated values from one tree to another.

Let us assume again that there are n metrics in the metrics tree and the statistics values for all metrics are changing at each time unit. After crossing the tf_1 time, we must aggregate all values of the tf_1 time period and insert them into the cube corresponding to the tf_2 period (Figure 5.11). Since there are n metrics in total, each metric requires one record ; thus, n records for all metrics together. Each record shows the statistics value of the corresponding metric in the completed time range. Thus, at each step any time unit change requires n aggregate updates into the higher-level cube. Therefore, using the parameters shown in Figure 5.11, formula 5.1 can be used to calculate the number of updates from one tree to another.

$$\psi_{all} = (tf_2/tf_1) \times n + (tf_3/tf_2) \times n + \cdots + (tf_n/tf_{n-1}) \times n. \tag{5.1}$$

The above formula calculates the required number of aggregate updates from one tree to a coarser level tree. The resulting value is a very small portion of all whole tree insertions :

Let us calculate the total number of insertion operations in all trees and compare that to the number of aggregate update operations. Suppose that for each tf_0 duration (the smallest time unit in the proposed time frame for which values can be gathered directly from the input trace events), the values of all metrics are changed. Thus, we will have n updates for each tf_0 duration. For the larger level, tf_1 duration, the number of insertion and update operations in the history will be :

$$\Phi_{tf_1} = (tf_1/tf_0) \times n. \tag{5.2}$$

Similarly, the number of insertions for each tf_2 duration :

$$\Phi_{tf_2} = [(tf_2/tf_1) \times ((tf_1/tf_0) \times n)] = (tf_2/tf_0) \times n. \tag{5.3}$$

And :

$$\Phi_{tf_m} = [(tf_m/tf_{m-1}) \times ((tf_{m-1}/tf_0) \times n)] = (tf_m/tf_0) \times n. \tag{5.4}$$

Totally :

$$\begin{aligned} \Phi_{all} &= \Phi_{tf_1} + \Phi_{tf_2} + ... + \Phi_{tf_m} \\ &= (tf_1/tf_0) \times n + (tf_2/tf_0) \times n + \cdots + (tf_m/tf_0) \times n = \frac{\sum_{i=1}^{i=m} tf_i \times n}{tf_0}. \end{aligned} \tag{5.5}$$

Therefore, the fraction of the aggregate updates with respect to all insertion and update operations equals :

$$Portion = \frac{\psi_{all}}{\Phi_{all} + \psi_{all}}. \tag{5.6}$$

For the example shown in Figure 5.9, this proportion is 0.00006 = 0.006 %. In other words, the aggregate update cost is a very small proportion of all operations and is not an issue. The challenging issue is the time required for the tree construction, which will be investigated in the Experimental Results section.

We use another heuristic to reduce the number of tree insertions : when a metric value is unchanged in two or more consecutive time units, no update is required in the tree. To do so, we store the current value of the metric in a temporary structure and wait for a change. After the first change, a node is inserted in the tree representing the value of the metrics for all unchanged time durations. This technique reduces the number of insertions in the history data store.

Sliding Window

One use-case of this research is to support the sliding window queries, both the fixed and moving sliding windows. In this subsection we propose a technique to support the sliding window queries.

As explained, at regularly defined time points, the algorithm aggregates the values of the current tree and inserts them into a coarse tree, belongs to a larger time frame.The fixed sliding window is obviously supported, because the values of any previous time ranges are available in the data structures. But to support the moving sliding window one inefficient way is to update the history for all new trace events. In other words the algorithm should update all trees at any granularity levels for each new event, after passing the first tf_0 time (e.g., a second). However, it would be too costly to update the tree structures, remove the old entries and insert a new one, for each new time unit tf_0 (e.g., a second).

The way we support the moving sliding window is by delaying the aggregate moving from one tree to another tree. In other words, we do not immediately move the aggregate value to another tree, but wait for another tf_1 time and then aggregate the tree and move to the coarser tree. Figure 5.12 depicts this technique (with respect to Figure 5.11).

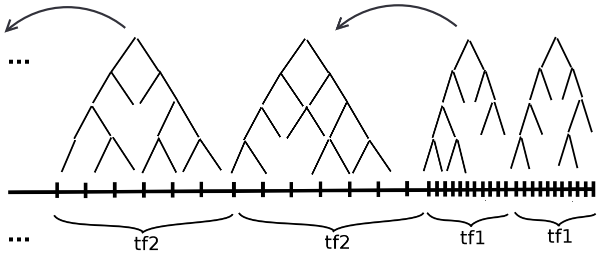


Figure 5.12 Supporting the moving sliding window by delaying the aggregate updates.

To illustrate this point, we use the values shown in Figure 5.9. Instead of aggregating the tree for the last 5 minutes and discarding the detailed tree, we keep these details in the memory and continue for another 5 minutes. At the end of the second 5 minutes, we simply aggregate the first tree, (the tree from the first 5 minutes), move it to a coarser level tree, and discard that tree from the memory. At the query time, when users ask for the last k, say 7, minutes, we can easily use these two trees (the trees of the first and second 5 minutes) to answer the query. As shown in Figure 5.12, the duplication and delay in aggregate propagation are used in all time frames, enabling the extraction of the desired statistics values for any arbitrary k time units with a varying time precision.

5.6 Query

The proposed method, as explained in the previous sections, reads the input trace events and extracts the statistics from the data. Then, the statistics data is stored in a tree-based history data store. In this history, the more details are stored for recent data and less for older data. Using this configuration, it is possible to reduce the details from the older history, while still satisfying the queries for recent history with a higher precision. In general, different types of queries are supported. We first look at the range queries and subsequently address the other types of queries.

5.6.1 Range Queries

The records in the history data store, collected from the trace events, represent the cumulative statistics values for the specified metrics and the corresponding time range between the start and end points. Thus, to retrieve the statistics value at any point, one can simply find and explore the corresponding tree and find a node that contains the required point. The extracted value represents a cumulative statistic between that query point and a base time point. In the same way, to extract the statistics values for a time range, one may perform two stabbing queries and subtract the results to yield the desired statistics value. Figure 5.13 shows an example of a range query.

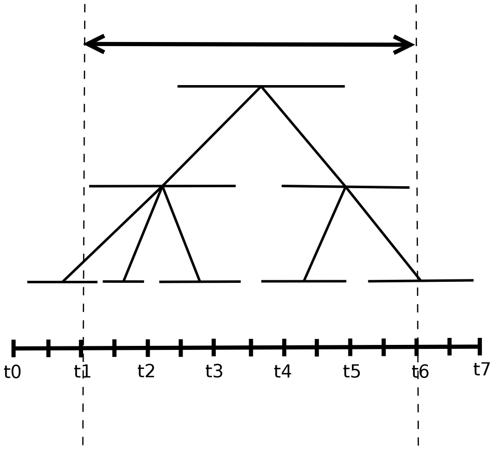


Figure 5.13 Performing range query in the stream history.

As shown in Figure 5.13, to extract the statistics values in any time range, say [$t1$, $t6$], two stabbing queries are required, one for $t1$ and another for $t6$. These two stabbing queries return two cumulative values with respect to a fixed start point. Then, the subtraction of the two values will provide the required statistics value (the increment within the interval).

Example : Suppose we have three records in the history : ([0,2), metric 1, 0), ([2,6), metric 1, 20), ([6,10), metric 1, 30). Each record contains a time interval, a key and a value. For instance, the first record shows that the value for metric 1 between times 0, 2 is zero. These records show that we have value changes in times 2, 6 and 10, since we create a new interval record only when a value changes, (here the GD value is 1). Using this history, the metric 1 statistics value between any time range in [0,10), say [3,9], will be the subtraction of the statistics value at 9 and 3 = 30 - 20 = 10, because time point 3 crosses the second interval record and time point 9 crosses the third record.

The stabbing query -finding the intervals that contain a given query point- is shown in Algorithm 3. Since our proposed solution does not force using the interval tree container, Algorithm 3 shows a general stabbing query for any typical interval trees. The maximum number of items in the result list L will be n, the number of metrics. Sometimes it is possible to have less than n intervals in the result list L. This means that some metrics may not have values for the given point, meaning that the corresponding resources were not active at that point, e.g., looking for the IO throughput of a process in a time prior to its starting time.

After performing a stabbing query, the result set L will contain statistic values of all metrics. Thus, one must filter out the result list L to find the value of the desired metrics.

ALGORITHM 3: Stabbing query.

```
Require: an interval tree v and a query point t.
if root node r contains the point t then
    add r to the result list L.
end if
if there is any children for node r then
    for any children of v like c(v) do
        call the algorithm for c(v) , t.
    end for
end if
return list L as result ;
```

As explained earlier, we have different trees for different time points : more details for recent times and less for older times. The proposed stabbing and range queries work for all cases, for a time range within a single tree or several trees. For all cases, we can find the statistic values for the desired time range, by using two stabbing queries for the boundary points of each interval.

The last point in this section relates to calculating the aggregate values of a tree and moving that to a coarser tree. To calculate the aggregated values of all metrics for any time range, one must perform two stabbing queries, one at the start and one at the end point of that time range. Each stabbing query returns the values for all n metrics together, so it is not necessary to repeat this algorithm for each metric separately. Therefore, calculating the aggregate values of a tree is not costly. It simply requires two stabbing queries for the start and end points of the tree.

Top-K Queries

Sometimes it is important to detect when the values exceed a predefined threshold, or find virtual machines or processes which consume more system resources than others. To support these query types, the system should be able to answer top-k queries, for any arbitrary k. To do so, we first use the mentioned range queries to extract the statistics values of all metrics, and then use an optimal sorting algorithm to find the top-k values from a maximum of n values.

The cost of this algorithm is $O(logm_1 + logm_2 + n \times logn)$, where m is the number of nodes in the history tree and n is the number of metrics. In this equation, $logm_1$ is the time required to perform the stabbing query for the history tree at the start point of the given query interval, $logm_2$ is used to extract the statistics value at the end point of the query interval, and $n \times logn$ is used to sort the n items, the output of the stabbing queries. The cost is obviously dependent on the count of metrics, n, and the depth of the tree, as one or the other may be more dominant depending on the situation.

5.6.2 Sliding Window Queries

This solution supports both the fixed and moving sliding window queries. As outlined earlier, the techniques used enable us to extract the statistics values for the last k time units for the fixed or moving values of k. An example of a fixed sliding window is reporting the statistics values after each k time units. In this case, after finishing the predefined k time units, say 1 second or 1 minute, the algorithm aggregates and returns the values for the desired time range. For instance, one may wish to retrieve a minute by minute report of the CPU usage. To do so, after finishing each minute, the program aggregates and reports the CPU usage for the preceding minute by summing up the values of small-scale chunks (e.g., by summing up the CPU usages of all 60 seconds of that whole minute).

Sometimes users wish to obtain values for a moving or sliding value of k, by taking into account the precision of the time unit. Suppose that we are in the 6th minute of the trace and the user asks for the CPU usage for the last 3 minutes. If we had aggregated the tree at the 5th minute, we would not be able to answer this query, since we need 2 more minutes from the history. However, we would not have the requested values with the desired precision if these values were already aggregated for that 5-minute interval. To solve this problem, the algorithm delays moving the aggregated values to a coarser unit tree for another time unit (e.g., another 5 minutes). For example, using the tree shown in 5.12, it is possible to provide values for any previous time range (less than the current time unit) e.g., last 3 minutes, last 58 minutes, last 11 hours, etc.

5.6.3 Multi-level Queries

Since in many applications, users may wish to perform some operations like group by, drill down or roll up, we investigate this type of queries here.

Due to the required storage space and processing time, it is not possible to generate and store all possible cuboids along the stream. To support multi-level queries, we propose two general solutions : minimal and partial cube materialization.

The first solution, minimal materialization, defines all metrics in the finer level - for the leaf nodes of the dimensions- and only stores the base cuboids (the history values for the leaf nodes of the metrics tree). Any other high-level metrics (none-base cuboids including apex cuboid) are computed on the fly using these low-level metrics. For instance, in the above example, it is possible to compute the IO throughput of virtual machines by performing aggregate functions (i.e., sum) over the low-level history values, i.e, by summing up the IO throughput of all processes belonging to each virtual machine. Figure 5.14-1 shows a view of this solution.

Partial materialization [69], the second approach, is used when the high-level nodes, for which the analytical data will be queried, are predictable. Therefore, the solution creates metrics for these high-level nodes and keeps track of history values for them as well. For instance, in Figure 5.7, suppose the system is notified in advance that users wish to retrieve the IO throughput for all virtual machines together in addition to each process separately. In this case, a metric node representing the desired granular level is created in the metrics tree and a history is kept within the history data store. Figure 5.14-2 depicts a view of this solution.

Depending on the number of high-level nodes, the partial materialization solution may require more storage than the minimal materialization solution to store the data for the coarser or finer granularity scales. However, the later may require more processing time to aggregate and compute the desired high-level statistics on the fly using the low-level information.

A tradeoff between the processing/response time, the storage space and the user and application requirements is generally required to determine which strategy should be used. In some applications, a small number of high-level nodes may be critical to users and therefore data should be kept to directly and quickly retrieve answers. In this case, a partial cube materialization method is used to only materialize the important high-level nodes in addition to all leaf-level nodes.

To extract the high-level statistics (rolling up) for a given time point, when the requested value is not directly in the history, one must perform a stabbing query at the given point, extract the values of all low-level nodes of the queried dimensions and finally aggregate the results (sum up, count or so on.). Since a single stabbing query will return all values for the given point, the rolling up query requires the same time as low-level queries, except for the extra time required to perform the aggregation over extracted values. For instance, to compute the IO throughput of a folder, one must aggregate the IO throughput of the files

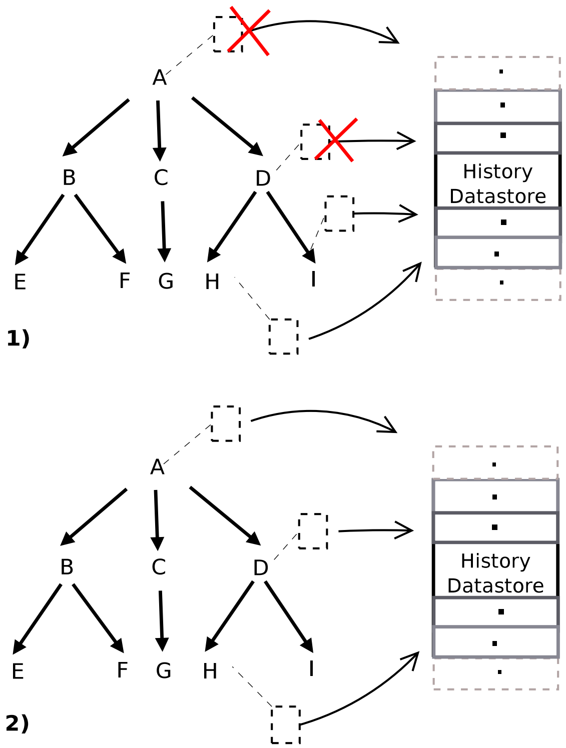


Figure 5.14 Hierarchical queries. 1) Minimal cube materialization, using aggregate functions to compute hierarchical values, 2) Partial cube materialization, storing data at the multiple levels.

inside that particular folder, which can be obtained directly with a single stabbing query.

5.7 Experimental Results

The experiments were performed on a Core i7 2.80 GHz system with 6GB of main memory, running Linux kernel version 2.6.38.6 instrumented with the LTTng tracer. The algorithms were programmed in Java using the Eclipse plug-in for Java and will eventually be contributed to the free software TMF (Tracing and Monitoring Framework) [2]. The tests were performed with real trace logs gathered from the LTTng kernel tracer. Since the original logs are too low-level, techniques are adopted from [49, 50] to abstract out the raw data to higher level and extract the desired statistics data. We also use the locally developed State History Tree [102] as the interval container for the interval values. With respect to the other approaches in the literature, this format works better for cases in which the input data arrives sporadically (in an unpredictable manner) and cases in which the tree is constructed incrementally. To generate the trace logs, system activity is generated using recursive operations like grep -r,

[2] http ://lttng.org/eclipse

wget -r -l, ls -R, etc.

The experiments will be discussed in three sections : processing time, memory usage and query response time.

5.7.1 Processing Time

In the first experiment, we aim to investigate the efficiency of the proposed trace analysis module, and whether a trace stream can be processed in real time (events are analyzed in less time than their rate of occurrence).

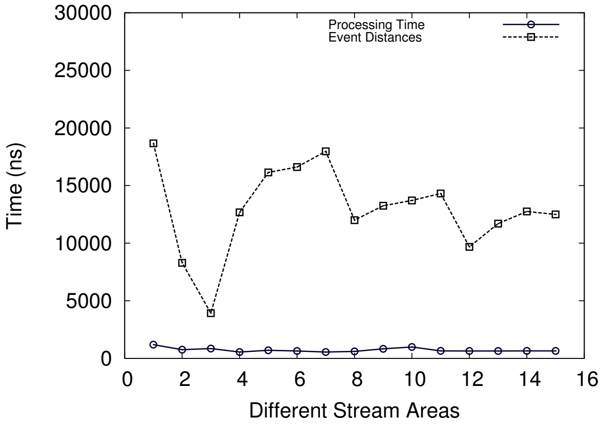


Figure 5.15 Delay between trace events and processing time for one event (average over a batch of 10000 events).

Figure 5.15 shows the average delay between trace events in different areas of a trace. Each delay is calculated as the average time for 10000 subsequent events. Similarly, the processing time to analyze the trace is computed as the average over 10000 events. The analysis time per event does not vary much. The average delay between events varies significantly depending on how busy the traced system is. In our tests, the processing time remains much lower so that the average delay and the analysis is thus efficient enough to accept a streaming trace in real time. In extreme cases, where events are much closer to each other, buffering the events and performing a delayed processing, or even dropping some events, will be required. We may investigate these techniques in more detail in future work.

Figure 5.16 shows the different times required for reading and processing the trace stream. The first case only reads the trace without processing any data. The other curve shows both the trace reading and simple processing of the events : reading the trace stream, extracting the statistics values and aggregating each base time unit (e.g., 1 second). However, it does not include the time needed to store this data in the data structure. Other cases show the time required to store the processed data to the different cubes in the history data store. In each case, the time frame is set so that the requested level cube is constructed. For example, in the case with only the first time unit, one cube is materialized, while in the case with two time units, the first and second levels are materialized and so on. Figure 5.16 shows that the first cube level requires the longest processing time. This is not surprising since the first cube level is updated for each base time unit (e.g., 1 second), while the others are called at granular time units (e.g., 5 minutes) and require less time.

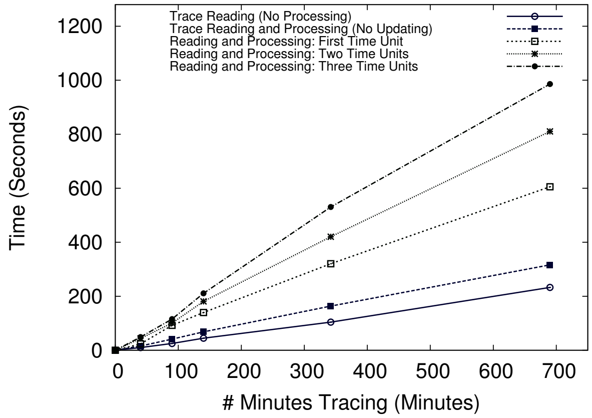


Figure 5.16 Processing time for different stream processing steps.

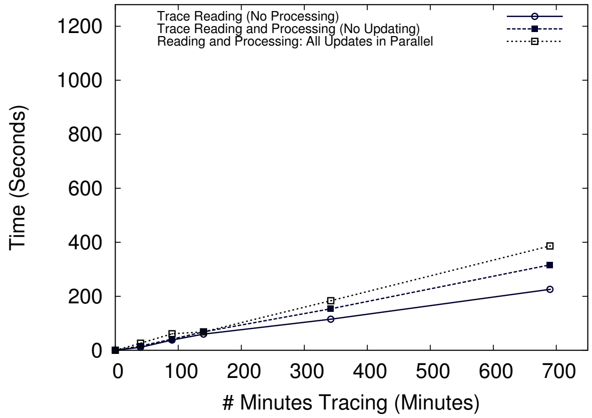


Figure 5.17 Processing time for parallel cube updating

Figure 5.17 shows the same processing steps while performing a parallel analysis, in which a separate thread is assigned to each single cube update. The parallel cube update is possible since different cubes correspond to different areas and the data gathered from unique trace portions can be written to the different cubes simultaneously. In this method, after performing a preliminary analysis over the trace and extracting the statistics data, a separate thread is assigned to each cube and, as a result, updates are done separately yet parallel to each other.

It is important to note that in all the above experiments, 1000 measures are used. As explained earlier, each metric is considered as a measure between two or more dimensions. For instance, the metric CPU usage is defined between the virtual machine, process and CPU dimensions and could also be shown as (virtual machine, process id, CPU number, CPU usage).

5.7.2 Memory Usage

One of the important aspects of any useful stream processing method is the ability to use as little memory as possible. In this section, we show the memory usage for our proposed method. The history data store, which records the temporary and intermediate values, stores the data on disk rather than in memory, enabling it to store a longer period of abstract streaming data.

As explained earlier, there are three cube materialization methods : full, partial and minimal. In all experiments below, the partial materialization stores 10% of higher-level cuboids in addition to the lowest level cuboids.

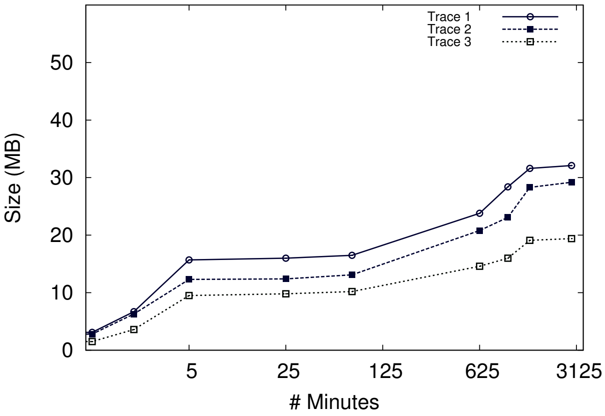


Figure 5.18 Memory usage for different trace areas.

Figure 5.18 shows the memory usage for different areas of the trace stream. Different trace areas may have more or less events depending on how busy the underlying system is. In Figure 5.18, the curve shown with Trace 1 belongs to a busy area of the trace (where the number of events per second is higher than in other places). The data used for drawing the curve shown with Trace 3 belong to a less active area of the trace, with fewer events. In all three cases, the number of metrics used is again 1000. Additionally, the time units are from

Figure 5.9. In other words, the memory is used to store three levels of cubes : the first level for the last 5 minutes, the second for the last 24 hours and the third for the last 12 days. However, we have used a one-day trace duration to test all three cube levels. The results show that the configuration used is desirable and readily usable. The maximum required memory is approximately 35 MB, which can easily reside in the main memory. However, as will be discussed shortly, the required memory may increase depending on the number of metrics or time units. This low memory usage is achieved at the cost of removing the more detailed history from the most distant time periods. It is obviously not feasible to store all information at the most detailed level for the whole trace duration, unlike for relatively small traces in offline tracing mode [50]. Figure 5.19 gives a comparison of the memory usage of two methods : storing all history for the whole tracing duration, and removing the old history from the data store (please note that a logarithmic scale is used for the y axis).

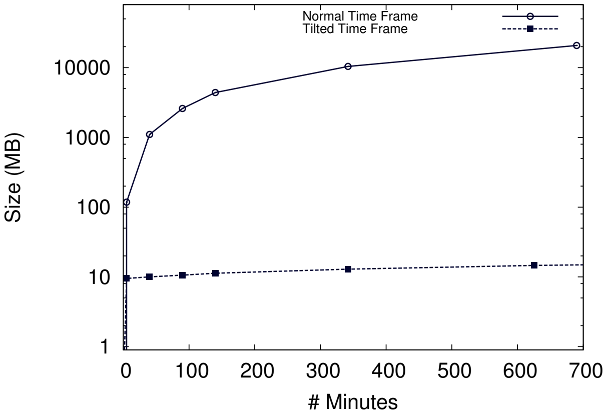


Figure 5.19 Memory usages comparison for tilted and non-tilted time units.

Another experiment was performed to see the effect of the number of metrics. It is possible to define different measures in different levels between the existing dimensions. For example, IO throughput can be defined between a file or folder in one hand,and a process or a group of processes in other hand. It is also possible to add a virtual machine to this measure, making it a threefold measure. Figure 5.20 shows the memory usage for different numbers of measures. In fact, the memory usage depends on both the type and number of measures. Indeed, the count and frequency of the events required to compute the values of a measure is the key factor to compute the memory required to store statistic values of that measure. The number of accesses to a specific web site requires events of type http that connect to the specified web site and can rarely be observed in the events. However, the IO throughput measure is based on the type of events (i.e., read or write events), which occur very often in any system execution, and thus require more storage space. The same metric, IO throughput,

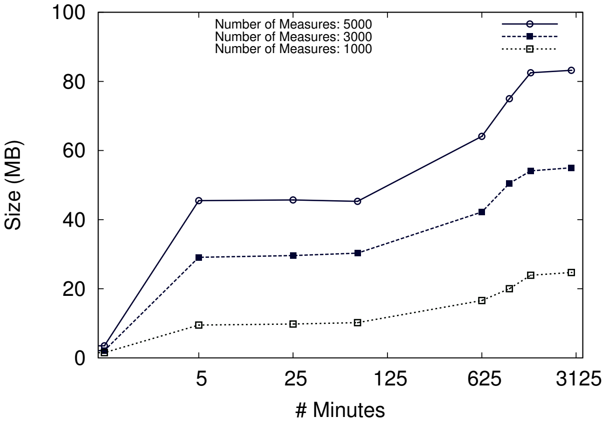


Figure 5.20 Memory usages for different number of measures.

defined between three dimensions, (virtual machine, process and file), is used to gather the results in Figure 5.20. To increase the number of measures, for instance from 1000 to 3000, we have added new tuples (virtual machines, files and processes) to the existing tuple list. The results show that increasing the number of measures has a direct (but not linear) effect on the memory usage.

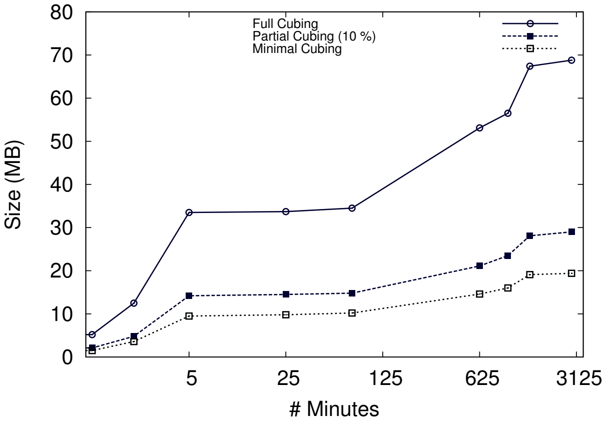


Figure 5.21 Memory usages for different cube materialization strategies.

The final experiment was conducted to investigate the different materialization methods. To do so, three modes are defined for 1000 measures : Minimal cubing, in which all measures are defined in the leaf nodes of the metrics tree (dimension tree) ; Full cubing, in which measures are defined in all levels and the history data is stored for all levels ; and Partial cubing, which is something in-between, defining all but 10 % in the leaf nodes. These 10% extra measures which are defined in the non-leaf nodes, can be considered as measures frequently requested by users, thus being less desirable to compute on-the-fly. Figure 5.21 shows the difference in memory usage for these three methods. Full cubing method demands more memory (three times or more to store the history in each and every level), while the partial and minimal cubing methods act very similarly.

The above memory usage experimental results demonstrate that the size of the data store is relatively stable and independent of the stream data size. Indeed, the design is such that it stores only minimal data for the distant history, and the memory usage therefore increases very slowly (logarithmic) with the size of the stream. This is a very important feature of the proposed data store that makes the solution scalable and usable for any size of input stream data.

5.7.3 Query Response Time

A proper response time is an important factor for most stream processing applications. Indeed, these tools may be used interactively to monitor the system runtime behavior and track problems. We have performed different analysis tests to measure the response time for different configurations.

In the first experiment, the single point query is examined using 1000 measures and for all three materialization strategies. To obtain a comparable result, the same points are queried in all three cube materialization modes. The results show that the base case is the minimal cubing mode, since in that case the history size is smaller than with the two others. Figure 5.22 depicts the comparisons of these three methods.

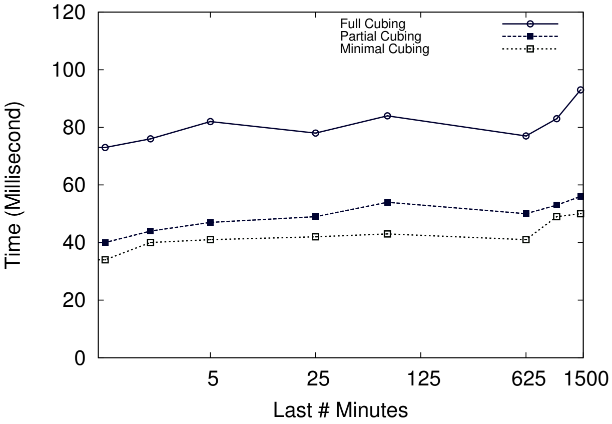


Figure 5.22 Response time for single point queries.

A similar comparison is undertaken for the range queries. As explained earlier, one of the continuations of this work is to support the range queries without having to count and aggregate the values for the whole range. In our solution, this is achieved by performing a

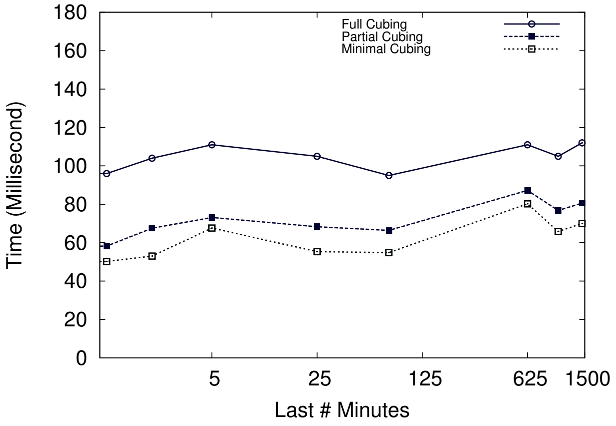


Figure 5.23 Response time for range queries.

few queries for the start and end points of the range and subtracting the values. Figure 5.23 shows this comparison. To obtain the comparison results, different time periods are examined within the last n minutes of the test. For instance, the values in time point 5, are obtained by testing time ranges within the last 5 minutes. Similarly, the results for point 25 are obtained by using random time points within the last 25 minutes. Since the time units are chosen randomly, they could occur in one or more time units. Further, the same ranges are used for all three cases. The results show that the time required to perform the range queries is related to the number of measures and materialization strategy, not the time interval duration.

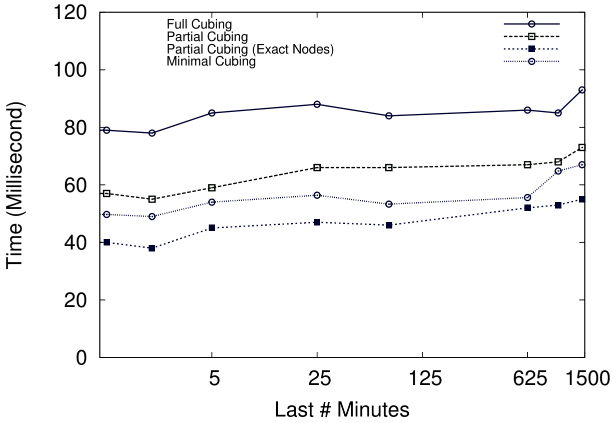


Figure 5.24 Response time for roll-up queries.

Figure 5.24 shows the response time for roll-up queries, to extract a high-level value given the low-level values. In the minimal case, to compute any higher level measure, the values from the leaf level (lowest level) should be extracted from the history data store and then aggregated on-the-fly. For the full cubing, all the values already exist in the history and simply need to be extracted. The partial cubing is somewhere in-between : some non-leaf measures have their own values in the history data store and some must be treated like the minimum cubing. We separate these two cases and consider them separately. Due to the values shown in Figure 5.24, the best query time belongs to the partial cubing with the 10% measures that data are stored. The minimal cubing method is similar but requires slightly more processing time. The results show that carefully choosing the non-leaf measures is an important factor, and can affect the response time. In this comparison, the same time points are used for all four cases and the results are computed for the single point queries.

In summary, partial cubing with carefully chosen non-leaf measures is considered the best solution. The memory usage for this method is almost equal to the minimal cubing, but the partial cubing has the best response time. However, selecting the potential measures to keep in non-leaf nodes is not an easy task. They can be chosen statically by a system expert or dynamically based on the users' feedback and experience. The memory requirements for the metrics (depending on the associated events) or usage statistics could be two important factors in dynamically selecting the non-leaf measures to materialize.

5.8 Conclusion and Future work

In this paper, a multi-level architecture, and corresponding data structures and algorithms are proposed to construct a cube storage for very large, theoretically unlimited, trace streams to enable different multi-level trace analyses. Reasonable memory usage, efficient response time and support of different query types (single point, range queries, drill-down and roll-up, sliding window queries) are important features of the proposed approach. A customized form of a so-called tilted time frame is used to compress the time dimension. In this configuration, a separate cube is constructed for each time frame, where the cubes for the most recent times are kept more detailed, while the cubes for older times are kept less detailed.

Each cube stores the statistics values in the interval forms (it stores the value and also the time range that value is valid) instead of storing the single values. Storing the data in interval forms enables the time range queries, not only within a cube, but also between the different cubes. This feature supports querying the system for any given time range of the input stream.

We have tested the proposed solution by using a stream of execution trace events gathered by the LTTng kernel tracer. The results show the possibility and efficiency of performing OLAP-based multi-level multi-dimensional analysis over a live trace stream. Having this possibility, this technique may be extended to monitor the system runtime behavior and detect different host and network based problems and attacks.

Several experimental results indicate the memory usage and response times of the proposed method for different cases and configurations. The results generally show that the memory and speed of the proposed method is reasonable and efficient. Indeed, for the range queries of any arbitrary length time ranges, the results show that the response time is unrelated to the size of the range. This achievement is important as the proposed solution enables us to efficiently perform long-lived historical (time-based) queries.

We defined the partial cubing using statically defined metrics. However, a possible future work will be to dynamically choose the non-leaf cuboids to be materialized, or to dynamically switch between two solutions (minimal and partial) based on the users' feedback or the defined queries. Extending the proposed solution to detect system problems and conduct complex analyses with data mining techniques is another possible future work.

CHAPTER 6

Paper 5 : Fast Label Placement Technique for Multilevel Visualizations of Execution Trace

Naser Ezzati-Jivan and Michel Dagenais

6.1 Abstract

Automatic label placement in a graphical display is an important problem in many domains and applications such as cartography, online maps and graph drawings. This paper describes a label placement technique that uses a collection of heuristics for placing labels along a number of parallel lines with designated points or line-segments that need to be labeled. The proposed label placement technique aims to maximize the number of labeled items as well as increase the quality of the labels and the efficiency of the method. Assigning multiple labels to each trace item is also supported in this method. The algorithm takes into account both the topological and semantic relationships between the trace items in order to achieve placements that are both quantitative and qualitative. The efficiency of the method is obtained by partially evaluating the candidate labels against the other labels and items. The proposed algorithm uses a dynamic preference method to increase the quality and readability of the assigned labels. The algorithm has been implemented and evaluated using different input trace sets. The experimental results show that considering the relationships between data items and dynamic preferences of the labels increase the labeling success rate and their quality, respectively.

6.2 Introduction

Execution traces are mainly used to analyze system runtime behavior and have applications in problem detections and execution comprehensions. One of the problem with execution traces is their size that can shortly become huge. Visualization is one solution for this problem that can enhance the trace analysis and comprehension [122].

There are many visualization techniques and tools to visualize execution trace events. These tools usually use space-time (timeline) diagrams to display the trace events, in which, one axis used to display the system resources (processes, modules and so on) and the other one to display the time dimension. In this diagram, the events, states or function calls are displayed along the horizontal line and usually labeled with a name.

Although the space-time diagram is used widely in the trace visualization tools, there are some problems with that. One problem is that it can only display a few elements in the screen, due to its limited display area size. With a limited screen size, one intuitive visualization style for massive data (e.g., execution traces) can be organizing data in a hierarchical fashion and displaying them at multiple levels of detail. In this style, the timeline view can show first the objects at a higher granularity level, and let users to move and pan in the trace and go up and down, by providing zooming and focusing operations.

But even with multiple visualization of trace events, all exiting objects of the view may not be displayed or labeled, due to the screen size limitation. Therefore, an efficient label placement algorithm may required to assign proper texts and labels to objects, increasing the overall readability of the view. Efficient placement of labels to trace events in a multi-level visualization tool is our motivation in this paper.

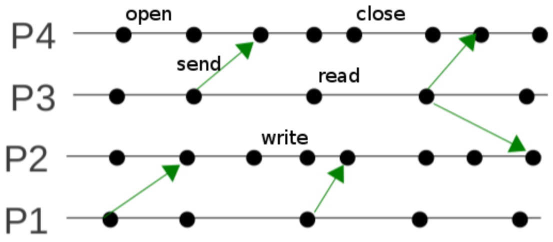


Figure 6.1 A view of the visualization pane.

The focus of the paper is on consistently and smoothly assignment of labels to trace items in multiple levels parallel timeline views. To explain better the problem addressed in this paper, we assume that the visualization pane is divided into separate lanes to place the events and their labels, representing the actions, messages and executions of active processes, as shown in Figure 6.1.

As the main contribution of the paper, we propose an efficient labeling algorithm and also the corresponding heuristic methods to dynamically assign qualitative and conflict-free labels to trace items at multiple levels of detail. This is done to ensure that the screen remains consistent and uncluttered by knowing in advance that : there are lots of events to display, the fact that events are not distributed uniformly and the scale changes by orders of magnitude.

Most of the previous work on label placement focus on graph labelling and cartography. The problem we aimed to solve is notably different. In our problem, the dimensions of the plane are used asymmetrically : labels along the x axis are much more acceptable than label offsets along the Y axis. The other important difference is that users can scroll and slide in all three dimensions : horizontal sliding for moving along the time, vertical scrolling for displaying the execution of other processes, and zooming in/out in Z axis for displaying the other detailed/overview levels of execution. In other words, in the general case of the problem, sliding can be done in 3D, making for more degrees of freedom, thus more complexity. These are the major differences that distinguish this work from most related art, and makes the algorithmic apparatus needed distinct from graph and cartographic labelling.

The cartography labeling techniques usually use a static cartographic preference to prioritize the candidate label positions, however, our proposed algorithm dynamically sets the preference places of the existing candidate labels, increasing the quality of the label-item associations as well as decreasing the ambiguity of the overall view. The method also takes into account the structural and semantic relationships between trace items to increase the number of assigning labels.

The structural relationship means using the geographical location of the items to inspect only the neighboring items (instead of all other items) that might conflict with them. This way avoids comparing a label against all other labels (and items) to discover the possible conflicts, increasing the speed of the labeling method.

The semantic relationship, in the other hand, is used to generalize and merge the consecutive items that are somehow related to each other (e.g., write events, send messages or other items that are being repeated consecutively or belong to the same high-level concept), when not enough space is available to place separate labels on all visible items. While this would not be useful in geographic applications (e.g., online maps), these relationships might be used very efficiently in labeling algorithms in the trace data context.

This research is part of a broader goal to develop a multi-level trace modeller and visualizer to display the operating system (kernel) level trace data at multiple levels of granularity. This paper focuses on placing labels on the kernel level trace items in a multi-level timeline view. The term ”trace items” is used to refer to the different types of data, including the events gathered directly by the LTTng kernel tracer [38] as well as the high-level information generated by different analysis modules [49, 50].

In the remainder of the paper, we firstly discuss the different automatic label placement approaches and their applications. Secondly, the details of the proposed algorithms for multi-level placement of trace items labels are described. Then, experimental results are presented and discussed, and finally the conclusion and specific areas of investigation for future development are outlined.

6.3 Related Work

We explain the related work from two points of view : the type of label placement algorithms and their complexities.

6.3.1 Static vs. Dynamic Placement Algorithms

Label placement originated from Cartography [149]. Although most of the related research is being conducted on map labeling problems [22, 57], label placement has other interesting applications including graph drawing [40, 76] and timeline visualization [89].

Label placement techniques mostly focus on static map labeling, in which the goal is to find the largest set of map objects that could be labeled, without conflicting with other labels and objects. Dynamic label placement methods [11], in the other hand, mostly refer to multi-level labeling by a support of repeated zooming, panning and hierarchical navigation. These methods are more concentrated recently [11, 12, 116].

Petzold et al. [116] propose a dynamic approach by using ”reactive and static” conflict graph data structures and processing the data within two phases. This solution is based on reducing the multi-level data set to the visible scale and applying a static labeling algorithm over the constructed conflict free labels set at the selected scale. Ken Been et al. [11] formulated a general optimization problem for dynamic labeling and addressed the consistency problem of the Petzold solution. However, neither consider the semantic relationships between the data items that can compress the labels and increase the efficiency of the method in the contexts that this kind of relationships is applicable (e.g., in the context of execution trace data).

The technique that considers relationships of the data is explained in [89]. Li et al. use a static exhaustive search algorithm to place labels on the visible data objects, in their LifeLane tool. They later enrich their placement algorithm by generalizing and merging the labels as well as using some interactive techniques, like highlighting a selected object and applying user feedbacks. Compared to this technique, our approach uses more aggregation methods (dynamic and static aggregations), and also supports placing multiple labels on one single item. Moreover, the topological relationships of the items and partial evaluation of the conflict graph are taken into account, which leads to the improvement in the efficiency of the algorithm.

It is important to note that most of the literature on label placement is concerned with graph labelling and cartography. Here the problem is notably different in that the two dimensions of the plane are used asymmetrically : label offsets along the x axis are much more acceptable than label offsets that cross a lane boundary (in Y). This is a major difference that distinguishes this work from most related art, and makes the algorithmic apparatus needed distinct from graph and cartographic labelling.

6.3.2 Complexity of Algorithms

Under specific aesthetic constraints deemed desirable, symmetric label assignment problems are NP-complete [22, 97]. Researchers thus use heuristic methods like simulated annealing [22], approximation [31] and exhaustive search methods [89] to alleviate the problem, in both the static and dynamic approaches. In our case, since the tool is aimed at visualizing massive trace items, with a possibly large number of user interactions, both the rendering speed and label placement time are important. We therefore use a modified version of the generic greedy algorithm. Greedy methods [22], while fast, only provide poor quality solutions, in which some items may left unlabeled. For this reason, a post processing phase is added to exploring the possibility of merging several labels into a single label, by taking into account the relationships between trace items.

Using a so-called conflict graph data structure is common in label placement algorithms, as used in [11, 147]. However, these solutions mostly describe how to use a pre-constructed conflict graph to find a maximal independent set (i.e. a conflict-free label set) of the input set. But, constructing such a conflict graph requires checking each item against other items to see whether their labels intersect or not, which can be a costly operation. Especially, when there is a large number of items, and also in the case of dynamic visualization, in which the number of labels on the visible screen are not completely known in advance. To solve this problem, we propose a solution to avoid constructing the complete conflict graph by partially materializing the conflict graph which increases the speed of the labeling algorithm.

6.4 Multi-level Label Placement

In this research, we aim to visualize trace items at different levels. The visible screen is divided into different lines, as shown in Figure 6.1. Each line corresponds to a distinct process. The trace items are divided based on their owning process names, and placed and shown in the line titled with that process name. The goal is to assign as many labels as possible to visible trace items at each selected scale. We assume that there is no limitation on the maximum number of visible items, and any number of trace items can be placed and labeled, regarding the size of the visible screen.

6.4.1 Data Items

The first step is to define what type of data items will be visualized and labeled. As explained earlier, in this project we use the trace data gathered by the LTTng kernel tracer. The different types of gathered trace items that are supported in this approach are as follows :

- Punctual trace events : the basic form of the data to visualize includes timestamped punctual trace events. Systemcall_start, systemcall_end, file open, read, write, and socket close are examples of these trace events. These items are similar to the point features in the map labeling applications.
- Synthetic events : trace events at higher level are usually synthesized by grouping the individual trace events [49]. Sequential file read, HTTP connection, process fork and file download are examples of these durative items. These items resemble the line features of the map labeling applications. The details of the abstraction techniques used to generate the different levels of synthetic events can be found in [49].
- Communication events : inter-process communications and send and receive messages are examples of these items. These are also considered as line features in the map labeling applications.

6.4.2 Label Positions

Defining proper positions for the labels is the first step to enhance the readability of the visual screen. Different positions have been proposed in the literature [108] :

Fixed position model : In this model, there is a constant number (i.e. 2, 4 , 6 or 8) of candidate positions around each item. The size of labels can be fixed or variable to resolve the conflicts. Figure 6.2 shows a 4-position model where each item, represented as a dot, has 4 candidate positions. Each position is represented by a rectangle.

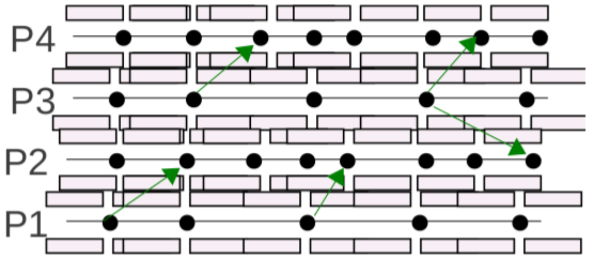


Figure 6.2 A view of 4-position model.

Sliding model : In this model, the label offset can be any x coordinate, therefore, there could be an infinite number of candidate places around each item [75]. A common feature of these places is that they all touch the corresponding item. This model can be one-way or two-way, as the labels can shift in one direction (e.g., left-right) or more (e.g., left-right and up-down).

In the proposed method, as shown in Figures 6.1 and 6.2, the visible screen is divided into equal height horizontal lines. Each line corresponds to a running process. The distance between two lines can be fixed or configurable. If fixed, the height is set as twice the label height and the label height is calculated using a standard font size. We consider the same (but not fixed) height for each label. However, the label width can be of variable size. In this model, there is no consideration of the font size and the algorithm should work for any label height and distances between two neighbouring lanes. For the different data items mentioned earlier, the models are selected as follows.

1. Punctual trace events : the 4-position model represents the label positions for each event (i.e. point), as shown in Figure 6.2.
2. Synthetic events : for consistency purposes, they are treated just like punctual trace events. We place the label around the middle point of the event.
3. Communication events : for each communication event, we consider two candidate positions, on left-side and right-side of its arrow line.

In all above configurations, unlike most techniques discussed in the literature, the priority of the candidate positions is dynamically computed by the labeling algorithm, which increases the association efficiency of the labels, meaning less ambiguity for the labels.

Label Positions Priority

Between the different candidate positions for each label, a priority was defined based on cartographic preferences of the label locations [97]. Figure 6.3 denotes the cartographic preferences of a fixed 4-position model.

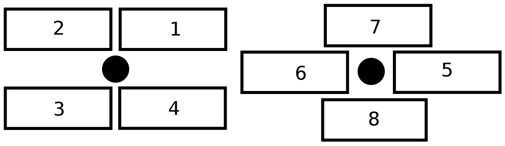


Figure 6.3 Candidate label positions and their preferences ranking.

As will be shown in the algorithms in the next section, the defined cartographic preferences order the selection of a position in the assignment algorithms, and can help to avoid conflicts between labels. However, this might result in an ambiguous association. As shown in

Figure 6.4-a, using the cartographic preferences may lead to an ambiguous association, e.g., it is not clear which label belongs to which item. However, not considering a prior priority between candidate positions, and setting the preference dynamically, can lead to a clear and unambiguous association (as shown in Figure 6.4-b).

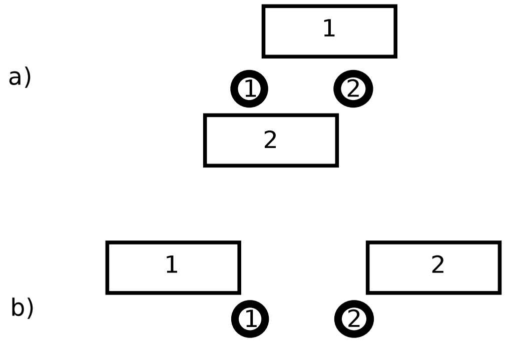


Figure 6.4 Static cartographic preferences (a) versus dynamic preferences (b).

In our approch, to solve the above problem and to achieve a high degree of unambiguity, a dynamic preference method is used. The method starts with the static preferences defined in 6.3, but it dynamically changes the preferences based on the ambiguity degree of each position. The algorithm that uses the dynamic preference method is explained in the next section.

6.5 Labeling Algorithms

The proposed labeling algorithm aims to label the maximum number of items and tries to assign appropriate labels for the input items, such that [149] : (i) labels should not be in conflict with other labels and items. (ii) there should be a clear association between labels and items. (iii) labels must be readable while the font size and style are reasonable.

Algorithm 4 represents the steps required to assign labels to different items on the screen. There are different levels of trace items and, at each level, the trace items hold different importance and priorities. The algorithm works as follows : initially, when users open the screen or change the resolution, the algorithm finds the right scale and outputs the labels for that scale. Then, a static labeling algorithm is applied to assign the labels to the items, while considering their importance and weight. This process may leave some items unlabeled, when there is not enough room to display all labels separately. Therefore, a post processing phase, considering the relations between items is applied to integrate the remaining items and assign labels to the new aggregated items (Algorithm 4).

Each step of the procedure shown in Algorithm 4 is discussed in details in the following.

ALGORITHM 4: multi-level labeling algorithm

```
Require: a hierarchy of trace items and the display size
find the most appropriate level in the hierarchy based on the visible window dimensions;
extract the labels;
perform a static labeling algorithm to assign labels;
if all items labeled then
    return the label assignments;
else
    perform a post-processing step and integrate the labels based on their relationship patterns;
end if
return the final set of overlap-free label assignments;
```

6.5.1 Appropriate Level Selection

As shown in Algorithm 4, the first step is selecting the right scale with respect to the input screen size. Since the number of pixels of the available screen is limited, to insure a clear output, we should firstly visualize the level that has a proper number of events with respect to the screen size.

To do so, a hierarchy of events is created using trace abstraction techniques, in the trace reading and processing phase, and is stored in tree based data structures [49, 50]. Trace abstraction usually use pattern matching or state machine techniques to match and integrate the trace events with the predefined execution models that are described in a pattern library. This hierarchy actually models the events at different levels of resolution, in which the highest level denotes more abstract data, while the lowest level shows more detailed data.

A threshold value is predefined for each scale, e.g., τ_1 for $level1$, τ_2 for $level2$, and τ_n for $leveln$. The threshold for each level denotes the number of events and labels for that level which can be shown with no (or few) conflicts. This number can be set by a program, by considering several previously studied experiments, or can be set manually by an administrator. For calculating the proper value, the number of events in the selected area of the traces should be known in advance. The number of events for any selected area, e.g., $\delta t = t_2 - t_1$, can be approximated by the following formula :

$$\eta = \frac{\delta t * \overbrace{N}^{\text{count of all events}}}{\underbrace{\Delta T}_{\text{whole trace duration}}}. \tag{6.1}$$

It is an approximate value because the trace events are not necessarily distributed uniformly within the trace duration. Some parts may have fewer events while some other parts that were more active may encompass more interactions and events. Although it is an approximate value, it can be used to estimate the proper scale. A post processing phase, as will soon be discussed, fulfils the process and recovers any accuracy possibly lost here.

Finally, using the number of events in the selected area, the proper scale value can be calculated using Formula 6.2 :

$$scale = \begin{cases} level\ \ 1 & \text{if } \eta \leq \tau_1 \\ level\ \ 2 & \text{if } \tau_1 < \eta \leq \tau_2 \\ \ \ldots & \\ level\ \ n & \text{if } \tau_{n-1} < \eta \leq \tau_n \end{cases} \tag{6.2}$$

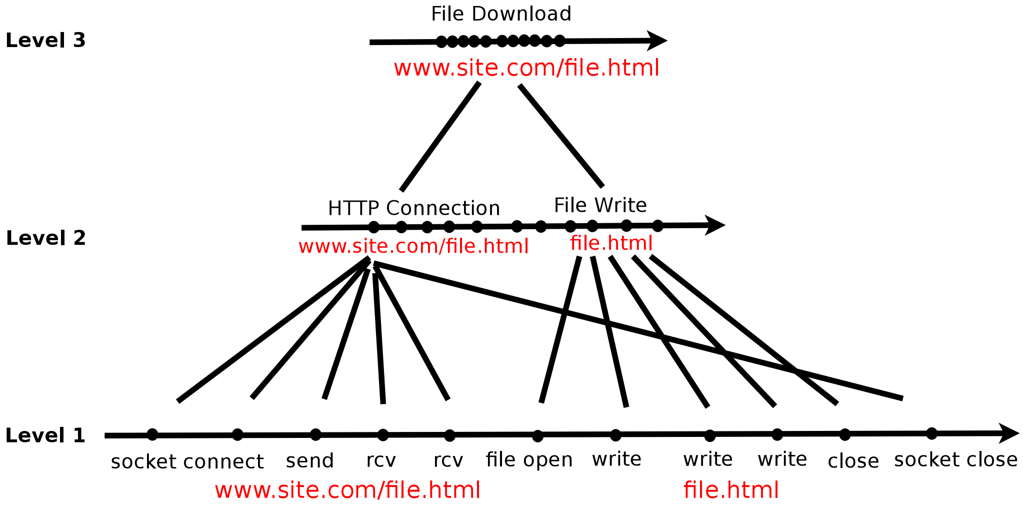


Figure 6.5 Hierarchy of events used as an input for multi-level labeling algorithm

6.5.2 Static Label Assignment Algorithm

Based on Algorithm 4, after finding the most proper scale and retrieving the labels list of the selected scale, it is the time to place appropriate labels to the trace items.

Having different $I_1, I_2, \ldots, I_n$ types of trace items, each with $C_1, C_2, \ldots, C_n$ possible positions, the number of all label positions can be gathered by the following formula.

$$C_1{}^{I_1} + C_2{}^{I_2} + \cdots + C_n{}^{I_n} = \textstyle\sum_{i=1}^{n} C_i{}^{I_i} \tag{6.3}$$

For instance, having two types, point and line items with 4 and 2 possible label positions for each, the number of all combination will be $4^{NL} + 2^{NM}$. The number of possible combinations increases exponentially when increasing the number of points.

Under specific aesthetic constraints deemed desirable, symmetric label assignment problems are NP-complete [97]. Our problem admits different criteria that make the plane asymmetric. Several heuristic methods are proposed to assign, in a reasonable time, conflict-free labels to the visual features (i.e. points, lines, areas), like simulated annealing [22], genetic algorithms [148], approximation [31], tabu search [146] and exhaustive search [89]. Our method to solve this problem is explained in the following section.

Overlap detection

The proposed labeling algorithms aim to assign conflict-free labels to trace items at the selected scale. To do so, the detection of overlaps and conflicts is an important step. For this purpose, we construct a conflict graph data structure to store the conflicts between labels. Conflict graphs are used by several researchers in the literature to model the conflicts between labels [105, 116, 147]. Using a conflict graph, the problem of overlap-free label assignment is mapped to find the maximal independent set of the constructed conflict graph [22].

In any conflict graph like G = (V , E), V is a set of possible label positions and E is a set of edges that describes the conflicting candidate positions. Different candidate positions for each item are also considered in conflict to insure that only one candidate position is retained. Figure 6.6 shows an example of labeling the three trace items (two point items and one line item). For each item, the candidate label positions are shown and the numbers (1 to 10) rank the candidate positions of the items. The corresponding conflict graph is also denoted in Figure 6.6. In the corresponding conflict graph, there are edges between labels of one item (e,g, nodes 1,2,3,4) and also between any overlapping labels (e.g., nodes 4,6).

We use an adjacency matrix to implement the conflict graph. In the adjacency matrix, each row corresponds to a label and denotes the conflict status of that label against the other labels. Having different $I_1, I_2, \ldots, I_n$ types of trace items, each with $C_1, C_2, \ldots, C_n$ possible positions, it is possible to associate a $\sum_{i=1}^{n} {C_i}^{I_i} * \sum_{i=1}^{n} {C_i}^{I_i}$ matrix to implement the corresponding conflict graph. Each $Cell_{i,j}$ in this matrix is defined as :

$$Cell_{i,j} = \begin{cases} 1 & \text{if } L_i \rightarrow L_j \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

For instance, for the example of Figure 6.6, there are 10 label positions. Therefore, its corresponding conflict graph can be implemented by an 10 * 10 adjacency matrix, as shown in Table 6.1.

Having constructed the conflict graph and adjacency matrix, and using the procedure

Table 6.1 Adjacency matrix for the conflict graph of Figure 6.6

L	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	1	0	0	0	0	0	0
3	1	1	0	1	0	0	0	0	0	0
4	1	1	1	0	0	1	0	0	0	0
5	0	0	0	0	0	1	1	1	1	0
6	0	0	0	1	1	0	1	1	0	0
7	0	0	0	0	1	1	0	1	0	0
8	0	0	0	0	1	1	1	0	0	0
9	0	0	0	0	1	0	0	0	0	1
10	0	0	0	0	0	0	0	0	1	0

ALGORITHM 5: Greedy label placement algorithm

```
Require: a set of item labels;
dynamically assign a dynamic number to each candidate label position based on the
  priority of the item and its position relative to other items;
set the output set S to empty;
for each entry n in the candidate labels; start from the most important node do
  if the entry n has no conflicts with all items of S then
    add the entry n to S;
  end if
end for
return the output set S as the early label assignment result.
```

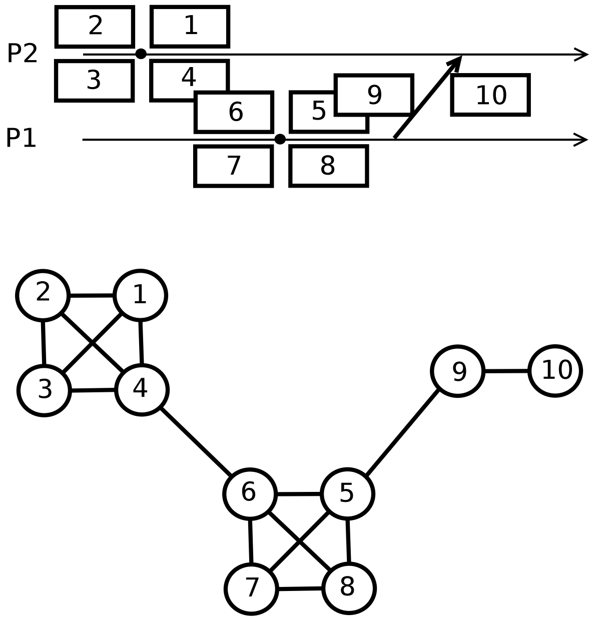


Figure 6.6 Candidate label positions and corresponding conflict graph

described in Algorithm 5, finding a maximum independent set of the labels that are conflict free is quite straightforward. The process starts by checking the labels one by one, based on their priorities. For each label, if there is no conflicting label in the output list S, it is added to the output(lines 4,5). This is the part where the conflict graph is used to specify whether the selected label overlaps the other labels or not. The labeling process finishes when it visits all labels. At exit, the output set S specifies a set of conflict-free labels that is a subset of all labels of the input set M : $S \subseteq M$. Figure 6.7 shows the algorithm running on the example of Figure 6.6.

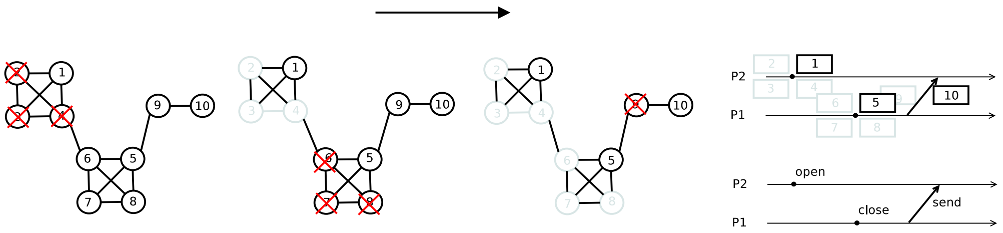


Figure 6.7 Labeling algorithm running on the example of Figure 6.6

One costly operation in the above algorithm is constructing the conflict graph. For constructing such a graph, each item should be checked against every other item, which requires $O(n^2)$ checks (n is the number of items). However, using the N-Space technique, that is explained in the following section, the required processing time can be reduced.

N-Space technique

Since we aim to place the labels along a horizontal line, it might not be required to check a label (item) against all other labels (items). Indeed, it is possible to determine an area around an item in which all possible overlapping items/labels can be found. In other words, all other items/labels that are outside of the neighbouring space (which we call N-Space) of an item, will not be in conflict with that item. For each item $k(t_k, p_k)$, which has labels of width w and height h, the N-Space can be estimated by a rectangle $[(t_k - 2w, p_k + 2h), (t_k + 2w, p_k - 2h)]$ around (Figure 6.8). A similar technique for labeling of map features is described in [105]. However, the N-Space technique presented here uses the characteristics of the proposed labeling space and is an explicit result of splitting the visible screen to different horizontal lines which does not require to define any artificial subdivision of the map space, as done in [105].

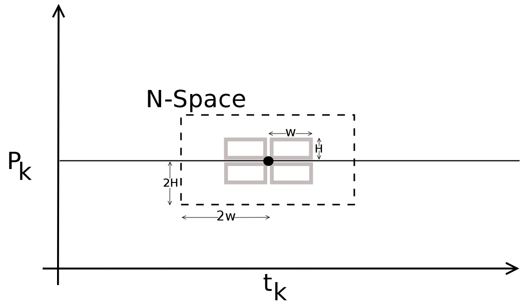


Figure 6.8 N-Space : an area around each item that might contain overlapping items

Having defined the N-Space of each item, it is enough to check the labels of one item only against the items inside its N-Space, instead of checking with all other items. In other words, by using the N-Space technique, it is not required to completely build the conflict graph, which reduces the number of checks, and thus the required processing time. Partially constructing the conflict graph, and the corresponding adjacency matrix, may be done right after fetching all items at the selected scale, as the position of each item (its x and y axis coordinates) are known in advance.

To calculate the number of checks, using this technique, let us assume for a moment that items are placed beside each other in a way that there is no conflict between their labels (one item at t_1, the next one at $t_1 + 2w$, the other one at $t_1 + 2w + 2w$ and so on). In this case, the N-Space area of each item will contain only 2 items (one before and one after). Therefore, each item must be checked against two items, instead of n. Thus, the number of checks will be $2 * n$, which is $n/2$ times faster than the previous explained approach.

In general, when there is in average m items in the N-Space area of each item, the solution will be n/m times faster than checking against all other items. The experimental comparison of these two approaches, complete and partial construction of conflict graphs, will be presented in the evaluation section.

6.5.3 Increasing the Quality

A clear and unambiguous association of labels to items is very important for the output readability. Labels should be assigned clearly to make the output understandable without needing any extra explanation. One way to make an assignment clear and less ambiguous is to reduce the number of conflicts between one label and other items. Therefore, among the different candidate labels of an item, the one with fewer conflicts with other items/labels should be selected.

The association function (AF) is defined to measure the quality of the label assignments. For each label, the association function assigns an association value by counting the items that are within its boundaries. The following formula (Formula 6.5) defines the set of conflicting items for each label (CI function). Formula 6.6 returns the count of the conflicting set members as the association value for label L_i.

$$CI(L_i) = \{I \mid x_{i1} \leq x_I \leq x_{i2} \;\; AND \;\; y_{i2} \leq y_I \leq y_{i1}\} \tag{6.5}$$

$$AF(L_i) = Count(CI(L_i)) \tag{6.6}$$

For instance, with respect to the items and labels shown in Figure 6.9, the association values of L_1 to L_4 are 2, 0, 1, 3 , respectively.

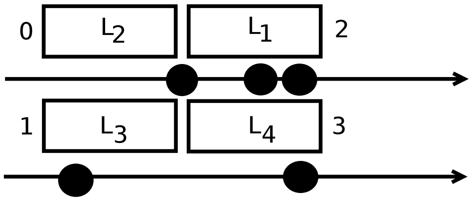


Figure 6.9 Association values of the different labels

In fact, The association value of a label specifies the ambiguity degree of that label. A smaller association value means fewer conflicts and more readability. Association values for the labels are preferred over their cartographic preferences, when the labeling algorithm aims to select a label to assign to an item. In other words, to select the labels for an item, the labeling algorithm firstly looks at the association value of each label and selects the one with the smallest value. When the association values are equal for the different labels, the algorithm uses the cartographic preferences of the labels. The association values change the preferences of the labels as : L2, L3, L4, L1. Figure 6.4 shows an example that illustrates how using the association values of the labels as a dynamic preference criteria can affect the readability of the labels.

6.5.4 Post Processing Phase

For dense cases, where there is a large number of items, Algorithm 5 may leave some items unlabeled. By considering the semantic relationships between trace items and integrating them, a post processing phase may label the unlabeled items. The post processing phase uses the following semantic relationships and integration techniques to integrate the unlabeled items :

1. Repetitive items : merging the same repetitive unlabeled items, and replacing one label instead of all, is the first integration technique. For instance, several repetitive ”send” events can be shown by one ”send” label. Later, when the user zooms-in and the screen gets larger, these items can be labeled individually, as shown in Figure 6.10.

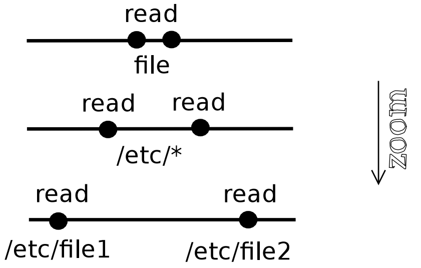


Figure 6.10 Merging the repetitive items

2. Generalization : in a kernel trace, there might be some low-level items that can be generalized to higher level items [49]. For example, there are different ways for the kernel to read a file (e.g., read, readv, pread64) or killing a process (e.g., kill, tkill, tgkill). All these items can be generalized to higher level items and merged possibly with each other, when there is not enough space to label them separately. For instance, any consecutive unlabeled ”read” and ”write” items can be generalized, and then merged and shown by a higher level ”IO” label. Figure 6.10 shows a generalization of file names for neighbouring ”read” items.
3. Static aggregation : static aggregation refers to using a pre-defined hierarchy of items (Figure 6.5) to group and merge items. This hierarchy is created in the early trace analysis steps, before the visualization phase. As an example, suppose a high-level DNS

(Domain Name Service) operation is defined as a combination of two "DNS request" and "DNS answer" operations. When there is no space to label the "DNS request" and "DNS answer" items individually, the algorithm just replaces them by the higher level "DNS" item. In other words, it combines two levels of the hierarchy to merge the labels and place a new integrated label instead of the separate labels. The static aggregation method can be used to put multiple labels on items, when there is enough space to put them. For instance, it can be used to put the labels of two levels of the hierarchy together (parent and children) to help the comprehension of the view. Figure 6.11 shows an example of this aggregation.

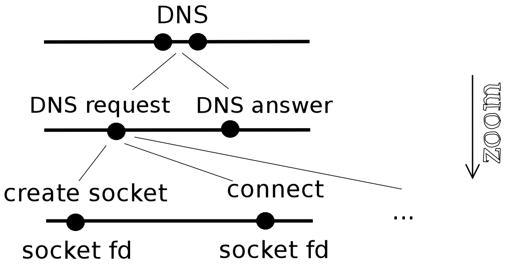


Figure 6.11 Aggregating the related items

4. Dynamic aggregation : for long duration traces, it is not feasible to store all hierarchical relationships between the different data levels. With respect to the available storage space, the algorithm is usually able to store a few, say 3 or 4 levels of data and the relationships between them. The other layers, between the predefined layers, should be built dynamically in the visualization step. To do so, we use a pattern library containing patterns of different operations at different scales. For the remaining unlabeled items, the post processing phase looks for the related patterns of the pattern library and tries to aggregate them, when the use of static hierarchical data is not applicable or could not help much. The aggregation of the resource names in Figure 6.10 is a simple example of the dynamic aggregation technique (etc/file1, etc/file2 → /etc/* → file).

6.5.5 Multiple Labels

In the literature, very little work mentioned the placement of multiple labels for an item [40]. However, in our case, each trace item may require more than one label. For instance, a trace item may require an additional label for the resource names, as well as a label representing the output value. For example, a socket read trace event might require three different labels to represent the operation name (i.e. read), socket address and the return value of the operation. We use a technique like the one presented in [40]. However, we extend the algorithm by adding a post processing phase to take into account the relationships between the items.

To do so, we call algorithm 5 for any extra label. For instance, if there exist at most N label areas for each item, the algorithm is called N times. At each step, one separate label is assigned to an item. After each step, we also perform a post processing step to see if more labels can be added by integrating and aggregating data. Figures 6.10 and 6.11 show examples of assigning more than one label to each item (i.e. the operation name and the resource name).

6.5.6 Complexity of the Algorithm

For each run of the labeling process, Algorithm 5 firstly selects the labels of the selected scale, then assigns labels and possibly performs the post-processing step. Later, the items are sorted based on their priorities and weights in $O(n\ log\ n)$, for n items. For labeling the items, the algorithm checks the items one by one based on their priority to see if there is a less ambiguous label (lines 3, 4, 5). To do so, each new item should be checked against the already labeled items in its N-Space to find the possible conflicts. This step requires again $O(n\ log\ n)$ time, instead of $O(n^2)$ thanks to the N-Space technique :

$$log\ 1 + log\ 2 + ... + log\ n \leq log\ n + ... + log\ n <= nlog\ n \tag{6.7}$$

Note that in Formula 6.7, at each step, $log\ (i)$ is the time required to extract the all items lying within the N-Space of an item (i.e. using a binary search tree over the x-axis values of the items). We assume here that the time required to compare the item with the extracted items of the N-Space, to find conflicts, is negligible.

For the post processing phase, looking at the hierarchical tree of the items could be done in a logarithmic time, again using a binary search method. It is important to note that looking for patterns in the pattern library, or searching in the hierarchy of items, may take more time than expected. However, since we expect that only a few of items will be left unlabeled, it should not be the case.

Therefore, the overall complexity of the algorithm is dominated by the time required for sorting the items, and checking against each other to find conflicts, which is $O(nlogn)$.

6.6 Experimental Results

The experiments were performed on a Core i7 2.80 GHz machine with 6GB RAM, running Linux kernel version 2.6.38.6 instrumented with the LTTng tracer. The algorithms are implemented in Java, using the Eclipse plug-in for Java and will be contributed to the open source TMF (Tracing and Monitoring Framework) software. Data abstraction techniques [49] are also used to analyze the trace events and generate the items hierarchy.

The proposed algorithms are evaluated and discussed in the following four parts.

6.6.1 Labeling Output

Figure 6.12 shows an output of the label assignment algorithm. The screen contains four separated lines, for displaying the trace items of four different processes. Each line denotes 50 random items (200 in total). The labeling success rate of this example is 88.5%, which means that 177 items (among the 200 items) are assigned conflict-free labels. The impact of changing this distance is shown in Figures 6.14 and 6.15, and discussed in the next experiment.

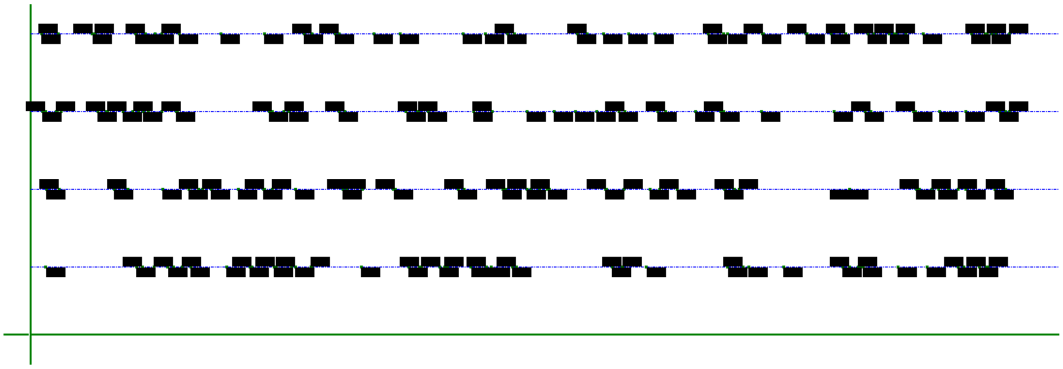


Figure 6.12 Output of the proposed algorithm for 200 items. 88.5% were labeled without conflict.

A similar experiment was performed considering the association between labels and items, which is shown in Figure 6.13. In this experiment, the dynamic label positioning method is used for labeling. The labeling success rate is almost the same as for the previous experiment. However, the association rate is increased (e.g., the labeling quality is increased from 34% to 52% for the first process).

6.6.2 Labeling Success Rate

For calculating the labeling success rate, we ran the algorithm 100 times for each given set of items and the average value is reported. The input data set is generated by the LTTng kernel tracer and abstracted out to three hierarchical levels. The first abstraction level contains sequences of system calls of file and network operations (file open, file read, file read, file seek, file close, socket create, socket connect, send, receive and socket close , etc.). The other levels contain different combinations of the items of the first level (items like sequential file read, HTTP connection, etc.). The experiment is undertaken for two cases, once for a normal

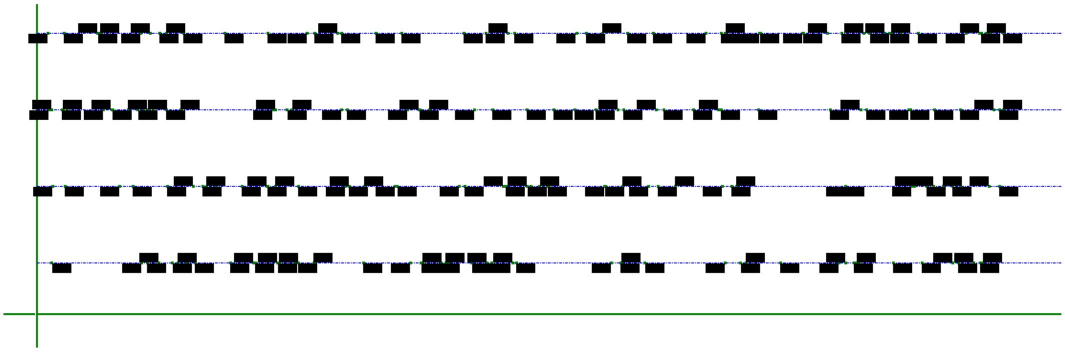


Figure 6.13 Output of the proposed algorithm for the same input as previous example, considering the association between labels and items.

screen and once for a tight screen, when there is no consideration of the distance between two neighboring lines (and conflicts can be horizontal or vertical). Figures 6.14 and 6.15 represent the results for the normal and tight screens, respectively.

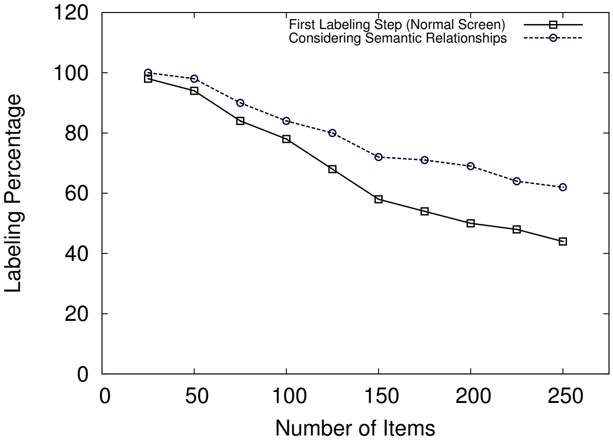


Figure 6.14 Percentages of conflict-free labels for different set of items in a normal screen

For each case, the graphs show the percentage of the labels that are successfully assigned by the algorithms : for the first step of the algorithm where the greedy algorithm tries to raise the number of labels assigned, and for the second step where the semantic relationships are taken into account.

Unsurprisingly, the success rate decreases when the distance between the process lanes is getting small. In other words, the success rate of the normal screen 6.14 is higher than for the tight screen 6.15. One reason is that, in the tight screen, items can conflict with the above and below items (vertical orientation), in addition to the next and previous items (horizontal orientation).

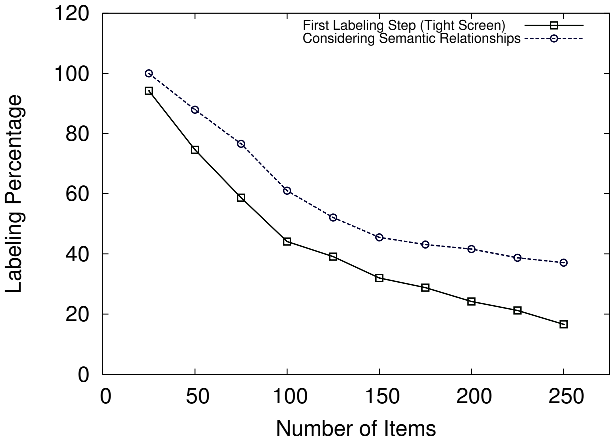


Figure 6.15 Percentages of conflict-free labels assigned in a tight screen

The figures also show that considering the semantic relationships and performing functions like generalization and aggregation and removing repetitive items can increase the success rate, and label some of the items that were left unlabeled in the first step of the algorithm.

6.6.3 Labeling Quality

As mentioned earlier, Formula 6.6 specifies the clearness and unambiguity of an assigned label. Sometimes, it might be important for users to know the clearness and unambiguity degree of a set of assigned labels, instead of a single label. This may help users to compare the clearness of two or more assignment sets. To define such a measure, suppose that there are n items, I_1 to I_N, and the labeling algorithm assigns labels to a subset (a subset of L_1, $L2$, ..., L_N). Then, the unambiguity degree (UD) for this set of items can be calculated using the following formula :

$$UD(I_1, \ldots, I_N) = \frac{Count(\{I|AF(L_I) = 0)\})}{\underbrace{N}_{\text{Count of all items}}} \tag{6.8}$$

Formula 6.8 defines the labeling quality, or the proportion of items that have been clearly assigned, (the number of items that have no ambiguity with other items on the total number of items N). Using this formula, one can easily calculate the clearness and unambiguity degree of a set of assignments and compare it against the other possible assignments.

We have done some experiments to calculate the clearness of the assignments that are performed by the proposed algorithms. Figures 6.16 and 6.17 represent the labeling quality degrees for the normal and tight screens, using both the proposed static and dynamic preferences respectively.

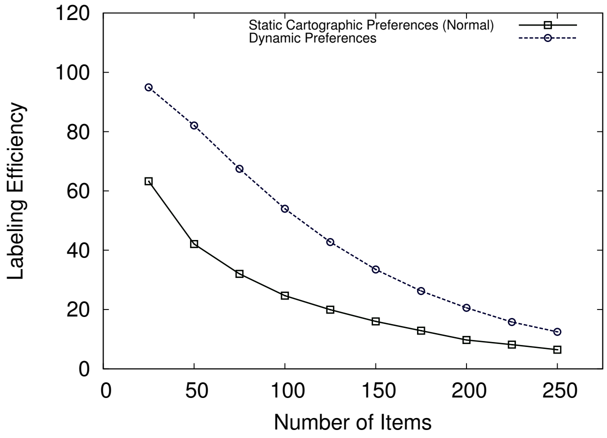


Figure 6.16 Percentages of conflict-free labels for different set of items

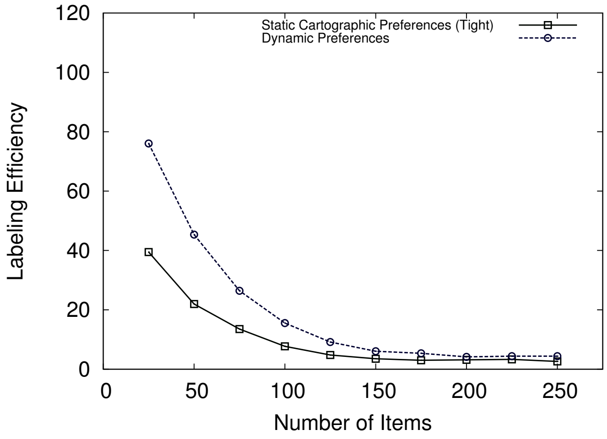


Figure 6.17 Percentages of conflict-free labels for different set of items

Figure 6.16 compares the labeling quality (thanks to the UD metrics) of the algorithm that uses static and predefined cartographic preferences with the algorithm that uses the dynamical label preferences. Figure 6.17 shows the same comparison for the tight screen. In both cases, the dynamic preference method results in higher unambiguity degrees, thus higher quality for the different input sets.

6.6.4 Execution Time

The last experiment is for comparing the execution time for the algorithms that construct partially or completely the conflict graph. Figure 6.18 shows the execution time of the algorithms in three cases. The first case is for the complete conflict graph construction with static cartographic preference (the case that does not consider the clearness of the assignments).

The second case is the same as the previous one, but with the dynamic preferences (it takes into account the clearness of the assignments). The third case is the partial construction of the conflict graph (considering the clearness of the assignments). The result shows that the time required for the case with partial construction grows very slowly with the number of items, and is much better than the other two cases. This actually proves the higher efficiency of the proposed algorithm in comparison with the same cases that use a complete construction of the conflict graph.

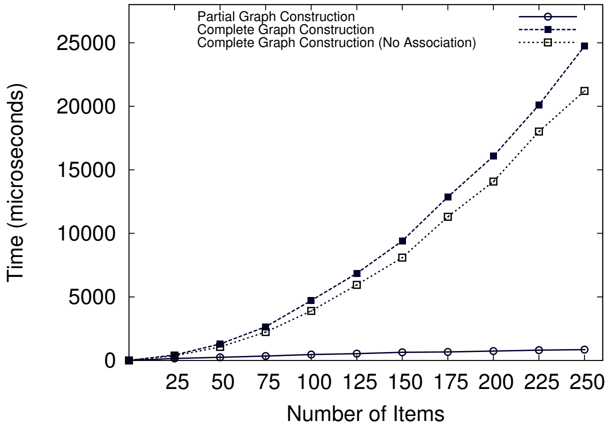


Figure 6.18 Execution time comparison for different algorithms

6.7 Conclusion and Future Work

In this paper, efficient algorithms for label placement of trace items are proposed. The algorithms assign labels to trace items that are separated in different lines based on their process name. We use the so-called conflict graph to find the conflicts between items. However, since constructing the complete conflict graph is a time consuming task, it was avoided by using a heuristic technique. In our system, we build the conflict graph only between neighbouring items (inside the N-Space of each item), instead of between all items, which reduces the number of checks from n^2 to $n * log(n)$ operations.

We have used a quick greedy algorithm to label the trace items. The speed of algorithm is an important factor for us. Because this technique will be used in an interactive multi-level visualization tool with lots of possible interactions, zoom and panning operations. Although the algorithm is fast, it may not label all items. For increasing the number of labeled items, we use a post processing step taking into account the semantic relationships between the data. Generalization, static and dynamic aggregations and removal of repetitive items are the techniques used in this research. The experimental data proves that the post processing step increases the number of labeled items.

We have also proposed an algorithm for placing multiple labels to each item. Extending that algorithm by increasing its efficiency is a possible future work. Another promising avenue is extending the algorithms to support label assignment to the line and area items directly, without considering them as point objects.

CHAPTER 7

Paper 6 : Multilevel Visualization of Large Execution Traces

Naser Ezzati-Jivan and Michel Dagenais

7.1 Abstract

Tracing generates valuable data about the underlying system execution. However, to extract useful information from this raw data, it should be presented in a meaningful way. Trace visualization, and especially multi-level (multi-scale) visualization, is a solution for this issue in which the large trace data is organized and visualized in a hierarchical manner. This paper presents a tool and a set of techniques to interactively visualize large trace data at multiple levels, allowing users to apply top-down exploration mechanisms and navigate hierarchically up and down through the entire trace data. This research investigates hierarchical management of large trace logs and proposes an interactive zoomable timeline view to visualize the multiple levels of trace information. This zoomable timeline view displays the multiple levels of trace data in a single graphical interface, in which the coarser layer is shown first and different exploration operators enable exploring and navigating through the different layers. The view supports both the semantic (content-driven) zooming and standard (structural or visual) zooming. This approach mainly facilitates the comprehension of the execution trace logs and can also be used to improve root cause analysis. Indeed, it provides an ”event location” feature, specifying where to look within the trace events for any selected high-level behavior (e.g., an alert, a system problem, a network attack, etc.). The paper also discusses the supporting hierarchical data model, implementation details and several experimental results. The proposed method is generic enough to be applicable to many areas and any trace data. However, the experimental evaluation of the method is based on timestamped events gathered from instrumenting different Linux kernel modules.

7.2 Introduction

Execution trace logs are used to analyze the system runtime behavior and detect its problems and misbehavior. However, the size of the trace logs is sometimes a serious challenge. Indeed, huge trace logs usually complicate analyzing and understanding the behavior of the system under study.

One solution to solve the size problem is to avoid displaying a huge amount of data at once and overwhelming the users, regardless of the size of the original data. This goal is achievable by organizing the data in a hierarchical fashion and enabling multi-level display and exploration [136]. In this approach, an overview of the trace data (e.g., overall statistics or aggregation of the underlying system execution) is presented first, and more details (low-level execution events e.g., process creation, file IO accesses, disk block accesses, etc) for any area of interest are prepared and displayed on demand.

This multi-level view uses the fact that, in many applications or many parts of system execution, displaying a general overview without digging much into the detailed data can be enough to understand those parts. For instance, to inspect a web server performance using trace data, it may not be necessary to dig too much into the initialisation section of the server, where it opens configuration files and checks some variables. Showing an overview (e.g., "reading configuration files") for this part of the execution, that might be an aggregation of tens of thousands trace events, can be representative enough for most users and can guide them to understand and glance over these areas quickly. For other parts of the execution, however, users may need to focus and demand more detailed information, which this multi-level view supports. This way can significantly speedup reading the trace data, understanding the execution behavior, and possibly detecting problems and finding their causes.

Different trace visualization techniques and tools are proposed to manage and display trace data in a hierarchical manner : Node+Link structures [119], Multiple Views [34], Treemaps [110] and so on. However, these existing trace visualization tools (and others like TMF [1] or Chrome tracing [2]) usually visualize data at a single semantic level, supporting only standard data zooming : when the screen gets larger, the objects are shown larger and possibly with some labels. They usually lack support for semantic zooming, where a larger display area will be used to not only show bigger objects and bigger labels, but also more objects, more labels and possibly other levels of data.

A proper multi-level trace visualization tool, displaying the system behavior at various levels of detail, can help users to get rid of the low-level trace data and allow them to perform a top-down exploration of trace data. They may first reason at higher abstraction levels and then zoom into any area of interest, to get a closer look and find the individual trace events. The ability to explore the execution trace, and follow the control flow of the system at more abstract and meaningful layers, can help to quickly understand the system execution and its (normal or odd) behavior.

The challenges of providing a multi-level visualization tool, with support for different

[1] http ://lttng.org

[2] http ://dev.chromium.org/developers/how-tos/trace-event-profiling-tool

navigation operations (standard and semantic zooming, drilling down, rolling up, etc.), are threefold. First, the different layers of representation must be generated from the given data set. Secondly, a data model must be developed to organize efficiently the hierarchical data and link the corresponding data between the different layers. Thirdly, proper visualization must be offered for the data hierarchy.

The different layers of the data can be generated using different trace abstraction techniques [48, 49, 99, 142] or gathered from different sources (e.g., kernel tracing, and user space tracing, application's syslogs, etc.), that is out of this paper's scope. Our focus in this paper is mainly on the second and third challenges : organizing the trace data hierarchically and visualizing them at multiple levels of detail. We aim to develop tools and techniques to support multi-level visualization of hierarchical trace data that enables a multi-scale navigation by providing standard and semantic (data) zooming, drill down and other navigational operations. The prototyped techniques and tool are generic enough to be used for any trace data. However, we use the trace data collected by the LTTng kernel tracer [37] to evaluate our prototype implementation.

The first contribution of this research is a data model to hierarchically organize the abstract trace data. Different data structures are proposed in this paper, and the benefits and limitations of each are discussed and compared. The proposed data model discusses the approaches to manage and organize the different levels of data in disk-based data structures. The model contains also a method to link the related data together. It is shown that the proposed data model is efficient in comparison with other models in terms of storage space and performance.

The second contribution is modeling the links between different elements in the trace data. The hierarchical organization of the trace data already describes a hierarchical relation. However, it is sometimes required to model and display other links and relations between data elements. A good example is a link between a file operation (e.g., read) and the corresponding disk block operations, which is modeled and supported in this research. The linking between high-level behaviors and individual trace events provides an ”event location” feature. This can be used in root cause analysis, to specify where to look exactly within the trace events for any selected high-level behavior (an alert, a system problem, a network attack, etc).

The final major contribution of this research is a zoomable timeline view to visualize the trace data at multiple levels of granularity. The view integrates different abstraction levels in a single display and provides useful navigation mechanisms (i.e., drill-down and standard and semantic zooming) to explore the trace data at various levels. It also uses label placement algorithms [55] to display readable and clear labels (representative of trace events) on the screen.

The remainder of this paper is organized as follows : we first present a survey of related work. This is followed by a problem statement, and a description of the data abstraction techniques used to generate different trace levels and the data model used to manage these multiple data levels. We conclude with a discussion about the implementation and performance results, examples of visualization output, other possible use cases, and a summary and final discussion.

7.3 Related Work

The relevant related work may be divided into two main categories : trace visualization techniques and spatial data structures and organizations. Each will be discussed in a separate subsection.

7.3.1 Trace Visualization Tools

Multi-level (Multi-scale) visualization has received much attention in the cartography domain. The challenges are on-the-fly aggregation, hierarchical organization, correspondence linking, multi-level geometric matching and multiresolution visualization [17, 73, 106, 136]. Although the generic ideas mentioned in these approaches are somehow applicable to trace analysis context, the research in this area mostly studies the various geographical aggregations and matching techniques which are not directly relevant to our research.

There are many trace analysis tools to visualize trace logs and provide different analytical views such as Paje [32], Triva[126], Jumpshot[150], LTTV, TMF (Tracing and Monitoring Framework)[3]. These tools often use a set of coordinated views (e.g., a main view and a set of supporting views) [143] to show the different aspects of the system runtime behavior. In these tools, a primary view is used to show the execution mainline, and a set of auxiliary views is used to display other aspects. Selecting an item in one view highlights a range of events in other views [109].

Multiple views techniques can be used to display hierarchical data as well. To do so, each view is specialized to reveal a selected hierarchy level. Although this technique helps users to get a better comprehension through various views, it is not well suited to display hierarchical data and their relationsips. Indeed, to understand a single behavior, this technique forces users to follow various separated views, which may be overwhelming when there are a large number of views, or when the views are not coordinated.

There are also some other specific techniques to visualize hierarchical data. Treemaps, a space filling technique in tool Triva [126], the Focus+Context technique in tool Ovation [114]

[3] http ://lttng.org

and the Multiple Levels view in Vampir [107] are also used to manage and visualize data at multiple levels.

These techniques also present some problems to visualize multi-level trace data. Treemaps, for example, focus more on the leaf nodes and have some problems to efficiently display large data sets and the different relations between data. Focus+context techniques are typically used to display data sets that have a clear categorized organization, but may have some problems with large unstructured trace data.

The Multiple levels technique is often implemented in some visualization tools in such a way that ”semantic zooming” [136] and content-based drilling down and rolling up are not supported, even though they are important features in hierarchical data navigation. These tools lack an elegant way to model and display multiple levels of hierarchical data, together in a single view with support for various content based zooming, focusing and navigational operations.

7.3.2 Data Structures

Spatial analysis tools (e.g, online maps) usually store spatial objects at different scales and resolutions in a multi-resolution /representation-database (MRDB) [79]. Using this kind of database, it is possible to store and then manage an object at different levels of precision and resolution. In such a database, an object representation in lower levels contains more details, but only a basic generalized representation is used at higher level [138]. There are three approaches for creating such databases [39] :

- Automatic creation of the database from a given set of objects. Similar to the trace abstraction techniques, this technique uses auto spatial aggregation and generalization methods to derive high-level objects from the input objects.
- Instead of automatic aggregation, which has limited capabilities, the second method stores different scales for each object. The different scales of an object are built manually or gathered from different sources. At display time, the method selects and shows the object at a scale related to the current visualization scale.
- In the third approach, similar to the second method, a multiresolution database is created for different sizes of the object, and the links between different object representations are stored as well. In this approach, each object knows its relevant objects (or relevant representations) in the other levels.

To relate the corresponding data, or the different representations, of an object together, these tools typically use the two following solutions :

- Geometric matching : in this technique, semantic attributes, metrics, geometrical and structural information of objects are used to match similar objects at different scales

[80]. This technique is used for linking road network intersections at different levels by Xiaomeng et al.[144].

- Scale-transition relationships : this method is used to directly link the different representations of objects (or object types) together. This kind of relationships enables various multi-scale operations like zooming, focusing and up and down navigation between different object scales. The details of the links can be gathered statically (in the process of aggregation) or dynamically using the matching techniques. This technique is used to link different-scale road networks by Devogele et al. [39].

Although it is possible to get ideas from the spatial analysis tools, the research in this area mostly discusses about the aggregations and matching techniques (and the precision of the matching), and rarely talk about the underlying data management structures. They often use existing database tools and technologies for storing the data and links. However, being dependent on a database tool will limit the capabilities of the analysis tool, especially in our application with very large trace data.

Various access methods and spatial index trees may be found in [61], which reviews the B-tree, Hb-tree, R-tree (and its variants : R+-tree and R*-tree), Quad-tree, and other data structures used for managing spatial and hierarchical information. Most of these techniques, while not optimal [71], and two trace data specific techniques, SHT [102] and SLOG [18], can be used to index data at different levels separately. However, this is different from managing several levels of events linked by different attributes. Some of those techniques can be used to manage the hierarchy, but they only support one form of hierarchy (e.g., time based hierarchy or aggregation hierarchy). Nonetheless, in kernel traces we simultaneously have hierarchies of different attributes (time based hierarchy, resource based hierarchy, data based hierarchy, etc), which should be supported in a new hierarchical data model.

7.4 Multi-level Exploration

7.4.1 What to be shown

Execution traces (especially kernel traces) contain valuable information about the underlying system execution. However, they are too low-level and voluminous. Therefore, a simple visualization of the original trace may only lead to a cluttered display, containing lots of low-level events and communications arrows. An example is shown in Figure 7.1 which shows a graphical view of kernel trace data. Only after zooming and limiting the display to only a small time duration, users can get an informative view of that selected portion.

Trace abstraction techniques are used to reduce trace size and complexity, and generate high-level meaningful events. The expressiveness of the abstract events can help to achieve

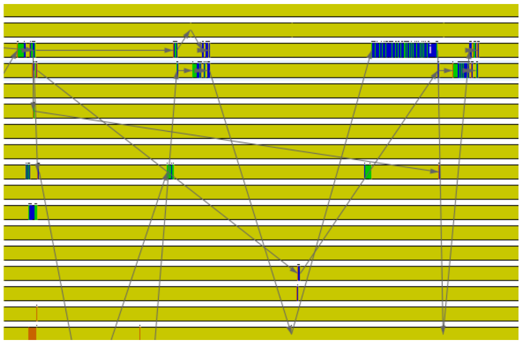


Figure 7.1 Single level visualization of kernel traces (TMF viewer).

a better visualization and comprehension of the underlying system execution. For example, a user could see some high-level events (e.g., displayed as labels, annotation, etc.) in the view shown in Figure 7.1 in addition to colored rectangles and arrows. In this case, it will be more informative about the high-level behavior of process execution, at different time areas, increasing the understanding and comprehension of the underlying trace data. Using this method, users may need to zoom only in some areas of interests, instead of all areas, to get more details and insights.

In this research, we propose a zoom-able time-line view that visualizes kernel trace data at multiple levels. In this tool, the default view displays a set of coarser events to render an overview of the execution ; each may represent hundreds or thousands of events. The finer levels, on the other hand, are also available on demand to provide more details. When users zoom in and focus on a selected area, these finer levels are fetched and displayed. The lowest level displays the original trace events, and can be used to reason about execution at the lowest data level.

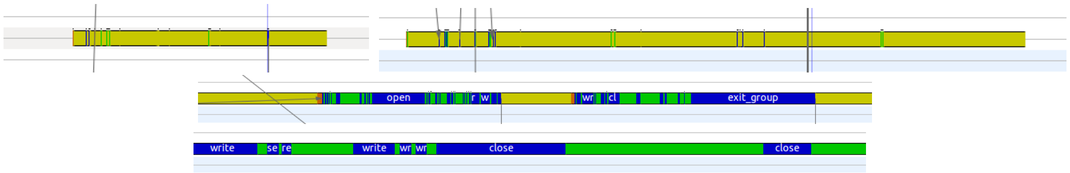


Figure 7.2 Standard zooming of the WGET process (different levels of background layer).

We divide the view into the background and the foreground layers. They are discussed separately, since their organization and visualization process are different. The background layer displays the kernel level trace states and events (system calls, interrupts, block waits, etc.), and the communications between them as rectangles and arrows. This is used as the background illustration of the view (as shown in Figure 7.1). However, the foreground layer

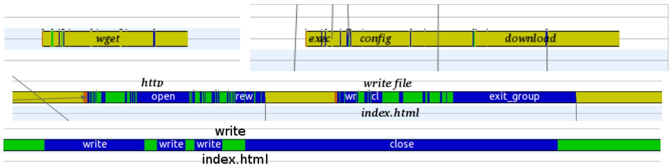


Figure 7.3 Semantic zooming of the WGET process (different levels of foreground layer).

displays a set of high-level events, integrated with the background view to facilitate trace reading and understanding. Please note that there is no actual separation between the foreground and background information from the user's point of view ; the tool overlays both layers and produces a single output view.

The zooming operation is supported in both layers, standard zooming in the former and semantic zooming in the latter. Zooming, in general, defines the screen space assigned to each object. Zooming can be used to get deeper into the data, and assign the visible space to a smaller/bigger set of data. Standard zooming refers to only enlarging/reducing the size of an object in the screen and assigning more/less space to show its single data level. By contrast, semantic zooming [136] refers to not only enlarging/reducing the size and shape of the objects, but also retrieving more or less detailed data. Semantic zooming is usually supported by online map tools (e.g., Google Maps) such that when users zoom into an area of interest, they get the screen enlarged as well as more detailed data about the selected area (e.g., show the city names, or the street names within a city, etc.).

In the proposed tool, zooming may either increase the scale, increase the size, show more detailed data, show more labels, split an event into smaller events or show more events (appearance of new events). For instance, when users zoom in and focus on a selected area, the tool enlarges the visible objects to an extent, possibly shows more labels and, eventually after an extent, starts to retrieve more detailed data (i.e., more events, more labels, etc.). The opposite operation occurs when users zoom out of the view. In this case, the tool decreases the size of the visible objects and, after a certain threshold, tries to aggregate the objects or retrieve data from a less detailed level.

Figure 7.2 gives an example of four layers of the background view. As explained, this view only supports standard zooming, increasing the size of objects when there is more space, and decreasing the size when there is less space. Some visual abstraction techniques are also used to visualize the background layer. Figure 7.3 displays the same view by integrating the foreground information. You can see that some labels (abstract event names, resource names, etc) are added to the view, increasing its readability and usability. This view supports semantic zooming and shows different sets of foreground information based on the display level granularity.

In this configuration, when users open the tool, a high-level view is shown, depicting an overall execution of the processes. This view contains some rectangles, colors and arrows (depicting the different execution states of each process, starting and ending points such as process creation, process execution, process exit, and also message or control transitions between processes) in addition to expressive labels and annotations (describing the high-level behavior of the system execution, the resource names, etc.). A typical view of the default high-level display, containing three different processes is shown in Figure 7.4.

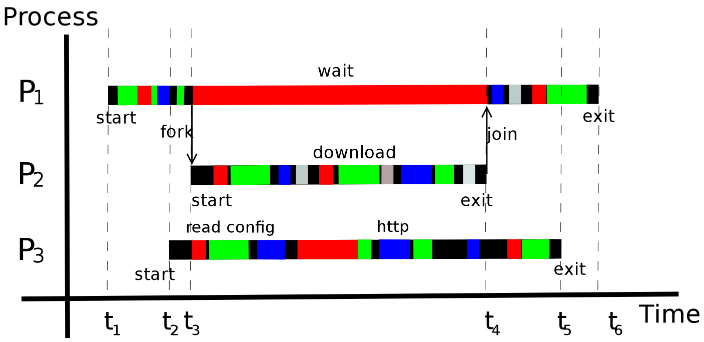


Figure 7.4 Highest level view of the multi-scale visualization tool.

7.4.2 Multi-level Data Generation

The first question is the ways required to generate the various levels of events. In the literature, this is done either by manually constructing the levels [141], reading each layer from different sources [39] or automatic data aggregations [49].

Various automatic techniques are proposed in the literature to convert the low-level data to high-level, to reduce its size, complexity and to generate various abstract events. Techniques like aggregation, generalization, filtering and reduction, exaggeration and displacement are used to visually abstract the data [73, 136]. Techniques such as pattern matching, frequent pattern mining, clustering and classification are used to abstract the data based on their context [49, 142]. Using the above trace abstraction and aggregation techniques, it would be possible to generate various levels of abstract events. In the following, we explain the data levels that are generated and used in our proposed tool (Figure 7.5).

The lowest level contains raw events generated directly by the LTTng kernel tracer [37]. A view of this level, containing LTTng kernel trace events, is shown in Figure 7.6.

An upper layer contains the system call events (e.g., file open, read, write, socket connect, etc.), and system states extracted using mapping tables or pattern matching techniques. To get more details about trace abstraction and state value extraction, please refer to [49, 103].

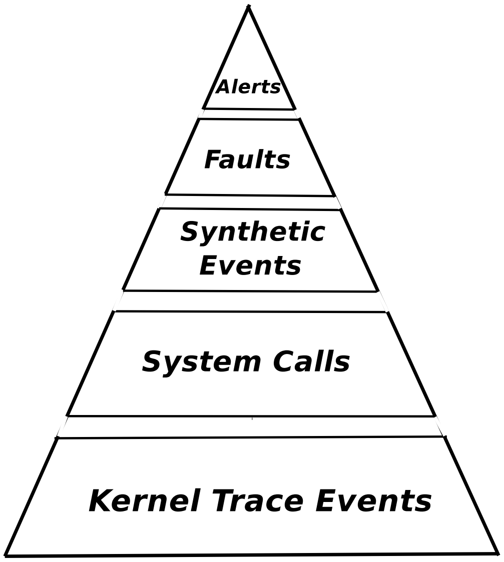
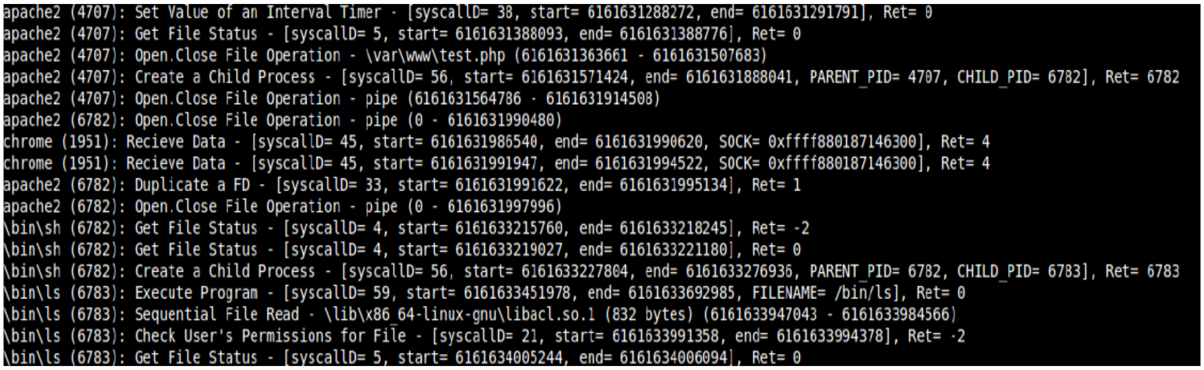


Figure 7.5 Different levels of data generated from the kernel trace events.



```
apache2 (4707): Set Value of an Interval Timer - [syscallD= 38, start= 6161631288272, end= 6161631291791], Ret= 0
apache2 (4707): Get File Status - [syscallD= 5, start= 6161631388093, end= 6161631388776], Ret= 0
apache2 (4707): Open.Close File Operation - \var\www\test.php (6161631363661 - 6161631507683)
apache2 (4707): Create a Child Process - [syscallD= 56, start= 6161631571424, end= 6161631888041, PARENT_PID= 4707, CHILD_PID= 6782], Ret= 6782
apache2 (4707): Open.Close File Operation - pipe (6161631564786 - 6161631914508)
apache2 (6782): Open.Close File Operation - pipe (0 - 6161631990480)
chrome (1951): Recieve Data - [syscallD= 45, start= 6161631986540, end= 6161631990620, SOCK= 0xffff880187146300], Ret= 4
chrome (1951): Recieve Data - [syscallD= 45, start= 6161631991947, end= 6161631994522, SOCK= 0xffff880187146300], Ret= 4
apache2 (6782): Duplicate a FD - [syscallD= 33, start= 6161631991622, end= 6161631995134], Ret= 1
apache2 (6782): Open.Close File Operation - pipe (0 - 6161631997996)
\bin\sh (6782): Get File Status - [syscallD= 4, start= 6161633215760, end= 6161633218245], Ret= -2
\bin\sh (6782): Get File Status - [syscallD= 4, start= 6161633219027, end= 6161633221180], Ret= 0
\bin\sh (6782): Create a Child Process - [syscallD= 56, start= 6161633227804, end= 6161633276936, PARENT_PID= 6782, CHILD_PID= 6783], Ret= 6783
\bin\ls (6783): Execute Program - [syscallD= 59, start= 6161633451978, end= 6161633692985, FILENAME= /bin/ls], Ret= 0
\bin\ls (6783): Sequential File Read - \lib\x86_64-linux-gnu\libacl.so.1 (832 bytes) (6161633947043 - 6161633984566)
\bin\ls (6783): Check User's Permissions for File - [syscallD= 21, start= 6161633991358, end= 6161633994378], Ret= -2
\bin\ls (6783): Get File Status - [syscallD= 5, start= 6161634005244, end= 6161634006094], Ret= 0
```

Figure 7.6 The lowest level events generated directly by the tracer module.

Synthetic events (e.g., port scan, a HTTP connection, file download, sensational file access, etc.) are generated by applying another set of patterns, typically complex patterns [49, 50], over trace events and system parameters. Synthetic events could also be a list of faults, attacks, attack attempts, alerts, etc. Examples of these events are : brute force attack, access to a closed file, fork bomb attack, denial of service attack and so on.

The above information forms the different layers of the background and foreground views. Foreground information is typically displayed as labels and annotations, while the background layer uses rectangles, colors, dots and arrows to depict the events. In the background layer, the events are displayed by rectangles while colors indicate the event type (system call, interrupt, block wait, etc.). In addition, arrows are used to display the message passing and transitions between processes. For example, when a process forks another process, or a process sends a message to another process, a corresponding arrow is displayed.

The background view may use some more visual abstraction techniques in its different layers. For example, when users zoom out and the screen space gets smaller to show a set of labels and events (rectangles), the labels are shortened or removed and one aggregated rectangle, possibly overlined as a visual cue to indicate that some events could not be shown. For example, a process could be mostly running but a few system calls happened and only one state, running, can be shown in the single pixel available to display this time interval. A dot (overline) is thus shown at the higher level in both Figures 7.3,7.2).

7.5 Data Model

In this section, we discuss the backend data model used to organize, manage and visualize the generated multi-level abstract events, as well as the hierarchical links between the data at different layers. We consider three possible approaches to model the data, provide comparisons, and discuss applications for each.

The first solution is to store only the lowest level (i.e., raw trace data) and generate the other layers on-the-fly, when they are needed. The second approach is to store all the data for the different layers in specific data stores and build a multiresolution database. The third approach is an hybrid solution in which the lowest level data, and some annotations about the high-level information, are stored. Figure 7.7 shows a view of these three solutions.

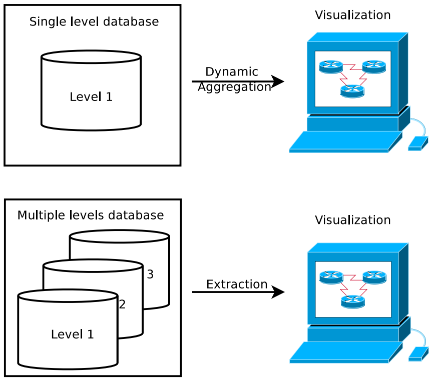


Figure 7.7 Different approaches to support multi-level visualization.

As shown in Figure 7.7, upper row, the first solution only stores the detailed level of the data (i.e., the original trace data). All other levels are then generated in the visualization phase using the stored detailed level. For generating the higher levels, the same abstraction approaches are used, as described earlier. The benefits of this solution is that it is storage efficient, avoiding to store regeneratable data. However, the time required for on-the-fly aggregation can be relatively high, making the solution costly. Indeed, to visualize an overview of a relatively large trace, a significant amount of processing time might be required to abstract out the logs and regenerate the requested overview. This solution, however, can be used to visualize live trace streams, when there is no extra time to pre-process and pre-generate different aggregation levels of trace events. In such a case, all data processing and visualization should be carried out live, simultaneously with the trace reading phase.

Storing all Events in a Data Store

The second solution is based on generating all abstract events in advance, in the trace reading phase, storing them in a data store and using them later in the visualization phase. This solution, as shown in Figure 7.7, lower row, creates several databases to store the different levels (usually a separate database for each level) and the possible links between levels. In this solution, there is no need to aggregate the data on-the-fly and, therefore, the visualization and navigation through the different levels are performed efficiently. The downside is more storage space cost, to store every single event in the data store. Figure 7.8 shows a schematic view of the different event layers. Some links between events are not shown to avoid cluttering the screen. Events and elements of all layers, and the links between the different layers, are stored separately in the data structures.

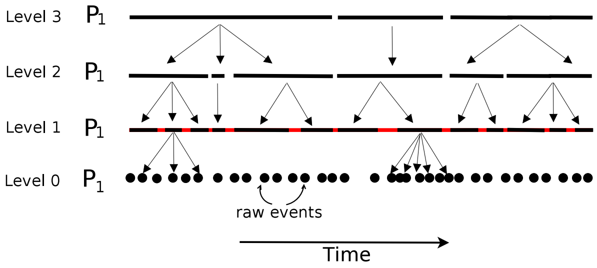


Figure 7.8 Different levels of events and hierarchical relations between them.

As shown in Figure 7.8, in this approach multiple levels of abstract events, and the links between them, are stored to be used later in the visualization phase. Each abstract event is an interval containing a start time, end time, a name (e.g., ”file open”), some optional arguments (e.g., url : www.x.com, destination port : 80, filename : /etc/passwd), the process owner for this event (e.g., Apache process), the level in the hierarchy, the pattern name, etc.

To store these intervals (abstract events), a customized interval container, supporting live streaming data is required. The State History Tree (SHT) [104] was proposed in the literature to manage the system state intervals [103]. Unfortunately, it manages only a single state level and does not directly support multiple levels of data, with possible links between them. In this research, we extended the SHT interval container to support storing abstract events in different levels of granularity, including a link data structure called S-Link. The details of the data structures will be discussed shortly. Also, experimental results and a comparison with other interval management systems (e.g., R-Tree) will be presented in the experimental results section.

The extended data structure used in our tool contains two main parts : Event Container and a structure called S-Link, as shown in Figure 7.9.

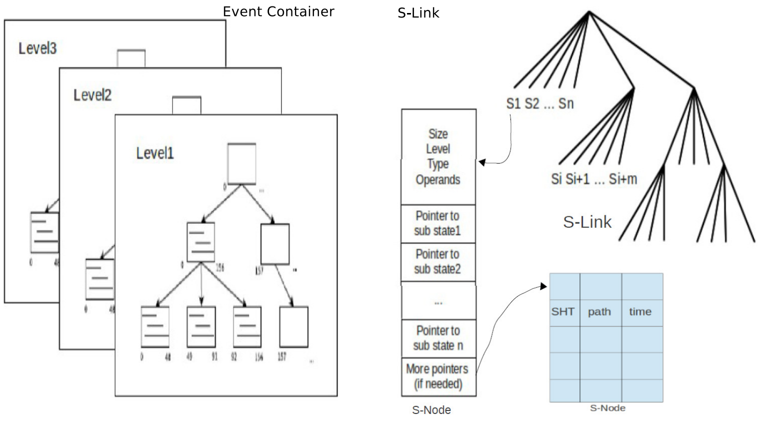


Figure 7.9 The data structure used to store the abstract events and the links between them.

The ”Event Container” part encompasses a set of separate interval trees to store the abstract events. Each tree, in turn, contains a set of nodes to store the events for only one level (e.g., system call events). Each node has a time bounding box (start and end time) which stores all events lying within this time interval. Organizing the events inside these time-dependent nodes enables users to easily query the system to find the events lying within the analysis time window (range queries) and to go back and forth in the trace (panning operation). The size and duration of the nodes in the different trees (trees for different layers) are not related to each other and have separate size and time ranges in their own trees.

Using the different event containers, one separate container for each level, enables the visualization algorithm to show the events of each layer separately. It also enables forward and backward movement within the events of each layer, as well as zooming to the upper and lower levels. When a layer is changed and a new layer should be displayed, the events in the current time window (or possibly a zoomed portion of it) are extracted and displayed. The detection of the correspondence between the layers (the events of the layers) are delegated to the users. This is similar to what typically happens in an online map system (e.g., Google Maps) : Users view sets of events belonging to the different adjacent layers and make conjunctive relations between them. In other words, no linking information is stored for the events.

However, it is sometimes required to make an explicit link between the events (which is usually not the case for online maps). For example, consider a case where users see an interesting behavior in an upper level and want to focus and dig into that specific behavior and fetch the individual events for that ; the user does not want to be lost among the possibly thousands of unrelated lower level events. Then, another strucure is required to model the links data. The second part of the data structure, S-Link, as shown in Figure 7.9, is used to support this important feature, linking the events together. S-Link is a tree of constant size S-Nodes. Each S-Node contains a header, the level of the event and a few entries (say k) in which the address of a related event is stored. The last entry is a pointer to another S-Node, if needed, to support a large number (more than k) of contained events.

An example is shown in Figure 7.10 for a DNS (Domain Name Service) connection event. This is an event generated using trace abstraction techniques and stored in the interval container. The links between this event and the contained events from the lower level layer is recorded in the S-Link structure. As shown in Figure 7.10, each entry of the S-Node record points to the address of one lower level event. Please note that there may be other events in between the DNS-connection events, but the S-Link structure allows to filter them out and show only the events related to the selected high-level DNS connection event.

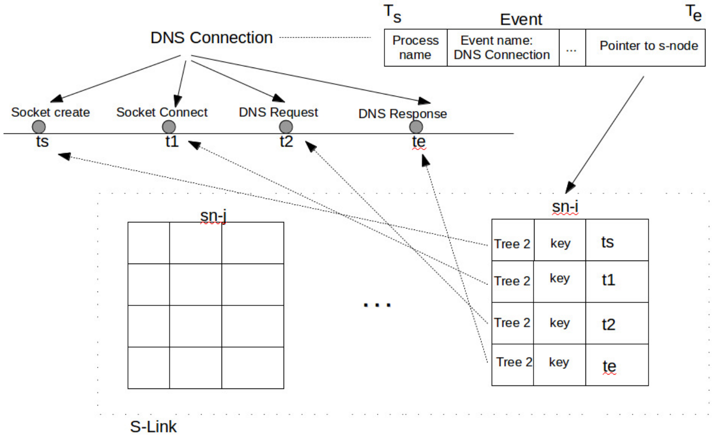


Figure 7.10 An example of linking the events using the S-Link data structure.

Hybrid Approach

An important benefit of the second approach is that the different data layers can be gathered from different sources (i.e. different tracers). For example, in our use-case, the highest level (e.g., the function calls) can be extracted from a user-space tracer while the low-level data (e.g., system calls and interrupts) is extracted from the kernel tracer. In this case, once the corresponding data in the different layers are linked together, it can easily be used to visualize the system execution at different scales.

However, one problem with this approach is that the abstraction process, used to generate the levels, does not have a sense of the visualization screen and graphical representation. In other words, the number of layers is set in advance, in this approach, by the event abstracter module, which does not have any information about the size of the visualization screen to adjust the size of abstract events to the size of the display. This may result in a less polished data visualization. Indeed, it does not know in advance if the layers are big enough to smoothly display the view, and avoid big jumps when hierarchically exploring the data, moving from one layer to another. This would be the case when the tool is aimed at visualizing different data sets with different sizes and properties. The hybrid approach is a solution to this problem, in which only some fixed layers and data elements are stored in advance, and other in-between layers and data elements are generated dynamically during the visualization phase.

The hybrid solution is in fact the result of a trade-off between the amount of storage space used and the speed of data retrieval and display. It actually combines the benefits of both approaches. It stores only some important data elements of each level, instead of all (the second approach) or none (the first approach), and generates in the visualization phase the whole level, on-the-fly using the stored information.

In this approach, only some abstract events, in either background or foreground layers, are stored and the rest is generated on the fly, in the visualization phase. Figure 7.12 displays a schematic comparison of the third (hybrid) and second solutions, in which only some snapshots (subsets) of events are stored in some checkpoints at each level, unlike the previous approaches (Figure 7.8). To select which events are stored at each level, we use the following algorithm :

Listing 7.1 Algorithm to decide about storing the events

```
Read the raw trace events.
Apply the abstraction techniques and generate high-level events
    (in different levels).
Store a snapshot of each level , after passing out each
    predefined processing time duration (e.g., each 10 seconds
    of processing).
Continue this process to the end of the input trace.
```

The checkpoints used to store the snapshots are of different sizes, because their duration is set dynamically, based on the processing time required to generate the abstract events. Unlike the other checkpoint methods using a fixed duration (fixed 10k events or 10 seconds of the trace e.g., in the partial history tree in [102]), here we use variable size checkpoints where the size is set dynamically based on the processing time for the abstraction process. For example, for the portions of the trace that contain a large number of events (and possibly a large number of abstract events), more snapshots are stored, as compared to parts with much fewer (abstract) events. This algorithm yields an almost balanced retrieval time for different trace areas.

At each checkpoint, a snapshot of events is stored. Each snapshot includes the important events lying within the previous and current checkpoint times. The importance of events is defined manually in advance or dynamically using some statistical information. For example, to analyze network applications, network-based events (socket create, send, receive, etc.) and packets can be ranked as important events. In the same way, to analyze an application from the file perspective, the file operations (file open, seek, read, etc.) are given high priority.

Using this approach, to extract and visualize the events, the state of the running processes is restored from the closest preceding checkpoint. The requested information is then generated and visualized by rereading and reprocessing the trace.

Figure 7.11 shows the way that the events of the previous example are stored and displayed (Figure 7.10) using the hybrid method. Checkpoints are displayed by dashed lines and the stored events are shown by arrows.

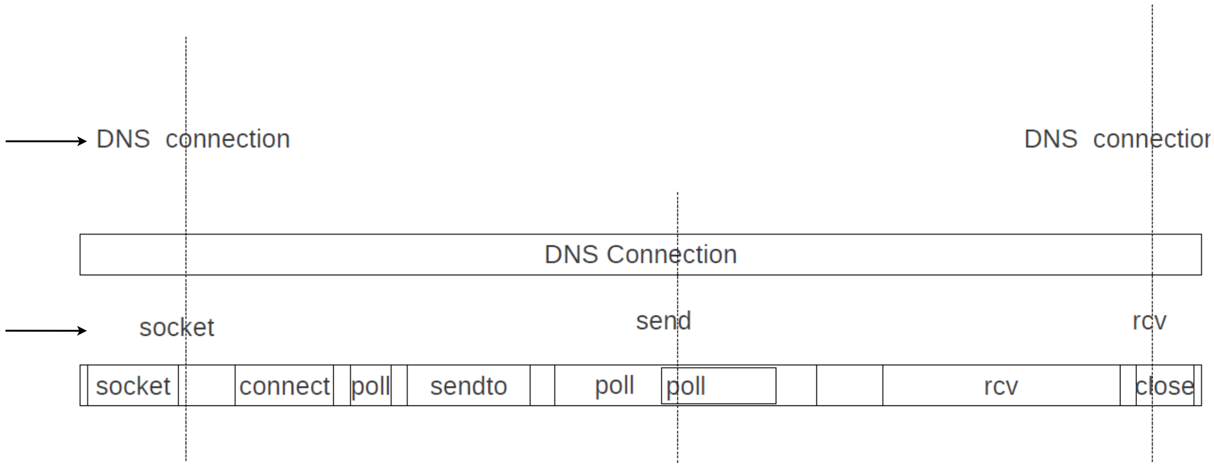


Figure 7.11 Hybrid solution and storing of only some important events in the checkpoints.

As shown in Figure 7.11 only some important events are stored in the checkpoints.

Checkpoints

A different checkpoint size is defined for each separate layer. The reason for defining a different checkpoint size for different layers is that the number of abstract events can be different from layer to layer. For example, the number of abstract events in the lowest levels (close to the bottom of the tree) should be higher than the number of events in the higher levels. In the proposed prototype, the higher levels events are generated by aggregating the events at lowest levels. However, this assumption can be somewhat relaxed in different applications. The only assumption here is that the size of checkpoint durations can be different for different levels, any layer could have a bigger or smaller checkpoint size.

In the designed configuration, the relation between checkpoint sizes is defined as :

$$CS_{Li} = 2 * CS_{Li-1} \tag{7.1}$$

where $CS_L i$ is the checkpoint size at layer i.

In other words, the checkpoint size of each layer is defined as twice the duration of the layer below, and half the duration of the layer above.

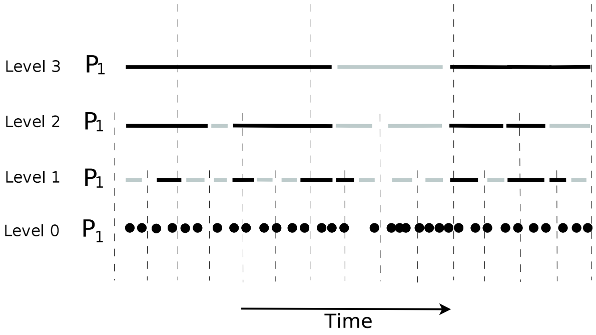


Figure 7.12 Store only some events (snapshots) of each level.

At each checkpoint, for each process, a list of m (a preset amount) important events, in addition to the internal states of the patterns, is stored. The interval states are used to regenerate the abstract events in case they are required in the visualization phase. The stored events are the important events that existed at the preceding checkpoint. By keeping track of the important events, we insure that at least these will be shown, when there is not enough space for all events to be shown in the view. In other words, for the areas of the trace with few events, the list may not be full, while for areas of the trace with many events, only a portion of them (up to m) will be recorded. In this case we insure that at least the important events will be shown, and the user has the option to zoom and see more events.

Each event is stored as an interval record containing the process name, event name, event arguments, bounding times, etc. It may contain a pointer to a corresponding S-Node record in the S-Link structure. This pointer is used to find the relevant information for this event. The importance of the events can be predefined statically by an administrator or can be measured dynamically by considering the size, type and frequency of events or using a set of patterns.

Some statistical information (e.g., the count of each event type, IO throughput, etc.) can also be stored in the checkpoints, when that statistical information should be shown, instead of (or in addition to) the events. Many trace visualization tools support multiple trace levels only by displaying the statistical information of traces. The proposed method can be used in these tools to manage the multi-level information.

7.6 Visualization Algorithms

The hybrid approach uses the same data structures used in the second approach, in which the events at each level (the ones selected to be stored) are stored in a separate tree, to be searchable by time or process name. For each event, a list of related events is stored in S-Link data structure, enabling to follow and retrieve the relevant events for each specific abstract event. In the following, the search and rendering algorithms are discussed.

Rendering Algorithm

Using the constructed data structure, users will be able to view and analyze the trace information in multiple levels. As explained earlier, the first view displays an overview of the trace execution, mainly a set of high-level process operations (fork, execute, running, block, wait, exit, etc). Background and foreground information for this view are stored (completely or partially depending on the approach used) in the data structure. Using a range query and dynamic trace abstraction, it will be possible to fetch the events, construct the view and render it for display. In the following, the algorithms to fetch and render the background and foreground layers are detailed.

Background layer

In the background layer, visual techniques are used to display the status of the underlying system execution. Rectangles, arrows, colors and labels are the most commonly used (visual) abstraction techniques for depicting the events in the display. Rectangles are used to display all events with a duration, e.g., system calls, interrupts, blocking waits, etc. Colors are in turn used to distinguishes those states, e.g., red for blocking wait, blue for system calls, green for user mode executions and so on. Arrows display any message passing (send, receive) or control flow transmission (fork, return) between processes.

The background layer only supports standard zooming. Therefore, to render its information (rectangles, colors, arrows, etc), it is sufficient to only store the events at the lowest level. Other layers will be built by using visual abstraction techniques. The rendering process of the background information is shown in Algorithm 7.2.

Listing 7.2 Algorithm to render the background layer view

```
1 Goal: visualizing a range [starting point, ending point]
2 Set query point t_q = starting point of the query (t_s).
3 Continue until query point t_q > ending point (t_e)
4 Query the first level tree of the data structure at time t_q
     and retrieve the event and operation at that point.
5 Draw the extracted event and color it, and possibly assign a
     label on it based on its type and size.
6 Find the ending point of this event (t_ec) and set the query
     point t_q to this new value (t_q = t_ec).
7 Go to line 3
```

To render the background layer, as shown in Algorithm 7.2, a range query is applied over the lowest level tree of the data structure. However, if the display time interval is large, there will not be enough space to render and display all the trace elements. Therefore, some visual abstraction techniques should be applied. Aggregating the rectangles, generalizing the events, colors, grouping the arrows, or diminishing or removing the labels are some of these visual abstraction techniques. For visual abstraction, further refinements are possible. For instance, when the whole display interval (the query duration) is less than a pixel on screen, one possible aggregation could be integrating the rectangles and putting a single dot as summary instead. To do so, the above algorithm is performed and continued until no more than one event remains in the given duration, then the algorithm puts a single dot on the screen and skips to the next interval.

Foreground layer

Unlike the background layer that uses only information stored at the lowest level tree, the foreground layer is extracted from all tree levels. There are stored pre-computed snapshots at each level (trees), but at visualization time, to obtain a smooth display, more abstract events might be generated using dynamic abstraction techniques. The dynamic abstraction techniques exploit the same pattern library used in the trace preprocessing phase and try to dynamically aggregate the events with respect to the screen size. As a result, some on-the-fly levels might be generated to better relate the views and keep zooming smooth and seamless.

The foreground layer information is visualized as labels overlaid on the background rectangles and colors. Labels show the name of abstract events generated in the trace processing and abstraction phase. A single area of the trace can have different names (abstract events), depending on the view level, from a single ”interrupt_entry”, to a ”buffer overflow attack”.

To render the labels on screen, the tool firstly locates the start and end points of the screen, in addition to the display level. It then queries the corresponding tree level and retrieves the events stored there. The whole process is shown in Algorithm 7.3.

Listing 7.3 Algorithm to render the foreground layer

```
1 Goal: retrieving the list of abstract events for a give range
     [starting point, ending point] and the given level no.
2 Set query point t_q = starting point of the query (t_s).
3 Continue until query point t_q > ending point (t_e)
4 Find and query the given level tree of the data structure at
     time t_q and retrieve the events and operations at that
     point.
5 Insert these extracted events in an ordered list (L).
6 Find the ending point of the containing node (t_ec) and set
     the query point t_q to this new value (t_q = t_ec).
7 Go to line 3
8 Return the ordered list (L)
```

In the above algorithm, a separate threshold value is assigned for each level, and when the given screen size crosses a threshold, the view changes level and shows the labels from the next (upper or level) level (i.e. semantic zooming). For zooming in/out between thresholds, the tool only enlarges the events and places more labels on screen if possible (i.e. standard zooming). The thresholds are determined statically in advance (as for online map tools), or set dynamically in relation to the whole trace duration and the average size of events at each level.

The above algorithm extracts the list of events from the data structures for any given time range. This list is then displayed as the foreground layer. However, the tool may not be able to display the whole list on screen. Indeed, when the screen space is small and too many labels are retrieved, label placement techniques like filtering, generalization and aggregation can alleviate the problem and put fewer but more important labels, increasing readability and display clarity. The detailed specification of the label placement algorithms can be found in [55].

Rendering the foreground layer, using the hybrid approach, is slightly different, as shown in Algorithm 7.4. In the hybrid approach, storing only a subset of abstract events should be considered.

Listing 7.4 Algorithm to retrieve the labels in the hybrid approach

```
Find the start and end points of the given range
Locate the corresponding checkpoints lying within the given
    range and with respect to the current display level

IF the input range is bigger than the threshold duration of
    that level (there is most probably some events there)
    query the corresponding tree at the checkpoint times and
        retrieve the ordered list of events
Else
        locate the previous checkpoint and retrieve the
            interval states of the patterns
     read the original trace and apply the trace abstraction
         techniques to re-generate the abstract events in the
         given time range
     if there is some generated events
         return the list of labels
     else return the list of one lower level events

Place the label in the screen
```

In the above algorithm, when the selected range is large enough, the labels can be obtained by querying the tree and reading the events of the checkpoint lying within the given range. However, a problem arises when there is no checkpoint across the given area in the current level. This happens when the selected area is too small or the distance between checkpoints is too large. In this case, the algorithm needs to reconstruct the events of the current level using the events from the lower levels. To do so, as shown in Algorithm 7.4, the tool queries the preceeding checkpoint and retrieves the internal states stored there. It then applies trace abstraction techniques (aggregation, generalization, etc) and generates the abstract events for the queried period. If aggregation is not possible, it simply returns the labels of the lower level and pushes them to the label placement algorithm to decide about which labels should be placed and shown on the screen.

Relating events

The proposed tool supports two kinds of linking between the different layers : structure (visual) based linking, and content-driven linking.

Structure-based linking connects the events of different layers based on their bounding boxes (i.e., timestamps). Events from multiple levels, shown for a selected time range, are actually related to each other because they display the same time range trace events, but at different scales. This kind of linking is typically used in online map systems and in graphical tools, to relate the different layers together. When users zoom in/out, the system retrieves the data that belongs to the same time range, and displays and relates visually the events. For example, if the tool shows a high-level ”file access” event on screen, when the user zooms in, he can follow that high-level event by viewing the ”file open”, ”file read” and other low-level events lying down within the corresponding time interval.

However, in the context of trace data, establishing another type of linking is required. For the ”file access” example, there may be some relevant low-level events beyond the selected time range, e.g., a ”file close” event that is actually part of the selected ”file access” event but occurring at a much later time. Thus, when zooming into this time range, this ”file close” event will not be shown, although it is a related event. Therefore, another type of linking between events in different layers is required, and is called content-driven linking. In content-driven linking, only the content-related events are fetched and displayed for a selected high-level event in the display. An example of this feature can be seen in Wireshark [4], when users want to only see the network packets belonging to a selected connection stream.

As explained earlier, the S-Link data structure is proposed to support this type of linking. S-Link is in fact a chain of down-pointers, with which the relevant data for any abstract event is stored and extracted when needed. Therefore, at visualization time, there will be two types of focusing, a focus based on the time axis (the semantic and standard zoom) using the timestamps to dig into the view, and content-based focusing available by double clicking on any high-level event to display the related lower level events.

The algorithm to fetch the related events for a selected abstract event (content-driven linking) works as following. It firstly finds the time range and the process ID of the query event and locates its position in the event tree. Each record in the event tree may contain a pointer to an entry in the S-Link data structure. By following this pointer in the S-Link structure, it will be possible to read the S-Node record(s) and return a list of related events. The other step is displaying these events on screen. In our prototype, these events are distinguished by highlighting. The process of fetching the related events is shown in Algorithm 7.5.

[4] http ://www.wireshark.org

Listing 7.5 Algorithm to fetch the related events

```
Goal: retrieving the list of related events for a given abstract event
Find the time range, the process ID and the S–Link pointer of the query abstract event.
Locate the corresponding S–Node record(s) in the S–Link structure.
Read the content of the S–Node record and fetch all its entries.
Return the list of related events and their time ranges to the displaying view.
```

7.7 Experiments

The proposed solution was prototyped in JAVA using trace logs generated by the LTTng kernel tracer. The Linux kernel version 2.6.38.6 is instrumented using LTTng (ver 1.x) and the tests were performed on a Core i7 2.8 GHz machine with 6 GB RAM. The numeric results are the average of ten separate runs. The traces are generated using a script built for generalizing traces up to a particular size. The trace logs of the following experiments are generated by running ”grep -r”, ”find -name”, ”ls -R” and ”wget -R”. The prototype tool will be contributed to the Eclipse Tracing and Monitoring Framework (TMF within the Linux Tools Project[5]).

7.7.1 Implementation and Outputs

The proposed tool prototype was written in JAVA and uses the LTTng kernel trace logs. Figures 7.13 to 7.17 show different layers of the trace events gathered by tracing the kernel level execution of the wget process. The ”wget -R” command was used to retrieve all files recursively from a web site.

In the proposed tool, the time is represented along one axis (X axis), and a list of active processes is shown along the other axis (Y axis). Behind the scene, several intervals and S-Link data structures are used to store the generated abstract events. The tool integrates different views and, instead of showing different coordinated views, displays the different layers in the same view by showing one level and hiding the rest. The views here only show the structure-based linking between the different layers. Zooming and panning enable users

[5] http ://lttng.org

to view other areas and layers. At each level, upon selecting a time duration and zooming into it, all lower level events lying within the time range of the selected event are extracted and shown.

Figure 7.13 shows a high-level ”download file” event and its corresponding events (the three Domain Name Service, DNS, connections and a HTTP connection). When zooming more on the first part of the view (shown by doted lines) the second screen is shown (Figure 7.14). In this figure, each DNS event is broken into two smaller events (dns-send and dns-recv). By zooming more and more, the source and destination of the ”DNS send” event (Figure 7.15) and then the kernel system calls are shown (Figures 7.16, 7.17).

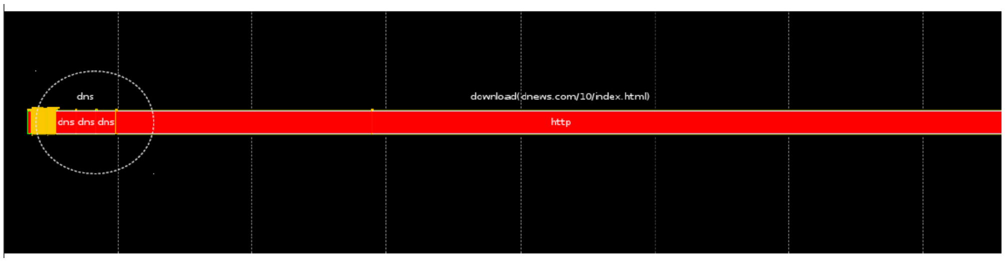


Figure 7.13 The higher level of the input trace log in the ”zoomable timeline view”.

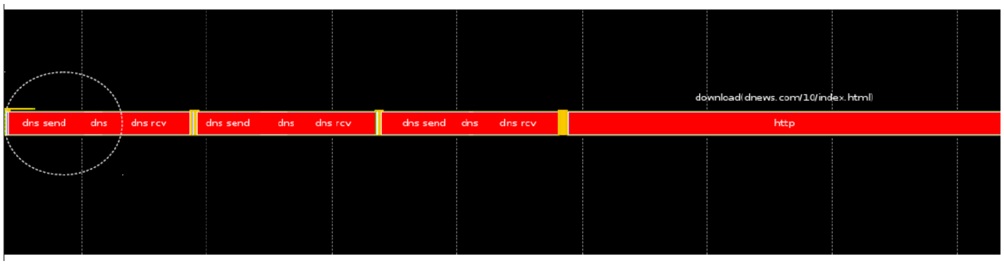


Figure 7.14 A middle level showing different DNS connections

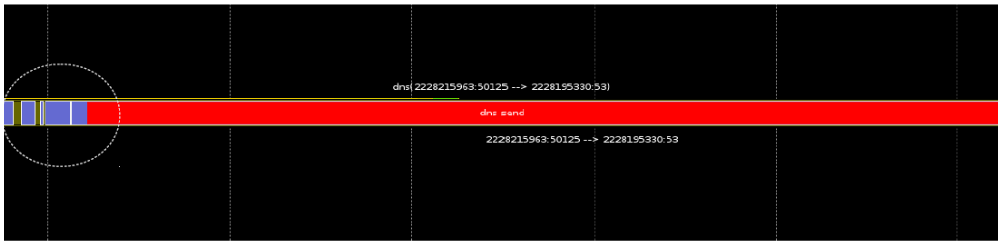


Figure 7.15 Details of a DNS connection.

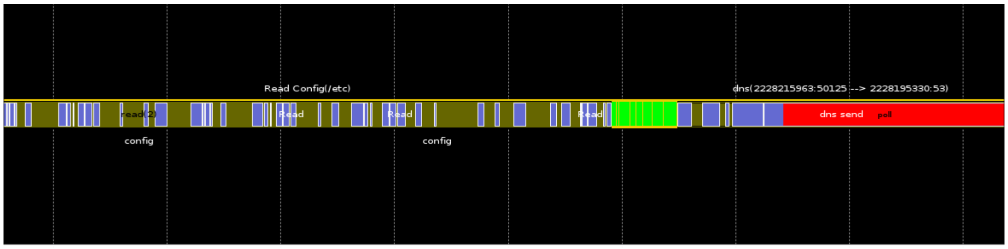


Figure 7.16 The system calls belong to a high-level DNS connection.

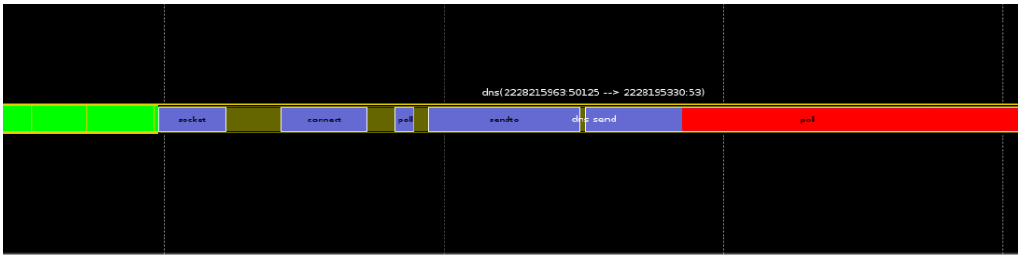


Figure 7.17 The lowest level of the input trace log in the ”zoomable timeline view”.

7.7.2 Reduction Rate

One reason to design and develop the proposed multi-level visualization was to display more events than with single level visualization. In this experiment, we compare the number of events displayed in a single level visualization versus a multi-level visualization. We use the example given in the previous section (shown in Figures 7.13 to 7.17) to compare the number of displayed events. When there is only a single level display, the number of events shown is 7 (Figure 7.17), while the number of events represented at the highest level is almost 600 events (Figure 7.13), including events for reading configuration files, three DNS connections, HTTP connection, etc. This method increases the actual number of displayed events, and can therefore be used to explore and understand traces in less time then with single level visualization.

7.7.3 Construction Time

The first experiment measures the time required to process the input logs and construct the data structures. This time consists in reading the trace events, applying the abstraction techniques, generating different levels of events and possibly writing them to disk. The construction time, however, is directly related to the size of the input trace and the number of patterns. A complete pattern set, to abstract file, network and process events, is used but, since the goal here is to compare the different proposed approaches, we use the same set of patterns in all experiments. Thus. in our results, we do not take into account the number of patterns.

Figure 7.18 shows the trace processing time for different situations : the case where all abstract events are stored in the data structures, the case where only the events are processed and abstract events are generated but not written to disk, and the case where only a part of abstract events is stored. The best processing time obviously belongs to the case where no abstract event is stored. As explained earlier, this approach can be used when it is known in advance that the trace size is too large, e.g., for live trace analysis.

The worst case is when the algorithm generates and stores all abstract events in databases. As explained earlier, a separate interval tree is used for abstract events at each level. Since there is more data to write to the databases, it takes more time.

For the hybrid solution, we consider two different checkpoint sizes (i.e, 1k and 15k checkpoints). The first stores data for each 1000 events, while second stores data for each 15000 events. We have defined the checkpoints based on the number of events, but it can be defined in time units as well (e.g., every 1 minute of execution, or 1 minute of processing). However, all those variants would work in the same way and lead to similar results. For prototyping, we have used the same checkpoint size for all processes. However, different sizes could be used. As shown in Figure 7.18, the time required for 15k checkpoints is less than for the 1k case. In both cases, almost the same processing time is required. However, with 1k checkpoints, there is more data to write to the database, increasing the overall cost as compared to 15k checkpoints.

Another experiment compares the construction time with a R-Tree. In the case where every layer is stored in the database, we could use a R-Tree as container for each abstract events layer. However, R-Trees are problematic with incrementally arriving data, because of the frequent re-balancing required as the tree is built. The proposed solution was designed to work with incrementally arriving trace data. Figure 7.19 shows this comparison. As explained, R-Trees require frequent re-balancing, causing a significant performance degradation.

7.7.4 Disk Size

To compare the disk space usage, we consider three cases : 1- All events are stored in each layer, 2- 1k checkpoints are used,3- 15k checkpoints are used. The results are shown in Figure 7.20.

In Figure 7.20, the worst case is again when every event is stored in all layers. A significant storage space is required to store all the generated abstract events. The best case is the hybrid solution using 15k checkpoints. After each 15k events, a snapshot of the previous events is generated and stored in the tree structures. The checkpoint size here is the size of checkpoints

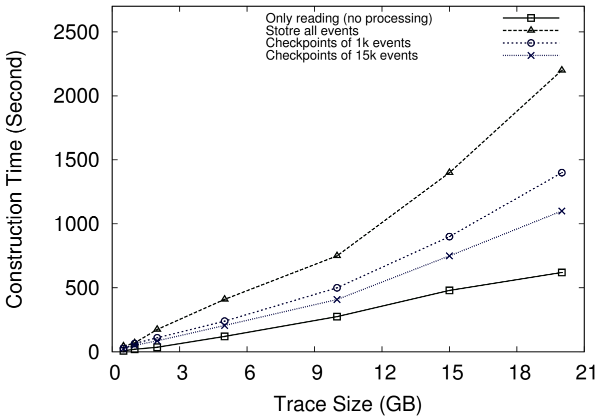


Figure 7.18 Construction time comparison of the different approaches.

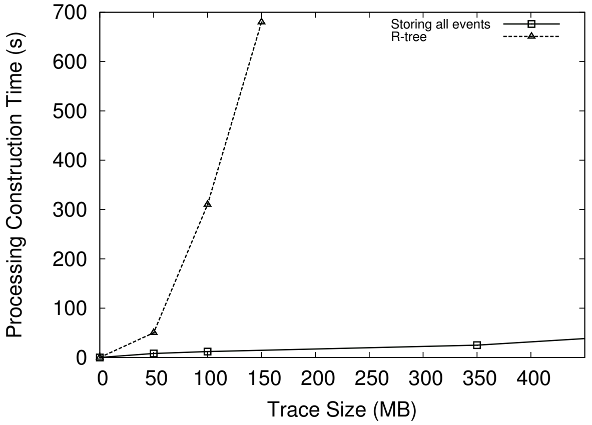


Figure 7.19 Comparison of construction time of our solution with R-Tree.

for the first level of the abstract events. For the other abstract levels, is this number is larger. We have simply set the checkpoint size of each layer as twice the size of the layer below.

As mentioned earlier, there is a difference between the type of data, and how it is stored and extracted, for the background and foreground layers. Unless otherwise mentioned, in all cases the focus is on the foreground layer which deals with different levels of abstract events. While the background stores the events in single layer, and create different layers on the fly abstracting these events, the foreground uses different interval trees for different layers. The goal here was to obtain an overview of the disk space usage for each solution.

7.7.5 Query Time

The query time is another important aspect of performance and was measured in this experiment, for the different approaches. Results are shown in Figure 7.21. Since we have

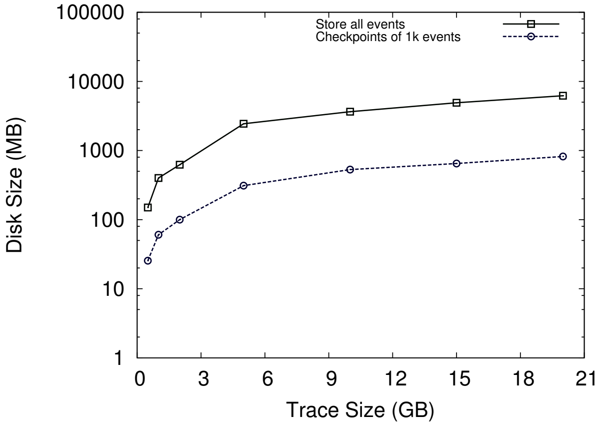


Figure 7.20 Disk usage comparison of the different approaches.

different sets of stored data, with different granularity for each solution, the query time is defined as the time required to extract ”the same set of outputs”. In other words, the hybrid approach was forced to reread the original trace data and regenerate some trace events. The experiment is executed multiple times for the first abstract levels and the resulting number is the average of these executions. For each execution, the query area is selected at random, with a duration larger than the checkpoint size of the level, to insure that all databases are queried.

Since the same set of output was expected with all solutions, some stored others needing regeneration, it was required to query the trace events, in addition to the abstract events containers. As shown in Figure 7.21, the worst case is again when all events are stored. We were actually expecting better results with this solution. The lower than expected performance could be related to the size of the database and how data is extracted. It required range queries upon large trees to fetch the needed data, which may be costly. In the hybrid method, on the other hand, a few stabbing queries required fetching events stored in the checkpoints, in addition to read the trace events and regenerate the abstract events. The later phase (reprocessing the trace events) slows down the approach, bringing the query time closer to the other approach (storing every event). Nonetheless, we were expecting a larger difference between the query times of the two approaches.

7.7.6 Discussion

Multi-level views are mostly used in spatial applications (like online maps) and rarely in the context of tracing. In this paper, we designed, modeled and prototyped a multi-level view for trace data and showed its interest and feasibility.

A general solution is proposed to visualize kernel trace events in multiple levels using a

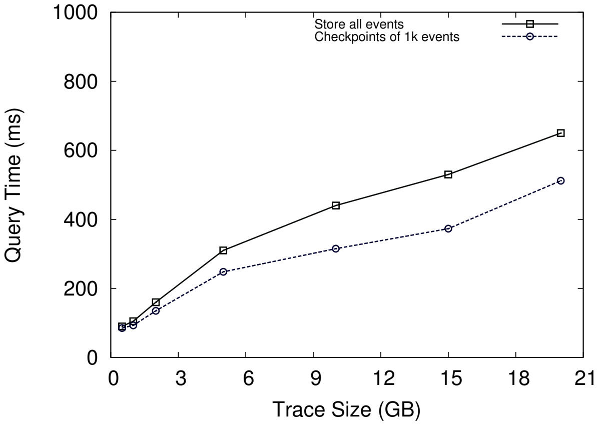


Figure 7.21 Comparison of time required to fetch the data from the stored databases.

zoomable timeline view. This multi-level tool helps users to get a quick and initial feeling of what is really happening in the underlying system. It can further be used to perform a multi-scale analysis, using the information presented in different layers. In essence, the tool can enable users to explore quickly very large traces and more easily understand their meaning.

Comparing the Approaches

Three approaches were presented in this paper to store abstract events, those displayed in the foreground. While there are performance differences between these approaches, it is in fact the application and use case that determines which should be used. In every case there is a tradeoff between the processing time, the query time and the amount of storage space used.

For instance, the solution that does not store any abstract event can be used in live trace analysis. The one that stores all events (in different databases) has a query time not better than the third solution but it can do without the original trace data. Once it reads the trace and generates the required files, it can drop the original trace. In some applications, with a long duration and too large trace, this solution can be useful. The hybrid (checkpoint) solution is an approximation method that stores only a snapshot at different points in the trace. It sometimes needs access to the original trace, when more details are requested, in the visualization phase.

As a consequence, one may ask if the checkpoint solution can be representative of the whole trace, especially at higher levels, while it stores only a snapshot of the events. The answer lies in several important factors that can make the solution complete, useful and efficient.

One factor is the distance between the checkpoints. With closely spaced checkpoints, the density of events and labels for the selected areas will be higher. Label placement techniques will be required to avoid collisions and make the labels readable and unambiguous. With a larger distance between checkpoints, the number of events to display will be too small for some areas, leaving the user with fewer clues to understand that portion. Users have to zoom to get more events, labels and details to understand what is happening there. It is important to note that users do not require a complete set of events at each layer, since they can zoom and focus to get more data. For example, online map systems (e.g., Google Map) only show the important city/province names to users, when there is not enough screen space to display everything. Our solution uses the same approach.

In each snapshot, an important set of events for that portion is stored. When the display area is too small and cannot show all available information to understand the trace at that point, zooming and focusing can help. Users can simply zoom in and focus on a selected area to get more insight and details. In that case, the algorithm queries and fetches data from lower layers or, in the worst case, loads the original trace and rebuilds that portion with the help of stored snapshots.

Correspondence Linking

The proposed solution visualizes the trace events at different layers. A main feature of the solution is linking together the related events at different layers. For each abstract event, it is possible to store a list of relevant events. Establishing links between an abstract event and its related raw trace elements provides an ”event location” feature which can be used in root cause analysis, to investigate a selected high-level problem. For example, the higher level in this tool may show a list of problems detected by the abstraction and detection module, and a list of related information is stored for each high-level problem. Then, using the proposed system, it would be possible to fetch more details about any selected problem, enabling a detailed study and identifying the root cause of that problem.

Other Use-cases

The proposed storage method, to link together elements from multiple levels for visualization, can also be used in other applications and use cases. In the following, we give some examples to outline how the proposed tool and techniques can be applied to other use cases.

One example is correlating the sender and receiver nodes in different machines. Suppose that there is a sender machine and one or more receiver machines and we would like to link the interacting processes and visualize them with the proposed tool. We can easily find out (e.g., IP addresses and ports) which nodes are communicating with each other on different machines. If we want to use the proposed tool to model and display the interactions, it is possible to store the events and the processes of each machine in a separate event container and the links between them in S-Link data structures. They can then be visualized with the proposed tool without modification. Figure 7.22 shows a view of this use case.

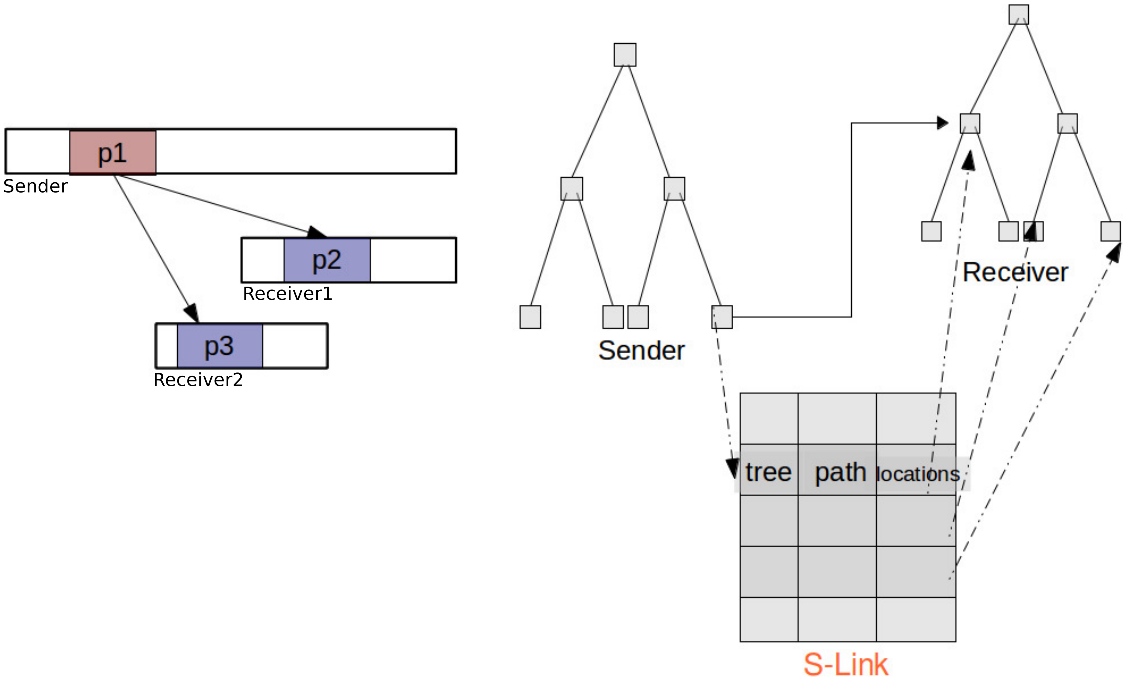
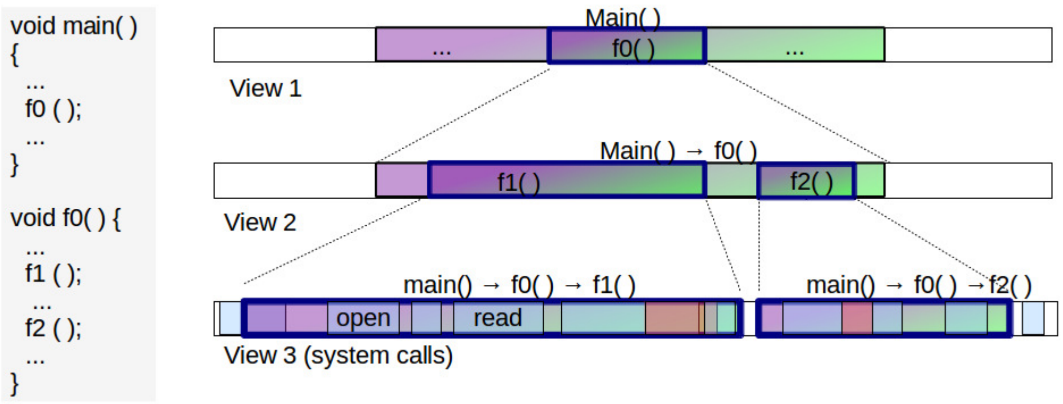


Figure 7.22 Linking the communicating nodes using the proposed tool.

In the graph shown in Figure 7.22, there is one sender and multiple receivers. A separate event container tree can be used to store the events of each machine. S-Links are also used to store the interactions. In the visualization phase, the first view will be the sender machine, but it would be possible to see the receiver events if the user double clicks on a communication arrow.

Another example is to link and visualize the trace events gathered from different sources. Figure 7.23 shows such a view. The higher level views display the function calls of the source code, as shown in the left side of the figure. In this use case, trace files collected from LTTng user space tracing and kernel space tracing are linked and visualized using the proposed tool. The user can zoom from the higher levels, showing the function calls, to the lower levels that display the system calls and kernel level events. The linking between events is based on the timestamps and the process name of the events. This example can be extended to be used for root cause analysis, showing system attacks at the higher level, and enabling users to dig into and zoom in to see the related kernel events, to find the underlying causes of the problems.



```
void main( )
{
  ...
  f0 ( );
  ...
}

void f0( ) {
  ...
  f1 ( );
   ...
  f2 ( );
  ...
}
```

Figure 7.23 Visualizing the function calls from user space trace data and kernel system calls simultaneously.

7.8 Conclusion and Future work

In this paper, a ”zoomable timeline view” and corresponding data structures and algorithms were presented. They provide multi-level visualization and analysis of trace events. The display first shows an overview and enables zooming in any area within the data. When zooming in/out each part of the view, the tool retrieves data from lower or higher levels and displays the result. To present a gradual transition, it sometimes integrates and shows data from two subsequent layers together.

The tool is implemented to simplify reading and understanding trace logs to better understand the system execution. Indeed, raw trace logs may contain a huge number of low-level events and are difficult and tedious to analyze manually. Since the tool generates different data layers, it enables a top-down comprehension. In other words, since each layer contains and increasing amount of data from higher to lower levels, it can be used to quickly analyze the trace (and therefore the system execution) using different granularity levels for different portions of the trace.

The tool establishes structure-based (or boundary-driven) and content-driven relations between data at different levels. The structural links can cover the relations between the data at different levels. However, for some applications, a direct link between two related events from different levels may be required. For instance, when a user zooms into an interesting area, he may want to see only the corresponding events, but not all events within the given time range. Making a direct link, and establishing content-driven linking, can be used in addition to the structure-based linking, to simplify following and understanding a specific behavior of the system and root cause analysis.

Data structures and algorithms to implement a prototype of the proposed tool is presented in this paper. It was experimented with large trace logs to study the performance of the approach. However, more efforts will be required to integrate this tool into a complete framework such as the Eclipse Tracing and Monitoring Framework [6]. Further refinements and optimisations are likely to be implemented.

Another avenue for future work is to combine this approach with network security tools, to visualize and analyze system problems and network attacks. Indeed, the root cause analysis enabled by the proposed tool would be most welcome in security tools.

Acknowledgement

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research, and Defence Research and Development Canada (DRDC) is gratefully acknowledged.

[6] http ://lttng.org

CHAPTER 8

GENERAL DISCUSSION

Execution traces contain valuable information about system runtime behavior. However, this information is hidden behind millions of low-level trace events. Efforts are needed to analyze these logs and extract the requested information. Moreover, the size, complexity and low-level nature of the trace logs make the analysis difficult.

In the previous chapters, we presented a framework to extract high-level information from very large trace logs and visualize it in a proper way to enhance the comprehension of the underlying program execution. Program comprehension is useful for maintenance, reverse engineering, optimization, reuse, etc. This framework is structured in three parts to align with the three major objectives presented in the Introduction :

- Part 1 :Perform multi-level analysis to extract several levels of synthetic and abstract events.
- Part 2 :Organize abstract events and the links between them in efficient data structures.
- Part 3 :Visualize and explore trace events in an efficient and effective manner.

The first part, multi-level trace abstraction, describes different techniques to analyze execution traces and extract various levels of information. These levels result from different techniques : data-driven (content-driven), metric-based and structure-based trace aggregation.

In the data-driven approach, presented in Chapter 3, the contents and parameters of the trace events are recruited to filter out the less-important data, generalize the events, and aggregate and group the related events together. This approach is mostly used to reduce the trace size and its complexity to generate several hierarchical levels of abstract events. The increased expressiveness of the generated abstract events is used to simplify the trace analysis and exploration, enhance program execution understanding, and detect system problems (host-based problems and network attacks). The lower level is a direct combination of trace logs, while the highest level is the result of more complex patterns. A pattern library containing patterns of different Linux operations and some high-level scenarios is developed and used in this approach. The patterns are used within the abstraction process to reduce the trace size and generate a set of meaningful and high-level events. The resulting set of abstract events is not necessarily a complete representation of the original trace. It depends on the defined patterns, scenarios and objectives. For example, replacing several events opening and sequentially reading a file by a higher level *Sequential file read* abstract event significantly reduces the trace size and eases the comprehension of the functional behavior of the program. There is not really any information lost from that point of view. However, from a performance and input-output request optimization point of view, knowing how the individual read requests were issued (their size, order and timing), is important and the new higher level abstract event would loose significant information. The solution is being developed as a stateful approach to share the common internal states and information between different processing threads, increasing its efficiency, in terms of trace processing time and consumed storage space.

A metric-based trace abstraction approach was presented in Chapter 4. Contributions of this work included : approaches to extract different statistical information from the trace events and organize them efficiently to enable point/range queries for any arbitrary duration within the underlying trace. This work also explores a structure-based trace aggregation to extract and organize the different system resources (dimensions) involved in the underlying system execution. This resource extraction and organization may help users access the statistical information of the resources easily and efficiently. Moreover, it can be used to compare the runtime parameters of different resources together. This technique also enables the drill-down and roll-up operations, as well as a multi-level, multi-dimensional, and multi-granular aggregation of statistical information (e.g., users can view statistics for a process, aggregate them for a group of processes, for processes of a virtual machine, or for the whole system). The solution can be used inside the different data analysis tools (even beyond trace data) to provide a multi-level view of the aggregated data.

Applying the above techniques over trace streams introduces new challenges : how to aggregate the data and organize them for a streaming trace (i.e., an unlimited flow of trace events). Chapter 5 presented a solution to this problem, where we performed a metric-based abstraction, considering the arrival speed of the trace stream, and extracted and organized the different resources in a way that enabled a multi-level multi-dimensional OLAP style analysis. The proposed solution supports the different dynamic and static window queries to reason about the underlying stream behavior. This approach can be used to monitor and control the execution of (a specific module of the) applications (e.g., their resource usage, the amount of internal and external communication between their different modules, etc.).

One major difference between the abstraction techniques presented in Part 1 and previous background work is that the proposed abstraction techniques are designed to work with kernel trace data. Kernel traces have some differences with other trace data that introduce new challenges. For instance, despite the application level traces, no continuous set of events exists for a single process because of the CPU scheduling events. The concurrency of execution, dealing with several kernel modules, and interpreting the interrupts are other special features of kernel based trace data. This technique is also different given the scalability of the solution that it can support very large traces. Main memory based solutions usually enforce a size limit for the supporting trace. However, the proposed techniques mainly use disk-based data structures to support large traces. They do not impose a limit on the supported size, and were tested and evaluated with traces of hundreds of GB in size.

Using the above techniques presented in Chapters 3, 4 and 5, it is possible to generate several levels of abstract events, ranging from system call events at the lowest level, to network or system attacks and overall statistical aggregation in the highest levels. If the only requirement were the extraction of useful information from trace data and problem detection, the above techniques would be sufficient. However, the requirement is to organize the extracted information and present in a way that is more convenient, accessible, explorable, and useful for users. It should help users easily explore the trace events, move forward and backward in the trace, and understand the reasons for the detected problems and misbehavior. The organization of trace events and their visualization were presented in Parts 2 and 3.

The second and third parts of the proposed framework refer to the techniques used to organize, represent, and visualize the extracted abstract events. As defined earlier, abstract events are the durative events (associated with a time interval) with time boundaries (start and end time), an event name, the owning process name, and some other information. Therefore, interval management structures are required to organize and store them.

The proposed solution supports two data storage strategies, complete and partial. Complete strategy records all events and their corresponding information. However, in the visualization phase, there may not be enough screen space to display all events. Therefore, techniques are required to display and label only the important events (Chapter 6). The partial solution employs this idea and stores only the important events, extracted in the trace pre-processing phase. The presented structures (both complete and partial) also model the relationships between items in the hierarchy, to be used later to follow high-level behaviors within the low-level trace events.

Structured trace events at different levels require a proper visualization technique to enhance the execution analysis and comprehension. Different solutions to visualize hierarchical information were reviewed in Chapter 2. We exploited the ”zoomable view” technique, a popular solution that is mostly used in the online maps applications, but rarely in the trace context, and integrated with timeline diagrams to introduce a ”zoomable timeline view” to visualize the trace events at different levels. This technique first shows a trace overview at the highest abstraction level, to give an initial understanding of the underlying system execution. Users can then pan around in the overview level and zoom into any selected area to access other view levels containing more trace events, details, and information. A solution to store these durative events and visualize them at multiple levels is presented in Chapter 7.

Such a multi-level view requires some heuristic methods to display the items on the screen. One heuristic is the placement of readable labels on the visible objects. Label placement techniques are typically used in spatial applications (e.g., Maps). We proposed a fast label placement algorithm be used in the trace visualization context. Unlike map-based label placement techniques, the proposed technique uses a different assumption : the two dimensions of the plane are used asymmetrically, and label offsets are more acceptable along the X axis than the Y axis. The technique also employs dynamic aggregation of the objects, which aggregates and compresses the labels when there is not enough space to place all labels on the screen. This technique, discussed in Chapter 6, is used in the zoomable timeline view to place labels on both the row and abstract events.

The above visualization techniques support a top-down and multi-level analysis of trace events, in which events in one level may be analyzed with respect to events and information available in other levels, leading to a better comprehension. Indeed, there are situations where none of the individual high-level events and low-level events can help to understand the meaning of an execution, but a multi-level and multi-scale analysis of the levels concurrently can complete the comprehension.

The proposed data structures and visualization techniques also enable feature/fault localization and can be used in root cause analysis. Suppose a misbehavior (e.g., network attack) is detected in the execution and reported in the highest level of the view. To find the reason and possibly solve the problem, system administrators need to identify both the problematic execution interval and the corresponding trace events that reveal the problem. Since the system execution is visualized in the timeline view at different granularity levels, and the links between the events are also maintained, the administrators can easily dig into the problem and find the related trace events and information for the detected problem. They can then analyze the related events, and navigate back and forth in the trace to analyze and comprehend what exactly occurred in the system to cause the reported problem. The limitations and future work are discussed in the conclusion.

CHAPTER 9

CONCLUSION

This thesis focuses on presenting techniques and tools to facilitate an operating system level execution trace analysis and enhance the comprehension of the underlying system execution. To facilitate analysis, we presented a trace analysis framework containing two important parts : trace abstraction and trace visualization.

Several trace abstraction techniques are presented in the first part to reduce the trace size and generate several levels of high-level events : data-driven, metric-based and structure-based techniques. The data-driven techniques are those that use trace event content and parameters to filter, generalize, aggregate and group events, to remove the noise and generate several levels of synthetic events. These techniques construct and use a pattern library containing a predefined execution model of some low-level Linux operations and high-level behaviors.

A metric-based technique is also presented to aggregate trace events based on some predefined metrics and measures. This technique can be used to measure system resource utilization and possibly to detect some misuse and overuse problems. For instance, resource usage that passes a threshold may be reported as a potential problem (e.g., brute force requests, non-optimal cache utilization, continuous uncontrolled file accesses and modifications, etc.).

A structure-based technique is also used to extract and organize the resources involved in system execution. This technique may help to analyze the events belong to a only specific resource, or compare the behavior of resources.

The second part of the framework focuses on multi-level data organization and trace visualization. A zoomable timeline view is introduced to facilitate a multilevel display of system execution. This view enables user to navigate through the hierarchy of events and follow a selected high level behavior to find its corresponding kernel-level events. Efficient label placement techniques are used to make the views more readable and effective.

In general, the framework proposes effective solutions to deal with exploration, analysis, and visualization of large kernel traces. This work resulted in 3 journal papers published or accepted and 2 additional journal papers submitted. The proposed abstraction and visualization techniques generated considerable industrial interest and are planned for inclusion in the Tracing and Monitoring Framework in the coming year.

Some techniques and directions for future work have already been mentioned in individual chapters of this thesis. Further suggestions for future work include :

Although we have tested the proposed framework (especially the visualization part) with

LTTng kernel traces, it was designed to apply to different trace types. However, further consideration is needed in order to improve the developed techniques for other data types and trace formats (e.g., Event Tracing for Windows format). We will investigate how to apply solutions over other types of data (possibly other trace formats) as a possible future work.

The pattern library includes some low and high-level operations and was constructed and used with the above abstraction techniques. However, the patterns were coded in an internal format defined specifically to support the techniques. More investigations will be required to describe and store the patterns in a flexible and easy to use interface format.

The proposed framework supports modeling, and establishing links between related events in the hierarchy, which can later be used to follow a high-level event within its related low-level data. This feature, as explained earlier, has applications in root cause analysis. In this work, we extracted, defined, and stored the links in an ad hoc manner. However, this may limit the extensibility of the system to support more data types. For instance, to use the proposed framework when the different abstract events have varied sources, it may not be easy to manually establish links between data elements. A flexible dynamic linking algorithm may alleviate this problem. In this method, some structural information in the objects may be used to match directly and dynamically the objects at different levels. Such a technique will be investigated in a future work.

LIST OF REFERENCES

[1] (2011). Nmap - free security scanner for network exploration and security audits.

[2] (2011). Rfc 793 : Transmission control protocol.

[3] ABDELLATIF, A. M. N. M. S., SLONIM, J., MCALLISTER, M. J. *ET AL.* (2011). Apparatus and method for dynamically materializing a multi-dimensional data stream cube. US Patent 8,024,287.

[4] AGRAWAL, J., DIAO, Y., GYLLSTROM, D. and IMMERMAN, N. (2008). Efficient pattern matching over event streams. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, SIGMOD 08, 147–160.

[5] ALAWNEH, L. and HAMOU-LHADJ, A. (2011). Pattern recognition techniques applied to the abstraction of traces of inter-process communication. *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, Washington, DC, USA, CSMR 11, 211–220.

[6] ARASU, A. and MANKU, G. S. (2004). Approximate counts and quantiles over sliding windows. *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, PODS 04, 286–296.

[7] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R. and WIDOM, J. (2002). Models and issues in data stream systems. *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, PODS 02, 1–16.

[8] BACON, D. F., CHENG, P. and GROVE, D. (2007). Tuningfork : a platform for visualization and analysis of complex real-time systems. *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 854–855.

[9] BEAUCAMPS, P., GNAEDIG, I. and MARION, J.-Y. (2010). Behavior abstraction in malware analysis. *Proceedings of the First international conference on Runtime verification*. Springer-Verlag, Berlin, Heidelberg, 168–182.

[10] BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R. and SEEGER, B. (1990). The r*-tree : an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19, 322–331.

[11] BEEN, K., DAICHES, E. and YAP, C. (2006). Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12, 773–780.

[12] BEEN, K., NÖLLENBURG, M., POON, S.-H. and WOLFF, A. (2010). Optimizing active ranges for consistent dynamic map labeling. *Comput. Geom.*, 43, 312–328.

[13] BLACK, J. P., COFFIN, M. H., TAYLOR, D., KUNZ, T. and BASTEN, A. A. (1994). Linking specification, abstraction, and debugging. Rapport technique, Computer Communications and Networks Group, University of Waterloo.

[14] BLANCH, R. and LECOLINET, E. (2007). Browsing zoomable treemaps : Structure-aware multi-scale navigation techniques. *IEEE Transactions on Visualization and Computer Graphics,*, 13, 1248–1253.

[15] BLIGH, M., DESNOYERS, M. and SCHULTZ, R. (2007). Linux kernel debugging on google sized clusters. *OLS (Ottawa Linux Symposium) 2007*. 29–40.

[16] CANTRILL, B. M., SHAPIRO, M. W. and LEVENTHAL, A. H. (2004). Dynamic instrumentation of production systems. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, ATEC 04, 2–2.

[17] CECCONI, A. (2003). *Integration of cartographic generalization and multi-scale databases for enhanced web mapping*. Thèse de doctorat, Universität Zürich.

[18] CHAN, A., GROPP, W. and LUSK, E. (2008). An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Sci. Program.*, 16, 155–165.

[19] CHAN, A., HOLMES, R., MURPHY, G. C. and YING, A. T. T. (2003). Scaling an object-oriented system execution visualizer through sampling. *In International Workshop on Program Comprehension*. IEEE, 237–244.

[20] CHATURBHAI, P. D. (2011). *MINING PATTERNS IN COMPLEX DATA*. Thèse de doctorat.

[21] CHEN, Y., DONG, G., HAN, J., WAH, B. W. and WANG, J. (2002). Multi-dimensional regression analysis of time-series data streams. *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, VLDB 02, 323–334.

[22] CHRISTENSEN, J., MARKS, J. and SHIEBER, S. (1995). An empirical study of algorithms for point-feature label placement. *ACM Trans. Graph.*, 14, 203–232.

[23] COCKBURN, A., KARLSON, A. and BEDERSON, B. B. (2009). A review of overview+detail, zooming, and focus+context interfaces. *ACM Comput. Surv.*, 41, 2 :1–2 :31.

[24] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J. and CHASE, J. S. (2004). Correlating instrumentation data to system states : a building block for automated diagnosis and control. *Proceedings of the 6th conference on Symposium on Operating Systems Design Implementation -Volume 6*. USENIX Association, Berkeley, CA, USA, 16–16.

[25] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T. and FOX, A. (2005). Capturing, indexing, clustering, and retrieving system history. *SIGOPS Oper. Syst. Rev.*, 39, 105–118.

[26] COLLINS, C. and CARPENDALE, S. (2007). Vislink : revealing relationships amongst visualizations. *IEEE Trans Vis Comput Graph*, 2007.

[27] CORNELISSEN, B. and MOONEN, L. (2008). On large execution traces and trace abstraction techniques. *Software Engineering Research Group, Delft*.

[28] CORNELISSEN, B., ZAIDMAN, A., VAN DEURSEN, A., MOONEN, L. and KOSCHKE, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35, 684–702.

[29] DAS, S. K. and JOHNSON, E. E. (1995). Accuracy of filtered traces.

[30] DE BERG, VAN KREVELD, OVERMARS and SCHWARZKOPF (2000). *Computational Geometry Algorithms and Applications 2nd ed.* Springer-Verlag.

[31] DE BERG, M. and GERRITS, D. H. P. (2012). Approximation algorithms for free-label maximization. *Comput. Geom. Theory Appl.*, 45, 153–168.

[32] DE KERGOMMEAUX, J. C., STEIN, B. and BERNARD, P. (2000). Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26, 1253 – 1274.

[33] DE PAUW, W. and HEISIG, S. (2010). Visual and algorithmic tooling for system trace analysis : a case study. *SIGOPS Oper. Syst. Rev.*, 44, 97–102.

[34] DE PAUW, W. and HEISIG, S. (2010). Zinsight : a visual and analytic environment for exploring large event traces. *Proceedings of the 5th international symposium on Software visualization*. ACM, New York, NY, USA, SOFTVIS 10, 143–152.

[35] DE PAUW, W., HELM, R., KIMELMAN, D. and VLISSIDES, J. (1993). Visualizing the behavior of object-oriented systems. *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. ACM, New York, NY, USA, OOPSLA 93, 326–337.

[36] DESFOSSEZ, J. (2011). *Résolution de problème par suivi de métriques dans les systèmes virtualisés*. Mémoire de maîtrise, Ecole Polytechnique de Montreal.

[37] DESNOYERS, M. and DAGENAIS, M. (2008). Lttng : Tracing across execution layers, from the hypervisor to user-space. *Linux Symposium*. 101.

[38] DESNOYERS, M. and DAGENAIS, M. R. (2006). The lttng tracer : A low impact performance and behavior monitor for gnu/linux. *OLS (Ottawa Linux Symposium) 2006*. 209–224.

[39] DEVOGELE, T., TREVISAN, J. and RAYNAL, L. (1996). Building a multi-scale database with scale-transition relationships. *International Symposium on Spatial Data Handling*. 337–351.

[40] DOGRUSOZ, U., KAKOULIS, K. G., MADDEN, B. and TOLLIS, I. G. (2007). On labeling in graph visualization. *Inf. Sci.*, 177, 2459–2472.

[41] DRAPER, G., LIVNAT, Y. and RIESENFELD, R. (2009). A survey of radial methods for information visualization. *IEEE Transactions on Visualization and Computer Graphics,*, 15, 759–776.

[42] ECKMANN, S., VIGNA, G. and KEMMERER, R. (2002). Statl : An attack language for state-based intrusion detection. *Journal of Computer Security*, 10, 71–104.

[43] EECKHOUT, L. and DE BOSSCHERE, K. (2004). Efficient simulation of trace samples on parallel machines. *Parallel Comput.*, 30, 317–335.

[44] EGELE, M., SCHOLTE, T., KIRDA, E. and KRUEGEL, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44, 6 :1–6 :42.

[45] EICK, S. G. and KARR, A. F. (2002). Visual scalability. *Journal of Computational and Graphical Statistics*, 11, 22–43.

[46] EIGLER, F. C., PRASAD, V., COHEN, W., NGUYEN, H., HUNT, M., KENISTON, J. and CHEN, B. (2005). Architecture of systemtap : a linux trace/probe tool.

[47] EZZATI-JIVAN, N. and DAGENAIS, M. (2014). Multiscale navigation in large trace data. *27th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE),2014*. 1–6.

[48] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2012). An efficient analysis approach for multi-core system tracing data. *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications (SEA 2012)*.

[49] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2012). Stateful synthetic event generator from kernel trace events. *Advances in Software Engineering*.

[50] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2013). A framework to compute statistics of system parameters from very large trace files. *ACM SIGOPS Operating Systems Review*, 47, 43–54.

[51] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). Cube data model for multilevel statistics computation of live execution traces. *accepted in Concurrency and Computation : Practice and Experience.*

[52] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). Fast label placement technique for multilevel visualization of execution traces. *submitted to Computing.*

[53] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). Multilevel visualization of large execution traces. *submitted to Journal of Visual Languages and Computing.*

[54] EZZATI-JIVAN, N. and DAGENAIS, M. R. (2014). Multiscale abstraction and visualization of large trace data : A survey. *submitted to The VLDB Journal.*

[55] EZZATI-JIVAN, N., SHAMELI-SENDI, A. and DAGENAIS, M. (2013). Multilevel label placement for execution trace events. *26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), 2013*. 1–6.

[56] FADEL, W. (2010). *Techniques for the Abstraction of System Call Traces.* Mémoire de maîtrise, Concordia University.

[57] FEKETE, J.-D. and PLAISANT, C. (1999). Excentric labeling : dynamic neighborhood labeling for data visualization. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, CHI 99, 512–519.

[58] FONSECA, R. (2008). *Improving Visibility of Distributed Systems through Execution Tracing.* Thèse de doctorat, EECS Department, University of California, Berkeley.

[59] FU, J. and PATEL, J. (1994). Trace driven simulation using sampled traces. *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*. vol. 1, 211 –220.

[60] FU, L. (2005). *Exploration and Visualization of Large Execution Traces.* Mémoire de maîtrise, University of Ottawa.

[61] GAEDE, V. and GUNTHER, O. (1998). Multidimensional access methods. *ACM Comput. Surv.*, 30, 170–231.

[62] GIANNELLA, C., HAN, J., PEI, J., YAN, X. and YU, P. S. (2002). Mining frequent patterns in data streams at multiple time granularities.

[63] GIRALDEAU, F., DESFOSSEZ, J., GOULET, D., DAGENAIS, M. R. and DESNOYERS, M. (2011). Recovering system metrics from kernel trace. *OLS (Ottawa Linux Symposium) 2011*. 109–116.

[64] GMBH, P. (1998). Vampir 2.0 tutorial. Rapport technique, Hermülheimer Straße 10 D-50321 Brühl, Germany.

[65] GOLAB, L. and ÖZSU, M. T. (2003). Issues in data stream management. *SIGMOD Rec.*, 32, 5–14.

[66] GUTTMAN, A. (1984). R-trees : a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14, 47–57.

[67] HAMOU-LHADJ, A. and LETHBRIDGE, T. C. (2002). Compression techniques to simplify the analysis of large execution traces. *Proceedings of the 10th International Workshop on Program Comprehension.* IEEE Computer Society, Washington, DC, USA, IWPC 02, 159–.

[68] HAMOU-LHADJ, A. and LETHBRIDGE, T. C. (2004). A survey of trace exploration tools and techniques. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, CASCON 04, 42–55.

[69] HAN, J., CHEN, Y., DONG, G., PEI, J., WAH, B. W., WANG, J. and CAI, Y. D. (2005). Stream cube : An architecture for multi-dimensional analysis of data streams. *Distrib. Parallel Databases*, 18, 173–197.

[70] HAN, J., CHENG, H., XIN, D. and YAN, X. (2007). Frequent pattern mining : current status and future directions. *Data Min. Knowl. Discov.*, 15, 55–86.

[71] HARLE, R. K. (2007). Spatial indexing for location-aware systems. *Proceedings of the 2007 Fourth Annual International Conference on Mobile and Ubiquitous Systems : Networking&Services (MobiQuitous).* IEEE Computer Society, Washington, DC, USA, MOBIQUITOUS 07, 1–8.

[72] HERMAN, I., MELANCON, G. and MARSHALL, M. (2000). Graph visualization and navigation in information visualization : A survey. *IEEE Transactions on Visualization and Computer Graphics,*, 6, 24–43.

[73] JABEUR, N. (2006). *A multi-agent system for on-the-fly web map generation and spatial conflict resolution.* these de doctorat en informatique, Universite Laval.

[74] JERDING, D. F., STASKO, J. T. and BALL, T. (1997). Visualizing interactions in program executions. *Proceedings of the 19th International Conference on Software Engineering.* ACM, New York, NY, USA, ICSE 97, 360–370.

[75] KAKOULIS, K. and TOLLIS, I. (1997). On the edge label placement problem. S. North, éditeur, *Graph Drawing*, Springer Berlin Heidelberg, vol. 1190 de *Lecture Notes in Computer Science.* 241–256.

[76] KAKOULIS, K. G. and TOLLIS, I. G. (2001). On the complexity of the edge label placement problem. *Computational Geometry*, 18, 1 – 17.

[77] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J. and BAHL, P. (2009). Detailed diagnosis in enterprise networks. *SIGCOMM Comput. Commun. Rev.*, 39, 243–254.

[78] KAPLAN, S. F., SMARAGDAKIS, Y. and WILSON, P. R. (1999). Trace reduction for virtual memory simulations. *SIGMETRICS Perform. Eval. Rev.*, 27, 47–58.

[79] K.H., A. and I., B. (2004). Mrdb approach to handle and visualise multiple dlms in a consistent. *In Proceedings of the 20th ISPRS Congress.* 12–23.

[80] K.J.DUEKER and J.A.BUTLER (2000). A framework for gis-t data sharing.

[81] KRANZLMÜLLER, D., GRABNER, S. and VOLKERT, J. (1997). Debugging with the MAD environment. *Parallel Computing*, 23, 199 – 217. Environment and tools for parallel scientific computing.

[82] KRIEGEL, H.-P., POTKE, M. and SEIDL, T. (2000). Managing intervals efficiently in object-relational databases. *Proceedings of the 26th International Conference on Very Large Data Bases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, VLDB 00, 407–418.

[83] KUMAR, S. (1995). *Classification and Detection of Computer Intrusions.* Thèse de doctorat, West Lafayette, IN, USA. UMI Order No. GAX96-01522.

[84] LAMPING, J., RAO, R. and PIROLLI, P. (1995). A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, CHI 95, 401–408.

[85] LANGE, D. and NAKAMURA, Y. (1997). Object-oriented program tracing and visualization. *Computer*, 30, 63–70.

[86] LAROSA, C., XIONG, L. and MANDELBERG, K. (2008). Frequent pattern mining for kernel trace data. *Proceedings of the 2008 ACM symposium on Applied computing.* ACM, New York, NY, USA, SAC 08, 880–885.

[87] LARUS, J. R. (1990). Abstract execution : A technique for efficiently tracing programs. *Softw. Pract. Exper.*, 20, 1241–1258.

[88] LEX, A., STREIT, M., KRUIJFF, E. and SCHMALSTIEG, D. (2010). Caleydo : Design and evaluation of a visual analysis framework for gene expression data in its biological context. *Pacific Visualization Symposium (PacificVis), 2010 IEEE.* 57 –64.

[89] LI, J., PLAISANT, C. and SHNEIDERMAN, B. (1998). Data object and label placement for information abundant visualizations. *Proceedings ACM Conference on New Paradigms in Information Visualization.* 41–48.

[90] LIN, J.-L., WANG, X. and S., J. (1998). Abstraction-based misuse detection : High-level specifications and adaptable strategies. *Computer Security Foundations Workshop, 1998. Proceedings. 11th IEEE.* 190 –201.

[91] LIN, X. and ZHANG, Y. (2008). Aggregate computation over data streams. *Proceedings of the 10th Asia-Pacific web conference on Progress in WWW research and development.* Springer-Verlag, Berlin, Heidelberg, APWeb 08, 10–25.

[92] M., A., A., G. and M., L. (1997). Defining a program behavior model for dynamic analyzers. *9th International Conference on Software Engineering and Knowledge Engineering.* 257–262.

[93] M., P. and BÉDARD, Y. (2008). Fundamental characteristics of spatial olap technologies as selection criteria. *Location Intelligence 2008.*

[94] MAKANJU, A., BROOKS, S., ZINCIR-HEYWOOD, A. and MILIOS, E. (2008). Logview : Visualizing event log clusters. *Privacy, Security and Trust, 2008. PST 08. Sixth Annual Conference on.* 99 –108.

[95] MALINOWSKI, E. and ZIMANYI, E. (2007). Logical representation of a conceptual model for spatial data warehouses. *GeoInformatica*, 11, 431–457.

[96] MALONY, A., HAMMERSLAG, D. and JABLONOWSKI, D. (1991). Traceview : a trace visualization tool. *Software, IEEE*, 8, 19 –28.

[97] MARKS, J. and SHIEBER, S. (1991). The computational complexity of cartographic label placement. Rapport technique.

[98] MATNI, G. and DAGENAIS, M. (2009). Automata-based approach for kernel trace analysis. *Canadian Conference on Electrical and Computer Engineering, 2009. CCECE 09.* 970–973.

[99] MATNI, G. N. and DAGENAIS, M. R. (2011). Operating system level trace analysis for automated problem identification. *The Open Cybernetics and Systemics Journal.*

[100] MEYER, M. and WENDEHALS, L. (2005). Selective tracing for dynamic analyses. *Comprehension through Dynamic Analysis*, 33.

[101] MIZOGUCHI, F. (2000). Anomaly detection using visualization and machine learning. *Proceedings of the 9th IEEE International Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises.* IEEE Computer Society, Washington, DC, USA, WETICE 00, 165–170.

[102] MONTPLAISI, A. (2011). *Stockage sur disque pour acces rapide d attributs avec intervalles de temps.* Mémoire de maîtrise, Ecole polytechnique de Montreal.

[103] MONTPLAISIR, A., EZZATI-JIVAN, N., WININGER, F. and DAGENAIS, M. (2013). Efficient model to query and visualize the system states extracted from trace data. A. Legay and S. Bensalem, éditeurs, *Runtime Verification*. Springer Berlin Heidelberg, vol. 8174 de *Lecture Notes in Computer Science*, 219–234.

[104] MONTPLAISIR, A., EZZATI-JIVAN, N., WININGER, F. and DAGENAIS, M. (2013). State history tree : an incremental disk-based data structure for very large interval data. *2013 ASE/IEEE International Conference on Big Data*.

[105] MOTE, K. (2007). Fast point-feature label placement for dynamic visualizations. *Information Visualization*, 6, 249–260.

[106] MUSTIERE, S. and DEVOGELE, T. (2008). Matching networks with different levels of detail. *Geoinformatica*, 12, 435–453.

[107] NAGEL, W. E., ARNOLD, A., WEBER, M., HOPPE, H.-C. and SOLCHENBACH, K. (1996). Vampir : Visualization and analysis of mpi resources. *Supercomputer*, 12, 69–80.

[108] NEYER, G. (2001). Drawing graphs. Springer-Verlag, London, UK, UK, chapitre Map labeling with application to graph drawing. 247–273.

[109] NORTH, C. and SHNEIDERMAN, B. (2000). Snap-together visualization : Can users construct and operate coordinated visualizations.

[110] ORSO, A., JONES, J. and HARROLD, M. J. (2003). Visualization of program-execution data for deployed software. *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, New York, NY, USA, SoftVis 03, 67–ff.

[111] PADDA, H. K. (2009). *CoMoVA -A comprehension measurement framework for visualization systems*. Thèse de doctorat.

[112] PATROUMPAS, K. and SELLIS, T. (2010). Multi-granular time-based sliding windows over data streams. *17th International Symposium on Temporal Representation and Reasoning (TIME), 2010*. 146–153.

[113] PAUW, W. D., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J. M. and YANG, J. (2002). Visualizing the execution of java programs. *Revised Lectures on Software Visualization, International Seminar*. Springer-Verlag, London, UK, UK, 151–162.

[114] PAUW, W. D., LORENZ, D., VLISSIDES, J. and WEGMAN, M. (1998). Execution patterns in object-oriented visualization. *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98*. 219–234.

[115] PESTANA, G., DA SILVA, M. and BEDARD, Y. (2005). Spatial olap modeling : an overview base on spatial objects changing over time. *Computational Cybernetics, 2005. ICCC 2005. IEEE 3rd International Conference on.* 149 –154.

[116] PETZOLD, I., GRÖGER, G. and PLÜMER, L. (2003). Fast screen map labeling—data-structures and algorithms. *Proc. 23rd Internat. Cartographic Conf. (ICC 03).* Durban, South Africa, 288–298.

[117] PIRZADEH, H., SHANIAN, S., HAMOU-LHADJ, A. and MEHRABIAN, A. (2011). The concept of stratified sampling of execution traces. *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on.* 225 –226.

[118] PITARCH, Y., LAURENT, A., PLANTEVIT, M. and PONCELET, P. (2009). Multidimensional data stream summarization using extended tilted-time windows. *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops.* IEEE Computer Society, Washington, DC, USA, WAINA 09, 250–254.

[119] PLAISANT, C., GROSJEAN, J. and BEDERSON, B. (2002). Spacetree : supporting exploration in large node link tree, design evolution and empirical evaluation. *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.* 57–64.

[120] PREMCHAND, U. (2003). *Intrusion Detection/Prevention Using Behavior Specifications.* Thèse de doctorat, Stony Brook, NY, USA.

[121] REISS, S. P. (2003). Visualizing java in action. *Proceedings of the 2003 ACM Symposium on Software Visualization.* ACM, New York, NY, USA, SoftVis 03, 57–ff.

[122] REISS, S. P. (2007). Visual representations of executing programs. *J. Vis. Lang. Comput.*, 18, 126–148.

[123] RENZ, M. (2006). *Enhanced Query Processing on Complex Spatial and Temporal Data.* Thèse de doctorat.

[124] ROBERTS, J. (2005). Tracevis : an execution trace visualization tool. *In Proc. MoBS 2005.* Citeseer.

[125] SCHMERL, B., ALDRICH, J., GARLAN, D., KAZMAN, R. and YAN, H. (2006). Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32, 454–466.

[126] SCHNORR, L. M., HUARD, G. and NAVAUX, P. O. A. (2009). Towards visualization scalability through time intervals and hierarchical organization of monitoring data. *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid.* IEEE Computer Society, Washington, DC, USA, CCGRID 09, 428–435.

[127] SCHNORR, L. M., LEGRAND, A. and VINCENT, J.-M. (2011). Multi-scale analysis of large distributed computing systems. *Proceedings of the third international workshop on Large-scale system and application performance*. ACM, New York, NY, USA, LSAP 11, 27–34.

[128] SEBRING, M., SHELLHOUSE, E., HANNA, M., and WHITEHURST, R. (1988). Expert systems in intrusion detection : A case study. *Proceedings of National Computer Security Conference*. 74–81.

[129] SELLIS, T., ROUSSOPOULOS, N. and FALOUTSOS, C. (1987). The r+-tree : A dynamic index for multi-dimensional objects. 507–518.

[130] SHAMELI-SENDI, A., EZZATI-JIVAN, N., JABBARIFAR, M. and DAGENAIS, M. (2012). Intrusion response systems : Survey and taxonomy. *SIGMOD Rec.*, 12, 1–14.

[131] SHAMELI-SENDI, A. S., DESFOSSEZ, J., DAGENAIS, M. R. and JABBARIFAR, M. (2013). A retroactive-burst framework for automated intrusion response system. *Journal Comp. Netw. and Communic.*, 2013.

[132] SHNEIDERMAN, B. (1992). Tree visualization with tree-maps : 2-d space-filling approach. *ACM Trans. Graph.*, 11, 92–99.

[133] SIVAKUMAR, N. and RAJAN, S. S. (2010). *Effectiveness of Tracing in a Multicore Environment*. Mémoire de maîtrise, Malardalen University.

[134] SRIVASTAVA, J. and LUM, V. Y. (1988). A tree based access method (tbsam) for fast processing of aggregate queries. *Proceedings of the Fourth International Conference on Data Engineering*. IEEE Computer Society, 504–510.

[135] STEINBERGER, M., WALDNER, M., STREIT, M., LEX, A. and SCHMALSTIEG, D. (2011). Context-preserving visual links. *IEEE Transactions on Visualization and Computer Graphics*, 17, 2249–2258.

[136] STOLTE, C., TANG, D. and HANRAHAN, P. (2003). Multiscale visualization using data cubes. *IEEE Transactions on Visualization and Computer Graphics*, 9, 176–187.

[137] TARJA, S., KOSKIMIES, K. and MULLER, H. (2001). Shimba : an environment for reverse engineering java software systems. *Softw. Pract. Exper.*, 31, 371–394.

[138] TIMPF, S. (1998). *Hierarchical structures in map series*. Thèse de doctorat.

[139] TU, L. and CHEN, Y. (2009). Stream data clustering based on grid density and attraction. *ACM Trans. Knowl. Discov. Data*, 3, 12 :1–12 :27.

[140] WALDNER, M., PUFF, W., LEX, A., STREIT, M. and SCHMALSTIEG, D. (2010). Visual links across applications. *Proceedings of Graphics Interface 2010*. Canadian Information Processing Society, Toronto, Ont., Canada, Canada, GI 10, 129–136.

[141] WALKER, R. J., MURPHY, G. C., FREEMAN-BENSON, B., WRIGHT, D., SWANSON, D. and ISAAK, J. (1998). Visualizing dynamic software system information through high-level models. *SIGPLAN Not.*, 33, 271–283.

[142] WALY, H. (2011). *A Complete Framework For Kernel Trace Analysis.* Mémoire de maîtrise, Laval University.

[143] WANG BALDONADO, M. Q., WOODRUFF, A. and KUCHINSKY, A. (2000). Guidelines for using multiple views in information visualization. *Proceedings of the working conference on Advanced visual interfaces.* ACM, New York, NY, USA, AVI 00, 110–119.

[144] XIAOMENG, L., YANHUI, W. and HAO, M. (2007). Linking multi-scale representations in a navigable database. *ISPRS Workshop on Updating Geo-spatial Databases with Imagery The 5th ISPRS Workshop on DMGISs.*

[145] XU, W., HUANG, L., FOX, A., PATTERSON, D. and JORDAN, M. (2009). Online system problem detection by mining patterns of console logs. *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining.* IEEE Computer Society, Washington, DC, USA, ICDM 09, 588–597.

[146] YAMAMOTO, M., CAMARA, G. and LORENA, L. A. N. (2002). Tabu search heuristic for point-feature cartographic label placement. *Geoinformatica*, 6, 77–90.

[147] YAMAMOTO, M., CÂMARA, G. and LORENA, L. A. N. (2005). Fast point-feature label placement algorithm for real time screen maps. *GeoInfo 05.* 122–138.

[148] YAMAMOTO, M., LORENA, L. A. N. and ESPACIAIS, N. P. (2005). A constructive genetic approach to point-feature cartographic label placement. *Metaheuristics : Progress as Real Problem Solvers.* Kluwer Academic Publishers, 285–300.

[149] YOELI, P. (1972). The logic of automated map lettering. *The Cartographic Journal*, 9, 99–108.

[150] ZAKI, O., LUSK, E., GROPP, W. and SWIDER, D. (1999). Toward scalable performance visualization with jumpshot. *Int. J. High Perform. Comput. Appl.*, 13, 277–288.

[151] ZHANG, R., KOUDAS, N., CHIN, B. and SRIVASTAVA, O. D. (2005). Multiple aggregations over data streams. *In Proc. ACM SIGMOD Int. Conf. on Management of Data.* 299–310.