Multi-scale Navigation of Large Trace Data: A Survey

Naser Ezzati-Jivan
Department of Computer and Software Engineering
Ecole Polytechnique Montreal
Montreal, Canada
n.ezzati@polymtl.ca

Michel R. Dagenais
Department of Computer and Software Engineering
Ecole Polytechnique Montreal
Montreal, Canada
michel.dagenais@polymtl.ca

ABSTRACT

Dynamic analysis through execution traces is frequently used to analyze the runtime behavior of software systems. However, tracing long running executions generates voluminous data, which is complicated to analyse and manage. Extracting interesting performance or correctness characteristics out of large traces of data from several processes and threads is a challenging task. Trace abstraction and visualization are potential solutions to alleviate this challenge. Several efforts have been made over the years in many subfields of computer science for trace data collection, maintenance, analysis, and visualization. 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 and methodological structure for categorizing tools and techniques aiming at multi-scale abstraction and visualization of execution trace data, and discuss the requirements and challenges faced in order to meet evolving user demands.

1. 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 [1]. Software comprehension is usually achieved by using static or dynamic analysis [2].

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 [3]. 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 usually 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) [1, 4]. However, there are other ways to dynamic analysis as well, for instance, VTune [5] and HPCToolkit [6] are two examples of sampling-based performance tools that add no instrumentation.

The use of dynamic analysis (through execution traces) to study system behavior is increasing among system administrators and analysts [7, 8, 9]. Tracing can produce precise and comprehensive information from various system levels, from (kernel) system calls [10, 11, 12] to high-level architectural levels [13], leading to the detection of more faults, (performance) problems, bugs and malwares than static analysis [14].

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 [15]. 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 [10].

In the literature, there are many techniques to cope with these problems: to reduce the trace size [12, 2], compress trace data [16, 4], decrease its complexity [17], filter out the useless and unwanted information [11] and generate high level generic information [18]. 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) [19, 20, 21].

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 [22, 23]. 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 [24, 25].

Of the many different trace visualization tools and techniques discussed in the literature [26, 20, 27, 3] and among those few interesting surveys on trace abstraction and visualization techniques [19, 15, 1], 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 techniques discussed in this paper are generic and applicable to any trace data, our focus will be more on the area of operating system (kernel) trace data and analysis, when this distinction is relevant. Kernel traces have some specific features that differentiate them from application (user) level traces. For example, there is mostly a discontinuity between the execution of events for a process, because of preemption and CPU scheduling policies. Another example is, unlike other (e.g., 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 1 depicts the topics investigated in this paper.

2. 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 [7, 8]. 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.

In the literature, various trace abstraction techniques are surveyed, including two recent systematic surveys [1, 2]. 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 four major categories: 1- Content-based (data-based) abstraction techniques, those based on the content of events. 2- Metric-based abstraction, the techniques to aggregate the trace data based on some predefined metrics and measures. 3- Visual abstraction, the techniques mostly used in the visualization steps. 4- Resource abstraction techniques, those techniques about extracting and organizing the resources involved within trace data.

2.1 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 [9] refers to trace only some selected modules/processes of the system, or gather only the interesting behavior, instead of the whole system. Abstract execution [28] 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 [3], a trace visualization tool, also uses a selective tracing method. LTTng Linux kernel tracer [29] can be configured to trace only the requested modules, say network, file system, etc, instead of tracing the whole operating system, resulting in a reduced amount of collected trace data which is useful and important for an efficient postmortem analysis.

Instead of processing all events, trace sampling selects and inspects only a variety of events from the trace data. Trace

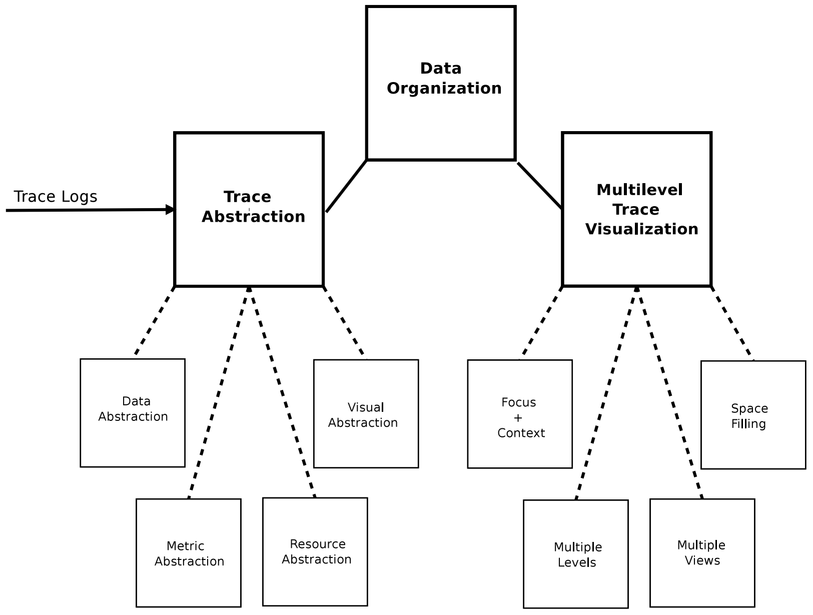


Figure 1: Taxonomy of topics discussed in this paper.

sampling is used in [30], [31], [32] and [33] and also in AVID visualization tool [26]. STAT (Stack Trace Analysis Tool) [34] is a scalable debugging tool that targets very large scientific applications. STAT processes the stack traces gathered during a sampling time to build a call graph prefix tree to extract and group common behavior classes within the program runtime space. In addition to sampling at the time of postmortem analysis, trace size reduction can be also done via sampling events at the time of measurement [35]. 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 [36, 37]. For example, an analyst may keep only the events related to socket and network operations and filter the rest out when he/she works only on the network behavior analysis. Fadel et al. [11] use filtering in the context of kernel traces to remove memory management events (out of analysis scope events) and page faults (noises) from the 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 [4]. 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. [16] studied both lossy and lossless compression techniques for memory reference traces and proposed two methods to reduce the trace size by discarding useless information [16]. 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 [38]. 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 [10]. PAPI interface [39] abstracts hardware specific events to a set of general and derived events (such as ratios of native and preset events and integration of events with system parameters). Using this interface users can dynamically specify the new derived events to be used in postmortem analysis and modelling.

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 [40]. In the literature, it is a broadly used method to reduce the size and complexity of the traces, and generate several levels of high level events [41, 11, 40, 42, 12, 18]. 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 [42].

For instance, Fadel et al. [11] used pattern matching to aggregate kernel traces gathered by the LTTng Linux kernel tracer [29]. Since trace data 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 [11], and some possible file system events between these two (Figure 2). A similar technique is exploited by [14] to group function calls into logically related sets, but in user-space level.

By 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 [42, 43].

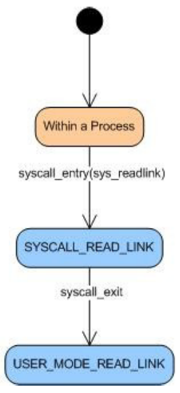


Figure 2: Recovering a "file read" event from trace events.

Source [11]

Wally et al. [12] 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 [11, 12], 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 [18].

Matni et al. [17] 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 [44]. 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 our previous work [10, 18]. We proposed a stateful synthetic event generator in the context of operating system kernel traces. An efficient trace aggregation method is designed for kernel traces considering all kernel specific features (extracting execution path, considering scheduling events, etc). The work shows 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 3).

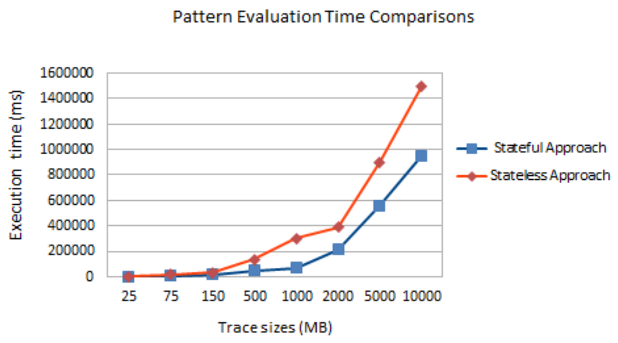


Figure 3: Trace aggregation time comparisons for stateful and stateless approaches [18].

Pattern mining techniques are also used to aggregate traces and extract high level information from trace logs [45]. 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 [46]. Han et al. [47] classified the pattern mining techniques into correlation mining (discovery of correlation relationships), structured pattern mining (similarity indexing of structured data), sequential pattern mining (mining of subsequent or frequently occurring ordered events), frequent pattern-based clustering (computing frequently occurring patterns in any subsets of high-dimensional data), and associative classification (discovery of different association between frequent patterns); 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. [8] and LaRosa et al. [7]. LaRosa et al. [7] 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 [7]. 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 [48, 49]. Productive evaluation of the specified patterns is closely related to multiple-query optimization in database systems [49] 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. [48]. 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 [18].

2.2 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 successful 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. [50], 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 [8, 51, 52]. Xu et al. [8] believe that system level performance indicators can denote the high-level application problems. Cohen et al. [52] 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 [51]. 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. [53] 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 [53, 54]:

- 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.
- The number (or area) of disk is mostly used, the latency of disk operations, and the distribution of seek distances.
- The network IO throughput, the number of failed connections.
- The memory usage of a process, the 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, Vampir [24], Jumpshot[27], TuningFork [21], 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 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 [55].

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) [23]. Many visualization tools use colors and shapes as the elementary elements to represent the trace events. Trace Compass and LTTV [56] 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 Trace Compass visualization tools [56], 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 of a message passed between different resources [57]. Labels, as a specific type of annotation, is also used in different visualization tools. Ovation [58], the Google chrome tracing tool [59] and Trace Compass [56] 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 [43].

2.4 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 gathers 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 [25, 41], filtering of uninteresting resources [60], or hierarchical organization of resources [61, 53, 62] are the related techniques used in the reviewed literature.

Montplaisir et al. [61] 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 4).

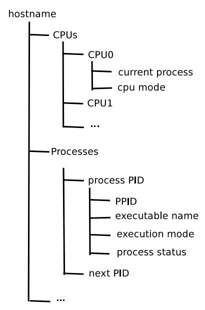


Figure 4: Hierarchical abstraction of resources extracted from trace events.

Another representation of resources extracted from trace events is presented in [53, 62]. 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. [25] 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 [41]. Two approaches were used: automatic clustering of processes based on their runtime inter-process communication intensity, and combining the inter-process communication information and the information of processes extracted from static source code [41].

2.5 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 [1]. 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. [63] 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 [38] uses a pattern matching approach to diagnose system problems. In [64], [65] and [66] similar techniques are also used to detect system problems and attacks. They mainly differ in describing and representing the patterns. STATL [64] models use signatures in the form of state machines, while in [65] signatures are expressed as colored petri nets (CPNs), and in MuSigs [66] directed acyclic graphs (DCA) are used to represent the security specifications. Chun Yuan et al. [67] provides an abstraction of system trace events by means of statistical learning technique and classifying system call sequences to automatic identification of root-causes of known problems (such as page loading error in Internet Explorer web browser).

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 [50, 17, 8]. The techniques used in pattern-based fault identification tools resemble the attack discovery 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.) [68].

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.

3. 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 [69]. 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.

3.1 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 [70], in which items are drawn radially, the higher levels at the center of the display and lower layers away from the center. Treemap [71] 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 [71]. It was later also used to visualize the trace data in Triva visualization tools [25], as well as in Gammatella [72] and LogView [73]. Figure 5 shows the visualization of one million items using the Treemap technique.

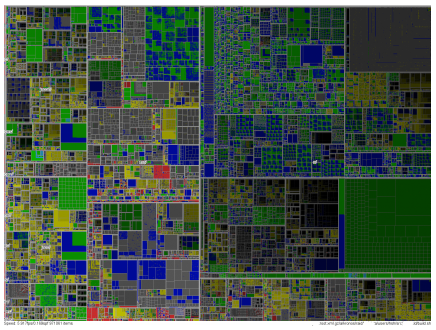


Figure 5: Visualization of one million items using treemap.
(source: wikipedia.org)

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.

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 [74]. 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. [74] 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 [75] has used the Hyperbolic tree and also machine learning techniques to find the tracks of an intruder in his anomaly detection system. Figure 6 depicts a view of the Hyperbolic technique.

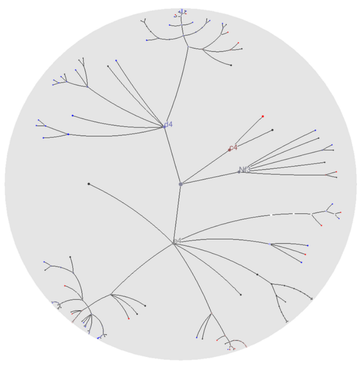


Figure 6: An example of a hyperbolic browser.
(source: wikipedia.org)

Multiple-levels Technique

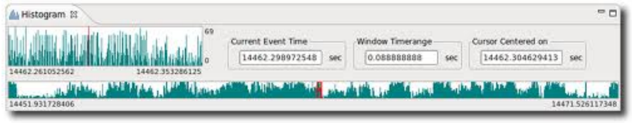
Focus+Context techniques are typically used for displaying and following data sets having clear hierarchies and categories. These techniques display both the context and focus within the same screen, but sometimes, when there is more detailed data for each selected area, it might not be possible to display both the details and the overview on the same display at the same time [76]. 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 [23], 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 [21], Jumpshot [27], Trace Compass [56] and Chrome tracing environment [59]), 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 [76]. 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) [43].

There is a special case of this method called overview + details, used in many visualization tools (Jumpshot [27], Vampir [24], Trace Compass [56], etc.), in which the overview and detailed views are shown next to each other (Figure 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 [76].



Current Event Time 14462.298972548 sec
Window Timerange 0.088888888 sec
Cursor Centered on 14462.304629413 sec

Figure 7: Overivew + Detail method (Trace Compass LTTng viewer).

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 [77].

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 [78].

This technique helps users to get a better comprehension through different views. Many trace visualization tools like Zinsight [20], TraceVis [57], Vampir [24], LTTV, Trace Compass [56], 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 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.

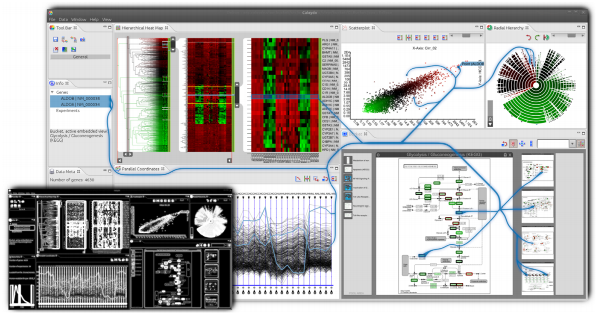


Figure 8: Context-preserving visualization of related items and links.

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 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 [79]. 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. [81], Waldner et al. [82] and Steinberger et al. [83].

Collins and Carpendale [81] 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 [84] to effectively present all relevant gene expressions and the relations between them and between multiple pathways. Waldner et al. [82] extend visual links to graphically present links between related pieces of items across desktop application windows. Figure 9 shows an example of the VizLink visualization.

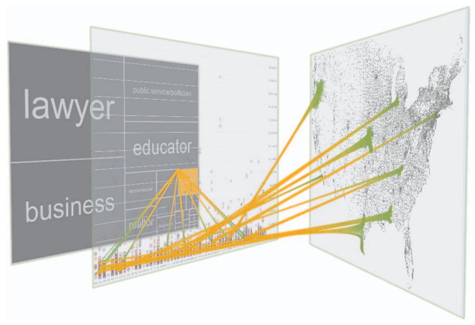


Figure 9: Visualization of related items and the links between them.

Steinberger et al. [83] believe that using this links visualization technique may lead to hiding valuable information. For solving this problem, they introduced context-preserving visual links [83]. 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 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.

3.2 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 [15, 20, 85, 22]. 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 [24] 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 [86]. 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 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 1.

Similar to Vampir, Jumpshot [27] 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 [87], 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.

The high-performance computing (HPC) field has developed a rich set of tools and techniques for trace collection, analysis, and visualization. Hardware performance counters, memory traffic, network traffic, call path samples, are some of the data collected by tracing tools in the HPC area. Except Vampir [24] and Jumpshot [27], there are other great HPC performance analysis tools such as Allinea's map [88], Intel's VTune [5], HPCToolkit's hpctraceviewer [6], TAU [89], and Scalasca [90]. Each of these tools has rich trace collection mechanisms and visualization interfaces. For example, hpctraceviewer [6] organizes its sampled traces in 3D, where the x-axis is the time, y-axis is is resources, and the depth-axis is the call stack depth. Scalasca [90] organizes its performance data in a CUBE format.

Significant efforts are made in visualizing HPC performance data. Isaacs et al. organize trace data in a logical time ordering for the ease of understanding performance anomalies [91]. Isaacs et al. [92] provide a rich survey of visualization tools for performance analysis. Landge et al. [93] devised 2D and 3D techniques to visualize the network traffic in massively parallel simulations. Bhatele et al. [94] devised a novel scheme to map performance data from a hardware topology to an application domain. They use intuitive colouring scheme and weighted arrows to indicate network bottleneck in a tree topology.

AVID (Architecture Visualization of Dynamics in Java System) [26] 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 [15].

Shimba [3] 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 [3]. 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 [95], 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. [95] 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 in some views.

Ovation [58] 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. [58] 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. [20], 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 fit in main memory [96].

TuningFork [21] 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 [57] 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 [25] 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) [25].

LTTV (Linux Trace Toolkit Viewer) [56], developed at Dorsal lab, is another visualization tool that shows events generated by the LTTng kernel tracer [29]. 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 Trace Compass [56] a full-featured Eclipse based LTTng trace viewer, with more views and features and supporting different visual abstraction mechanisms (e.g., labels, etc).

4. 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

Table 1: Trace visualization tools and their features.

Trace 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	✓		✓	✓	

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 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 opens and reads the trace from the nearest previous checkpoint to the given query point and rerun the patterns for this small set of events and regenerates the desired information. The checkpoint method is used in LTTV (the LTTng trace viewer) and was used initially in Trace Compass [56].

Related to the checkpoint method, LTTngTop, developed by Desfossez et al. [54] at Dorsal lab [1], 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. [53] 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. 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 [53] works well with offline traces and was tested for very large traces, since they store snapshots and trace events at regular-basis periods, the size of data stored behind their work grows linearly with the trace size and can become unexpectedly large for long streaming traces. The cube data model [62] 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. [97, 98] 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 [61]. 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 [97]. 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 [98, 61], 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 [87].

Chan et al. [87] 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.

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

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. [87] 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 [27]. 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. The 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 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.

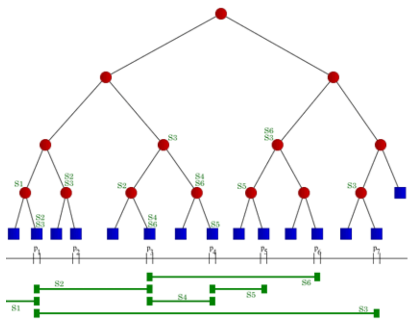


Figure 10: Examples of the segment and interval tree.

(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 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)).

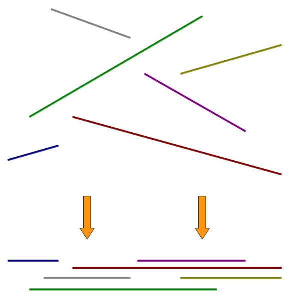


Figure 11: Segments versus intervals.

The Relational Interval Tree (RI-tree) proposed by Kriegel et al. [99], 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. [97] experimented with storing traces in databases but incurred a considerable overhead in both storage space and access time.

R-tree [100] 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 [101, 102].

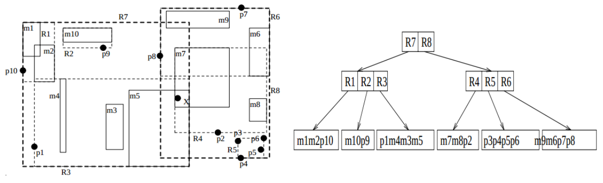


Figure 12: Set of 2D rectangles indexed by R-tree and the corresponding structure (right)

(source: wikipedia.org)

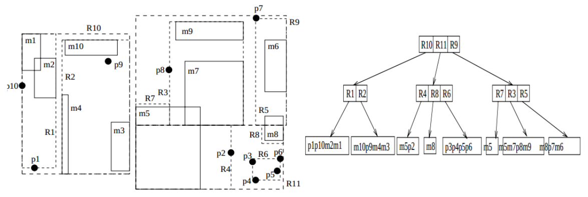


Figure 13: An example of the R+-tree to split entries in the tree to remove overlaps
(source: wikipedia.org)

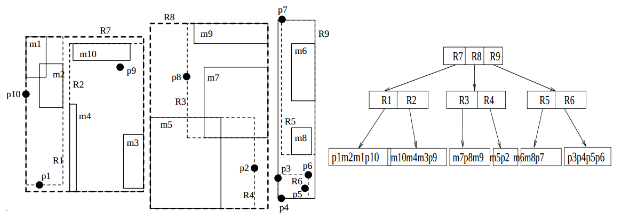


Figure 14: An example of the R*-tree which uses a combined strategy (little overlap but better query performance)
(source: wikipedia.org)

In general, the R-tree and its extensions do not work well for sequences of long intervals with significant overlap [103]. Indeed, splitting and merging (re-balancing) the nodes cause 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 [104], 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 [87] and State History Tree [98].

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.

5. 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 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 [25] node-link tree-representations [58], 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 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 [105] to explore very large hierarchical data (e.g., 700k nodes amongst 13 levels). A summary is shown in Table 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 efficient 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 [106, 107, 108]. 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.

6. 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 for 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 explo-

Table 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 un-structured 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

ration 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.

7. REFERENCES

[1] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, 2009.

[2] B. Cornelissen and L. Moonen, "On large execution traces and trace abstraction techniques," *Software Engineering Research Group, Delft*, 2008.

[3] S. Tarja, K. Koskimies, and H. Muller, "Shimba: an environment for reverse engineering java software systems," *Softw. Pract. Exper.*, vol. 31, pp. 371–394, April 2001.

[4] A. Hamou-Lhadj and T. C. Lethbridge, "Compression techniques to simplify the analysis of large execution traces," in *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC 02, (Washington, DC, USA), pp. 159–, IEEE Computer Society, 2002.

[5] Intel, "Intel vtune amplifier: Modern processor performance analysis," 2016.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 685–701, Apr. 2010.

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

[8] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM 09, (Washington, DC, USA), pp. 588–597, IEEE Computer Society, 2009.

[9] M. Meyer and L. Wendehals, "Selective tracing for dynamic analyses," *Comprehension through Dynamic Analysis*, p. 33, 2005.

[10] N. Ezzati-Jivan and M. R. Dagenais, "Stateful synthetic event generator from kernel trace events," *Advances in Software Engineering*, January 2012.

[11] W. Fadel, "Techniques for the abstraction of system call traces," Master's thesis, Concordia University, 2010.

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

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

[14] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, pp. 6:1–6:42, Mar. 2008.

[15] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON 04, pp. 42–55, IBM Press, 2004.

[16] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson, "Trace reduction for virtual memory simulations," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, pp. 47–58, May 1999.

[17] G. N. Matni and M. R. Dagenais, "Operating system level trace analysis for automated problem identification," *The Open Cybernetics and Systemics Journal*, April 2011.

[18] N. Ezzati-Jivan and M. R. Dagenais, "An efficient analysis approach for multi-core system tracing data," in *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications (SEA 2012)*, 2012.

[19] S. P. Reiss, "Visual representations of executing programs," *J. Vis. Lang. Comput.*, vol. 18, pp. 126–148, Apr. 2007.

[20] W. De Pauw and S. Heisig, "Zinsight: a visual and analytic environment for exploring large event traces," in *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS 10, (New York, NY, USA), pp. 143–152, ACM, 2010.

[21] D. F. Bacon, P. Cheng, and D. Grove, "Tuningfork: a

platform for visualization and analysis of complex real-time systems," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 854–855, ACM, 2007.

[22] L. M. Schnorr, A. Legrand, and J.-M. Vincent, "Multi-scale analysis of large distributed computing systems," in *Proceedings of the third international workshop on Large-scale system and application performance*, LSAP 11, (New York, NY, USA), pp. 27–34, ACM, 2011.

[23] C. Stolte, D. Tang, and P. Hanrahan, "Multiscale visualization using data cubes," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 9, pp. 176–187, April 2003.

[24] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[25] L. M. Schnorr, G. Huard, and P. O. A. Navaux, "Towards visualization scalability through time intervals and hierarchical organization of monitoring data," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID 09, (Washington, DC, USA), pp. 428–435, IEEE Computer Society, 2009.

[26] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak, "Visualizing dynamic software system information through high-level models," *SIGPLAN Not.*, vol. 33, pp. 271–283, Oct. 1998.

[27] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with jumpshot," *Int. J. High Perform. Comput. Appl.*, vol. 13, pp. 277–288, Aug. 1999.

[28] J. R. Larus, "Abstract execution: A technique for efficiently tracing programs," *Softw. Pract. Exper.*, vol. 20, pp. 1241–1258, Nov. 1990.

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

[30] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying, "Scaling an object-oriented system execution visualizer through sampling," in *In International Workshop on Program Comprehension*, pp. 237–244, IEEE, 2003.

[31] J. Fu and J. Patel, "Trace driven simulation using sampled traces," in *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, vol. 1, pp. 211 –220, Jaunary 1994.

[32] L. Eeckhout and K. De Bosschere, "Efficient simulation of trace samples on parallel machines," *Parallel Comput.*, vol. 30, pp. 317–335, March 2004.

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

[34] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–10, IEEE, 2007.

[35] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto, "Scalable fine-grained call path tracing," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 63–74, ACM, 2011.

[36] A. Malony, D. Hammerslag, and D. Jablonowski, "Traceview: a trace visualization tool," *Software, IEEE*, vol. 8, pp. 19 –28, September 1991.

[37] S. K. Das and E. E. Johnson, "Accuracy of filtered traces," 1995.

[38] U. Premchand, *Intrusion Detection/Prevention Using Behavior Specifications.* PhD thesis, Stony Brook, NY, USA, 2003.

[39] S. Moore and J. Ralph, "User-defined events for hardware performance monitoring," *Procedia Computer Science*, vol. 4, pp. 2096–2104, 2011.

[40] A. M., G. A., and L. M., "Defining a program behavior model for dynamic analyzers," in *9th International Conference on Software Engineering and Knowledge Engineering*, pp. 257–262, June 1997.

[41] J. P. Black, M. H. Coffin, D. Taylor, T. Kunz, and A. A. Basten, "Linking specification, abstraction, and debugging," tech. rep., Computer Communications and Networks Group, University of Waterloo, 1994.

[42] R. Fonseca, *Improving Visibility of Distributed Systems through Execution Tracing.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[43] N. Ezzati-Jivan, A. S. Sendi, and M. R. Dagenais, "Multilevel label placement for execution trace events," in *CCECE*, pp. 1–6, 2013.

[44] F. Perrad, "Smc - the state machine compiler," 2015.

[45] L. Alawneh and A. Hamou-Lhadj, "Pattern recognition techniques applied to the abstraction of traces of inter-process communication," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR 11, (Washington, DC, USA), pp. 211–220, IEEE Computer Society, 2011.

[46] P. D. CHATURBHAI, *MINING PATTERNS IN COMPLEX DATA*. PhD thesis, 2011.

[47] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Min. Knowl. Discov.*, vol. 15, pp. 55–86, August 2007.

[48] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD 08, (New York, NY, USA), pp. 147–160, ACM, 2008.

[49] R. Zhang, N. Koudas, B. Chin, and O. D. Srivastava, "Multiple aggregations over data streams," in *In Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 299–310, 2005.

[50] M. Bligh, M. Desnoyers, and R. Schultz, "Linux kernel debugging on google-sized clusters," in *OLS (Ottawa Linux Symposium) 2007*, pp. 29–40, 2007.

[51] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, “Capturing, indexing, clustering, and retrieving system history,” *SIGOPS Oper. Syst. Rev.*, vol. 39, pp. 105–118, oct 2005.

[52] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, “Correlating instrumentation data to system states: a building block for automated diagnosis and control,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design Implementation -Volume 6*, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2004.

[53] N. Ezzati-Jivan and M. R. Dagenais, “A framework to compute statistics of system parameters from very large trace files,” *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 43–54, 2013.

[54] J. Desfossez, “RÃl’solution de problÃÍme par suivi de mÃl’triques dans les systÃÍmes virtualisÃl’s,” Master’s thesis, Ecole Polytechnique de Montreal, 2011.

[55] I. Herman, G. Melancon, and M. Marshall, “Graph visualization and navigation in information visualization: A survey,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 6, pp. 24–43, Jan 2000.

[56] Ericsson, “Open source application for viewing and analyzing traces.”

[57] J. Roberts, “Tracevis: an execution trace visualization tool,” in *In Proc. MoBS 2005*, Citeseer, 2005.

[58] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman, “Execution patterns in object-oriented visualization,” in *In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS âĂŹ98*, pp. 219–234, 1998.

[59] Google, “The trace event profiling tool.”

[60] D. KranzlmÃijller, S. Grabner, and J. Volkert, “Debugging with the {MAD} environment,” *Parallel Computing*, vol. 23, no. 1âĂŞ2, pp. 199 – 217, 1997. Environment and tools for parallel scientific computing.

[61] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data,” in *Runtime Verification*, pp. 219–234, Springer Berlin Heidelberg, 2013.

[62] N. Ezzati-Jivan and M. R. Dagenais, “Cube data model for multilevel statistics computation of live execution traces.” 2013.

[63] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, “Behavior abstraction in malware analysis,” in *Proceedings of the First international conference on Runtime verification*, (Berlin, Heidelberg), pp. 168–182, Springer-Verlag, 2010.

[64] S. Eckmann, G. Vigna, and R. Kemmerer, “Statl: An attack language for state-based intrusion detection,” *Journal of Computer Security*, vol. 10, no. 1/2, pp. 71–104, 2002.

[65] S. Kumar, *Classification and Detection of Computer Intrusions*. PhD thesis, West Lafayette, IN, USA, 1995. UMI Order No. GAX96-01522.

[66] J.-L. Lin, X. Wang, and J. S., “Abstraction-based misuse detection: High-level specifications and adaptable strategies,” in *Computer Security Foundations Workshop, 1998. Proceedings. 11th IEEE*, pp. 190 –201, jun 1998.

[67] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma, “Automated known problem diagnosis with event traces,” *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 375–388, Apr. 2006.

[68] A. S. Shameli-Sendi, J. Desfossez, M. R. Dagenais, and M. Jabbarifar, “A retroactive-burst framework for automated intrusion response system,” *Journal Comp. Netw. and Communic.*, vol. 2013, 2013.

[69] H. K. Padda, *CoMoVA -A comprehension measurement framework for visualization systems*. PhD thesis, 2009.

[70] G. Draper, Y. Livnat, and R. Riesenfeld, “A survey of radial methods for information visualization,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 15, pp. 759–776, Sept 2009.

[71] B. Shneiderman, “Tree visualization with tree-maps: 2-d space-filling approach,” *ACM Trans. Graph.*, vol. 11, pp. 92–99, Jan. 1992.

[72] A. Orso, J. Jones, and M. J. Harrold, “Visualization of program-execution data for deployed software,” in *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis 03, (New York, NY, USA), pp. 67–ff, ACM, 2003.

[73] A. Makanju, S. Brooks, A. Zincir-Heywood, and E. Milios, “Logview: Visualizing event log clusters,” in *Privacy, Security and Trust, 2008. PST 08. Sixth Annual Conference on*, pp. 99 –108, oct. 2008.

[74] J. Lamping, R. Rao, and P. Pirolli, “A focus+context technique based on hyperbolic geometry for visualizing large hierarchies,” in *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI 95, (New York, NY, USA), pp. 401–408, ACM Press/Addison-Wesley Publishing Co., 1995.

[75] F. Mizoguchi, “Anomaly detection using visualization and machine learning,” in *Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, WETICE 00, (Washington, DC, USA), pp. 165–170, IEEE Computer Society, 2000.

[76] A. Cockburn, A. Karlson, and B. B. Bederson, “A review of overview+detail, zooming, and focus+context interfaces,” *ACM Comput. Surv.*, vol. 41, pp. 2:1–2:31, Jan. 2009.

[77] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky, “Guidelines for using multiple views in information visualization,” in *Proceedings of the working conference on Advanced visual interfaces*, AVI 00, (New York, NY, USA), pp. 110–119, ACM, 2000.

[78] C. North and B. Shneiderman, “Snap-together visualization: Can users construct and operate coordinated visualizations,” 2000.

[79] L. Xiaomeng, W. Yanhui, and M. Hao, “Linking multi-scale representations in a navigable database,” in *ISPRS Workshop on Updating Geo-spatial Databases with Imagery The 5th ISPRS Workshop on DMGISs*, 2007.

[80] K.J.Dueker and J.A.Butler, “A framework for gis-t

data sharing," 2000.

[81] C. Collins and S. Carpendale, "Vislink: revealing relationships amongst visualizations," *IEEE Trans Vis Comput Graph*, vol. 2007, 2007.

[82] M. Waldner, W. Puff, A. Lex, M. Streit, and D. Schmalstieg, "Visual links across applications," in *Proceedings of Graphics Interface 2010*, GI 10, (Toronto, Ont., Canada, Canada), pp. 129–136, Canadian Information Processing Society, 2010.

[83] M. Steinberger, M. Waldner, M. Streit, A. Lex, and D. Schmalstieg, "Context-preserving visual links," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 2249–2258, Dec. 2011.

[84] A. Lex, M. Streit, E. Kruijff, and D. Schmalstieg, "Caleydo: Design and evaluation of a visual analysis framework for gene expression data in its biological context," in *Pacific Visualization Symposium (PacificVis), 2010 IEEE*, pp. 57 –64, march 2010.

[85] S. G. Eick and A. F. Karr, "Visual scalability," *Journal of Computational and Graphical Statistics*, vol. 11, no. 1, pp. 22–43, 2002.

[86] P. GmbH, "Vampir 2.0 tutorial," tech. rep., HermÃijlheimer StraÃ§e 10 D-50321 BrÃijhl, Germany, 1998.

[87] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Sci. Program.*, vol. 16, pp. 155–165, Apr. 2008.

[88] Allinea, "Allinea map - c/c++ profiler and fortran profiler for high performance linux code."

[89] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 287–311, May 2006.

[90] I. Zhukov, C. Rössel, M. Geimer, M. Knobloch, B. Mohr, and P. Saviankou, "Scalasca v2: Back to the future," in *Proc. of Tools for High Performance Computing 2014*, pp. 1–24, Springer, 2015.

[91] K. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 20, pp. 2349–2358, Dec 2014.

[92] K. E. Isaacs, A. GimÃľnez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the Art of Performance Visualization," in *EuroVis - STARs* (R. Borgo, R. Maciejewski, and I. Viola, eds.), The Eurographics Association, 2014.

[93] A. Landge, J. Levine, A. Bhatele, K. Isaacs, T. Gamblin, M. Schulz, S. Langer, P.-T. Bremer, and V. Pascucci, "Visualizing network traffic to understand the performance of massively parallel simulations," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, pp. 2467–2476, Dec 2012.

[94] A. Bhatele, T. Gamblin, K. E. Isaacs, B. T. N. Gunney, M. Schulz, P.-T. Bremer, and B. Hamann, "Novel views of performance data to analyze large-scale adaptive applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 31:1–31:11, IEEE Computer Society Press, 2012.

[95] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Revised Lectures on Software Visualization, International Seminar*, (London, UK, UK), pp. 151–162, Springer-Verlag, 2002.

[96] W. De Pauw and S. Heisig, "Visual and algorithmic tooling for system trace analysis: a case study," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 97–102, Mar. 2010.

[97] A. Montplaisi, "Stockage sur disque pour accÃÍs rapide dâĂŹattributs avec intervalles de temps," Master's thesis, Ecole polytechnique de Montreal, 2011.

[98] A. Montplaisir-Goncalves, N. Ezzati-Jivan, F. Wininger, and M. Dagenais, "State history tree: an incremental disk-based data structure for very large interval data," in *Social Computing (SocialCom), 2013 International Conference on*, pp. 716–724, IEEE, 2013.

[99] H.-P. Kriegel, M. Potke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB 00, (San Francisco, CA, USA), pp. 407–418, Morgan Kaufmann Publishers Inc., 2000.

[100] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, pp. 47–57, June 1984.

[101] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," pp. 507–518, 1987.

[102] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, pp. 322–331, May 1990.

[103] M. Renz, *Enhanced Query Processing on Complex Spatial and Temporal Data*. PhD thesis, 2006.

[104] V. Gaede and O. Gunther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, pp. 170–231, June 1998.

[105] R. Blanch and E. Lecolinet, "Browsing zoomable treemaps: Structure-aware multi-scale navigation techniques," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, pp. 1248–1253, Nov 2007.

[106] G. Pestana, M. da Silva, and Y. Bedard, "Spatial olap modeling: an overview base on spatial objects changing over time," in *Computational Cybernetics, 2005. ICCC 2005. IEEE 3rd International Conference on*, pp. 149 –154, april 2005.

[107] P. M. and Y. BÃľdard, "Fundamental characteristics of spatial olap technologies as selection criteria," in *Location Intelligence 2008*, april 2008.

[108] E. Malinowski and E. Zimanyi, "Logical representation of a conceptual model for spatial data warehouses," *GeoInformatica*, vol. 11, no. 4, pp. 431–457, 2007.