AN EFFICIENT ANALYSIS APPROACH FOR MULTI-CORE SYSTEM TRACING DATA

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

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

ABSTRACT
Trace data usually contains information about the underlying system execution such as running processes, memory usage, disk and file accesses and other runtime information. However, this raw data is not what the system administrators are looking for. For instance, in a multi-core system, the system administrator may need to know what is the utilization of each core or even what are the performance bottlenecks of the system. This analytical information is not available directly from the trace data and, in fact, is hidden behind the mountains of trace raw data. The trace data needs to be analyzed to extract the valuable information. Thus, efficient trace analysis tools and techniques need to be developed to handle large trace data and extract and provide useful and analytical information. In this paper, we propose a stateful trace analysis and abstraction approach that shares the computation and storage of the common information between parallel abstraction processes. This technique leads to relatively simple patterns compared to other pattern-based techniques. Furthermore, since it computes and stores the common state once, shared between related processes, it has a better computation and storage efficiency than stateless methods. The architecture of the method, its applications, and evaluation are detailed in this paper.

KEY WORDS
kernel tracing, trace analysis, data abstraction, pattern matching, stateful approach

1 Introduction

Execution traces are increasingly used to analyze system runtime behavior among administrators and analysts. Trace files, especially the kernel traces, usually contain valuable information of the system execution such as running processes, file and disk operations, memory management, etc. This information can be used to reason about the system execution at runtime and can also be used to identify the system bugs and problems. However, the tracing data may contain a large number of events, even for a short tracing period, which makes analysis difficult. Moreover, this data is full of the low level system calls that complicate the reading and understanding.

Normally, it is appropriate for analysts to look at relatively abstract and high level events, which are more readable than the original data and reveal the same behavior but at a higher level of the system execution. To generate such abstract events efficiently, it is required to develop effective algorithms and tools that read trace events, detect similar and related sections, and generate meaningful synthetic and high level events.

Several approaches have been reported for generating abstract events and reducing the trace size [1, 2, 3, 4, 5]. These approaches usually use pattern-based techniques such as pattern matching and pattern mining. Although the pattern-based techniques are useful methods to extract meaningful information from massive traces, they have some weaknesses. The first problem is that, in the studied tools, the patterns are defined just over the trace events. However, there are still other types of faults and synthetic events that are difficult to detect with this technique, and need more information about the system resources. In other words, there is valuable information hidden behind the system state that can be gathered directly from the execution traces and can be used, in addition to the events, as inputs for the pattern-based techniques.

Moreover, in the most pattern-based trace abstraction approaches, patterns are analyzed separately without sharing common information (e.g. common system state). Since their patterns mostly examine events belonging to a small set of the system processes and resources, it should be possible to share internal states between different but related patterns. For example, by storing the open files in a common state store, all processes and patterns could share this information. However, in existing tools, each pattern tries to describe separately such thing as an opened file.

The gathered state information can be modeled uniformly in such a way that it can be shared between all the concurrent patterns. Without that, it has to be recreated and recomputed for each pattern that requires this information. For example, the state of a file (even is opened, read, written or closed by a process) can be stored once and used by any pattern that tries to work with that file. We named this method "stateful trace abstraction". Stateful trace abstraction method uses a state database to store the intermediate information, and state of the pattern matching engine. In other words, this database contains all information required by the analyzer to inspect efficiently the input events, group them and generate high level meaningful information.

The proposed stateful approach has great applications like supporting partial trace abstraction, simplifying the patterns, reducing the storage space required to store and manage the patterns, increasing the overall computation efficiency, and finally reducing the complexity of the trace analyzer.

For evaluating the proposed method, trace generated by Linux Trace Toolkit next generation (LTTng) [6] is used. LTTng is a powerful, low impact and lightweight [7] open source Linux tracing tool, that provides a detailed execution trace of kernel operations and user space applications. This trace contains data about running processes, CPU scheduling, memory usage, file, disk operations, and network packets [8].

The rest of the paper is organized as follows: after discussing the related work, we explain the challenges of a scalable, efficient and flexible trace abstraction technique. Then, we present the architecture of the proposed framework. Eventually, we will present experiences and evaluation of the system, and some outlook of related future work.

2 Related Work

Using kernel trace data to analyze and monitor system execution is explained in [8, 9]. However, because of the trace files size it is usually difficult to analyse the system runtime behavior using execution trace data. Trace abstraction techniques like filtering, sampling, compression, generalization and aggregation have been developed to remove the noise, highlight the important elements, and generate high level events, and, consequently, reduce the size of trace files [1, 2, 3, 5].

Trace filtering and sampling remove redundant and unnecessary information from the trace files, and highlight events with some specified attributes. They can be based on event timestamp, event type, event size, method name, process name and also the priority or importance of an event [10, 11].

Trace compression has two common forms: lossy and lossless. A trace is compressed by finding similarities and removing redundancies [12].

Generalization is the process of extracting common features from two or more events, and combining them into a generalized event [4]. We use generalization technique to convert the operating system related events to more canonical events. For example, the read, readv and pread64 system calls generate events that can be generalized to a regular read operation.

Trace aggregation or grouping, aggregates sets of low level events to a group of compound events using pattern matching, pattern mining, pattern recognition and other techniques. In these techniques, related events, participating in an operation, are grouped together to produce larger events [13]. Trace aggregation is a broadly used method for aggregating the trace events [1, 13, 14]. It combines groups of similar events into compound events and, by eliminating the unnecessary events, reduces the complexity of the input trace [14].

Fadel et al. [1] and Wally et al. [2] use pattern matching approach to abstract out the raw kernel events gathered by LTTng Linux kernel tracer to generate higher level events. The focus of the former is on generating a pattern library; however, the latter is focused on creating a language for defining the abstract event patterns and scenarios. Matni et al. [3] used an automata-based approach over kernel traces to detect problems like "escaping a chroot jail" and "SYN flood attacks".

They use state machine language to describe the attack patterns. Although they present useful examples and applications of the trace abstraction in the system fault detection field, their solution lacks the scalability to meet the demands of sizeable trace sizes, and their work should be optimized to support large traces. Moreover, since they use disjoint patterns for generating the high level entities, for large traces and for a large number of patterns, it will be time-consuming to inspect all patterns separately.

Efficient evaluation of patterns has been studied in several experiments [15, 16]. Productive evaluation of the specified patterns is closely related to multiple-query optimization in database systems [17, 16] 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. [15]. They proposed an automaton model titled "NFA^b" for processing the event streams. They use the concept of sharing execution states and storage spaces among all the possible concurrent matching processes to gain efficiency. However, applying this idea for kernel trace data needs more investigations. We show how this approach can be used to share storage and computation between different trace abstraction processes for traces that are generated from the Linux kernel execution.

In the following chapter, we review some of these challenges and limitations in detail.

3 Challenges of the kernel based trace abstraction techniques

Kernel based trace abstraction techniques usually face some important challenges:

- representation of the patterns in a rich language,
- efficient evaluation of the patterns,
- discontinuities of the execution (because of the cpu scheduling events),
- generality, scalability, and flexibility of the method.

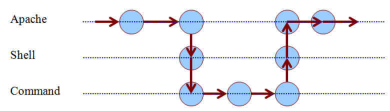


Figure 1. Extracting execution path from kernel traces.

There are different languages for representing the fault scenarios and patterns in IDS (Intrusion Detection Systems) systems, that can also be used in tracing tools as well (e.g. automata-based languages, Imperative languages) [18]. Regardless of the language, it is important to provide a tool that enables the analysts and pattern writers to design the patterns and scenarios in a simple way, and with less complexity. We will show that one of the results of the proposed stateful approach is helping users to design simpler patterns.

The second challenge is the efficiency of the method. Most of the aforementioned methods, normally use disjoint patterns for the trace aggregation, and evaluate and inspect each pattern separately, although, they typically use the same system resources. Sharing joint information and states leads to simple and more efficient patterns. We use a common state model to manage the shared states and information. Without an appropriate state model, many patterns will simply attempt to recreate and recompute those shared states and information, leading to degradation of the overall method performance. The challenge here is making the abstraction method efficient enough, and sharing as much as possible the data and computations between the different pattern matching processes. This challenge can be addressed by using the stateful approach. The primarily result of using a stateful trace abstraction approach is increasing the speed of the pattern detection process. In this case, the shared information will be extracted and calculated once. Effective sharing of common states between various matching processes allows storage and processing costs to be reduced from the total number of patterns to the distinct number of them. Comparison of the stateful and stateless approach and other applications of the proposed method will be shown in Applications and Experiments section.

Another challenge, the discontinuous nature of the Linux kernel execution, necessitates extracting the real execution path from several execution chunks, for any high level kernel trace analysis. Since the scheduling events split an execution of a process in different executions chunks, the execution path of a process in kernel is different than its execution at user level. Figure 1 shows different execution paths for a user level operation, and the real extracted path for the same operation from kernel traces.

The other challenge is making the solution as generic as possible. By designing a tool, independent of the kernel version and trace format, it would be possible to achieve a generic abstraction tool that is not dependent on a specific version. Similarly, the approach must be scalable and

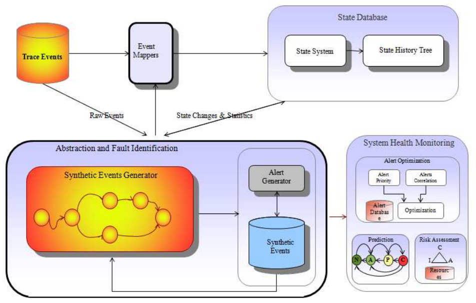


Figure 2. Architectural view of the stateful trace abstractor.

support different trace sizes. Designing a special external memory based data structure allows us to support arbitrary trace data size. Furthermore, it should be possible to generate arbitrary synthetic events, either for reducing the trace size or detecting the system problems and attacks. We show that using different kinds of patterns and following real execution paths for the different process executions enables us to generate various types of synthetic events.

4 Architecture

Figure 2 depicts the architecture of the proposed trace abstractor. In the following, we explain its different modules.

Event Mapper

In the Linux kernel, there is often no single way to implement some user space operations. Thus, by categorizing the events for similar and overlapping functionalities, we achieve a new set of semantic events that hide the kernel concepts for these operations [4]. For example, both the sys_open and sys_dup Linux system calls are used to open a file. We define a new semantic "Open File" event instead of the events for those two system calls. The reason we use the semantic event notion is that it makes the trace abstractor method generic and independent of a specific kernel version and trace format. The event mapper is used to convert the input raw events to environment-independent semantic events. The relationship between raw and semantic events is actually n:m, which means that a raw event or a group can be converted to one or more semantic events. For example, a Linux_sched_schedule(p1, p2) event can be mapped into two, process_stop(p1) and process_running(p2), semantic events.

State Database

As discussed earlier, it is essential to use the system state values, in addition to the events, to generate complex abstract events. Although state values are extracted from trace events, storing and using them can help for detecting complex patterns in an efficient way. For that, a state database, named "modeled state" is used to store the state values. This database contains the state values of the resources that are required by the synthetic events patterns, such as execution status of a process (running, blocked, waiting), mode of a CPU, various state values of file descriptors, disks, memory, locks, etc., for different areas of the trace. In this method, we use the modeled state to store both the system state, as well as the state machines intermediate states. Based on this idea, common methods are used for storing, retrieving, and exploring both state types.

Synthetic Event Generator

Synthetic event generator is the core of the system. It accepts the pattern library, modeled state, and also the trace events as inputs, parses them, looks for patterns in the trace events, and finally generates a hierarchy of abstract events. In this project, State Machine Language is selected to describe the patterns, because of its expressiveness and simplicity. In addition, it is visually friendly and domain-independent [19]. It has also been used to define several attack scenarios in STATL [19] that can be converted to our format, and can be used in our project. The synthetic event generator uses patterns of semantic events and system state values to generate synthetic events. Previously generated synthetic events may be passed again through the synthetic event generator, which then can be used to generate complex higher level events. For example, one can generate "A file download" synthetic event from a sequence of several "reading from a HTTP connection" and "writing to a file" synthetic events.

The output of this module is a set of high level synthetic events that can be displayed in the user interface. They also may be sent again to the analyzer to participate in the other phases of the trace abstraction. The defined patterns may generate alerts representing the existence of a potential misbehavior or an attack on the system. These alerts are subsequently sent to the system administrator or any predefined monitoring system.

The proposed module generates abstract events at three levels:

- kernel level abstract events: these events are generated using pattern matching over lower level information. Examples of these events are: "sequential file read", "process fork", "port scan", "file download", "HTTP connection", etc

- System problems and faults: this level includes the events resulting from applying the fault identification patterns. Examples of these events are: "hogging CPU process", "inappropriate use of an application cache", "denial of service attack attempt", "access to a restricted file", etc.

- System execution statistics: this level contains statistics of different system metrics. Examples are: "number of different event types", "IO throughput", "CPU usage", etc.

Pattern Library

We developed a prototype pattern library that covers most of the ordinary user operations and some system problems patterns. By using this pattern library, we could extract various levels of file operations (sequential and non sequential file read and write, etc.), network connections (e.g. TCP connection, port scanning, etc.), and Process management (process fork, process termination, etc.) from kernel execution trace events. It is also possible to detect a number of system problems.

The proposed pattern library contains two general types of patterns:

- Trace size reduction patterns: this type contains patterns for reducing the size of the original trace. Linux kernel uses several system calls to accomplish user space operations. However, from users' point of view, some of them look similar. The synthetic event generator looks for patterns of the various system calls, classifies and replaces them by high level entities. Also, some filtering patterns are defined in the pattern library. They are used to filter events that are often not of interest and show page faults, kernel timers, etc. Since kernel traces are full of low level information, it is better to reduce the size using these types of patterns before applying other types of fault identification patterns.

- Fault identification patterns: this type contains patterns for detecting system problems and faults. It uses both the state values and events to detect system misbehavior. For example, a set of patterns is defined for computing the system resources overloads. By keeping track of the system load and usage (e.g. CPU usage, I/O throughputs, Memory usage, etc.) and aggregating them per process, per user, per machine, and also in different time intervals, it becomes possible to check the resource loads against predefined threshold values. The library also contains patterns for detecting denial of service attacks, for instance SYN flood attack, and process fork bomb attack. Further details about the prototyped pattern library, and also the size reduction results of applying these patterns, may be found in [5].

5 Applications and experiments

We have implemented a Linux Java tool that read LTTng trace files and applies the proposed techniques to create high level synthetic events. It has been used to test the performance of the proposed model with real trace data. For this purpose, Linux Kernel version 2.6.38.6 is instrumented using LTTng and the tests are performed on a 2.8 GHz system with 6 GB RAM on kernel traces of different sizes. To generate these trace files, some patterns like "fork bomb detection", "port scan detection", "file access pattern" (e.g. to check whether a file is accessed sequentially or not"), "file IO overuse", "HTTP connection", "UDP connection", "file download", etc. are used. Also, "grep -r", "wget -r", "ls -R","nmap" and another developed "recursive process forker" commands and tools are used to generate the required workload.

In the following, we discuss important features and experimental results of the proposed approach.

Computational efficiency

As explained earlier, most of the studied approaches consider patterns independently. However, they may use the same base information. Also, the shared states have to be gathered and stored separately, leading to wasted space and long computation time.

By sharing the common states of the patterns, the method computes and generates the shared states once rather than for each pattern separately, leading to computational efficiency in the abstraction technique. In other words, since the patterns are shared among different processes, and analyzed together, they may share some computations like redundant condition checking. Suppose that there exists k coexisting patterns, each requires n computations, and the time required to perform each computation is t. In this situation, Formula 1 and 2 show the evaluation time for each pattern and for all patterns respectively in the stateless mode.

$$EvalTime(p_i) = n * t; \tag{1}$$

$$EvalTime\_stateless(All) = \sum_{i=0}^{i=k} EvalTime(p_i) = k * n * t; \tag{2}$$

Now, suppose that the patterns share m computations. In this case, the evaluation time for each pattern can be obtained by Formula 3 given below:

$$\begin{gathered} EvalTime(\text{shared part}) = m * t \\ EvalTime(\text{non-shared part of } p_i) = (n - m) * t \end{gathered} \tag{3}$$

Similarly, the evaluation time for all patterns can be calculated by Formula 4:

Using the above formulas, computational efficiency of the stateful method can be easily calculated using Formula 5:

Regarding Formula 5, for 20 concurrent patterns (k = 20) where each contains 10 states, and computations of four states are shared between all patterns, we gain 38 % in computations time.

Please note that the above explained formulas are simply here to validate the experminetal results that are shown in the Figures 3, 4, and 7 and will be explained in the following paragraphs.

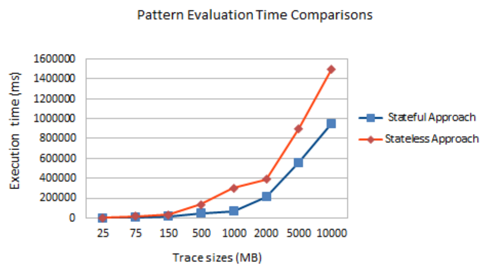


Figure 3. Execution (pattern evaluation) time comparisons

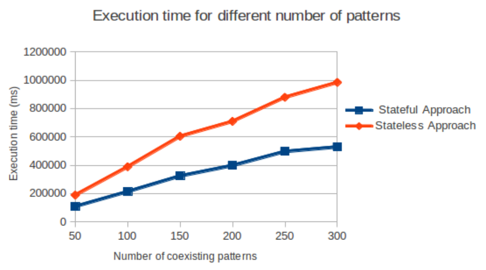


Figure 4. Execution (pattern evaluation) time comparisons for different number of coexisting patterns

Figure 3 compares the execution time of both the shared state and separate state approaches for the different kernel trace sizes.

The figure shows that the stateful approach is faster than the stateless approach for different trace sizes. However, other factors may affect the results, such as the complexity of patterns, and the number of coexisting patterns. In this experiment, since we aim to compare the execution time of two stateful and stateless approaches, we use the same set of patterns for both approaches.

In another experiment, we fixed the size of the trace to 2000 MB and tested the proposed method for various numbers of coexisting patterns from 50 to 300. The results are shown in Figure 4.

As shown in Figure 4, the method works better for the larger number of coexisting patterns. This is explained by the fact that with a large number of parallel patterns, more common state will exist and be shared, leading to increase the differences between the evaluation time required for evaluating the patterns in the both solutions.

Storage efficiency

The stateful abstraction method uses a shared memory to store the common intermediate states and information, as shown in Figure 5. With this technique, since several patterns do not need to store separately the shared states, less storage is required. Figure 7 depicts a comparison of mem-

$$
\begin{aligned}
EvalTime\_stateful(All) &= EvalTime(\text{shared part}) + \sum_{i=0}^{i=k} EvalTime(\text{non-shared part of } p_i) \\
&= (m * t) + \sum_{i=0}^{i=k} (n - m) * t; \\
&= [m + (n - m) * k] * t;
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
\text{Computational Efficiency} &= \frac{|EvalTime\_stateful(All) - EvalTime\_stateless(All)|}{EvalTime\_stateless(All)} \\
&= \frac{|EvalTime\_stateful(All) - EvalTime\_stateless(All)|}{EvalTime\_stateless(All)}; \\
&= \frac{|(m + (n - m) * k) * t - (k * n * t)|}{k * n * t}; \\
&= \frac{m * (k - 1)}{n * k};
\end{aligned} \tag{5}
$$

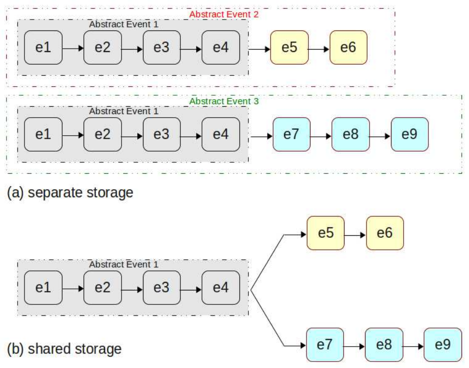


Figure 5. Shared storage for common data.

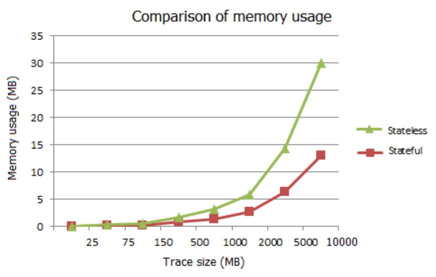


Figure 6. Memory usage comparisons

ory usage for both the stateful and stateless approaches.

As shown in Figure 7, the stateful approach uses less memory to store the intermediate states. Indeed, in the stateless case, all the intermediate states are stored for each pattern separately, leading to more memory usage.

Patterns complexity

The proposed abstraction method helps users to write simpler patterns than with the other pattern based systems, for mainly two reasons:

1. Semantic events allow pattern writers to develop generic scenarios, i.e. patterns that are not specific to a particular version of operating system or tracing tool.

2. Furthermore, by using high level abstract events and a modeled state system, the patterns become simpler. Indeed, by sharing the common states and information between patterns, the number of conditions is decreased and several calculations and condition checking are replaced by a simple query to the shared state. In other words, for the states that are shared between a group of patterns, it suffices to refer them directly instead of defining how to compute them separately.

As an example for the latter case, suppose that we have a few patterns that require the list of open files to look

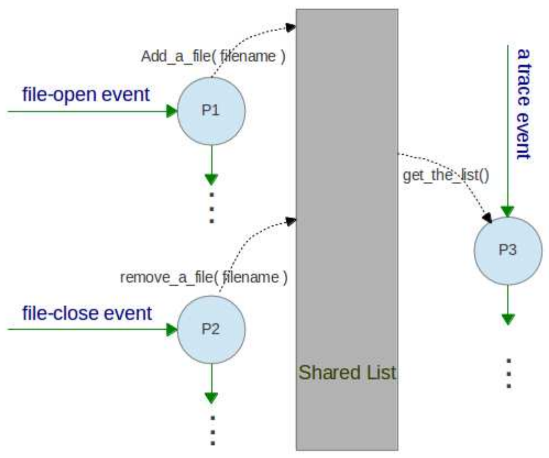
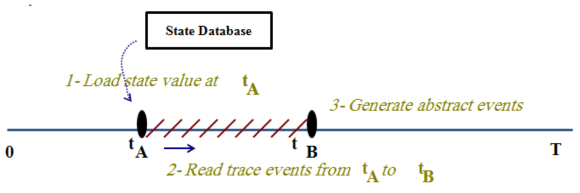


Figure 7. An example of sharing resources between patterns

for special behavior of IO operations in the kernel trace files (e.g. looking for IO overuses, or accesses to specific files like /etc/passwd). In this example, each pattern can calculate and keep track of a list of open files for itself. However, using a shared open files list, makes it possible for the patterns just to update and use the shared list, instead of handling it separately. Sharing of the common parts decreases the required calculations and therefore, the count of patterns states, leading to have simpler patterns. In this case, all redundant calculation parts of a pattern are replaced with a simple referring to the shared database. Figure 7 depicts the graphical view of this example.

Extracting execution path

Taking into account the execution path is essential for any kernel based fault and attack detection. The CPU scheduling operations split the execution of a process in different execution chunks. For instance, suppose that you want to detect and report any remote shells (as the purpose of many buffer overflow attacks) executed by the Apache web server. From kernel traces, it is not obvious to discover which process is the "real" owner of the detected remote shell. In the Linux kernel, when Apache (like any other process) is forced to execute a shell, it firstly creates a new process (possibly with a different name and PID) and then executes a shell program within that process. Since many pattern matching techniques use separate patterns matching phases for separate processes [1, 2, 20], it will not be possible to discover the main owner of the shell execution. Instead, it will just detect a shell executed by the new process and mark the new process as the owner. However, the main owner is the Apache process. Therefore, it is important for the detector to keep track of the execution path of a process, in an efficient index, to extract on demand and perform correct pattern matching. Figure 1 shows a graphical



State Database
1- Load state value at t_A
3- Generate abstract events
0
t_A
t_B
T
2- Read trace events from t_A to t_B

Figure 8. Partial trace abstraction in the stateful approach.

view of this example. This feature is one of the important results of the proposed method and can be used to detect complex attack types and system problems.

Partial trace abstraction

The other application of the proposed stateful method is being able to perform partial trace abstraction. The method supports partial trace abstraction, which makes users able to seek, go back and forth in the trace, select an area and abstract out the trace events lying within the selected time area. For instance, suppose we see there is a high load at a specific time, we can jump to that area in the trace, load the system state information and run the trace abstraction algorithm for that given area to get meaningful and high level understanding of the system execution. As shown in Figure 8, for any given area, it suffices to load the state values of the system at the starting point of the interval, re-read and re-run the trace events until the end point, and regenerate the abstract events without reading the previous trace events.

6 Conclusion and future work

Execution traces can be used to analyze the system runtime behavior and detect its bugs and problems. However, the size of the trace may grow rapidly and complicate the analysis. Trace abstraction technique is used to reduce the trace size, generate high level meaningful events, and detect system problems and misbehavior. As discussed in the literature review, most of the trace abstraction approaches, based on pattern matching and pattern recognition techniques, consider and check the patterns separately. However, many patterns use the same base information, and it is possible to compute and extract the common states only once, and share and reuse these repeatedly.

We proposed an architecture for a stateful trace abstractor, which uses a state database to manage and store the common data, and uses it in generating the synthetic events. Sharing information between different patterns reduces the computation effort required to compute the intermediate states. It also decreases memory usage required to process patterns. The memory usage and evaluation time reduction will be significant in a large trace with numerous coexisting processes and patterns. The performance results of the proposed technique were presented, and compared to the developed stateless approach.

Possible future work is to extend the detector engine to use other methods like pattern mining to detect complex system problems. Extending the pattern library for detecting more system and network faults, and comparing the solution with common IDS systems will be interesting to investigate in the future.

Acknowledgment

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.

References

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

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

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

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

[5] N. Ezzati-Jivan and M. R. Dagenais, "A stateful approach to generate synthetic events from kernel traces," *Advances in Software Engineering*, April 2012.

[6] 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.

[7] N. Sivakumar and S. S. Rajan, "Effectiveness of tracing in a multicore environment," Master's thesis, Malardalen University, 2010.

[8] F. Giraldeau, J. Desfossez, D. Goulet, M. R. Dagenais, and M. Desnoyers, "Recovering system metrics from kernel trace," in *OLS (Ottawa Linux Symposium) 2011*, pp. 109–116, June 2011.

[9] K. Yaghmour and M. R. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2000.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," *SIGMOD Rec.*, vol. 29, pp. 249–260, May 2000.

[18] M. Couture, R. Charpentier, M. R. Dagenais, P.-M. Fournier, G. Matni, and D. Toupin, "Monitoring and tracing of critical software systems - state of the work and project definition," tech. rep., Defence Research and Development Canada, dec 2008.

[19] 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.

[20] 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.