# Enhanced execution trace abstraction approach using social network analysis methods

Ji Wang
jw17tl@brocku.ca
Brock University, St. Catharines, Canada

Naser Ezzati-Jivan
nezzati@brocku.ca
Brock University, St. Catharines, Canada

## Abstract

In this paper, we propose an improvement in system execution tracing by applying social network analysis techniques on the trace data. We perform a 3-step analysis: collection of trace data on operating system kernel; community analysis on the data; and PageRank algorithm within each community. The proposed analysis focused on the following problems: useless information contained in the data and the enormous size of the data. We propose two use cases: one on kernel trace filtering and the other on virtual machine clustering. Our evaluation shows that the proposed method provided a concise and more comprehensive view of the trace data. This can help shorten the time and assist in building infrastructural functions in analyzing system execution.

## 1 Introduction

Tracing has been a revolutionary way to analyze and debug operating systems. It provides complete and exhaustive information about system executions. Most of the time, the underlying challenges in tracing are the huge size of data and lacking of functions in analyzing tools. There have already been several different methods to address these challenges: noise filtering, data abstraction, profiling and parameter based abstraction.

This paper focuses on the trace abstraction method, in which we employ social network analysis technique to highlight most important system entities. We mimic the algorithms and analysis in this technique by treating entities in tracing data as people and communities in social media data: leave out unimportant entities and highlight the significant ones.

To start, we collect tracing data from a certain operating system. Then the collected data is used to generate a view of the system executions. At the end, we apply social media network analysis on the data and modify the view accordingly.

Contributions of this work are: adopting and applying social network analysis on system kernel tracing data and showing that social network analysis algorithms are useful in system analysis.

## 2 Methodology

The proposed method is comprised of several steps: data collection and pre-processing, social network construction, community analysis, pageranking and finally visualization. Architecture of the proposed method is shown in Figure 1.

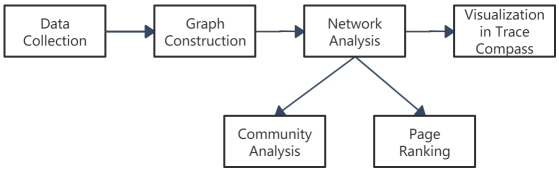



Figure 1: Architecture of the proposed method

### 2.1 Data collection and pre-processing

We firstly obtained trace data from a Linux operating system. To achieve this, we use an open-source tool named LTTng[2]. This tool can recognize and extract traces of various types: kernel space and user space applications in different programming languages.

Secondly, we import the trace data into another open-source tool called Trace Compass[5] which is an open source application to analyze tracing data. The trace extracted on kernel level contains enormous details of system execution. In preparation for data modelling and analysis, we specify the events of interactions, which we collect from trace data.

### 2.2 Constructing thread interaction's graph

We define a social network graph G(V,E) in system analysis context as follows: a graph G is comprised of a set of nodes V and a set of edges E. In this context, a node represents a thread existing in the trace data. An edge represents an interaction between two threads. The metrics are shown in Table 1.

Our algorithm iterates over all trace events and processes the data correspondingly to their types. Each sched_switch event adds an edge $Vprev\_tid \rightarrow Vnext\_tid$ to the graph.

In the context of this particular analysis, G is a directed and weighted graph consisting of threads as nodes and interactions as edges. We construct an adjacency matrix M of G as follows: the matrix's size is

Table 1: The parameters used in the graph structure

| Metric | Description |
|---|---|
| $V_i$ | A thread |
| $V_i \rightarrow V_j$ | Edge of interaction from $V_i$ to $V_j$ |
| $w_{i_j} (i \neq j)$ | The weight of edge $V_i \rightarrow V_j$ |

$|V| \times |V|$. Each entry $m_{V_1,V_2}$ marks the weight of the edge from $V_1$ to $V_2$.

### 2.3 Graph community analysis

Once the weighted graph is created, a graph community analysis algorithm is applied to partition nodes into distinct modules. Here, we use the Louvain algorithm [3] which has been widely used in many applications to identify the community structures in graphs. This algorithm has the advantages of rapid convergence, high modularity, and hierarchical partitioning.

The Louvain method consists of two phases. In the first phase, the algorithm sequentially sweeps over all vertices and moves them to one of their neighbors based on the gain in the modularity cost function. This iterative procedure continues until no movement yields a gain.

In the second phase, these communities become supervertices by aggregating each community into a single node. Two supervertices are connected if there is at least one edge between nodes of the corresponding communities, in which case the weight of the edge between the two supervertices is the sum of the weights from all edges between their corresponding partitions at the lower level.

These two phases are then recursively applied to the supergraphs in a hierarchical way. The algorithm terminates until communities become stable. Typically, the Louvain algorithm converges very quickly and can identify communities in a few iterations.

### 2.4 Ranking processes

In this step, each community is considered as a separate graph and processes within each partition are ranked. The PageRank (PR) algorithm [1] is employed to assign a rank to each process. The idea is to rank the processes that are more significant based on interactions with other processes.

Let $p$ and $v$ be two processes in community $c$ and $E_c$ the set of directed edges in $c$. The PageRank of $p$, $PR_j(p)$, is defined using the following recursive formula:

$$PR_t(p) = \alpha \sum_{(p,v) \in E_c} \frac{PR_{t-1}(v)}{out - degree(v)} \qquad (1)$$

where $\alpha$ is a normalization factor for the total rank of all processes and $t$ shows the iteration number. The PageRank values are initialized as $\frac{1}{n}$ for each process, where $n$ is the number of nodes in each community. The PageRank values get updated in each iteration, until it converges into a stable value.

## 3 Use case

In an operating system such as Linux, the kernel is responsible for accessing and managing resources. Due to the fact that Trace Compass [5], the tracing tool we use, collects everything at the kernel level, most of the trace data will not have critical meaning at application level. However, there is no simple way in Trace Compass to filter out irrelevant data and to provide an updated view of that.

In this work, we provide an implemented method to better facilitate Trace Compass's functions and integrate everything using Trace Compass's EASE scripting engine[5]. The work will cover data collection, data processing and enhanced filtering. After the work is done, users are provided an update view of critical threads.

### 3.1 Trace data collection

The interactions between threads are traced using the Linux Trace Toolkit Next Generation (LTTng) [2]. The tool is able to trace the kernel and applications and we want the entire collection of executions done at the kernel level.

### 3.2 Data processing using script

Trace Compass will provide default views such as control flow and resources to us. We then create a new JavaScript file within this workspace and name it script.js. In the script, we firstly extract events using Trace Compass's analysis module. From the event data we can search for certain event type and extract the TID(thread ID), CPU(CPU number) and PID(process ID) for further analysis. In order to perform community analysis, we make a directed and weighted graph out of the data by setting each thread as a node and each interaction as an edge. For the ease of data abstraction, we created an adjacency matrix in the script. Using the analysis module, we store the attributes in the matrix. Now the data is ready for more advanced analysis.

### 3.3 Enhanced filtering and view updating

After the data is extracted and stored, we employ the Louvain algorithm [3] to perform a first phase of community detection. The initial large community is partitioned into smaller but more related communities. Within each small community, threads interact more often between each other.

Then, we employ the PageRank algorithm[1] to identify the most active thread(s) within each community. Using the adjacency matrix we created, we calculated the outdegree and indegree of each thread and apply Equation 1. We modify the algorithm to make it iterate once due to the small scale of the partitioned communities. In the script, the PageRank

Table 2: The metrics of the graph structure

| Metric | Description | Value |
|---|---|---|
| $\|V\|$ | Number of threads | 509 |
| $\|E\|$ | Number of interactions | 6015 |
| $W$ | Total weight of edges | 58505 |

values $PR_t(p)$ are stored in the diagonal entries $V_{i_i}$ of M. Based on the values, we sorted the threads and picked the ones with the largest value.

After that, we make use of the global filter function in Trace Compass to update the view. The function will prompt for limiting criteria such as thread ID or running time and highlight anything that meets the criteria. Different views are synced to provide better correspondence. Figures 2 and 3 shows the initial and updated view in Trace Compass.

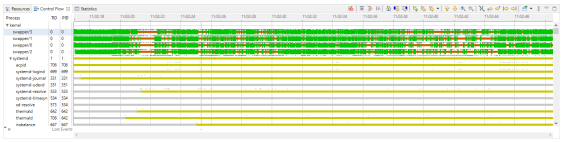


Figure 2: Initial Views in Trace Compass

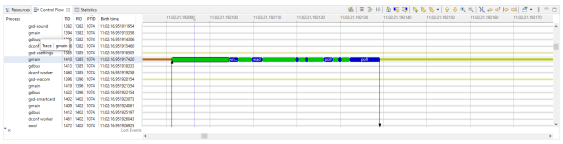


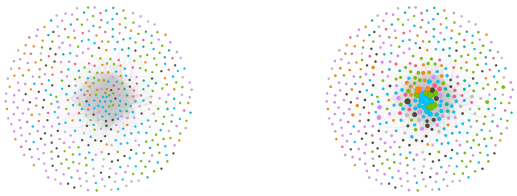
Figure 3: Highlighted Views in Trace Compass

## 4 Evaluation

The tracing was done using LTTng 2.10[2] on a Ubuntu 18.04 system with a Quad-core CPU and 16 GB RAM. Our analysis shows that enabling each event imposed 14 ns in the execution time of the program. And the overall overhead is based on how many times this event appears in the trace. In this use case, our analysis shows a 5.3% slowdown in the execution of the program under analysis. This is because the sched_switch event is a frequent event which occurs 213385 times over a time period of 26 seconds in this use case.

The second part is to evaluate the trace analysis part. Firstly, the network modelling is implemented in the EASE script. We extracted 509 threads and 6015 distinct interactions with a total weight of 58505 as listed in Table 2. This is aligned with our assumption before the implementation that the interactions between threads would be repetitive and recurrent. Then we fed the metrics to Gephi[4] for visualization. The trace size in this use case is 316 MB and it took 1599 milliseconds to extract the metrics.

Then we performed the Louvain algorithm[3] on the metrics. Figure 4(a) shows the result of the modularity test, an implementation of Louvain algorithm in Gephi. The partitioning appears to be very clear and unambiguous.



(a) Modularity Algorithm (b) PageRank Algorithm

Figure 4: Results from Gephi

Lastly, we performed PageRank algorithm[1] using Gephi's PageRank test. Figure 4(b) shows the outcome. We expected the outcome of the threads within each community being ranked. The threads with the greatest ranking values are the most significant ones we want to highlight in Trace Compass view. From this figure, we can see that the algorithm ranks the threads efficiently.

## 5 Conclusion

The presented work was a successful application of Social Network Analysis on system tracing data. The results provide a more effective and efficient approach to monitor and debug operating systems. In general, the work validates the feasibility of applying community analysis and PageRank algorithm on data-sets other than social media data. Furthermore, the presented work enlightens the possibility of combined use of existing techniques in various field. With the help of emerging techniques such as Artificial Intelligence and Machine Learning, more revolutionary applications can be brought to network analysis.

## References


[1] W. Xing and A. Ghorbani. "Weighted PageRank algorithm". In: *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.* May 2004, pp. 305–314.

[2] 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.* 2006, pp. 209–224.

[3] V. D. Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008.

[4] *Gephi: The Open Graph Viz Platform.* `https://gephi.org/`. Accessed: 2020-07-26.

[5] *TraceCompass.* `https://projects.eclipse.org/projects/tools.tracecompass`. Accessed: 2020-07-26.