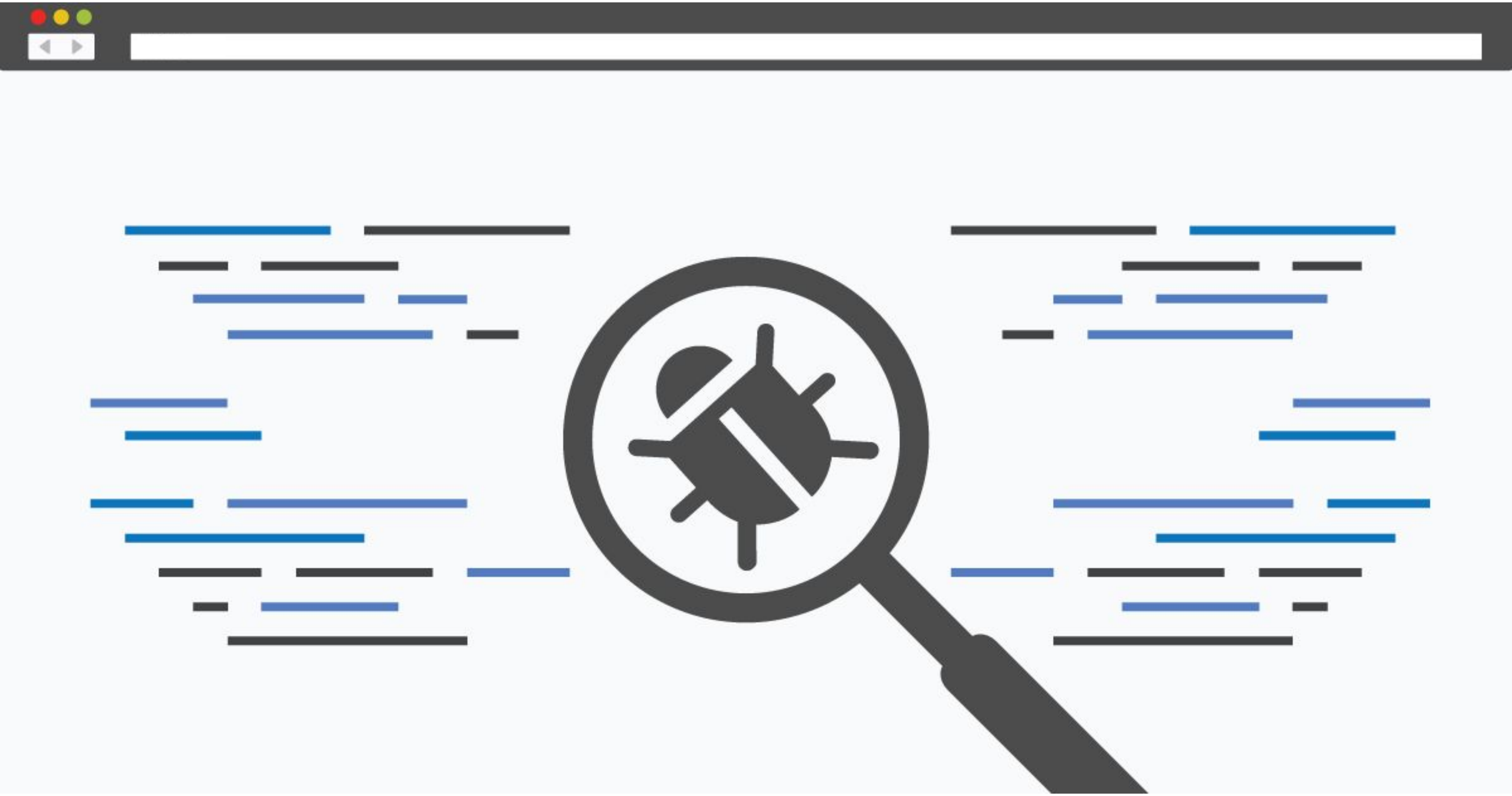
Multilevel Analysis of Software Systems

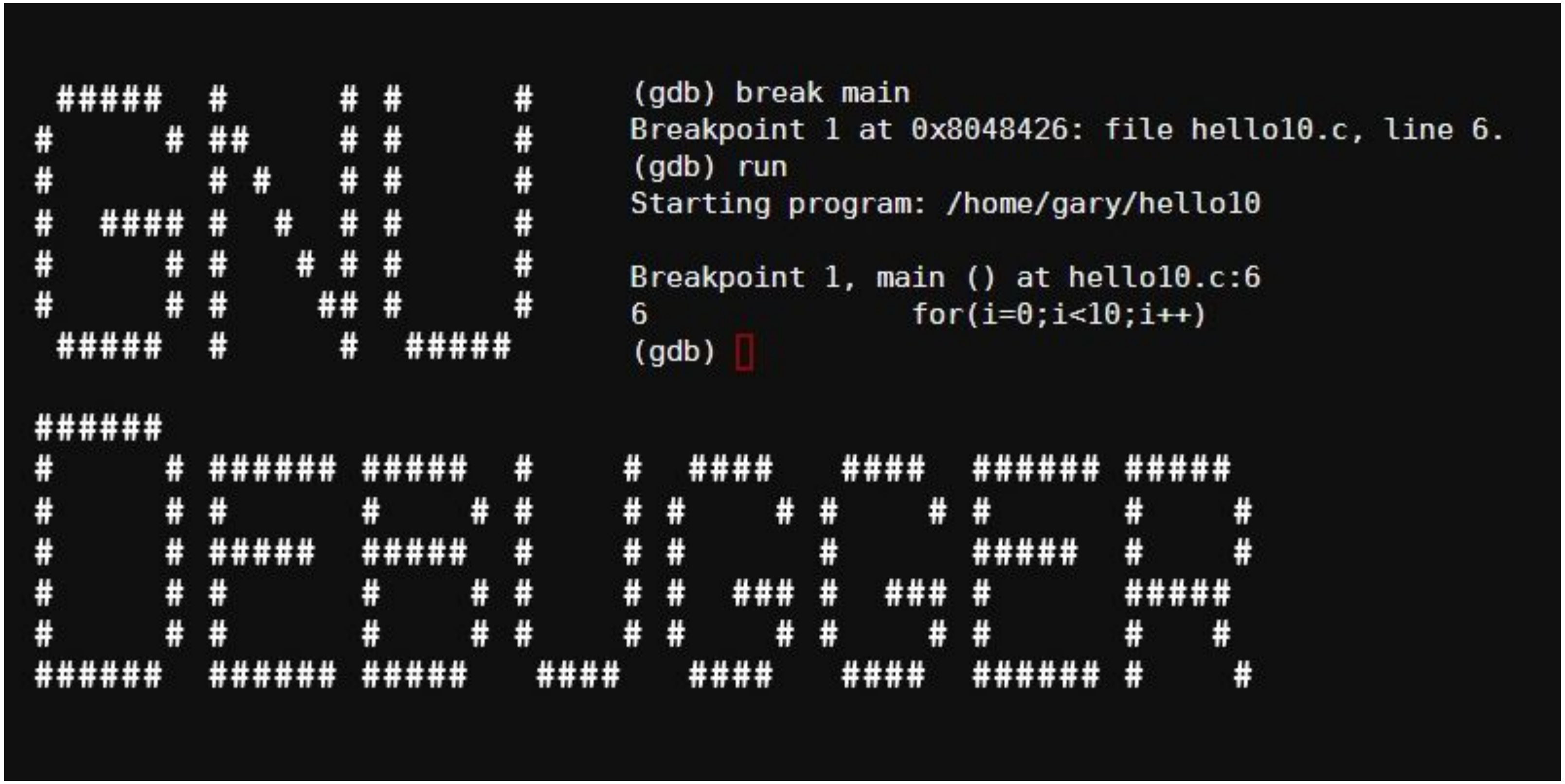
Naser Ezzati

Brock University
www.github.com/naser/ubrock

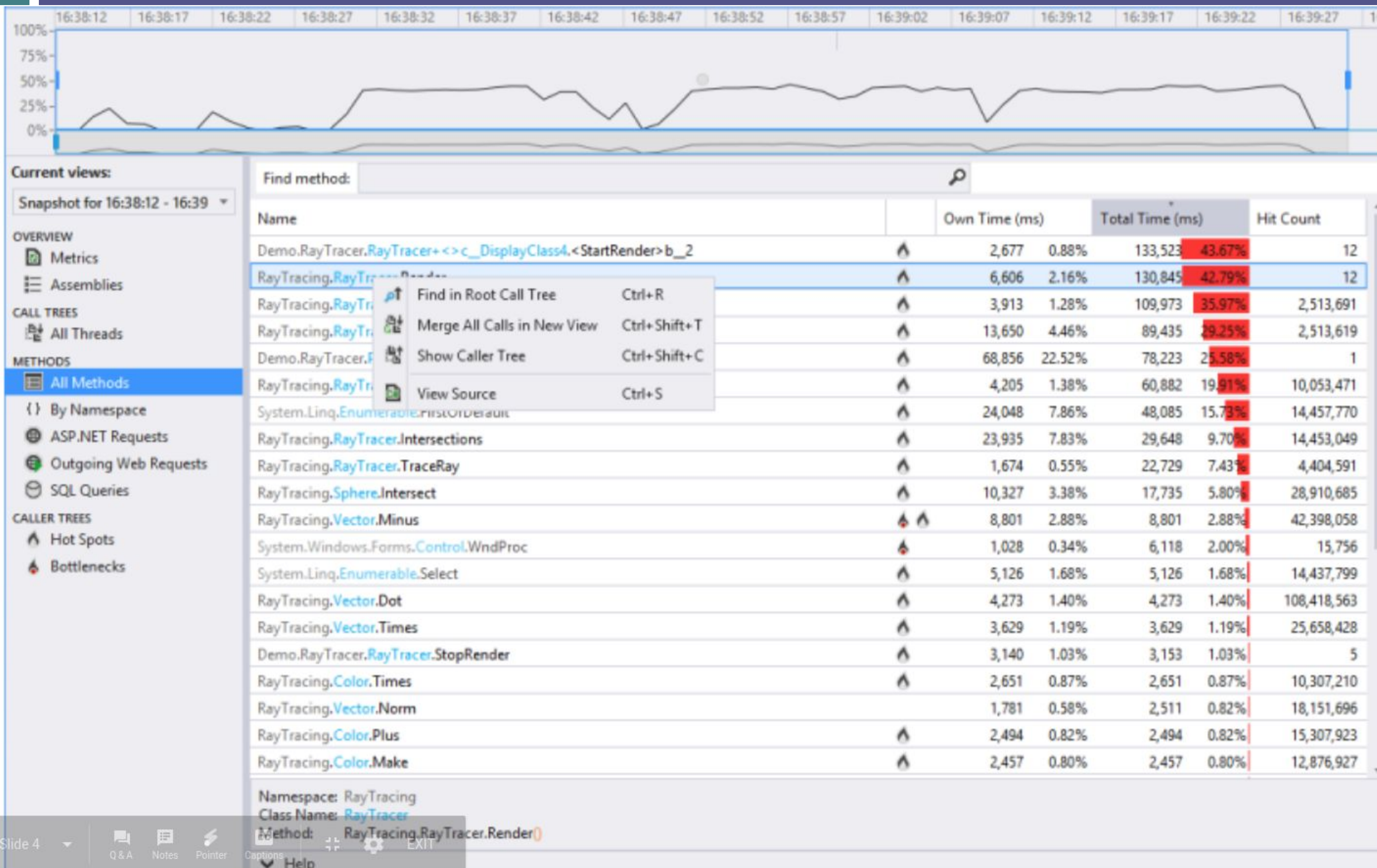
Goal: Debug Software in Production



Debuggers and Profilers



Profilers



Current views:
Snapshot for 16:38:12 - 16:39
OVERVIEW
Metrics
Assemblies
CALL TREES
All Threads
METHODS
All Methods
By Namespace
ASP.NET Requests
Outgoing Web Requests
SQL Queries
CALLER TREES
Hot Spots
Bottlenecks
Find method:
Name
Own Time (ms)
Total Time (ms)
Hit Count
Demo.RayTracer.RayTracer+<>c__DisplayClass4.<StartRender>b__2 2,677 0.88% 133,523 43.67% 12
6,606 2.16% 130,845 42.79% 12
3,913 1.28% 109,973 35.97% 2,513,691
13,650 4.46% 89,435 29.25% 2,513,619
68,856 22.52% 78,223 25.58% 1
4,205 1.38% 60,882 19.91% 10,053,471
24,048 7.86% 48,085 15.73% 14,457,770
Find in Root Call Tree Ctrl+R
Merge All Calls in New View Ctrl+Shift+T
Show Caller Tree Ctrl+Shift+C
View Source Ctrl+S
RayTracing.RayTracer.Intersections 23,935 7.83% 29,648 9.70% 14,453,049
RayTracing.RayTracer.TraceRay 1,674 0.55% 22,729 7.43% 4,404,591
RayTracing.Sphere.Intersect 10,327 3.38% 17,735 5.80% 28,910,685
RayTracing.Vector.Minus 8,801 2.88% 8,801 2.88% 42,398,058
System.Windows.Forms.Control.WndProc 1,028 0.34% 6,118 2.00% 15,756
System.Linq.Enumerable.Select 5,126 1.68% 5,126 1.68% 14,437,799
RayTracing.Vector.Dot 4,273 1.40% 4,273 1.40% 108,418,563
RayTracing.Vector.Times 3,629 1.19% 3,629 1.19% 25,658,428
Demo.RayTracer.RayTracer.StopRender 3,140 1.03% 3,153 1.03% 5
RayTracing.Color.Times 2,651 0.87% 2,651 0.87% 10,307,210
RayTracing.Vector.Norm 1,781 0.58% 2,511 0.82% 18,151,696
RayTracing.Color.Plus 2,494 0.82% 2,494 0.82% 15,307,923
RayTracing.Color.Make 2,457 0.80% 2,457 0.80% 12,876,927
Namespace: RayTracing
Class Name: RayTracer
Method: RayTracing.RayTracer.Render()
Help

Tracing


Can Fly
FLIGHT RECORDER DO NOT OPEN
FLIGHT RECORDER DO NOT OPEN

Instrumentation

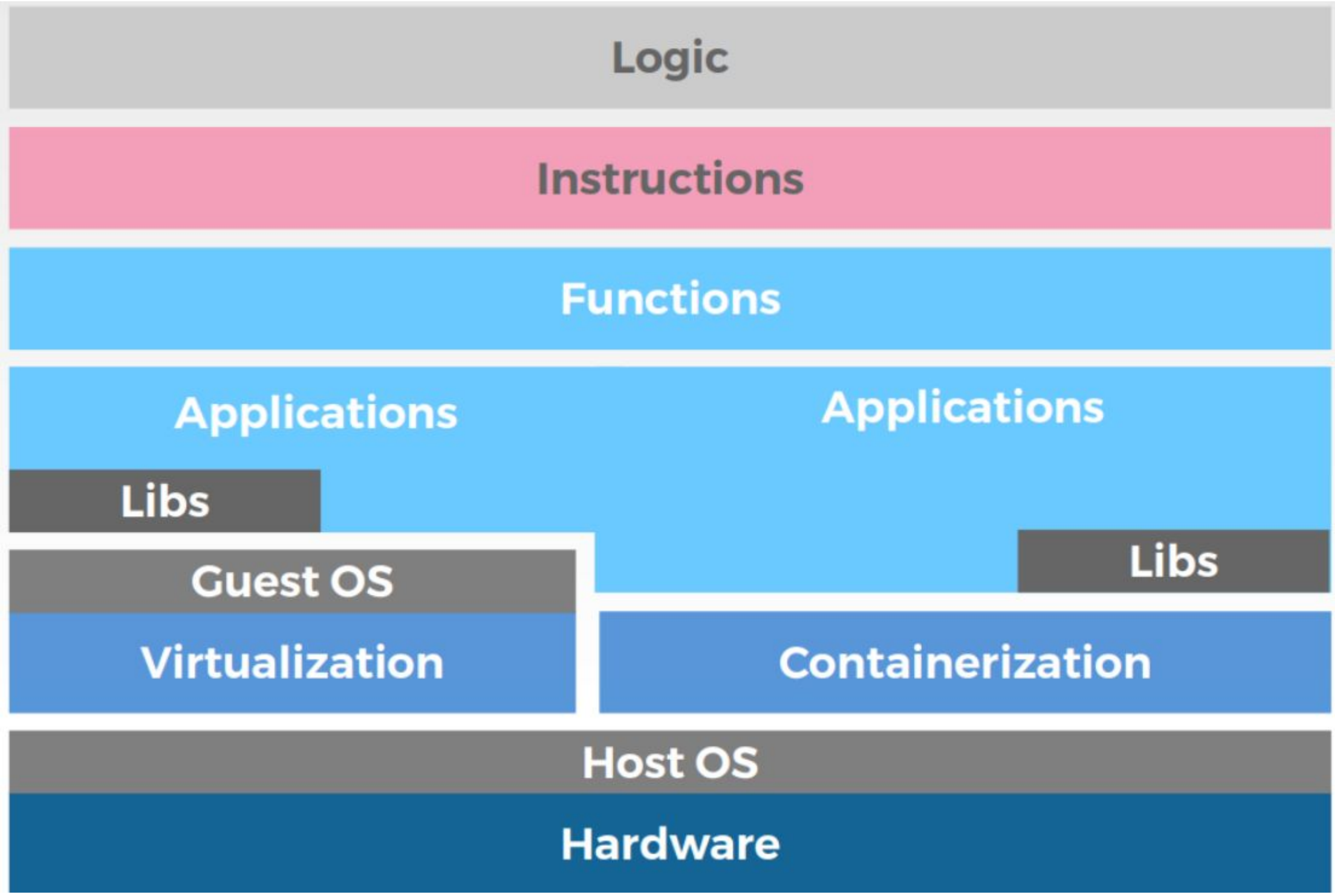

```c
int my_func(void* my_value) {
  int i, ret;

  printf("%s: Entering my function", timestamp)  //<--
  for (i = 0; i < MAX; i++) {
    ret = do_something_with(my_value, i);
    printf("In for loop %d", i);                 //<--
  }
  printf("%s: Done: %d",  timestamp, ret);       //<--

  return ret;
}
```

```
$ ./myprog
12345674:Entering my function                 // <-- event
12345675:In for loop 0                        // <-- event
12345676:In for loop 1                        // <-- event
12345677:In for loop 2                        // <-- event
12345678:Done: 10                             // <-- event
```

Multilevel Tracing



Models, logic

Branches, calls

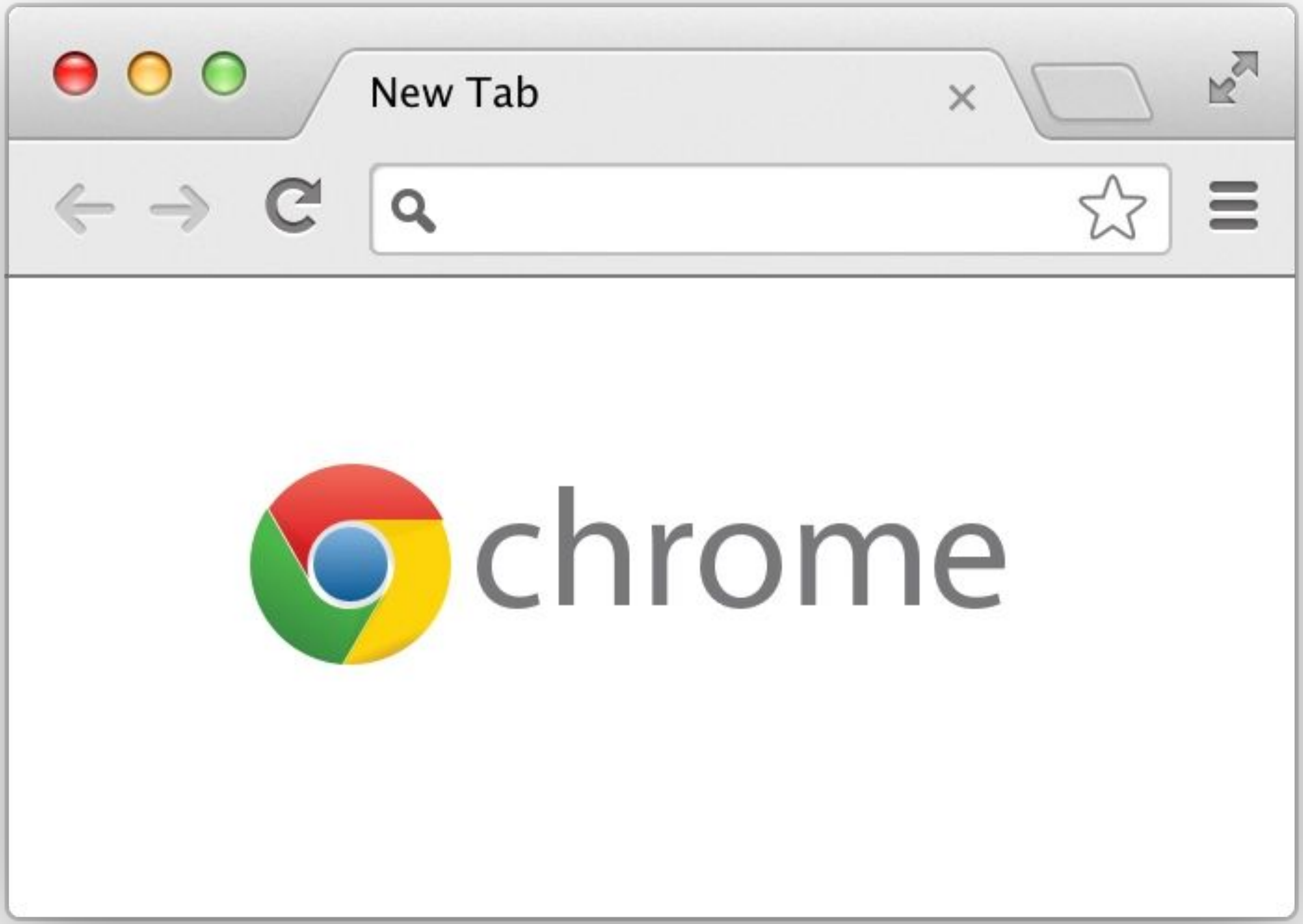
Application functions

Library functions, Syscalls

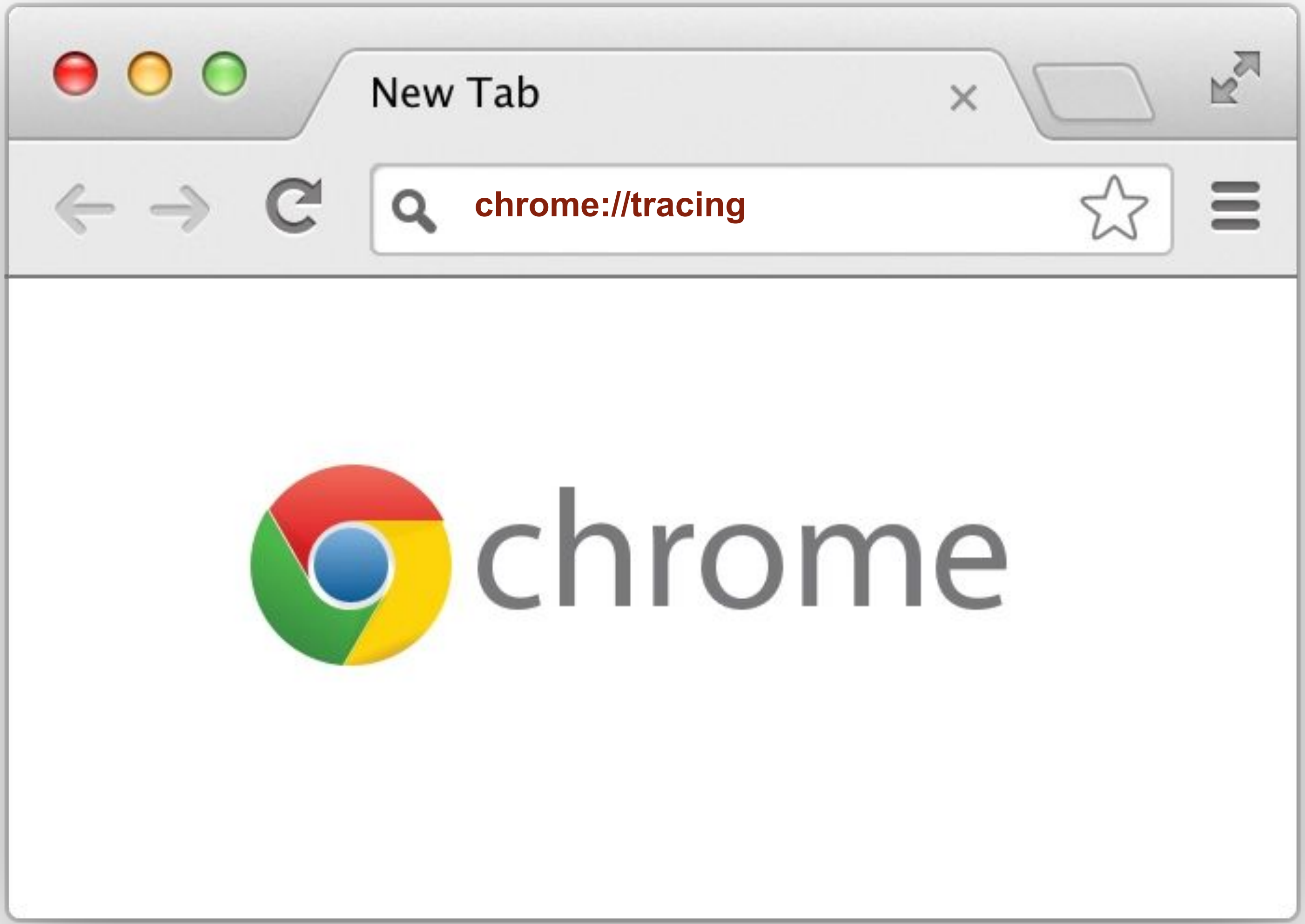
Kernel function, Hypervisor

Kernel function, Hardware perf counters

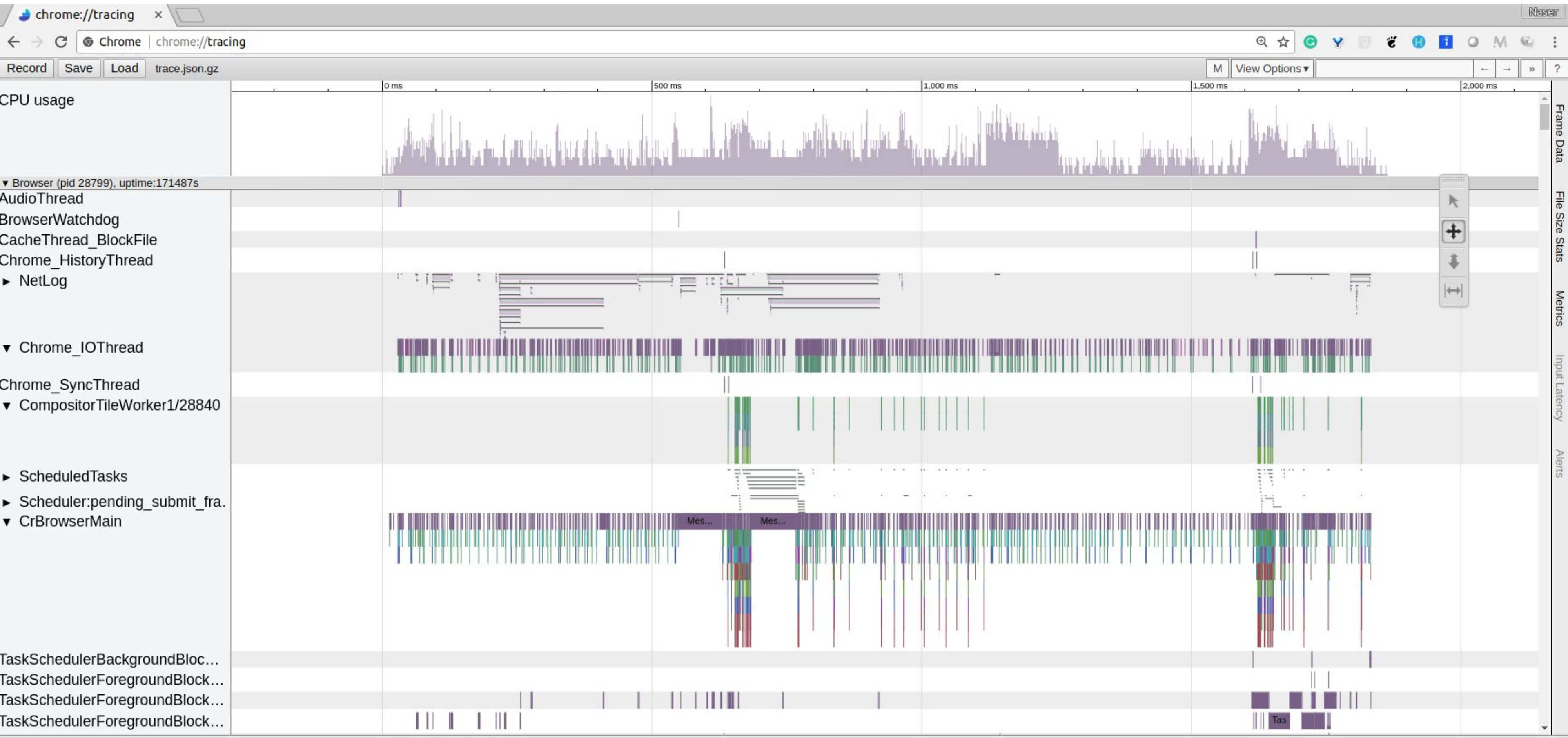
Why Multilevel Analysis?



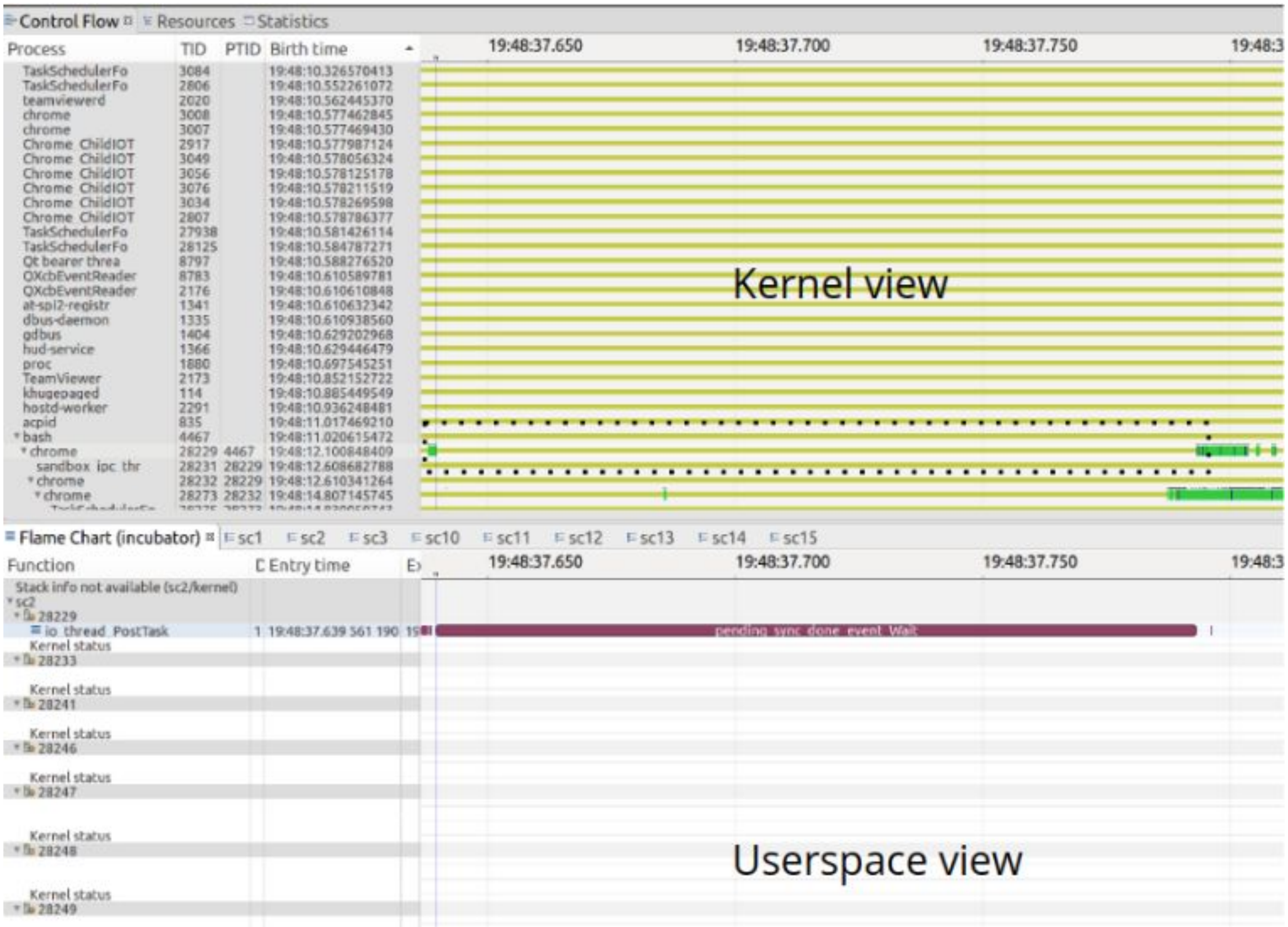
Why Multilevel Analysis?



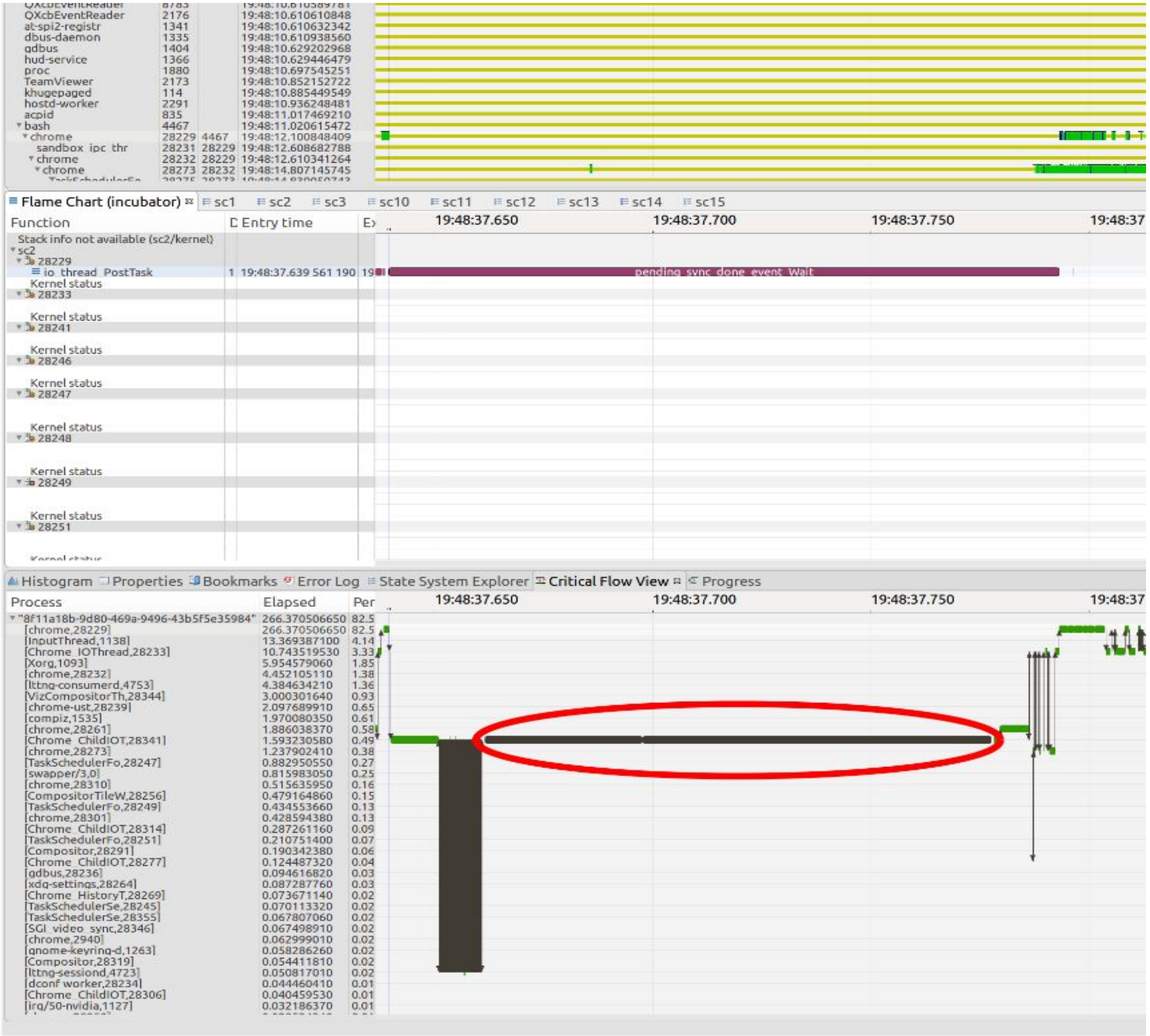
Why Multilevel Analysis?



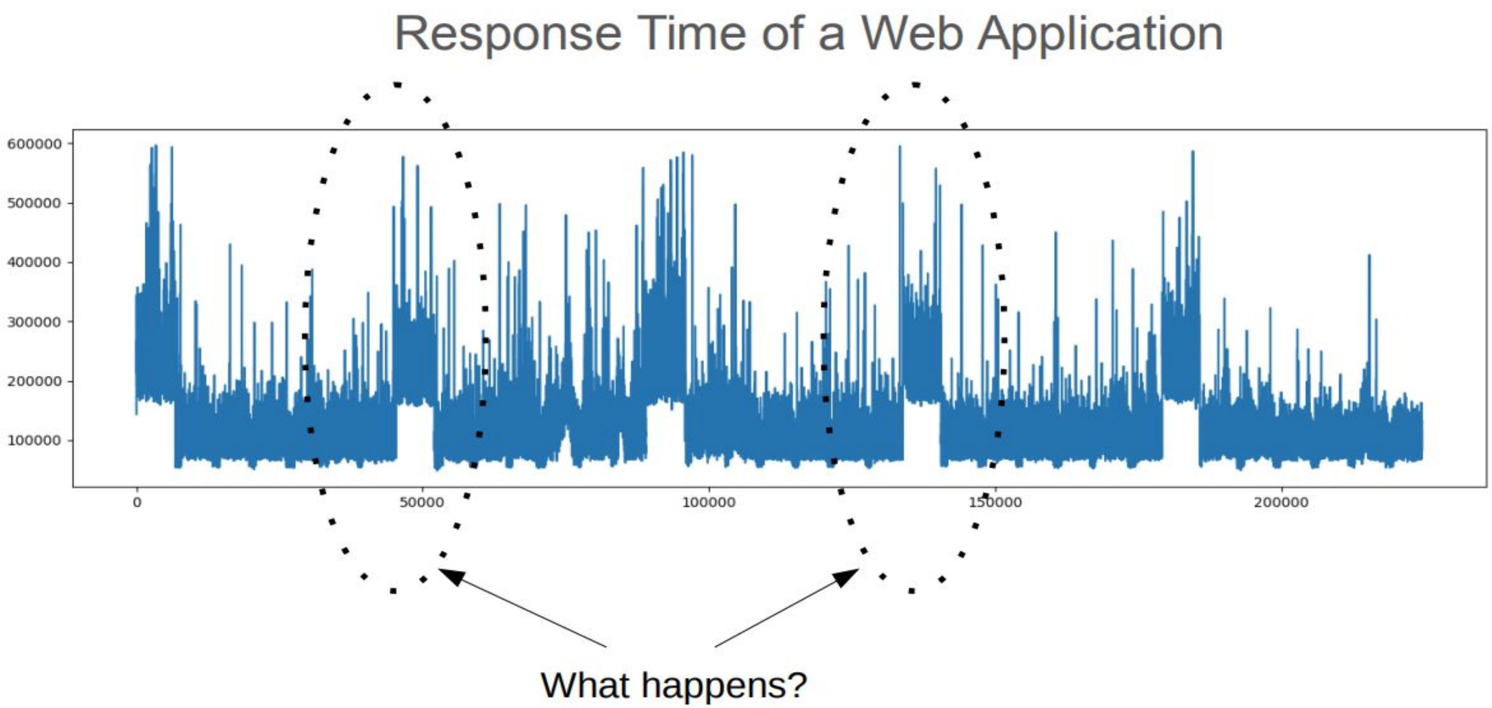
Why Multilevel Analysis?



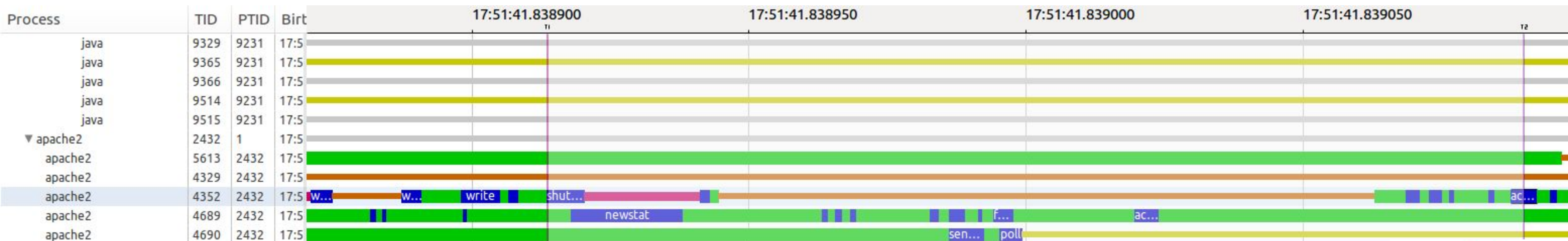
Why Multilevel Analysis?



Web Requests Latency Problem



Manual Investigation?

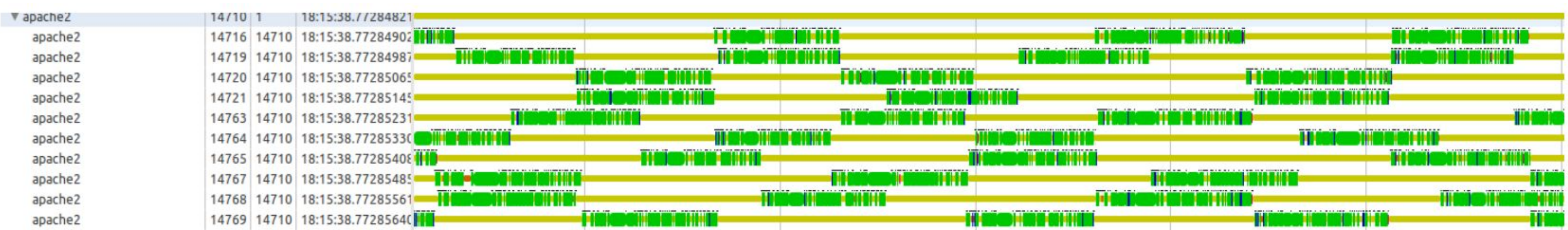


Trace View

- Huge amount of information not related to the observed problem.

Behavioral Analysis

- Can we **cluster** executions/requests **based on their behavior**?
- Can we **identify the differences** between groups of requests?
  - Can we find similar requests within a cluster?
  - Can we find different requests between clusters?



Converting Trace to State Sequences

ap_R → ap_R → my_U → my_R → ap_P → ap_R → ap_R →...

time: 10ms

ap_R → ap_R → my_U → my_R → ap_P → ap_R → my-R →...

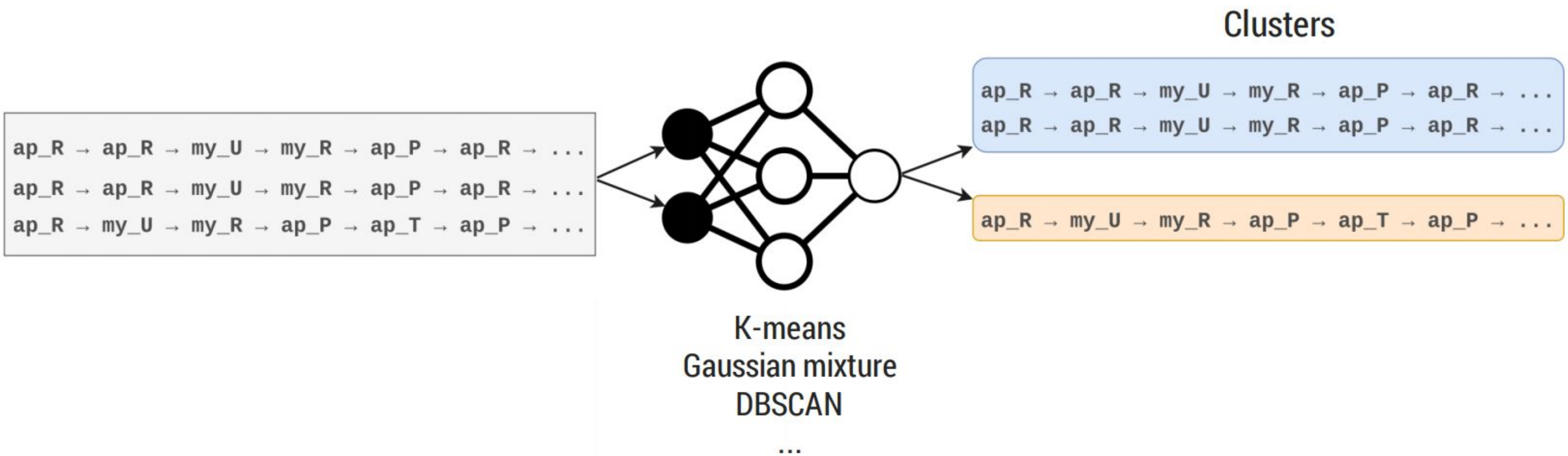
time: 14ms

ap_R → my_U → my_R → ap_P → ap_T → ap_P → ap-R →...

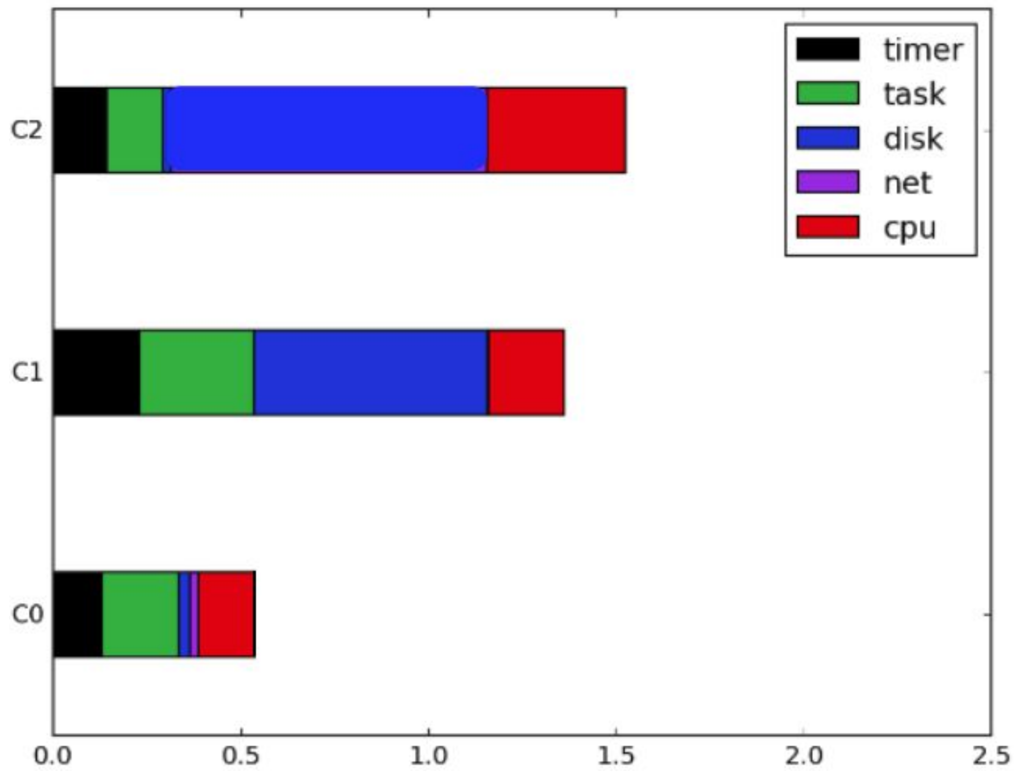
time: 36ms

...

Clustering

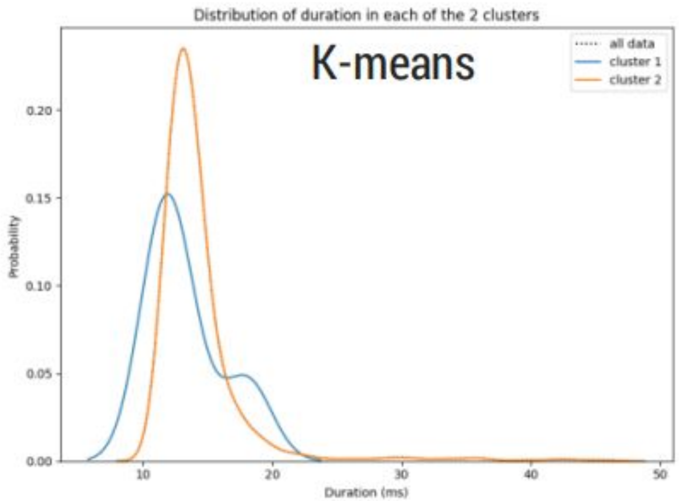

ap_R → ap_R → my_U → my_R → ap_P → ap_R → ...
ap_R → ap_R → my_U → my_R → ap_P → ap_R → ...
ap_R → my_U → my_R → ap_P → ap_T → ap_P → ...
K-means
Gaussian mixture
DBSCAN
...
Clusters
ap_R → ap_R → my_U → my_R → ap_P → ap_R → ...
ap_R → ap_R → my_U → my_R → ap_P → ap_R → ...
ap_R → my_U → my_R → ap_P → ap_T → ap_P → ...

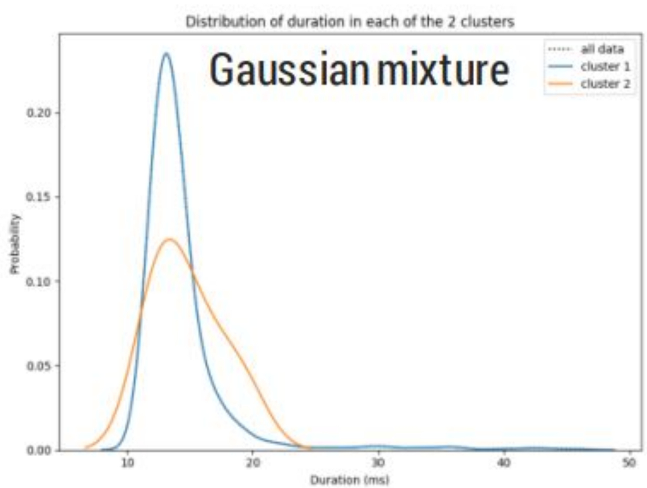
Resource Usages Clustering

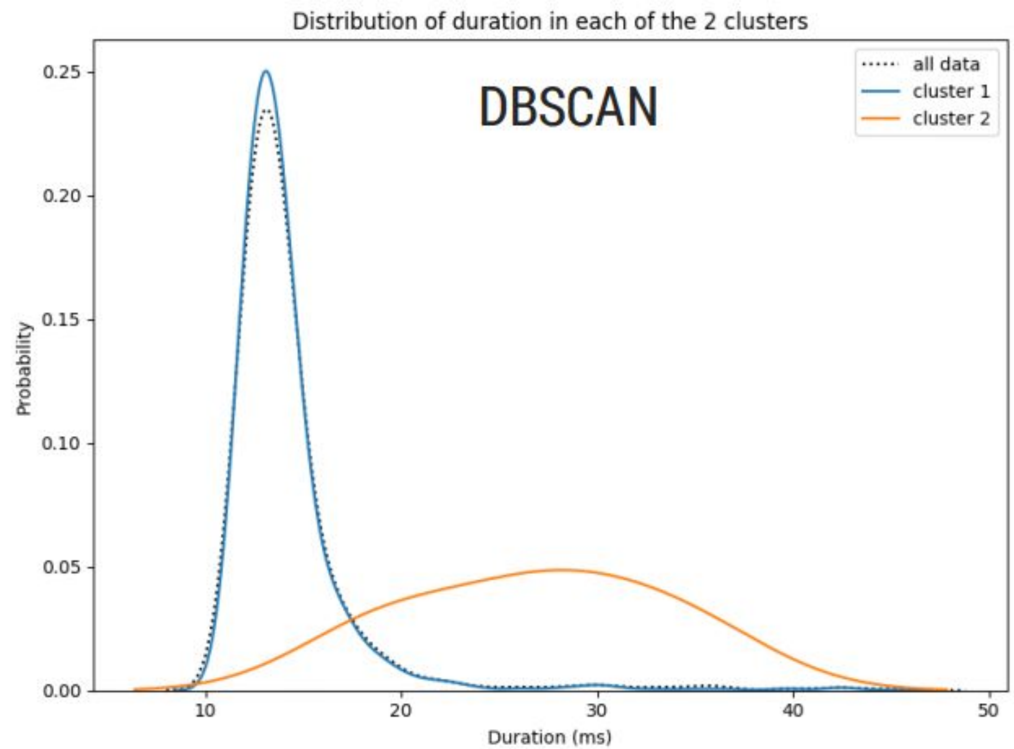


Metrics:
Wait AVG for Disk average time
Wait AVG for Net average time
Wait AVG for Timer average time
Wait AVG for Task average time
Wait AVG for CPU average time
The frequency of wait for Disk
The frequency of wait for Net
The frequency of wait for Timer
The frequency of wait for Task
The frequency of wait for CPU
The frequency of read from Disk
The frequency of write to Disk
Total Block Disk Read
Total Latency Disk Read
Total Block Disk Write
Total Latency Disk Write
Disk Cache Hit/Miss Ratio

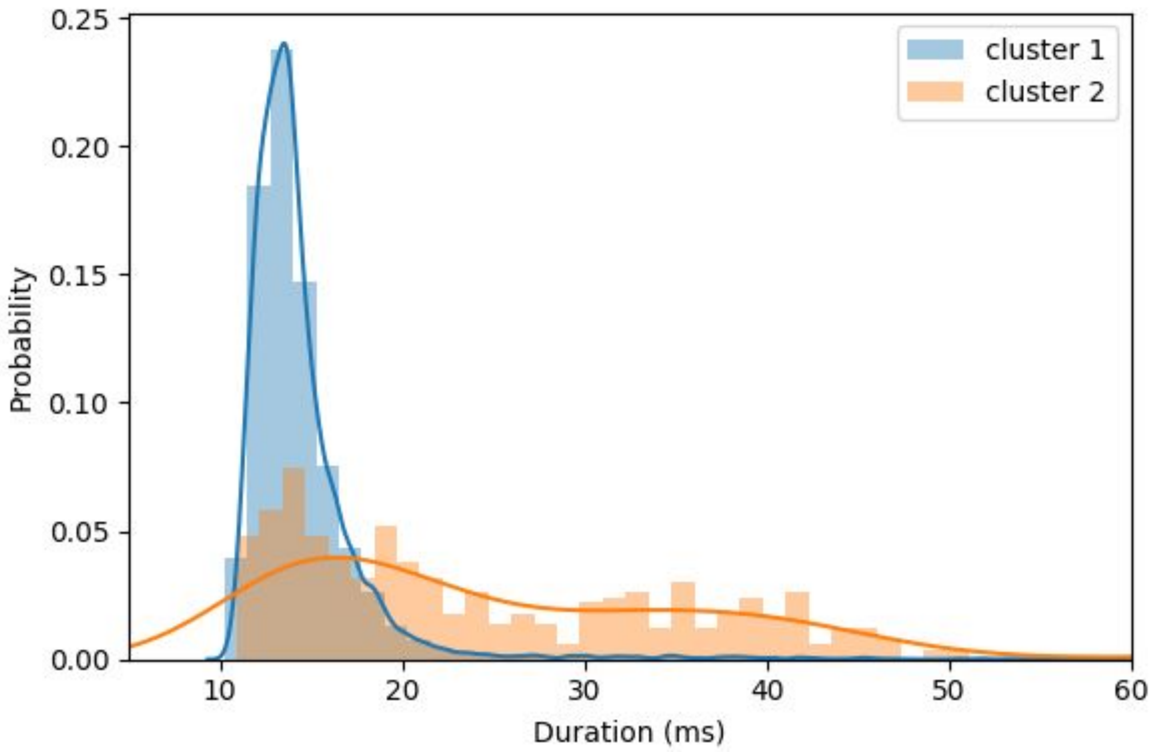
Clustering Result







DBScan Clustring



Analyse the Clustering Result

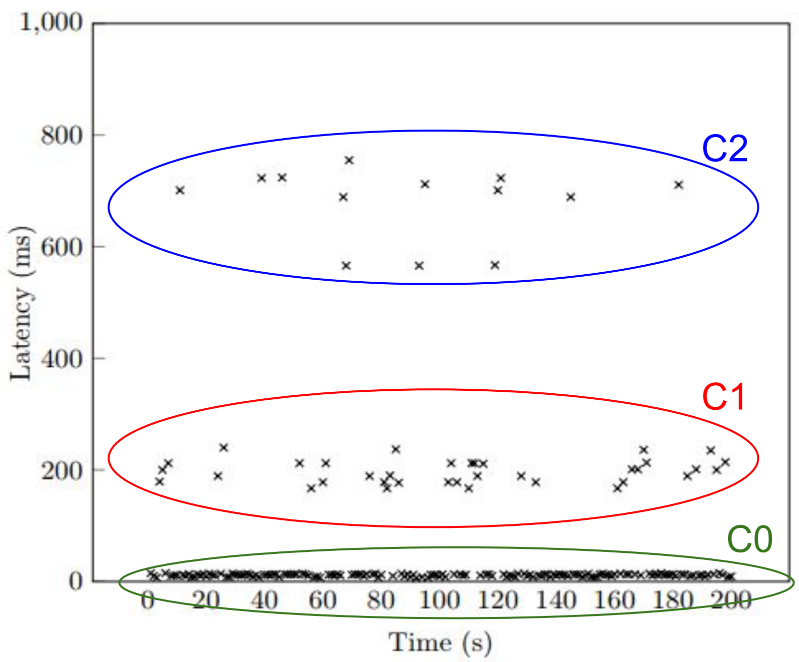
(C0) A latency of 10 ms. Most requests belong to this category

(C1) A latency of [180 ms, 220 ms]

(C2) A latency of [600 ms, 800 ms]

C0: No disk!

C1: and C2 partially/totally served from disk!

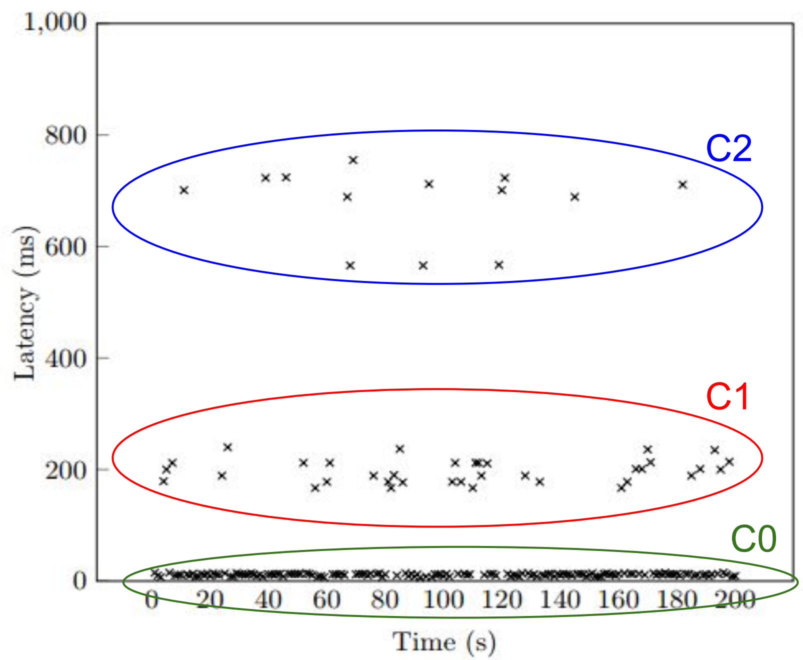


Analyse the Clustering Result

(C0) A latency of 10 ms. Most requests belong to this category

(C1) A latency of [180 ms, 220 ms]

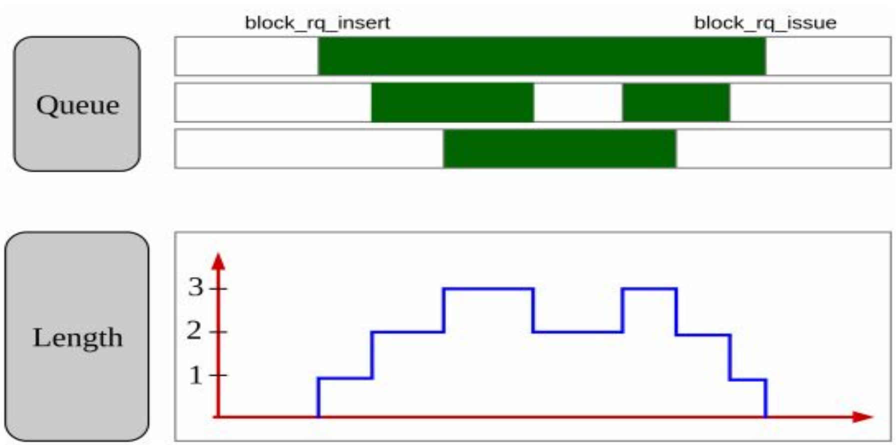
(C2) A latency of [600 ms, 800 ms]

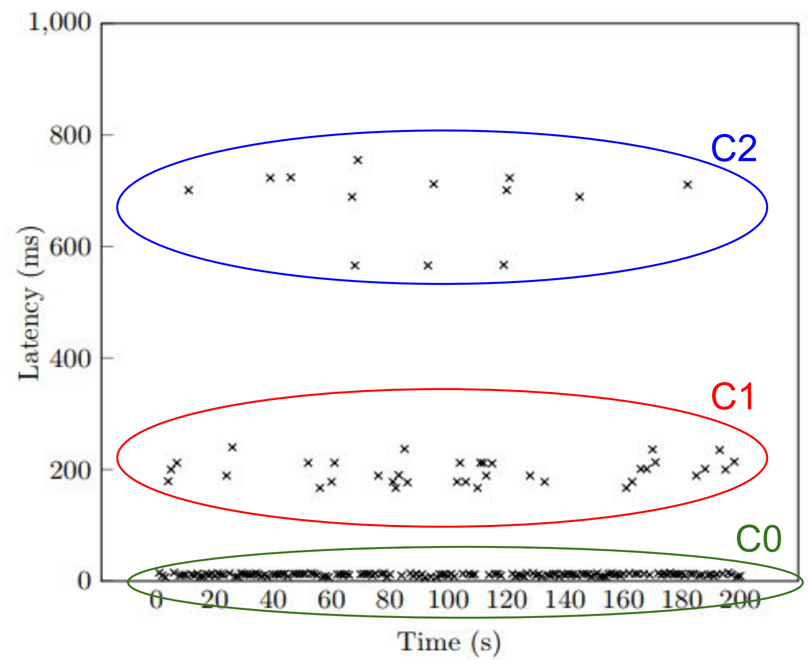


Cache Hit Feature helps to understand the origin of slowness of some requests, but

Why C2 requests take more time to read from the disk, while they have the same cache his/miss ratio?

Analyse the Clustering Result



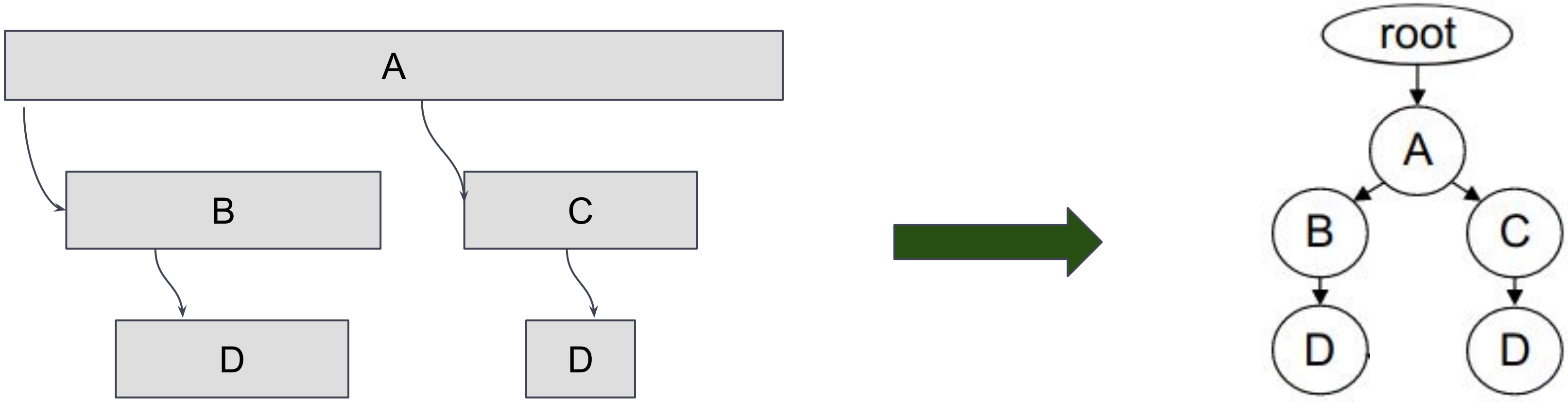


C2 requests are processed when the disk drive is busy processing other requests from other processes.

- A backup.sh process
- Other web requests

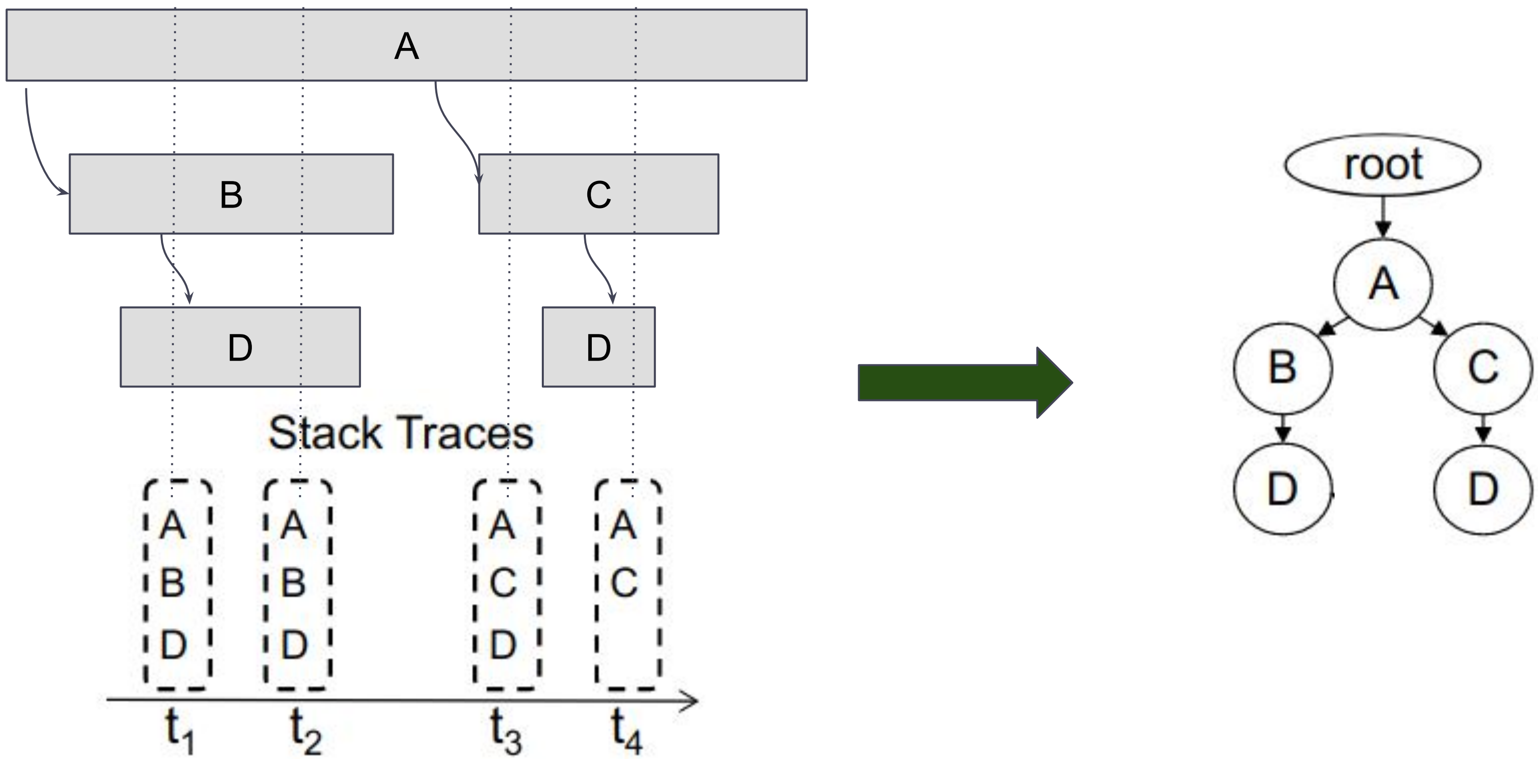
Automatic Investigation

- Extract CCT for each request



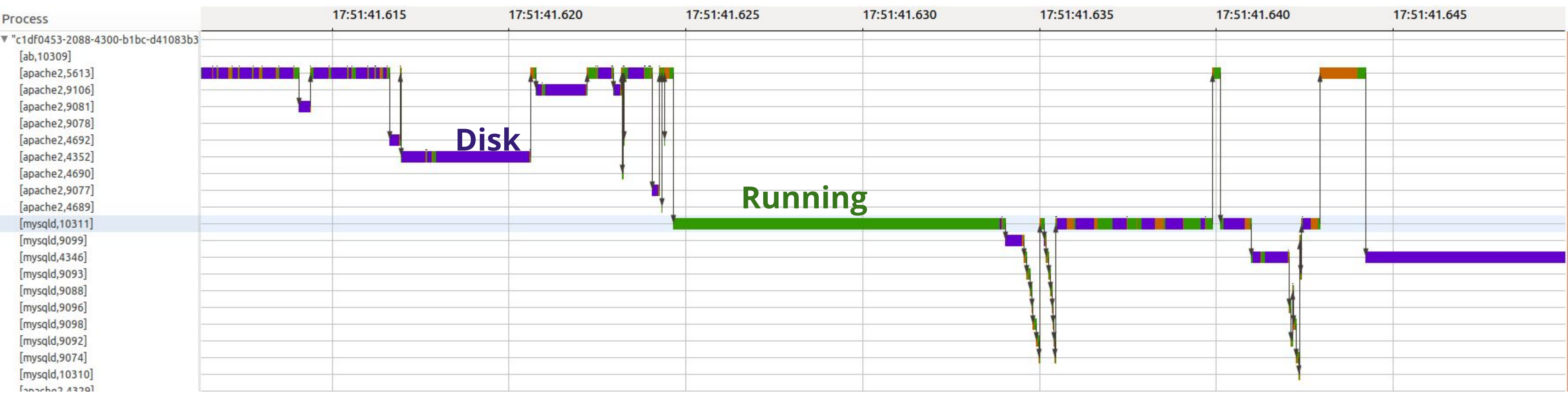
Automatic Investigation

- Extract CCT for each request



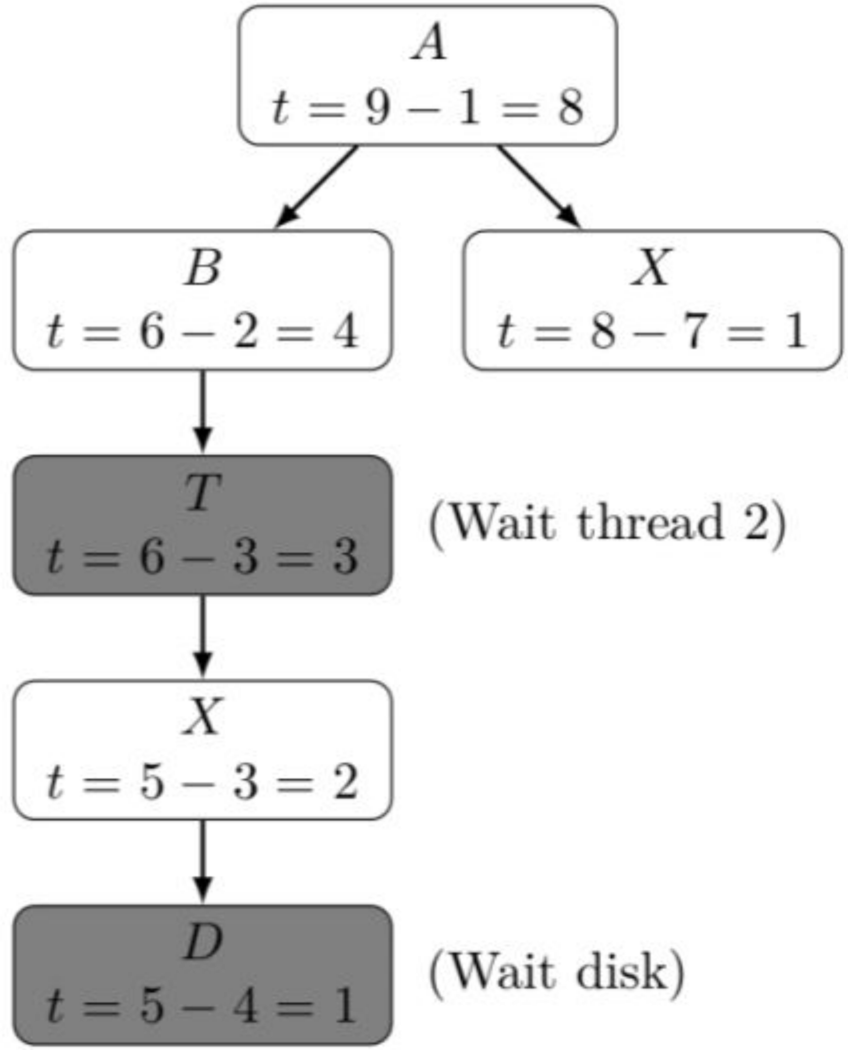
Automatic Investigation

- Extract CCT for each request
- Extract critical path for each request
  - Graph of dependencies between threads



Merging CCT of Related Threads

Time	Thread 1	Thread 2
1	Call A	
2	Call B	
3	Wait thread 2	Call X
4		Wait disk
5		Return X
6	Return B	
7	Call X	
8	Return X	
9	Return A	



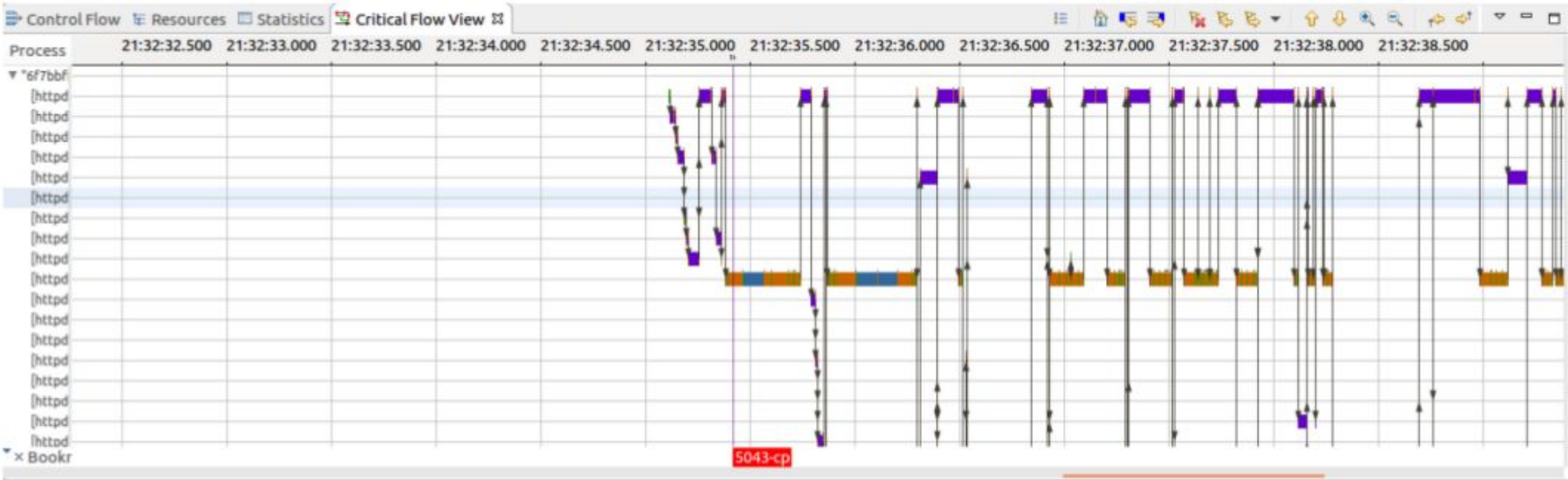
Comparing the Requests

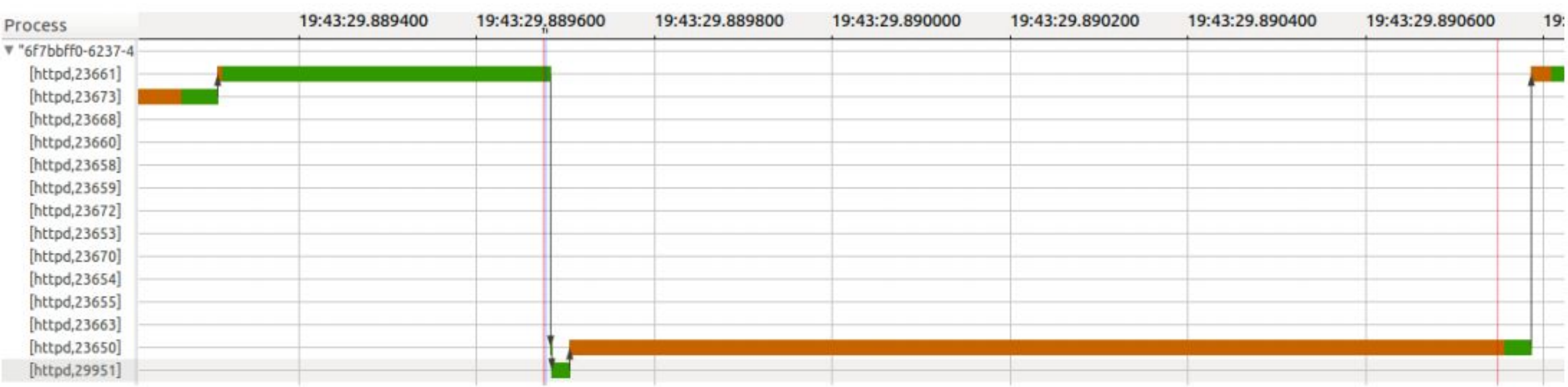
- Comparing the requests ---> comparing the CCTs
- General solution:
  - Make critical CCT for each request
  - Cluster requests based on different features
  - Compare the groups to find the differences and reasons of slowness

Use-cases

- Some period slow web requests
- One slow mysql request out of 10000 requests
- Real-time process was preempted with a low-priority thread doing some slow tasks
- Latencies related to:
  - CPU usage
  - Disk / network contention
  - Dependencies between threads
    - Futex/lock/etc.

Web Requests Periodic Slowdown





Performance

- How costly is this work?
  - Tracing
    - For every kernel event it adds 100 ns overhead
    - For the web application test it has 8.6 % overhead
  - Analysis
    - For 5 minutes of tracing (130 MB), it took 30 seconds to analyse and save the results and the size was 12.8 MB (10% of the original trace size)

Conclusion

- Automatic root cause identification of latency problems between multiple similar requests
  - Dynamic analysis through execution trace
  - Multilevel trace collection
  - Automatic grouping of requests based on their runtime behavior
  - Comparing different groups
- Support more latency problems
  - VMs
    - Containers
    - Distributed requests

References

[1] Houssem Daoud, Naser Ezzati-Jivan, and Michel R. Dagenais. Dynamic trace-based sampling algorithm for memory usage tracking of enterprise applications. In 2017 IEEE High Performance Extreme Computing Conference (HPEC 2017), pages 1–7, Sept 2017. URL: https://doi.org/10.1109/HPEC.2017.8091061, doi:10.1109/HPEC.2017.8091061.
[2] Simon Delise, Naser Ezzati-Jivan, and Michel R. Dagenais. Integrated modeling tool for indexing and analyzing state machine trace. submitted to Elsevier Journal of Computer and System Sciences, 2018.
[5] Jean-Christian KOUAME, Naser Ezzati-Jivan, and Michel R. Dagenais. A data driven approach for pattern matching, filtering and analysis of execution traces. submitted to Springer Empirical Software Engineering, 2018.
[3] K. Kouame, Naser Ezzati-Jivan, and Michel R. Dagenais. A flexible data-driven approach for execution trace filtering. In 2015 IEEE International Congress on Big Data, pages 698–703, June 2015. URL: https://doi.org/10.1109/BigDataCongress.2015.112, doi:10.1109/BigDataCongress.2015.112.
[4] Alexander Montplaisir, Naser Ezzati-Jivan, Florian Wininger, and Michel R. Dagenais. State history tree: An incremental disk-based data structure for very large interval data. In 2013 International Conference on Social Computing, pages 716–724, Sept 2013. URL: https://doi.org/10.1109/SocialCom.2013.107, doi:10.1109/SocialCom.2013.107.
[5] Loic Prieur-Drevon, Raphael Beamonte, Naser Ezzati-Jivan, and Michel R. Dagenais. Enhanced state history tree (esht): A stateful data structure for analysis of highly parallel system traces. In 2016 IEEE International Congress on Big Data (BigData Congress 2016), pages 83–90, June 2016. URL: https://doi.org/10.1109/BigDataCongress.2016.19, doi:10.1109/BigDataCongress.2016.19.
[6] Fabien Reumont-Locke, Naser Ezzati-Jivan, and Michel R. Dagenais. Efficient methods for trace analysis parallelization. accepted in Springer International Journal of Parallel Programming, 2019.
[7] Naser Ezzati-Jivan. Multi-Level Trace Abstraction, Linking and Display. PhD thesis, École Polytechnique de Montréal, 2014. URL: http://publications.polymtl.ca/1402/.
[8] Doray, F. & Dagenais, M. R. (2016). Diagnosing performance variations by comparing multi-level execution traces. IEEE Transactions on Parallel and
Citation : Distributed Systems, 28(2), p. 462-474. doi:10.1109/tpds.2016.2567390
[9] Naser Ezzati-Jivan and Michel R. Dagenais. A stateful approach to generate synthetic events from kernel traces. Adv. Soft. Eng., 2012:6:6–6:6, January 2012. URL: http://dx.doi.org/10.1155/2012/140368, doi:10.1155/2012/140368.
[10] Naser Ezzati-Jivan, Genevieve Bastien, and Michel R. Dagenais. High latency cause detection using multilevel dynamic analysis. In 2018 Annual IEEE International Systems Conference (SysCon 2018), Jan 2018.
[11] Naser Ezzati-Jivan and Michel R. Dagenais. An efficient analysis approach for multi-core system tracing data. In Proceedings of the 16th International Conference on Software Engineering and Applications (SEA 2012), 2012. URL: http://dx.doi.org/10.2316/P.2012.790-053, doi:10.2316/P.2012.790-053.

Questions?

LTTng and Trace Compass

http://lttng.org

http://tracecompass.org

Sources:

http://github.com/naser

Contact:

ezzati@gmail.com

Silhouette Score

- Considers both cohesion and separation and is computed for a given sample RQi
    - Si = (out i − In i) / max(In i, out i)
        - In i: average distance of RQi to all other samples in its cluster
        - Out i: minimum average distance between RQi and out-cluster samples
    - The value of silhouette coefficient lies between -1 and 1, where a positive large value is more desirable.
    - We would like to have *In i* as close as possible to zero for a very cohesive cluster, and to have out i as large as possible representing very large separation from the closest neighbor cluster.

Silhouette Score

- while the quality of a clustering algorithm plays an important role in finding good clusters, having cohesive and well separated clusters also largely depends on the nature of the data and the distance metric.
- The silhouette coefficient can be averaged over a cluster or the entire samples to yield a silhouette score per cluster or for the entire clustering, respectively.
- We found that applying Kmeans to find three coarse clusters would yield the best total silhouette score of 0.42 and per cluster silhouette of C0 = 0.38, C1 = 0.35, C2 = 0.48.