Course Notes on Software Monitoring & Tracing

Martin Monperrus
KTH Royal Institute of Technology

Created April 2019, last updated April 6, 2023

"Program execution traces provide the most intimate details of a program's dynamic behavior." Bhansali et al. 2006

This document contains course notes about monitoring and tracing of software execution (prepared for the KTH course "Automated Testing and DevOps"). The monitoring in production aspect of DevOps is essential to close the infinite loop.

Feedback: `martin.monperrus@csc.kth.se`
URL: `https://www.monperrus.net/martin/monitoring.pdf`

Contents

1 Core Concepts

Execution trace:

- an **execution trace** is a set of events resulting from the execution of a program
- it can be huge: "Nine billion instruction boot of FreeBSD"[1]
- it is most most of the time incomplete (it only gives partial information about the execution, it is not able to recompute the final result or recreate an intermediate program state), unless you use time-travel debugging.
- a trace has different granularities, it can be for a single thread, a single program or for a system distributed over thousands of machines.

Usages of monitoring:

- Testing: code coverage
- Problem and anomaly detection
- Performance analysis and optimization
- Debugging (the absolute monitoring is time-travel debugging and execution replay)
- Security: intrusion detection (defense), reverse engineering, see for example this post
- Art (eg Web stalker)

Monitoring Terminology

- Monitoring and tracing is the art of collecting execution traces. Both terms are considered equal in this document.
- It is related to logging: monitoring is meant to be automatic while logging is traditionally manual.
- Monitoring can be achieved by a number of different techniques, including hardware interrupts, instruction set simulation, performance counters, operating system hooks (see below), code instrumentation, dedicated runtime support (in interpreters, VMs, libraries).
- dynamic program analysis (or dynamic analysis for short) is based on monitoring to infer some knowledge, such as coverage or invariants.
- profiling is a specific kind of monitoring dedicated to performance analysis.

Core Trade-offs

- Performance overhead in production
- Amount of data (beware of the monitoring data lake)
- Privacy (we must not log passwords in clear)

Observability:

- In dynamic program analysis, observability is the property of a system referring to the ability of collecting information of interest.
- Observability has an engineering cost (it can take months to otain a specific kind of information).
- Observability has a performance cost (monitoring can slow down the whole process1).
- Observation has a storage cost (traces are quickly huge).

[1] https://github.com/panda-re/panda

Figure 1: Example of code coverage

2 Types of traces

2.1 Logs

- It is straightforward to do logging
- it is harder to do proper centralized logging
  - See centralized log management tools such as `https://github.com/Graylog2/graylog2-server`, fluentd, Apache Flume
  - A known practice is to put all the logs into ElasticSearch `https://www.elastic.co/blog/get-system-logs-and-metrics-into-elasticsearch-with-beats-system-modules`
- Most application-level monitoring is done with logging. However, there exists more generic solutions such as JMX

2.2 Coverage

- The code coverage, also known as spectrum, is an interesting aspect of execution.
- Open question: for a given software stack, how to do it in production with acceptable low overhead(remove unused code, A/B testing code, decrease technical debt).

2.3 Instruction stream

An instruction stream is a sequence of consecutive events at a low-level (machine code, byte code), see for example this instream trace

- Intel PIN generates instruction opcode traces
- Bytecode traces (`-XX:+TraceBytecodes`, see bytestacks) hello world trace
- A branch trace logs successful branch instructions
- If the trace is completely captured, it enables time-travel debugging.

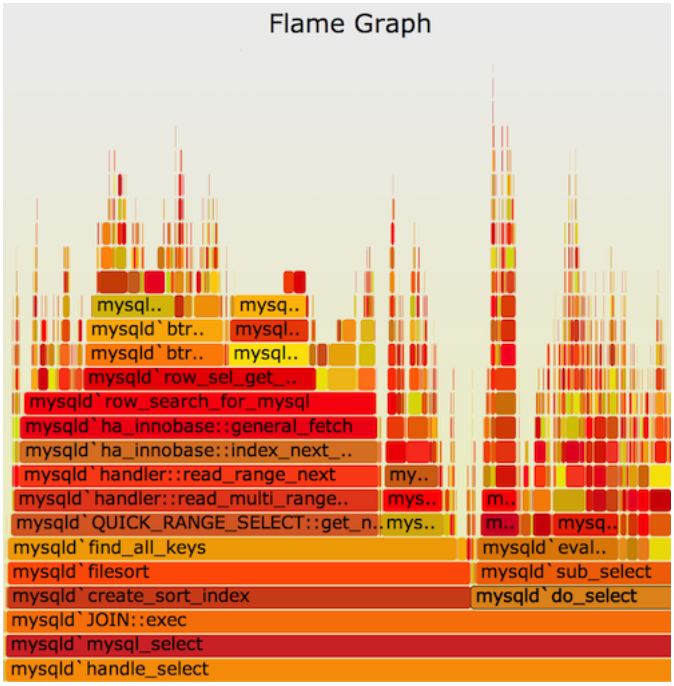


Figure 2: A flame graph is a nice visualization of a call tree

2.4 Memory trace

- TracerGrind is a Valgrind tool (plugin) which can generate execution traces of a running process. example trace

2.5 Call tree

- The basic block of a call tree is a call trace (Kernel call trace with ftrace)
- A call tree is the concatenation of all consecutive call traces in a tree.
- Example: `https://raw.githubusercontent.com/monperrus/traces/master/perf.txt`
- Can be visualized `http://www.valgrind.org/docs/manual/manual-core.html#manual-core.xtree`, eg in a flame graph see Figure 2
- Capturing call tree in Java `https://github.com/castor-software/yajta`
- Capturing call tree in Javascript `https://maierfelix.github.io/Iroh/`

2.6 Dynamic call graph

A call graph is a graph representation of a call tree.

- Jdcallgraph: `https://github.com/dkarv/jdcallgraph/`
- Java-callgraph: `https://github.com/gousiosg/java-callgraph`

2.7 Event/message trace

2.7.1 Crashes

It is a common practice to collect crashes in the wild, with plenty of open-source and commercial technology (many of them being under the umbrella term application performance management. Crash handlers are all possible levels in the stack:

- at the kernel level (kernel oops)
- at the VM level Thread.setDefaultUncaughtExceptionHandler.
- at the cloud level openstack

Random sample of technology:

- `https://sentry.io/welcome/`
- `https://github.com/mozilla-services/socorro`
- `https://chromium.googlesource.com/crashpad/crashpad/+/master/README.md`

2.7.2 HTTP traces

- `https://github.com/buger/goreplay/`
- `https://glowroot.org`

2.7.3 Database connections

Example `https://glowroot.org`, `https://github.com/onecodex/chrononaut`

2.7.4 Distributed system tracing

- Distributed system tracing consists of tracing messages between machines / actors / microservices.
- Key enabler: use the same network library everywhere
- Core idea: tag all requests with a unique identifier
- For sake of performance, one can trace only a sample of requests
- See tools below

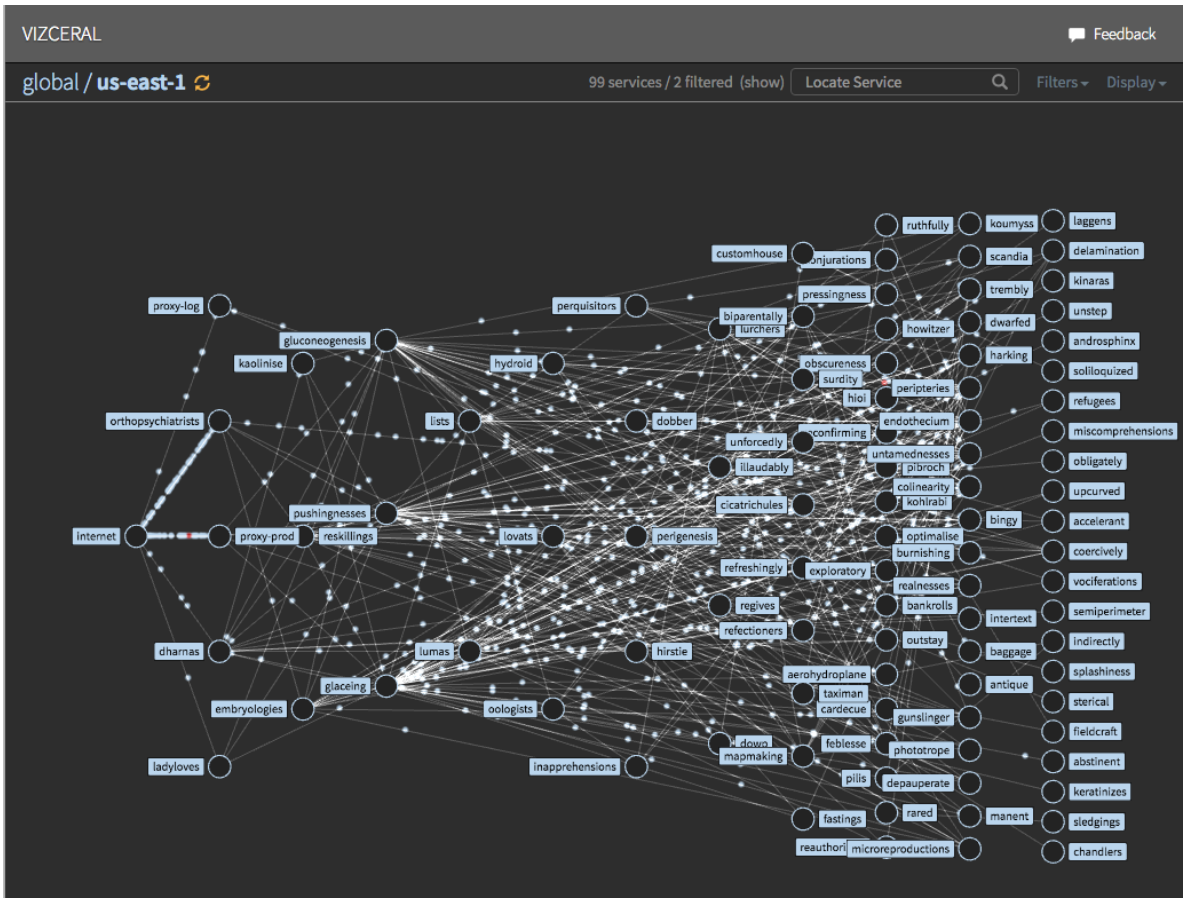
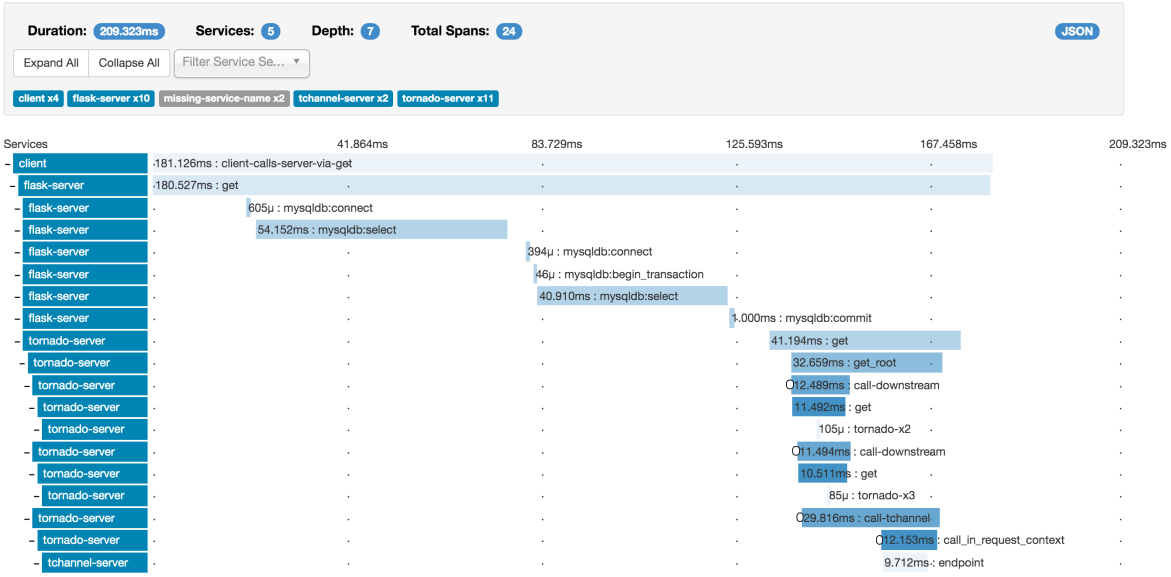


Figure 3: Dashboard of Vizceral for animated traffic graphs youtube1 youtube2



3 Implementation techniques

- instrumenting either the program source code or its binary executable form.
  - Source code instrumentation, eg `https://github.com/INRIA/spoon/`, hundreds of techniques
  - Binary code instrumentation, eg `https://asm.ow2.io/`, hundreds of techniques, see gdoc, incl. standard compiler techniques (gcov, lvvm/xray), aspect technology
- using an instruction set simulator or an interpreter
- using kernel support (post), see subsection 4.4
- using hardware support (interrupts and performance counter)
- using debug interfaces
  - JVM: Java Debug Interface see `https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/`
  - Browser/Node/V8: Chrome Remote Debugging Protocol see `https://www.igvita.com/2012/04/09/driving-google-chrome-via-websocket-api/` (working example)
- native support in virtual machine and interpreters
  - NodeJs: `node -trace`
  - JVM:
    - `jstack -l <pid>`
    - JVM options, see for instance `https://github.com/cl4es/bytestacks`
    - JVM Tool Interface and agents,
    - jconsole is a graphical tool for monitoring a JVM `https://openjdk.java.net/tools/svc/jconsole/`
    - Java agents Martin's list of Java agents
    - Visualvm, see `https://visualvm.github.io/`

4 Monitoring Tools

4.1 Java

- Clover: *https://bitbucket.org/openclover/clover* (coverage)
- Jacoco: *https://github.com/jacoco/jacoco* (coverage)
- Yajta: *https://github.com/castor-software/yajta* (execution tree)
- Btrace: *https://github.com/btraceio/btrace*
- Kieker: *https://github.com/kieker-monitoring/kieker*
- Bytefrog: *https://github.com/codedx/bytefrog*
- Bytestacks: *https://github.com/cl4es/bytestacks*
- Jdcallgraph: *https://github.com/dkarv/jdcallgraph/*
- Java-callgraph: *https://github.com/gousiosg/java-callgraph*

- Java Flight Recorder: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm
- Glowroot: *https://glowroot.org/*
- Anttracks: *https://glowroot.org/*
- Android/Dalvik: https://github.com/FrenchYeti/dexcalibur
- Quarkus - Micrometer: https://quarkus.io/guides/micrometer

4.2 Native

- gcov (coverage)
- LLVM
  - llvm-tracer: https://github.com/ysshao/LLVM-Tracer
  - Llvm-cov: https://llvm.org/docs/CommandGuide/llvm-cov.html
  - Panda: https://github.com/panda-re/panda

4.3 Javascript

- Iroh: https://maierfelix.github.io/Iroh/
- Istanbul: https://istanbul.js.org/
- V8's system-analyzer: https://v8.github.io/tools/head/system-analyzer/index.html

4.4 Linux Kernel

- Linux process
  - strace
  - tracer (recursive tracer), strace-graph trace-children
- perf, ftrace, bcc, dtrace, dtrace4linux, LTTng, strace
- See also post https://jvns.ca/blog/2017/07/05/linux-tracing-systems/

4.5 Trace format & trace analysis tools

- See list of formats at https://www.eclipse.org/tracecompass/
- Common trace format: http://diamon.org/ctf/
- trace-diff: https://github.com/ryosuzuki/trace-diff
- Babeltrace: https://github.com/efficios/babeltrace
- OpenMetrics: https://github.com/OpenObservability/OpenMetrics, inspired by the Prometheus exposition format.

4.6 Distributed Tracing

- https://github.com/openzipkin/zipkin/
- https://github.com/Netflix/zuul
- https://github.com/jaegertracing/jaeger
- SigNoz: Open source Observability platform https://github.com/SigNoz/signoz

Proprietary:

- https://lightstep.com/ Infra and app monitoring, distributed tracing

4.7 Integrated / System level

- Nagios
- Zabbix

4.8 Others

- Pin https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
  - https://github.com/SideChannelMarvels/Tracer
  - https://github.com/hasherezade/tiny_tracer
- Uftrace: https://github.com/namhyung/uftrace
- Valgrind: http://valgrind.org/
  - https://github.com/SideChannelMarvels/Tracer/tree/master/TracerGrind
  - https://github.com/wmkhoo/taintgrind
- Instream: https://github.com/dwks/instream/ (x86 trace)
- Panda: https://github.com/panda-re/panda
- DynamoRio: http://dynamorio.org/. See module 'DrCov' for coverage http://dynamorio.org/docs/page_drcov.html
- Dyninst: https://github.com/dyninst/dyninst
- For Rust:
  - https://github.com/anelson/tracers
- For Windows API call: https://github.com/microsoft/Detours

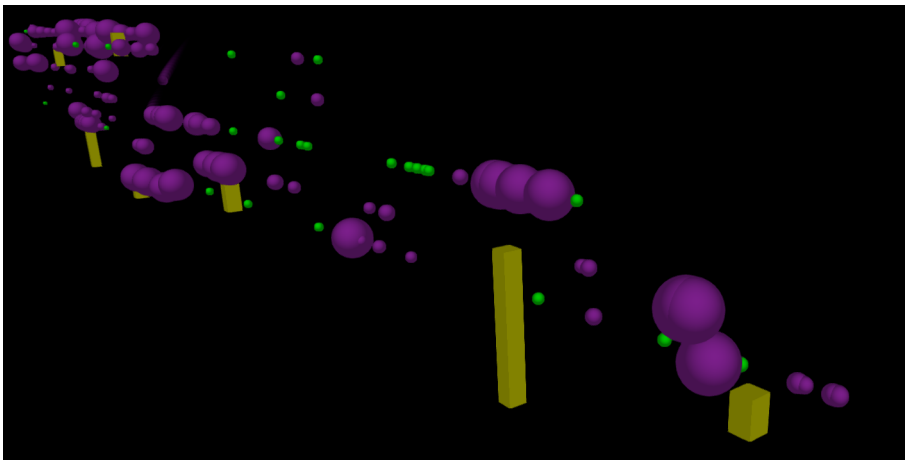


Figure 4: Visualization of execution by Iroh.js. Live video at https://maierfelix.github.io/3d-code-vizard/

5 Visualization

- https://maierfelix.github.io/Iroh/
- https://glowroot.org/
- https://github.com/ncatlin/rgat
- https://thlorenz.com/visulator/
- https://github.com/adrianco/spigo
- https://www.explorviz.net/
- https://github.com/brendangregg/FlameGraph
- https://github.com/chrishantha/jfr-flame-graph
- https://github.com/epickrram/grav

6 Intercession

- Once you do monitoring, the next step is to do intercession (changing/impacting the execution)
- Behavioral change: hot patching
- Fault injection: chaos engineering
- Synthetic monitoring is active monitoring in production, using synthetic workloads.
- Security: capture and block calls to 'eval' (see https://maierfelix.github.io/Iroh/)