

Instrumentation and Performance Analysis of Distributed Systems with Freud

Stefano Taillefert

May 2021

Supervised by
Prof. Antonio Carzaniga

Bachelor Project Report

Abstract

Engineering software systems for performance is a particularly complex task. The performance of a software system depends on many factors, such as the algorithmic nature of the software, the features of the execution environment (including memory, storage, and CPU) and the complex interactions between components, whether internal or external. A common way to support performance analysis is to create models of the system. This can be done, for example, with traditional profilers, which characterize the distribution of the CPU time expenditures on functions and methods.

Freud is a software performance analysis tool that derives similar but more expressive models called performance annotations. In essence, such annotations characterize a relevant performance metric (e.g. CPU time) as a function of one or more input parameters or features. In particular, Freud derives performance annotations automatically based on measurements of a software system. The goal of this project, called *Jung*, is to extend Freud to instrument and collect data from *distributed* systems. This means augmenting the existing implementation to instrument and collect performance metrics from a number of distributed components. Jung applies this instrumentation and collects data through a communication framework, and subsequently links and merges independent component traces using causal relations. The resulting unified trace is then fed into the Freud statistical analyzer to produce meaningful annotations.

Jung can therefore be used to support the performance engineering of applications that use and connect to external resources, such as databases and other services.

Contents

1	Introduction	2
2	Performance analysis	3
3	Project design	5
3.1	Freud	5
3.2	Requirements and analysis	5
4	Implementation	7
4.1	Technologies and tools used	8
4.2	Issues	8
5	Evaluation	9
5.1	Testing	10
5.2	Demo	11
6	Conclusions	12
6.1	Future work and possible developments	12

Chapter 1

Introduction

Freud [1] is a software performance analysis tool that derives performance annotations from measurements of running systems. What does that mean? Freud requires the system in binary form, although it can also benefit from the source code for additional analysis. In essence, Freud performs a dynamic analysis of the system, meaning that it uses information derives from the execution of the system under workloads chosen by the performance engineer. Given the program executable, compiled to include debugging symbols, Freud instruments the system to measure a series of metrics, such as running time and memory usage, while also tracing the function calls and their parameters. Freud then analyzes this data (offline) to produce significant statistical models that characterize the performance metrics as functions of one or more input parameters (or “features”).

The goal of this project is to extend Freud to collect and analyze data from a *distributed* software system. This includes desktop or server applications that might use external services, typically through remote procedure calls (RPC), or a web application components that query external servers such as a database. To highlight the relation with the Freud tool and technique, we decided to name this project Jung — after influential Swiss psychiatrist and psychoanalyst Carl Jung¹ who was a colleague and collaborator of Sigmund Freud.

In essence, Jung instruments a distributed system so that each distributed component would measure various performance metrics of interest. The measurements are initially stored as separate traces within each component. Jung then takes care of collecting and merging these separate traces so as to assemble a unified trace that would then be processed through the statistics module of Freud. In particular, the instrumentation provided by Jung traces related calls *across* distributed components using unique identifiers. The relation between calls amount to causal relations and therefore are used as a basis for the merge operation. Freud uses a powerful binary instrumentation framework called Pin [3]. In the initial prototype we developed during this project, we opted for a simpler solution, and therefore decided to inject our instrumentation directly within the source code, manually. A fully automatic instrumentation could be added in future developments to reduce the overhead of using Jung.

¹https://en.wikipedia.org/wiki/Carl_Jung

Chapter 2

Performance analysis

Performance engineering amounts to measuring, modeling, and ultimately improving the performance of software systems. Performance engineering is crucial for many software systems. For example, a company such as Google that runs global-scale services and applications is very concerned about performance. The response time of applications, such as GMail, can severely influence their adoption and therefore their success or failure. The same is definitely true for web search and many other applications. Even relatively minor speed-ups or slow-downs can make a significant difference in usability and therefore profitability. In sum, performance engineering is important.

Performance engineering is also difficult. Many factors affect performance in many often subtle ways. There is the algorithmic complexity, but there are also other factors, including hardware features and, crucially, the interactions between components within applications, and across applications. For example, a cache system (memory or otherwise) can significantly improve the performance of a system. However, the mixed use of the same caching system by many components or systems can drastically reduce its effectiveness overall.

Performance analysis is primarily a type of dynamic analysis, and therefore it is based on the measurement of running systems. Almost every kind of application can be instrumented to collect a variety of performance metrics. Consistently with the description of Freud, we use the term *metric* to denote the values of performance indicators such as execution time (or “wall clock” time), actual CPU time or cycles, memory usage, lock holding time, and many more. These metrics can be used in various ways and are chosen by the performance engineer accordingly to the type of application and the type of analysis. For example, lock-holding time is not a very significant metric for a single-threaded application, but it might be of crucial importance for a highly parallel application such as most modern client-side and server-side applications. Performance analysis have several tools at their disposal to perform instrumentation, measurement, and summary analysis. For example, there are a lot of existing tools and solutions out there—one of which will be discussed in section 3.1—that can *instrument* the executables: a dedicated tool injects some special code directly in the compiled binary file, following some particular compiler flags. This generated code then measures different parameters, keeping track for example of starting and ending times, memory allocations, and so on. At the end, an information dump is produced to summarize the data, which can then be analyzed in various ways.

In this project we focus on a specific problem and context, namely the collection and combination of measurement traces for the analysis of distributed systems.

```
File: client_log.txt
1 0 do_stuff1 FUNC_START param=int&0 useless=double&12.200000
2 0 do_stuff1 malloc 0
3 0 do_stuff1 pagefault 1214 0
4 0 do_stuff1 FUNC_END
5 0 do_stuff2 FUNC_START param=int&1 useless=double&12.200000
6 0 do_stuff2 malloc 1
7 0 do_stuff2 RPC_start
8 3 do_stuff2 RPC_end 1
9 3 do_stuff2 RPC_start
10 4 do_stuff2 RPC_end 2
11 4 do_stuff2 pagefault 1412 0
12 4 do_stuff2 FUNC_END
13 0 do_stuff3 FUNC_START param=int&2 useless=double&12.200000
14 0 do_stuff3 malloc 2
15 0 do_stuff3 RPC_start
16 1 do_stuff3 RPC_end 3
17 1 do_stuff3 RPC_start
18 2 do_stuff3 RPC_end 4
19 2 do_stuff3 RPC_start
20 2 do_stuff3 RPC_end 5
21 2 do_stuff3 RPC_start
22 2 do_stuff3 RPC_end 6
23 2 do_stuff3 pagefault 1429 0
24 2 do_stuff3 FUNC_END
```

Figure 2.1: An example of performance trace (in this case, produced by Jung)

Chapter 3

Project design

3.1 Freud

This project follows the steps of Freud, a tool developed by Daniele Rogora, an USI PhD student. This software has multiple components; the first two (`freud-dwarf` and `freud-pin`) are tasked with instrumenting the executable via a special tool called PIN [3], developed by Intel. This library—as explained earlier — inserts the instructions necessary to collect the data during the execution. Basically, coding this part means writing code that will write some code to put into other code.

The other part of Freud, which is the one we will be interacting with, is `freud-statistics`. As the name implies, this module (based on the popular R library¹) processes the statistics, resulting in a regression or clustering, given the output of the instrumentation.

3.2 Requirements and analysis

The main idea was to start small and build features incrementally, to ensure that we always had a minimal working prototype. The main objective of Jung is to instrument multiple systems, collect the data and format it. To achieve this, we divided the project in three main components:

- The actual instrumentation library, with an API to track the various metrics
- The merger, which has to collect all the log files from the different systems and merge them in a single coherent trace
- The dumper, which is tasked with creating the binary output that will be passed to `freud-statistics`

The main milestones of the developments were the following: first we had to develop a simple distributed application, based on an RPC library, to be used as an initial test environment. For this part we chose to put together a relatively simple program that sends back and forward some text messages; the server has some delays set up in answering some queries to simulate a long computation.

Following that, we needed to develop an instrumentation for the client side, server side, and — crucially — the RPC library: the idea was to measure some meaningful statistics on all sides involved. The metrics we settled with are:

- Execution time
- Memory usage
- Major and minor page faults
- Lock holding and waiting time
- Possible memory leaks (experimental)

¹<https://www.r-project.org/>

The execution time is computed and accounted for separately between client (total, end-to-end, synchronous), server, and network. All other metrics are computed and accounted for the client and server side. All the chosen metrics except for the memory leak estimate are supported by `freud-statistics`. Possible memory leaks are detected by tracing and matching the amount of `malloc` and `free` calls. However, this computation can only offer an approximate leak detector.

Then, we had to devise a method to save and retrieve the measurement logs from all the components. This has been done by dumping the collected information in a text file with some special encoding (see section 4.2 for details) for data types and more complex values.

The next step was to design an algorithm to merge the logs from all the systems into a single coherent trace. This was tricky because we had to correctly identify all the remote calls to allow to trace back which server execution corresponded to which client function. We don't want to debit someone with the computing costs of someone else, therefore we assigned unique IDs to each RPC call to allow to trace back which client function requested which server resource.

Integrating said trace in the existing statistics tool (`freud-statistics`) to derive the performance annotations was the last objective for the coding part. For this we had to rely on the help of its creator, Daniele, to fully understand how to format the data. There were a few hiccups due to some technical issues, but the process concluded perfectly.

Unfortunately, the only abandoned part of the project was to identify some third-party non-trivial distributed applications and analyze them with the created tool. Due to the fact that Jung doesn't work on the binary executable, we needed to access the source code of the application we wanted to analyze. This heavily reduced the spectrum of possible targets, and also considering the little time left we decided to abandon it.

Last but not least, we had to write the report, prepare the poster and the presentation. Due to the current situation, we as a class decided to not hold any presentation this year, therefore this point got reduced to the document you're currently reading.

Have a pizza: this is still on hold but will definitely be completed sooner or later. Every project needs a celebratory pizza at the end.

Chapter 4

Implementation

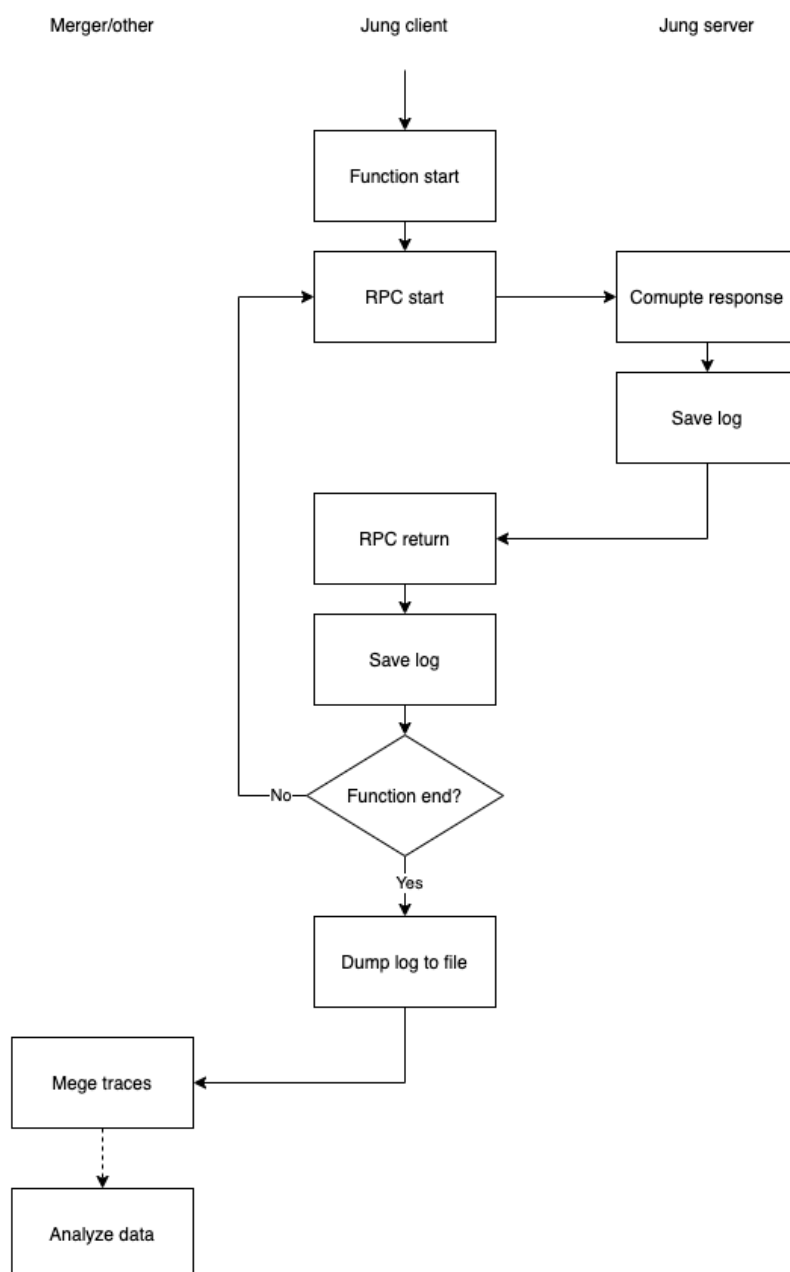


Figure 4.1: Jung's general functioning scheme

4.1 Technologies and tools used

We used gRPC [2] as the RPC library, since it seemed pretty straightforward and simple to use. There are plenty of examples in the documentation and a lot of languages are supported, including C++. On the other hand, Jung is independent from any library: its functions can be used directly from the code, even in a custom RPC implementation.

The choice of language was pretty easy as well: Freud is built entirely in C/C++, so using the same language would increase compatibility. The only choice we had to make was regarding the version: we went with C++17 to benefit from the `filesystem::exists()` function, which we used to check for the existence of previous log files.

To simplify development and testing, we also packaged an auto-building Docker image (<https://hub.docker.com/repository/docker/steeven9/jung>) with the example server in it. This way we could keep the server running on another machine and, for example, test the network times from another remote location.

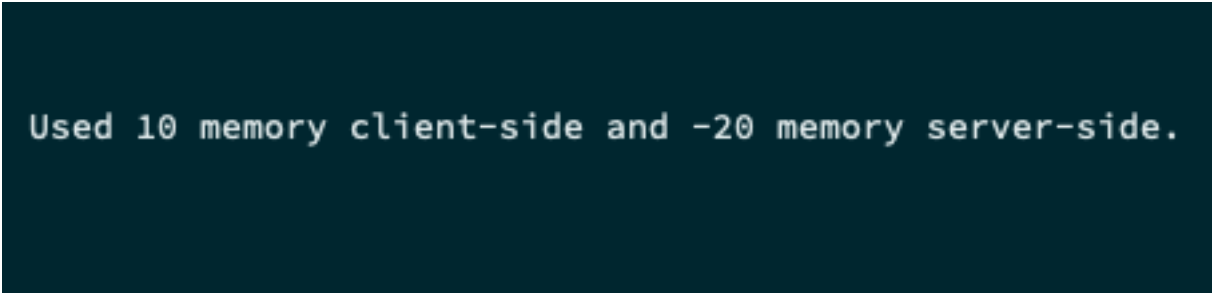
4.2 Issues

The only major problem we had - which we believe to be very common - was that we made some design choices in order to initially simplify the job, but then, with the addition of new features or to fix certain problems, we had to change approach. The direct consequence was that we had to rewrite some large parts of the code and data structures to adapt to the new requirements.

One instance of the aforementioned problem is that we had to develop some particular encoding to represent the function parameters, since we had to convert from data (the running code) to text (the log file) and vice versa. More specifically, we chose to represent the values as `name&value` (e.g. `asd&12` for an integer named `asd` of value 12); this added a supplementary layer of encoding/decoding when saving and reading the data to/from the text files.

When we first started testing the integration with `freud-statistics`, since we was unable to see what was in the binary file, we had troubles checking if we was dumping the metrics correctly, resulting in some weird-looking statistics. With Daniele's help, we managed to make sense of the output and find the offending code parts to patch.

And of course, we had our fair share of miscellaneous issues with pointers, segmentation faults and general weird behaviors. As a positive note, in the process we somehow created a program that frees memory as you run it:



```
Used 10 memory client-side and -20 memory server-side.
```

Figure 4.2: A very peculiar memory usage

Another problem we ran into was that initially we were unable to get `freud-statistics` to find a regression, despite our data was clearly a good candidate for it. Turns out we simply didn't have a good enough p-value, which was easily fixed by getting more samples (details will follow in the next chapter).

Chapter 5

Evaluation

personal experience (?)

Referring to the task list in section 3.2, we can assert that we successfully implemented all the required features. The results the library provides are accurate (to a certain extent) and repeatable.

To test Jung's functionality we created a demo program that simulates a real use-case of this library: a client program (`jung_client`) sends some requests to a server (you guessed it, `jung_server`) which computes the results and sends them back. To make it more realistic, the server takes roughly the time in seconds equal to the parameter received (with some added random margin), since it simulates a computation. Being a single-threaded sever, this has been achieved by making the server thread sleep for the given input. On the other hand, the client uses a for loop to send the requests, so the parameter's value increases gradually; the maximum number of iterations has been set to 20 and the number of points to 5, but those can easily be changed in the code (`NUM_MSG` and `NUM_POINTS` respectively).

Alas, time and coding constraints limited the amount of testing we could do, meaning that the performance analysis has been tested only with a few scenarios. The first one is illustrated in the plot below:

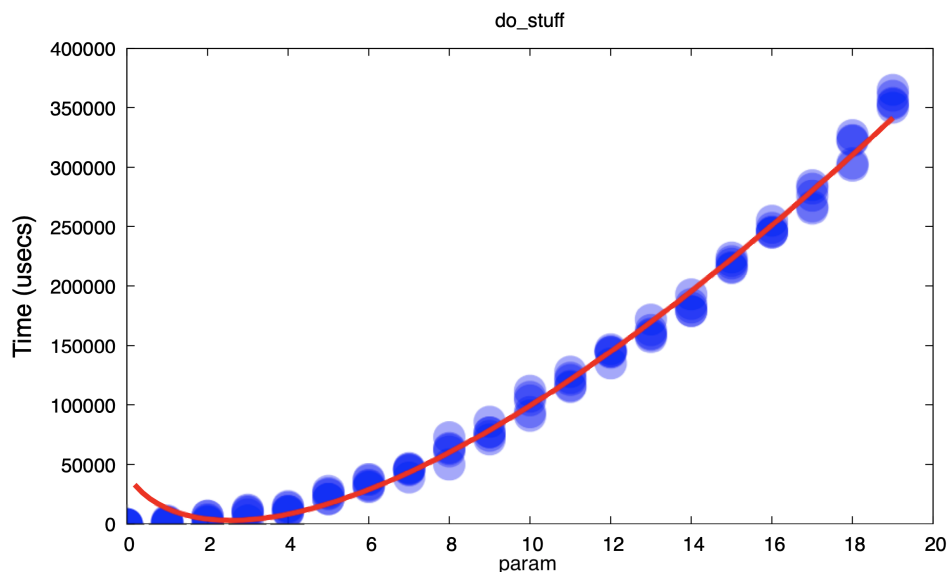


Figure 5.1: The first plot produced by `freud-statistics`

As we can see, our data fits nicely in a quadratic regression, as expected; our client program can be represented with the following algorithm:

Algorithm 1: Quadratic simulation algorithm

```

for i ← 0 to NUM_MSG do
  for j ← 0 to NUM_POINTS do
    for k ← 0 to i do
      sleep(k + random_margin());
    end
  end
end
end

```

Another scenario can be simulated by making the RPC calls a constant amount of times instead of depending on the parameter, which produces the linear result in the plot below:

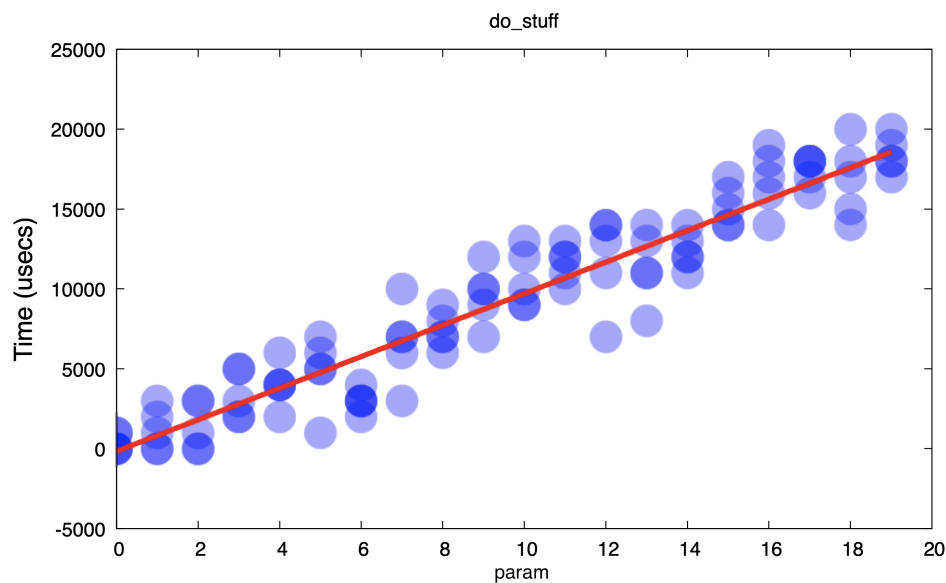


Figure 5.2: The second plot produced by `freud-statistics`

And the corresponding elementary algorithm:

Algorithm 2: Linear simulation algorithm

```

for i ← 0 to NUM_MSG do
  for j ← 0 to NUM_POINTS do
    sleep(i + random_margin());
  end
end
end

```

5.1 Testing

Due to the complexity and particular nature of the software, we decided to not implement any automated tests. Opposed to test-driven development, we had a more unstructured way of iterating over the code, since the functionality had to be checked by hand by comparing the produced log files with the program. Therefore, the effective coverage is 0%, but as someone once said:

"Tests don't prove correctness"

5.2 Demo

In this section you can find instruction on how to run a demo program and analyze the obtained data. To install and compile all the required software, simply run the `install.sh` script from the Jung repository. In case of failure, please refer to the READMEs of each project for more information.

First, start the server:

```
$ ./jung_server
```

While in another shell, run the client:

```
$ ./jung_client
```

Note: if you run the server on another machine, you can pass the `--target=HOSTNAME` sentence above render correctly dunno why leave it like that argument to the client.

Then, with both the client and server log files in the root folder, merge the traces:

```
$ ./trace_merge
```

This will produce the binary data (under `symbols/`) and a summary of the execution (`trace_log.txt`). We can then run Freud's analysis tool:

```
$ cd ../freud/freud-statistics
```

```
$ ./freud_statistics 3 0 0 do_stuff ../../Jung/symbols/do_stuff/
```

Note: this is an example, refer to Freud's documentation for details and usage.

This should find a nice regression and plot it. That's it!

Chapter 6

Conclusions

We consider the obtained results pretty satisfying: we managed to realize a working application that complies with the requirements and could be used in the field to measure actual systems (with some limitations, of course). As the evaluation shows, Jung correctly tracks and aggregates metrics, producing an output which can be interpreted by another tool. Different behaviors are correctly picked up and produce different outputs when analyzed, as expected.

From a more personal point of view, the whole project was a great learning experience, both in terms of development and project management; in addition to refreshing my C++ skills, I experienced carrying out a large-ish project on my own, with deadlines and constraints. I would like to add that the way my advisor and I organized the work was very helpful: small steps and weekly meetings allowed to iterate quickly over the code and to keep relatively on track with the schedule. As a matter of fact, the only discarded feature, testing on third-party applications, was not mandatory for the completion of the project.

Acknowledgements

I'd like to thank Antonio, my advisor, for the opportunity and for guiding me throughout this project; Daniele, Freud's author, for his help understanding his amazing software; my friends Sasha and Brites for their incredible support; and last but not least everyone that supported me during these tiring times. Thank you!

6.1 Future work and possible developments

While this project can be considered a success, there are still a number of improvements that could be implemented. For example a better handling of the data dumping via a separate thread: as of now every function writes the buffered metrics to the log file when it finishes executing. While this is barely noticeable on a small scale, it could add a significant overhead under large loads or in performance-critical systems. A crucial missing part is an actual instrumentation like Freud's, either through PIN or another similar library, allowing the user to work on the executables instead of having to change the source code. Other possible improvements would be to support more complex features as parameters, like structs, automatically infer the type of the parameter (opposed as having to manually specify it), and export the data in a more universal format, like csv, to be able to use it in a wider variety of analysis tools.

Bibliography

- [1] GitHub Inc. `usi-systems/freud`: Freud, a tool to create Performance Annotations for C/C++ programs. <https://github.com/usi-systems/freud>, 2021. Accessed on 15.05.2021.
- [2] gRPC Authors. Documentation | gRPC. <https://grpc.io/docs/>, 2021. Accessed on 27.04.2021.
- [3] Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool. <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>, 2021. Accessed on 22.05.2021.