ODP: An Infrastructure for On-Demand Service Profiling

John Nicol*
jnicol@linkedin.com
LinkedIn Corp

Chen Li*
cnli@linkedin.com
LinkedIn Corp

Peinan Chen*
pchen@linkedin.com
LinkedIn Corp

Tao Feng*
tofeng@linkedin.com
LinkedIn Corp

Haricharan Ramachandra
hramacha@linkedin.com
LinkedIn Corp

ABSTRACT

CPU and memory profiling of services are commonly-used methods to identify potential performance and cost optimizations. However, the tooling solutions for profiling are often nonstandard, not centralized, inconvenient for users, and costly, leading to limited adoption.

Additionally, with projects and companies employing Agile methodologies such as the microservices model, the service diversity, number, and frequency of changes can drastically increase, further limiting adoption due to scalability concerns and needs for varied profiler technologies.

To address these challenges, we present the *ODP* ("On-Demand Profiling") framework. This is a scalable, language- and platform-independent framework designed to enable on-demand CPU and memory profiling of microservices, and centralized storage, sharing, and analysis of the resulting data.

CCS CONCEPTS

• **Software and its engineering → Software performance**;

KEYWORDS

microservices, performance, profiling

ACM Reference Format:
John Nicol, Chen Li, Peinan Chen, Tao Feng, and Haricharan Ramachandra. 2018. ODP: An Infrastructure for On-Demand Service Profiling. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3184407.3184433

1 INTRODUCTION

Service profiling is a form of dynamic program analysis which is commonly used to aid in service optimization. Two primary analyses are CPU and memory profiling.

CPU profiling is a technique to analyze the execution time of methods of a service; this can find service "hotspots" which can be optimized. This technique has a long history; for example, the *prof* command has been implemented in Unix since the early 1970s [18].

Memory profiling can refer to analysis of whole memory (often called "core" or "heap") or to memory allocation events. These techniques are often used to find memory leaks, but also can be used to optimize memory usage. Optimization of memory can also have effects on application latency; this can be due to reduced garbage collection in managed languages, but can also be due to reduced allocation and deallocation events, copying of data, and improved paging and caching.

Although CPU and memory profiling are old techniques, there are few standards for their use, especially across platforms or languages. As a result, unless a profiler tool is supported internally, users may need to configure the tool, acquire licenses, and request installation on remote hosts before profiling. In addition, viewing the profiled results often requires manual data transfer or setting up a tunnel from the production environment to the development environment. Furthermore, sharing and comparing profiling data is difficult or effectively impossible, especially when profiling runs are captured by different users with different profiler settings, or even different profiling tools.

Additionally, companies and projects have begun to adopt Agile software architectures such as microservices, where software applications are designed as suites of independently deployable services [7]). (LinkedIn uses a microservice and continuous deployment model itself [1]. Thus for the remainder of the paper, we will use the terms "service" and "microservice" interchangeably.)

Although microservice architectures are very useful, they can drastically increase the total count of services, their diversity, and the frequency of their changes and deployments. More services and more frequent deployments of those services lead to more frequent profiling; this generates more data, thus leading to challenges in scalability. Higher diversity in services can also necessitate more varied profiler technologies. These further limit profiler adoption due to scalability concerns and needs for varied profiler technologies.

The scalability concerns for microservice profiling are not limited to hardware and software; there are user scalability concerns as well. Each microservice requires independent optimization, and versioning of dependencies becomes a significant issue. For example, if a newer version of a library is optimized, not all of the services depending upon that library will immediately choose to consume this new version; service owners will often be unaware of the optimizations, or won't prioritize the upgrades unless they're known to improve the service behavior.

*These authors contributed equally to the work.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184433

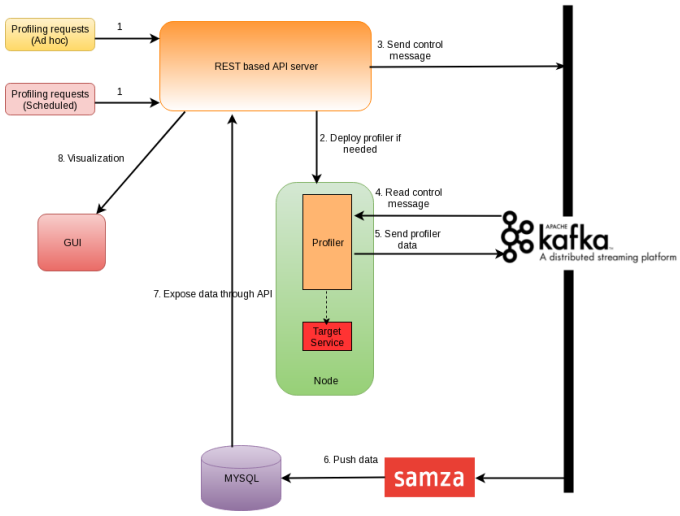


Figure 1: High-level architecture of the ODP framework

Clearly there are challenges in significant adoption of profiling in the Agile methodology. But there are huge benefits as well. Rapid changes allow for rapid optimizations; improvements to libraries, for example, can quickly be consumed by all dependers, resulting in profound optimizations from simple analyses.

We have developed the *ODP* ("On-Demand Profiling") framework to address these challenges.

The ODP framework is itself a collection of microservices. It consists of: RESTful APIs for starting and stopping profilers and for accessing reported data; profiler plugins for different profiling modes, languages, and platforms; scalable messaging and data storage; scalable data processing (including Common Issue Detection); and a GUI for users to visualize and share results.

2 ARCHITECTURE

We now present the high-level architecture of our solution (the ODP framework). As shown in Figure 1, the key components are the Profiler request, the Profiler API server, a Profiler service with profiler plugins, a messaging backbone, and a data pipeline and datastore.

A user or a scheduled job (from an arbitrary host machine) requests a specific service ("target service", also called "STP" for the Service To Profile) be profiled on a specified host machine ("target host").

The request to profile is passed to a REST-based API service. The API service deploys the Profiler service on the target host if necessary, and then signals the Profiler service on the target host to profile the STP.

The Profiler sends its data through a scalable pipeline. After post-processing, the data can be viewed via a GUI, or consumed directly via another REST-based endpoint.

2.1 Profiling requests

Profiling requests can come from both users and approved services (e.g., automated performance testing). In addition to on demand requests, profile requests can be scheduled for regular events, such as traffic shifts. Profiling requests are simple HTTPS calls in REST style.

Requests can be made to start or stop STPs on specified hosts. Additionally, requests can be made to simply deploy or undeploy the Profiler service, without actually executing plugins. All requests and their current state are stored within the Profiler database for monitoring and analysis.

The framework requires authentications to protect services from untrusted or duplicated profile requests. Once a request is verified, the framework will take care of additional authentications in production clusters to ease developers' job.

For flexibility, the framework supports authenticated requests from all hosts in the network, and allows requests to any hosts that are approved to run the Profiler service.

2.2 REST-based API

Our REST-based API service acts as an interface for the profiling requests. The profiling requests come from CLIs (Command Line Interfaces) or GUIs, or are scheduled jobs stored within the Profiler database.

For a "deploy" request, the API service queries the target host (or the Profiler database) to determine if a Profiler service is already running, and if not, deploys the Profiler service.

For a "start" request, the API service executes a "deploy" request, and then sends a command to the target host's Profiler service via an Apache Kafka message (see Section 2.5) in order to start a profile of the STP.

For a "stop" request, the API service sends a similar Kafka message to the target host's Profiler service to stop a profile of the STP (if it is running).

For an "undeploy" request, the API service queries the target host (or the Profiler database) to determine if a Profiler service is already running. If it is, the API service can either undeploy the service immediately (thus terminating any ongoing profiles on that host), or wait until all profiler requests have finished on the target host.

2.3 Profiler Service

The Profiler is a microservice that's either deployed on-demand or is continuously running on target hosts where STPs are requested. Each language-specific CPU profiler and each language-specific memory profiler is implemented as a plugin of the Profiler.

The Profiler polls for Kafka messages from the Profiler API service. Then it verifies that those messages are authenticated, and activates itself when a message to profile a service on the given host (the STP) arrives. The Profiler verifies the existence of the requested STP on the host, and determines its characteristics (e.g., process ID, language, version, and flags).

Based on these characteristics, the Profiler executes the appropriate profiler plugin. If multiple profiling requests arrive contemporareneously to the same host, multiple plugins (or multiple instances of the same plugin if necessary) can be invoked.

2.4 Profiler Plugins

Each Profiler plugin can profile one or more combinations of profiler mode (e.g., CPU, memory allocation tracking, heap dump), platform (e.g. Linux, MacOS, or Windows), and language (e.g., JVM, .NET, C++, or Python).

There are varied technologies and techniques that can be used to profile such as [12] and [14]. The ODP framework is agnostic to these choices.

The data generated from a profiling plugin often has high redundancy, as the stack traces or classes (for memory allocation profiling) are expected to recur frequently. To avoid the potential for network congestion, this data is periodically aggregated, which dramatically reduces the data redundancy. As aggregation will take CPU and memory resources, the aggregation frequency must be carefully balanced to ensure that the Profiler's CPU and memory footprint remain small.

This aggregate data is then sent to Kafka for later consumption by the data pipeline; see Sections 2.5 and 2.6. Some profiler plugins cannot periodically aggregate (for example, heap dumps) and can only send all of their data at profile completion. This is supported functionality, but additional care must be taken within such plugins to not overload memory or network considerations for the Profiler service.

A lightweight sampling-based JVM CPU/memory profiler is currently used at LinkedIn, which supports late attach functionality and avoids restarting service in production environment before/after profiling requests.

2.5 Messaging Backbone

We use Apache Kafka both to send command messages to the Profiler instances and to publish profiling data generated by the profiler plugins.

Kafka is a distributed streaming platform that, among other functionality, can "read and write stream data like a messaging system" [3] and "store streams of data safely in a distributed, replicated, fault-tolerant cluster" [3]. Kafka also supports authentication and SSL encryption/decryption, which resolves potential security concerns such as message spoofing.

2.6 Data Store

The profiling data sent to our Kafka topic is processed and written to a data store via Apache Samza [4]. Samza is a scalable, lossless, streaming processing framework that uses Kafka for messaging; if multiple Profiler services send data simultaneously to the Kafka topic, Samza catches up with the produced messages and pushes them to our remote data store.

We have chosen MySQL [19] for our data store. This is a very popular and well-supported database technology. It works well for our needs (simple tables, low/medium traffic, large storage). A simple sharding technique allows scaling for the storage of profiler data.

To ensure a manageable database size, we have scheduled jobs to deduplicate redundant data and delete old data.

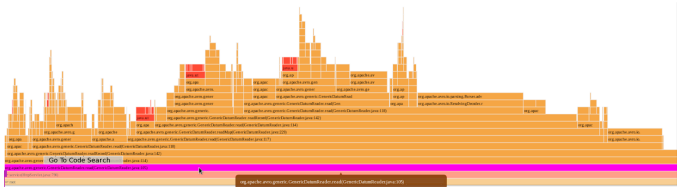


Figure 2: A Flame Graph: cell width indicates percentage of method within the graph; higher cells are child methods of lower cells. Note the clicked context menu linking to a code search tool.

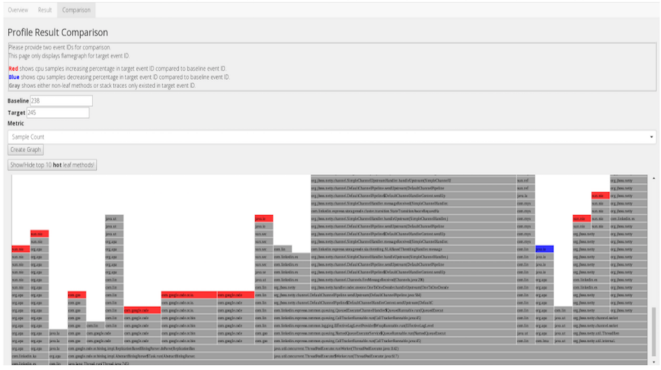


Figure 3: Comparison Flamegraph: Red coloration at leaf nodes indicates an increase compared to the baseline, while blue indicates a decrease.

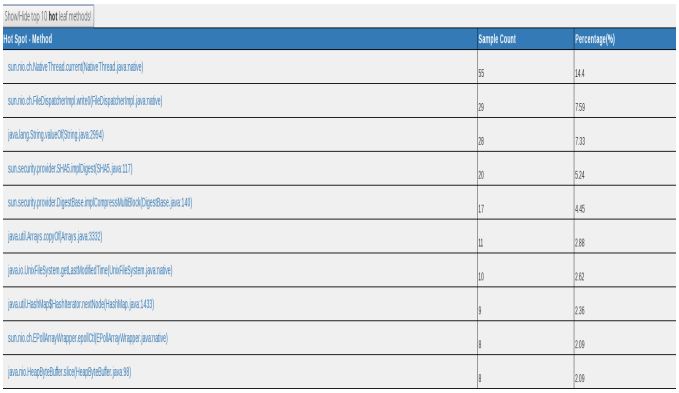
2.7 Visualization

The CPU and memory profiling results are visualized through an internal web application. Visualizing via a native application would also be an option, as would be a plugin to an IDE (Integrated Development Environment), but we have found the current approach to be simple and to allow easy sharing of URLs.

These profiling results and visualizations have been quite popular at LinkedIn; similar ones have also experienced success at other companies [13]. As a result, there are a great deal of open-source resources to help implement elements of this web-based visualization ([13]).

The visualization is done with Flame Graphs [13], see Figure 2. The flame graph is rendered as an interactive SVG (Scalable Vector Graphic) allowing easy zooming into and out of stack traces. Additionally there are options to filter and highlight based on regular expressions, and functionality to compare profiling results (see Figure 3). Some teams have used the comparison functionality to generate regular performance assurance reports to guard against regressions, see Section 4.

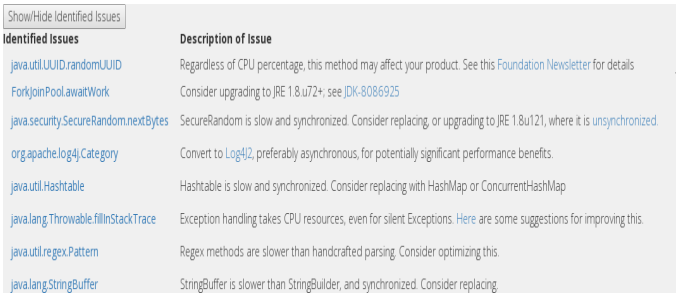
There is also a context menu associated with each cell. This menu contains contextual information for both the cell's method and the full stack trace. From this menu we have both: a link to a code-search tool, which allows us to view the source code associated with a given stack trace (internally we use [5], but there are certainly



Show/Hide top 10 **hot** leaf method(s)

Hot Spot - Method	Sample Count	Percentage(%)
sun.nio.ch.NativeThread.current(NativeThread.java:native)	55	14.4
sun.nio.ch.FileDispatcherImpl.write0(FileDispatcherImpl.java:native)	29	7.59
java.lang.String.value(Of(String.java:2964)	28	7.33
sun.security.provider.SHA5.implDigest(SHA5.java:117)	20	5.24
sun.security.provider.DigestBase.implCompressMultiBlock(DigestBase.java:141)	17	4.45
java.util.Arrays.copyOf(Arrays.java:3332)	11	2.88
java.io.UnixFileSystem.getLastModifiedTime(UnixFileSystem.java:native)	10	2.62
java.util.HashMap$HashIterator.nextNode(HashMap.java:1433)	9	2.36
sun.nio.ch.EPollArrayWrapper.epollCtl(EPollArrayWrapper.java:native)	8	2.09
java.nio.HeapByteBuffer.slice(HeapByteBuffer.java:90)	8	2.09

Figure 4: Hot Leaf Calls: A list of the most common "leaf" methods in a profile, linked to their occurrence within the profiled service.



Show/Hide Identified Issues

Identified Issues	Description of Issue
java.util.UUID.randomUUID	Regardless of CPU percentage, this method may affect your product. See this Foundation Newsletter for details
ForkJoinPool.awaitWork	Consider upgrading to JRE 1.8.u72+; see JDK-8086925
java.security.SecureRandom.nextBytes	SecureRandom is slow and synchronized. Consider replacing, or upgrading to JRE 1.8.u121, where it is unsynchronized.
org.apache.log4j.Category	Convert to Log4J2, preferably asynchronous, for potentially significant performance benefits.
java.util.Hashtable	Hashtable is slow and synchronized. Consider replacing with HashMap or ConcurrentHashMap
java.lang.Throwable.fillInStackTrace	Exception handling takes CPU resources, even for silent Exceptions. Here are some suggestions for improving this.
java.util.regex.Pattern	Regex methods are slower than handcrafted parsing. Consider optimizing this.
java.lang.StringBuffer	StringBuffer is slower than StringBuilder, and synchronized. Consider replacing.

Figure 5: Common Issue Detection: A list of issues autodetected by ODP, linked to their desriptions and their occurrence within the profiled service.

others such as [11]); and a link to crowdsource known issues, see Section 3.

Additionally, we have a table-based "hot leaf call" (see Figure 4), which is often more convenient than the graphically-intensive flame graph.

3 COMMON ISSUE DETECTION

Many of the issues found during analyses of profiling results have been found to pop up repeatedly for different microservices. This can result in a great deal of repeated analyses and differing optimizations as common performance patterns (and their solutions) are rediscovered. This becomes even more noticeable for larger and decentralized groups for whom information sharing is more difficult.

We have developed a crowdsourcing technique and technology that helps alleviate this [15]. By centralizing the repeated issues and automatically detecting them, we can save valuable development resources. This technique is applied to both future profile results and to previous results, which are re-analyzed to link to newly-discovered patterns.

Additionally, these detected patterns can be sent directly to affected teams; this may be of use when, say, a profiler result is generated on a scheduled basis and a newly-detected issue is not noticed by a user. If an issue is found to occur in a library that is consumed by several teams, this auto-alerting can help prioritize a resolution to the underlying library problem.

Issue detection can be applied to aggregate stack percentages (as in Figure 5), with a customizable threshold percentage to avoid flagging low-impact issues. It can also be applied to detect when certain library versions or runtime options are used by the profiled service.

4 RESULTS

The ODP framework has gained significant traction at LinkedIn. In sixteen months, it has grown to incorporate analyses of roughly 150 of the services used in production. This represents well over 50% of the critical, expensive, or heavily-used services (e.g., those with a high number of instances, custom or particularly expensive hardware, or those services directly corresponding to user latency). This adoption rate compares quite favorably to that of other LinkedIn performance tools such as Redliner [20].

Dozens of improvements have been applied based upon analyses from ODP profile results. Most of these have been applied to common internal libraries, thus affecting many services. Some have even been applied to external open-source libraries, helping the larger software community.

The effects of these improvements are not always easy to predict; for example, some improvements based on CPU profiling may have dramatic effects on average latency, some on P99 latency, some on service throughput, and some on CPU usage. Improvements based on memory profiling may reduce overall memory usage or reduce time spent in garbage collection; these will have effects on latency, throughput, and CPU usage as well, but again those effects are often hard to predict.

The results at LinkedIn have been undeniable, however. A synchronization fix led to a roughly 40% increase in throughput of a critical service. A series of smaller improvements led to a roughly 25% increase in throughput of another critical service. Both of these improvements have enabled reductions in hardware needed to support those services, resulting in substantial cost savings for the company.

Scheduled profile results helped detect and mitigate a library regression, which if it had made it to production, would have reduced a critical service's throughput by 40% and would have likely caused cascading failures.

Other improvements have resulted in significant latency reductions in critical services: over 10% average latency reduction for one, and over 20% P99 latency reduction for another.

A key point: many of these improvements were made to internal libraries (and some to open-source external libraries). As such, there are synergistic effects to other services as well.

5 OVERHEAD

As with other profilers, ODP adds some overhead to services. The overhead varies depending on the service, but we found that the current profiler plugin in ODP generally adds 11% to 13% throughput overhead based on SPECjvm2008 benchmarks [9], and less than 5%

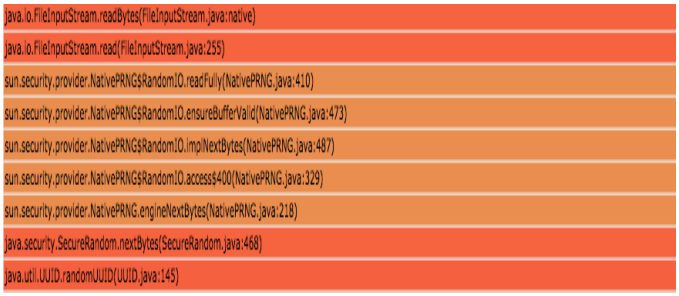


Figure 6: Case study: problematic stack traces.

latency overhead on LinkedIn servers. ODP also adds network overhead when it sends results to Kafka. To reduce this, ODP cuts and compresses results to small snapshots and sends them periodically at a configurable rate.

6 CASE STUDY

An example of how LinkedIn has used ODP is with the backend service of the People You May Know (PYMK) feature of LinkedIn. In order to diagnose why this service was experiencing high latency in production, its service owners used ODP to capture a CPU profile. Through their profile, they found that the routine shown in Figure 6 occupies more than 18% of CPU time for their service.

This routine involves using the UUID.randomUUID call to generate tracking entity IDs in a highly concurrent fashion. UUID.randomUUID internally uses NativePRNG to generate random numbers which uses randomness from /dev/urandom and has a global lock within the implNextBytes method (as shown in Figure 6).

To address this problem, we proposed that the service owners write their own UUID generation call based on another non-blocking security algorithm which we found provides comparable randomness and security with improved performance benefit. After this change, about 12% of 90% latency has been reduced based on production data.

7 RELATED WORK

Profiling and debugging services that are running in production environments is a critical requirement for projects and companies that need to resolve performance regressions.

7.1 Profiling tools

There are standalone profilers that support remote profiling [10] [8], but production clusters usually require additional authentication checks and make it difficult or impossible to use those standalone profilers directly.

7.2 Frameworks

[1]Frameworks			
Features	ODP	GWP	Vector
On demand/Always on	On demand	Always on	On demand
CPU information	Yes	Yes	No
Memory information	Yes	No	No
Machine level metrics	No	Yes	Yes
Pluggable profilers	Yes	No	No
Flexible sampling rate	Yes	No	No

Frameworks similar to ODP have been created for system and application metrics (such as Netflix's Vector [2]) as well as for daily or scheduled profiles of a sample of hosts (such as Google's Google-Wide Profiling [16]). The ODP framework differs from these as it focuses on diagnosing performance problems through on-demand requests rather than monitoring the state of systems – although it is occasionally used for that purpose. The most similar framework that could be found is Alibaba's ZProfiler and ZDebugger [6], which provides profiling and detected issue sharing features.

8 FUTURE WORK AND CONCLUSION

For certain languages and platforms, employing particularly lightweight profilers such as Linux Perf [12] would enable an "always-on" approach, which would help support continuous performance assurance. An always-on approach would significantly increase data processing and storage, but the scalability of the ODP framework allows this. Unlike similar efforts [16], we plan to allow both on-demand and always-on requests. This is a straightforward generalization: one can think of "always-on" as simply a long-running on-demand request.

Profiling of embedded devices (for example, mobile devices) represents a challenge. There exist profiling tools for devices, emulators, and simulators, but we are not currently aware of any that could be adapted as a Profiler plugin.

The technologies for virtualized or containerized services are in flux in the industry [17]. Some of these technologies would require additional permissions or additional steps to ensure the STP is visible to the Profiler service and its plugins; as such, we must stay in sync with those efforts.

In this submission we have presented the ODP framework, our solution for on-demand CPU and memory profiling of microservices. This is a scalable, language- and platform- independent framework that has been widely adopted within LinkedIn.

REFERENCES

[1] 2013. The Software Revolution Behind LinkedIn's Gushing Profits. (April 2013). https://www.wired.com/2013/04/linkedin-software-revolution/
[2] 2015. Introducing Vector: Netflix's On-Host Performance Monitoring Tool. (April 2015). https://medium.com/netflix-techblog/introducing-vector-netflixs-on-host-performance-monitoring-tool-c0d3058c3f6f
[3] 2017. Apache Kafka. (Sept. 2017). http://kafka.apache.org
[4] 2017. Apache Samza. (Sept. 2017). http://samza.apache.org
[5] 2017. JARVIS: Helping LinkedIn Navigate its Source Code. (Sept. 2017). https://engineering.linkedin.com/blog/2017/08/jarvis--helping-linkedin-navigate-its-source-code

[1]Data are based on existing paper/doc, and might not reflect the current state of the frameworks

[6] 2017. Java at Alibaba. (Sept. 2017). https://jcp.org/aboutJava/communityprocess/ec-public/materials/2017-02-14/Java_at_Alibaba.pdf
[7] 2017. Microservices: a definition of this new architectural term. (Sept. 2017). https://martinfowler.com/articles/microservices.html
[8] 2017. NetBeans. (Sept. 2017). https://profiler.netbeans.org/docs/help/5.5/attach.html
[9] 2017. SPECjvm2008 Benchmark. (Dec. 2017). https://www.spec.org/jvm2008/
[10] 2017. YourKit. (Sept. 2017). https://www.yourkit.com/docs/java/help/attach_agent.jsp
[11] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 681–682.
[12] Arnaldo Carvalho de Melo. 2010. The new linux perf tools.
[13] Brendan Gregg. 2017. Flame Graphs. (Sept. 2017). http://www.brendangregg.com/flamegraphs.html
[14] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2015. Lightweight Java Profiling with Partial Safepoints and Incremental Stack Tracing. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2668930.2688038
[15] John Nicol, Chen Li, Peinan Chen, Tao Feng, and Hari Ramachandra. 2017. Common Issue Detection for CPU Profiling. (Sept. 2017). https://engineering.linkedin.com/blog/2017/09/common-issue-detection-for-cpu-profiling
[16] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* (2010), 65–79. http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68
[17] Mathijs Jeroen Scheepers. 2014. Virtualization and containerization of application infrastructure: A comparison.
[18] Ken Thompson and Dennis M Ritchie. 1973. *Unix Programmer's Manual* (4 ed.). Bell Telephone Laboratories, Inc.
[19] Michael Widenius and David Axmark. 2002. *MySQL reference manual: documentation from the source.* " O'Reilly Media, Inc.".
[20] J. Xia, Z. Zhuang, A. Rao, H. Ramachandra, Y. Feng, and R. Pasumarti. 2017. RedLiner: Measuring Service Capacity with Live Production Traffic. In *2017 IEEE International Conference on Web Services (ICWS)*. 628–635. https://doi.org/10.1109/ICWS.2017.75