Research Statement

Xu Zhao

My research interests are in building tools to automate postmortem failure diagnosis in distributed software systems. Distributed software systems are used to power many of the most widely used software services, such as Google and Facebook. Production failures in these systems can be catastrophic. For example, Google's blackout in 2013 caused a 40% drop in global Internet traffic. More recently, an hour-long outage of the Amazon website on its 2018 Prime Day caused the company to lose a hundred million dollars of revenue. When a failure occurs, diagnosing it to recover the service is the top priority.

Unfortunately, diagnosing failures in a production environment is notoriously difficult and extremely time-consuming. In most production failures, system logs provide the only source of diagnostic information. Therefore, engineers have to map the printed logs to the program code in order to understand the failure code path. This process is highly inefficient and engineers could be misled by logs irrelevant to the failure. Consequently, developers today spend 60% of their development time on debugging.

My research goal is to automate failure diagnosis and minimize service downtime. Towards this goal, I have been working on two lines of research. First, I pioneered the area of non-intrusive distributed system failure diagnosis. Existing diagnosis tools, such as interactive debuggers and dynamic tracing techniques, are all intrusive. These intrusive approaches require the target software to be instrumented, which incurs non-negligible performance overhead in production environment. Meanwhile, we observe that system logs have rich diagnostic information and can be acquired easily. This idea led to two research projects, *lprof*[4] and *Stitch*[3]. *lprof* is able to reconstruct software execution flow from the logs by analyzing the code of a single software component. *Stitch*, on the other hand, is capable of reconstructing the execution flow across the entire software stack by utilizing the *Flow Reconstruction Principle*, a principle that programmers intuitively follow in postmortem debugging.

The efficacy of postmortem debugging ultimately relies on the quality of logs. However, software logging has typically been a black art; there is no best practice guiding developers on where to log. The log printing code usually undergoes frequent reviews and modifications before it finally matures. The second line of my research is to systematically improve the quality of software logging. We built a tool, *Log20* [2], that can determine the optimal placement of log printing statements under a performance overhead budget without any human involvement. The key challenge was to measure the informativeness of logs, which we addressed by applying the ideas from information theory.

My research have had real world impact. *lprof* has been licensed by a world-class IT company as part of their developers' debugging toolset. *Log20* was featured by *the morning paper*, and developers from AngularJS, the Opera web browser, and Amplifinity Inc. have started projects to use *Log20* as the logging system. *Log20* has also been acquired by multiple research groups and open source projects, which in turn, will provide new sources of inspiration for future research.

Non-intrusive Distributed System Failure Diagnosis

My research is based on two key observations. First, to diagnose a failure, developers must understand and reconstruct the failure execution path. Second, systems already output information about failure execution paths in the logs: otherwise, developers wouldn't have enough information to debug. The key challenge of automating this process is that logs from the concurrent execution paths are intertwined and interleaved in the log files. The insight behind *lprof* [4] is that execution flow can be reconstructed by analyzing the program code. Specifically, developers will insert variables in the log printing code to serve as an identifier for an execution path. *lprof* is able to reconstruct an execution path by grouping the logs contain the same identifier.

My second research approach is to embody users' intuition on how to reconstruct the execution flow across the software stack by correlating the logs. After two years of research, we were able to condense our insights into the following *Flow Reconstruction Principle*:

- Developers will output logs at critical points in the control path so as to enable postmortem diagnosis.
- Developers will output object identifiers in the logs to differentiate concurrent executions.
- Developers will output a sufficient number of identifiers in the *same* log to uniquely identify the object involved.

It is a strong principle: every developer should follow it in order to effectively reconstruct all execution flows a posteriori.

The Flow Reconstruction Principle enabled us to build *Stitch*, a tool that is radically different from all existing tools. Since the Principle implies that only identifiers are necessary to reconstruct an execution flow, we can simply treat each log as a set of identifiers and ignore all string constants. *Stitch* works in three steps. First, it analyzes the system executables on each cluster node in order to parse the identifier values from the logs. Second, *Stitch* sends all parsed logs to a centralized server, where it analyzes the identifier relationships to recognize system objects and their hierarchical relationships such as object creation. Finally, *Stitch* displays results in a web-based user interface that provides intuitive system run-time information such as object hierarchies and their lifetimes.

Strengthening Software Logging Quality

Logs are indispensable in failure diagnosis, but printing logs introduces performance overhead. Developers have to carefully choose *where* to print a log to achieve the optimal balance between performance overhead and informativeness of the log printing statements (LPS) for postmortem debugging. Consequently, manual logging is hard to get right. We conducted a study on 14,863 LPSes from three popular open source distributed systems. On average, each LPS is modified 1.93 times, and 42% of the LPSes were modified at least once since they were first introduced. Our study also shows that a large number of LPS revisions were about changing the verbosity level, where we found developers frequently debated about what is the correct verbosity. This highlights the difficulty of choosing the right balance between performance overhead and informativeness when deciding where to place LPSes.

The key challenge of the LPS placement problem is how to define and quantify the informativeness of an LPS placement. The informativeness of an LPS placement corresponds to its ability to disambiguate execution paths as a result of the log sequence printed by this placement. Moreover, we quantify informativeness by applying the concept of *entropy* from information theory; this takes the probability of each execution path into consideration. Intuitively, an LPS placement with an entropy value H indicates programmers have 2^H possible paths to disambiguate during postmortem analysis. For example, the most informative LPS placement will have zero entropy, meaning each unique program path outputs a unique log sequence.

By using *entropy* to quantify the informativeness of any LPS placement, we were able to design and implement *Log20* [2], a tool that gives a near-optimal LPS placement under a user specified performance overhead threshold. First, *Log20* collects the basic-block level execution paths in the system by inserting tracing instructions. Second, after a user specifies an overhead threshold, Log20 uses a dynamic programming algorithm to find a optimal LPS placement, i.e., a set of basic blocks to log such that they will generate log sequences achieving the smallest entropy under the current workload. Finally, Log20 will deploy the new LPS placement by hot patching the system's bytecode.

Future research directions

My ultimate goal is to fully automate the process of failure diagnosis by identifying the principles that developers intuitively follow in debugging. In our characteristic study [1], I reproduced tens of real-world distributed software failures. My observation is that failure diagnosis is a highly mechanical process that can be automated by the programs which precisely capture the intuitions developers unconsciously follow in the manual debugging process.

One of my next steps is to embody the developer's intuition about where to place logs. *Log20* solves the problem of "where to log" by finding the LPS placement that has the smallest entropy under a specified overhead threshold. However, it relies on runtime traces of execution paths and is not flexible to workload changes. On the other hand, programmers "predict" which basic blocks are the most log-worthy at development time. I would like to explore the intuitions behind programmers' logging decisions and develop new technology to predict these basic blocks and give logging recommendations at the software development stage.

Another direction is to extend the idea of non-intrusive diagnosis to other types of failures such as network failures. Diagnosing network failures is challenging because network devices such as switches and routers are usually integrated as part of the hardware firmware and the code cannot be easily accessed or modified. Developers have a limited source of information when debugging network failures. Therefore, I believe it is be a great application scenario for the non-intrusive failure diagnosis techniques.

Today, it is common practice that developers diagnose software system failures with manually inserted log printing statements and seek through the logs to understand and reconstruct system execution flow. Such a practice heavily relies on developer's human expertise and is highly inefficient. A prolonged failure diagnosis time means longer service downtime, which to a lot of companies is a matter of life or death. Now is the time to revisit this process and develop novel tools to fully automate the failure diagnosing process.

References

[1] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 14)*, pages 249–265, Broomfield, CO, USA, 2014. USENIX Association.

[2] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP' 17)*, pages 565–581, New York, NY, USA, 2017. ACM.

[3] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 16)*, pages 603–618, Savannah, GA, USA, 2016. USENIX Association.

[4] X. Zhao, Y. Zhang, D. Lion, M. FaizanUllah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI' 14)*, pages 629–644, Broomfield, CO, USA, 2014. USENIX Association.