;login:
OCTOBER 2014 VOL. 39, NO. 5
ELECTRONIC SUPPLEMENT

Conference Reports

WiAC '14: 2014 USENIX Women in Advanced Computing Summit

June 18th, 2014, Philadelphia, PA

Summarized by Amy Yin

Opening Talk: Becoming a Researcher: Practical Strategies for Taming the Angst and Changing the World

Professor Jeanna Matthews, Clarkson University

Jeanna, a program co-chair for WiAC '14, commenced the 2014 conference with lessons about being a researcher that can be generalized to industry. As an associate professor at Clarkson with stints at VMware and Intel, Jeanna has been successful in both realms, and she focused her talk on how to find and solve problems by using community to your advantage. She stressed picking good conferences, joining communities, and learning from existing work as keys to success.

Research is angsty. Jeanna could not have stressed this more. It is one of the first times in your life that you will have to define your own questions and then prove to yourself and others that you have solved these questions. It is not enough to solve a problem and leave others to write up the solution or figure out how to apply the results. However, joining a community can be hugely helpful at all stages of this process.

The first step: Scope out communities by identifying good conferences. Find out who is going to be at the conference; read the titles of the papers and the sessions, and the names of the program committee and the other likely attendees. They will help you form a targeted list of questions. One of my favorite pieces of advice: When you get coffee, don't talk about the weather! Every moment is a valuable networking opportunity. If you don't like the idea of networking, think of it as an open opportunity to pick the brains of people you want to emulate.

Finding a community early on makes identifying an important problem that much easier. Figure out what the community knows by reading, going to conferences, and talking to others in the field. Read with purpose and with varying levels of depth. At the very least, Jeanna recommended reading the titles, authors, and abstracts published in the past 5–10 years and talking more deeply about the pieces that interest you in reading groups with peers. Once you figure out what the community wants to know but doesn't yet, make one of those things your area of research. In my opinion, this is similar to what you can do in industry. Look at the commits and codebase of your team. Read your teammates' in-company documentation, review new and old source code, and figure out what issues have arisen in the past, whether through conversation, old meeting notes, or company-wide agendas. Figure out what your team has done and what it still needs to do. Then do it without being asked (or at least propose the idea so that your teammates know you are on the same page as they are).

In contrast, it is much more difficult to find a problem and then a community. By finding a community first, you allow yourself to be guided to a question with genuine curiosity surrounding it, even by people you haven't met before! Look at what a few researchers in that community are doing—what are they publishing? What have they done? And follow them closely. You never have to speak (although at some point you might want to!), but their work can inform and guide your own contributions. I see the same for programmers—find someone whose career success at your own company you would like to emulate, and figure out how that person got there. You can do this by studying code, proposing ideas, talking to that person or colleagues, following him or her on social media, or reading the books that you know have influenced that person.

After explaining the importance of community for choosing a research question, Jeanna went on to discuss her Repeated Research model, in which you position yourself for follow-on work. She challenges you to examine papers and ask, "Do I understand how this was generated and what 'gotchas' might be hiding?" Ask yourself, if I were the researcher and had this setup and equipment, could I have done better? What else would I have tested? Don't be afraid to repeat research when you feel you can take it meaningfully further! Further, look for methods, not just results. Jeanna always asks herself, What data did they use in this paper? What systems?

Finally, get concrete. Get your hands dirty and play with some APIs, write and throw away some code, and have something cooking on the side so that if you get frustrated with one project, you can focus on another.

In all Jeanna could not stress more that, yes, research is hard. If we knew the answer, then it wouldn't be research, but with the tricks detailed above, she hopes that entering research will seem less perilous. The talk resonated deeply with many women at the conference, including those in industry. I later overheard a PhD student comment, "I wish I had gone to this talk BEFORE I did five years in graduate school!" and Jeanna said she wished she had, too!

Diagnosing Production-Run Concurrency-Bug Failures

Professor Shan Lu, University of Wisconsin-Madison

Professor Shan Lu opened the second talk of the day on a personal note. The name Shan means "mountain" in Chinese, and her father gave her this masculine name because he always had thought that he would have a son. Shan's family hails from Heifei, the largest city in Anhui province in Eastern China, and she has a one-year-old daughter with whom she plays *Animal Crossing*, a life simulation video game. Shan talked about how, in the game, she has to shake the apple trees to collect the apples for money to upgrade her house, but sometimes the computer would freeze and lose all her tree-shaking progress. However, she doesn't like to save her progress because to do so she has to go to her virtual home and sleep.

The story was a funny metaphor for the subject of the talk: bugs and debugging. Shan started out by identifying four categories: in-house bug detection, in-field failure recovery, in-field failure diagnosis, and in-house bug fixing. She focused on in-field bug detection software (software used in production), which involves a lower overhead compared with in-house, which is more focused on high accuracy.

A challenge described for in-field bug detection software was concurrency bugs, which could be anything from untimely accesses among threads (buggy interleavings) to shared variables. Shan commented that since most machines are now multi-core—e.g., Intel Core 2 Duo—concurrency bugs are becoming even more prevalent. They are the same type of bug that caused the NASDAQ glitch in the Facebook IPO.

In-field bug detection requires a certain degree of user consent and cooperation, but Shan found that very few users actually click the "Send Error Report" button. When users do feel confident about their privacy or feel that the company will actually do something if they send a report, developers often get a very generic call dump. Her first challenge was figuring out what to collect from production runs, and how to collect and how to process information while maintaining performance, good UI, capability, and low latency.

Core dump has a huge performance advantage and does not have runtime overhead, but the messages take developers hours to dig through. In contrast, replay and bug detectors have lower performance and require huge amounts of overhead and data, but developers save themselves the manual effort that core dump requires.

Shan refused to believe in this all-or-nothing collection method. She first tried applying an existing technique called Cooperative Bug Isolation (CBI) on concurrency bugs. For each user, CBI has the software randomly decide which branches should be recorded. Statistical analysis then figures out which predicate is most correlated with failure without nearly as much overhead.

This technique is powerful but not perfect. If a "throw error" branch is taken, we don't need statistics to tell us that failure has happened, so Shan designed new types of program properties, instead of branches, to sample at runtime. She also designed thread-coordinated, bursty sampling to test these new properties. This way, she would know which thread certain data were coming from by continuing to sample for at least a few memory accesses beyond the start.

Shan concluded her talk with a reminder that debugging tools need not be custom made. She was able to greatly improve the speed of this CBI by accessing the program counter with no change to the hardware, since that information was already available based on the machine's architecture.

Machine Intelligence

Neha Pattan, Google researcher

Neha began her talk on machine intelligence by discussing the famous Turing Test, by which Alan Turing proposed testing whether or not a machine could be considered "intelligent." In a version of "The Imitation Game," if a human, messaging a machine, could not accurately judge if her interlocutor were machine or human, then the machine would have successfully answered the question, "Are there imaginable digital computers which would do well in the imitation game?"

Neha's point in discussing the Turing Test was this: For machines to be made useful, they must be able to understand context and environment, common-sense reasoning, actions, temporal representation, spatial representation, and natural language disambiguation. To illustrate this, she pulled up a clip from *Small Wonder*, a TV show she watched as a kid. A family buys a little girl robot and asks her, "Coffee, please." The machine does not understand, so the mother instructs her husband to be more specific. He asks, "Can you pour me coffee in my mug, please?" so the robot grabs the mug and starts pouring coffee into the mug and doesn't stop even when it overflows over his lap.

For the little girl robot to have properly poured coffee into the father's cup, she would have needed to understand that her context and environment were helping with breakfast in the kitchen. She lacked the common sense necessary to know that she should have stopped pouring when the cup was full. The only way for her to have completed the command would have been to break down the actions: Lift the carafe, pour the coffee, place the carafe back on the table. Spatial representation was needed to understand the distance of her hand from the carafe, and temporal representation to understand how events cause, overlap, and relate to each other. Finally, she would have had to have basic natural language understanding to interpret the sentence, "Can you pour me coffee?"

Neha ended her talk with the quote, "We can see only a short distance ahead, but we can see that much remains to be done," and by affirming Jeanna Matthews' comment that being as good as a human is too easy. The Turing Test is just one benchmark along the long road ahead for machine intelligence.

Fast and Flexible Development

Meg Green, Life360 engineer

Meg began her talk with a famous quote from Thomas Edison, "I have not failed, I've just found 10,000 ways that won't work." She discussed applying Edison's precept to her own work as an engineer, specifically while working with the application Tomcat, easing application deployments and tracking configuration changes, to get to a best platform for rapid development work.

Apache Tomcat is an open source Java application server that she used as a Genentech software infrastructure engineer (she made special note to thank the open source communities that made her work possible). With Tomcat, multiple engines (aka containers) can share a single set of files needed for the core Tomcat server, which gives the engineers fewer places to maintain the full Tomcat server and makes adding more individual containers cheaper and more convenient. However, this centralization went too far and led to oversharing, making it difficult for projects to use libraries with different versions and harder to maintain in the long run. Meg recounted battling these overcentralized libraries, toning down places that were over-automated, recoding premature optimizations, and chasing—instead of managing—configurations.

Because of the lessons learned while working on Tomcat, the community of Java developers at Genentech can rely on quickly generated platforms to support the research organization's culture of moving from ideas to investigation in hours or days instead of weeks.

Operating System Innovation: Engineering Complete, Integrated, and Automated Software in Oracle Solaris

Liane Praza, Oracle senior principal engineer

Liane talked about how she and her team at Oracle put a whole operating system together in a meaningful way, specifically the UNIX operating system Solaris. Solaris is "Big Memory at Big Scale," and Liane has been a part of the construction of this Oracle OS for the last 15 years, witnessing everything from its new virtual memory system to ZFS data management.

Over this time, Liane has seen an interesting progression in operating systems: At first, a computer was huge and took up an entire room, and computing power was carefully doled out among many people. Then personal computers put an operating system in every lap. Now, with the explosion of computing infrastructures, operating systems control massive numbers of systems and share that power among many people.

She talked about improvements in hardware fault detection and isolation—instead of taking down whole systems when one bit was corrupted, a particular piece of memory that fails can now be isolated and dealt with separately. Oracle has built self-healing systems with a telemetry à prediction à diagnosis à restart à offlining à notification progression.

She talked about Solaris Zones—which is built-in, free virtualization: a shared kernel—and how reduced administration overhead because of this shared kernel has led to a fundamental management paradigm shift. Before, there was one admin per OS instance. That admin would babysit the OS instance and rescue it when it went down. But with a shared kernel, admins are able to monitor many systems at once.

Liane concluded by saying that systems are already managed as collections, and operating systems no longer end at the hardware boundary. Cloud platforms are a natural progression, and she sees Oracle at the frontier of these advances.

From Backend to Mobile Development, Career Transitions at Facebook

Lavinia Petrache, Facebook engineer

Lavinia opened with a bright anecdote about her career as a young programmer. Before she discovered programming, she wanted to be a journalist, a translator, a teacher, and then a lawyer, but at some point, she figured out she was good at math and became a software engineer. In Romania, where Lavinia grew up, only operating systems and compilers were considered "serious." She carried this mentality with her to Facebook, her first job out of college. She knew the theory of Linux and how to handle hypothetical problems, and wanted her colleagues to know she was a serious coder, so she worked on spam detection infrastructure, checking whether messages were genuine or not (she joked that she used to love visiting Brazil and Turkey, but now grimaces at the mention of either, because they are some of the biggest producers of spam content on the site).

After a year at Facebook, Lavinia did a hack-a-month with Android because all her friends in Romania were using Gingerbread, one of the older operating systems. In Romania, she had done only "serious" distributed systems work, not mobile, and PHP was not emphasized, so Facebook gave her a week of training in Android, starting with "hello world," and she ended up loving the product side. She thrived on strict deadlines associated with products and one-month release cycles. Her infrastructure team tended to work on its own schedule, but in a customer-facing unit she got to think and code for millions of Android users every day.

Lavinia emphasized that whatever everyone else thinks is cool, which in her case was distributed systems in Romania, is not necessarily going to be the most satisfying or best career path for you personally. If she hadn't tried Android for a month, she would never have ended up as happy as she is now on the Facebook Android photos team, and she hopes that other women and engineers will similarly look to unfamiliar terrain for inspiration.

Release Engineering as More Than a Part-time Past-time

Dinah McNutt, Google

Dinah is a mechanical engineer by training and has been actively involved in USENIX as program chair of the USENIX Release Engineering Workshop '14, chair of LISA VIII, and other roles. In her job at Google, she is a release engineer, which means, in her words, "accelerating the path from development to operations."

Traditionally, release engineering has been an afterthought. To their own detriment, startups have always wanted engineers to build features, not think about release. It is much cheaper to put good practices in place early instead of battling legacy code. Releng, the Google slang for release engineering, works with developers and SREs. Release engineers must understand how code should be built and deployed and then define these processes.

Dinah described releng "building blocks" as consisting of source code management, building configuration files, learning to deploy "as fast as makes sense for the product," Logsaver, automated build systems, building identification mechanisms, packaging (versioning, naming), reporting/auditing, and best practices. With these tools releng can ensure a continuous delivery of new products, early bug identification, repeatability, enforcement of policy and procedures, and an airtight, hermetic build process.

Dinah sees the future of releng changing. As of now, few early-stage companies have the foresight to hire a release engineer, and big companies that need to scale and maintain huge systems do not have a reliable way of identifying qualified job candidates. Job descriptions are all over the place, and there is no standard hierarchy or job ladder. She envisions a future with industry standards for job ladders and descriptions, best practices, metrics, and compliance, as well as college curricula and classes and more end-to-end solutions from vendors.

Untitled Talk

Yuanyuan Zhou, University of California, San Diego

Yuanyuan, who was Professor Shan Lu's graduate advisor, has co-founded two startups, Emphora and Pattern Insight, and is the Qualcomm Chair Professor at UCSD. Her talk focused on logs and how they can be useful for debugging.

Software bugs are the most labor-intensive type of bugs and are also difficult to diagnose. Of all the NetApp customer issues from hardware fault, 25% were from software misconfiguration.

Troubleshooting is expensive and downtime is costly for customers. On average, downtime costs a customer 18.35% of TCO (total cost of ownership), which is the total cost of purchasing and operating a server or technology product over a lifetime. Vendors spend an average of 8% of total revenue and 15% of total employee cost on customer problem support. Cloud computing has only deepened the problem.

Production run failure diagnosis is hard to reproduce, and the inputs leading to the failure are often not available. NetApp collects 40 million log messages a day and 99% of organizations collect logs, but the question becomes, what do we do with this data?

At her former company, Pattern Insight, Yuanyuan developed a tool called Log Insight. Log Insight analyzes large amounts of machine-generated data (e.g., logs) in real time and allows users to quickly diagnose and fix problems. It saves the vendors many phone calls and allows the customer admin teams to reduce downtime. Yuanyuan sold Log Insight to VMware in 2012, a move that she joked made investors very happy because of the return on investment and made her very happy because VMware kept the name she chose.

She also talked about when to log. She frequently found that a developer would check an error but wouldn't log it, making it difficult to diagnose a problem when a user would ask for help but the troubleshooter could only see the logs. There have been 5409 log enhancements in Apache's history over five years because developers have found that log messages can provide greater detail and shed light on bug complexities with more features.

She finished the talk with the classic Fault-Error-Failure model. Logging the fault (i.e., the root cause of the failure) is hard. These faults lead to abnormal behaviors, called errors, which may not manifest themselves to the user or may be silently handled by the system. However, a few propagate and will cause the program to crash, hang, give an incorrect result, etc. These perceivable errors are easy to log. In a study Yuanyuan conducted in 2012, 77% of user-reported failures were concrete error patterns (e.g., error return codes, switch statement "fall-through"), yet 57% of these easily detectable errors were not logged, dramatically increasing the time to resolve the problems (2.2x).

Yuanyuan concluded her talk by emphasizing that there were many instances in which logging an error would be simple for the developer and would dramatically improve the lives of the engineers supporting that code as well as improve the software's performance at runtime. However, little empirical evidence exists about how well existing logging practices work, so engineers do not have a set of "best practices" to follow when deciding where to log. Yuanyuan developed a tool called Errlog that adds only 1.4% logging overhead yet can speed up failure diagnosis by 60.7%, which showed that the usefulness and importance of good logging will only continue to grow.