Teaching Software Performance Evaluation to Undergrads: Lessons Learned and Challenges

Diwakar Krishnamurthy
University of Calgary
dkrishna@ucalgary.ca

ABSTRACT

Recent high-profile performance-related outages and problems in industry clearly establish the importance of imparting performance evaluation skills to students at the undergrad level. Yet, performance engineering is rarely a required course in most software engineering programs around the world. The typical undergrad student is naturally drawn towards coding courses and courses on topics that they think are likely to be in demand in industry, e.g., data science. While sympathetic, curriculum designers often cite student pressures and other factors such as accreditation requirements from engineering bodies to argue against mandating a performance evaluation course. As a long time instructor of a mandatory, undergrad software performance evaluation course, I describe some of my experiences operating in such a climate. Specifically, I outline key strategies I have followed to motivate students and overcome their resistance to the somewhat analytical nature of performance analysis. I also offer my observations on how undergrad curriculums can be tuned to instil a performance-aware mindset into students. Finally, I point out ongoing challenges to stimulate future solutions.

1. INTRODUCTION

Performance considerations are critical in real-world software systems. Recently, there have been many instances of systems failing because performance considerations were not addressed adequately. A well-known example is the failure of the heathcare.gov Web site, which was plagued by many problems including poor performance. For example, there were reports that the Web site could not even handle 500 concurrent users when it went live [1]. More recently, performance problems have affected even Fortune 100 companies that have access to large scale computational resources and state of the art scaling techniques [3].

Since ignoring performance considerations can clearly have adverse real-world implications, it is imperative that organizations have access to engineers with performance evaluation skills. Specifically, organizations need personnel that can design systems that reduce the likelihood of operational problems related to performance. They need engineers who can build effective scaling strategies and who can trouble shoot performance problems on the rare occasions they occur.

Unfortunately, most organizations hire undergrads [5] and most undergrad software engineering curriculums do not prescribe mandatory performance evaluation courses. Performance courses are more common at the graduate level. Some universities may offer a performance course at the undergrad level, but such offerings are typically not mandatory. In effect, we as educators are leaving performance-related training to chance. I argue that this is quite reckless given how critical the topic is!

In the remainder of the paper, I discuss the main challenges in including performance evaluation courses in undergraduate software engineering curriculums. I will then focus on one of these challenges – student buy-in. I will describe some strategies I have observed to be effective in getting students interested in performance evaluation. Finally, I will present a discussion of open pedagogical problems that need to be addressed by the SIGMETRICS community.

2. PERFORMANCE EDUCATION AT UNDERGRAD LEVEL: CHALLENGES

At the outset, the lack of performance evaluation training might seem like an easy problem to fix. All that needs to be done is to create the appropriate course and decree that it should be taken by all students. However, it is a bit more complicated to operationalize this idea for several reasons. One key reason is that past curriculum standardization efforts do not explicitly include performance evaluation as mandatory material in their recommendations. For example, the IEEE/ACM reference curriculum [4] mentions performance adjacent topics such as software quality but does not explicitly emphasize topics such as performance modeling and measurement. This silence has the unintended consequence of making performance evaluation and engineering sound like a niche field thereby complicating their inclusion in undergrad programs.

Another issue that complicates curriculum design is that in some jurisdictions such as Canada engineering degrees, including software engineering degrees, must be accredited by an external body. In the case of Canada, this is the Canadian Engineering Accreditation Board (CEAB). The CEAB, for example, requires engineers to have depth in their own domain but also breadth in subjects such as math, physics, other engineering disciplines (e.g., thermodynamics, and statics), and complementary topics (e.g., role of engineers in society). While the pursuit of breadth and linkages to other engineering disciplines is laudable, in practice, such requirements make it very hard to carve out space for courses that focus on topics such as performance evaluation.

To illustrate this, software engineering students at my uni-

versity only take one programming course in year 1! Year 2 has a 50-50 split between software (e.g., Java programming) and non-software (e.g., optics and digital circuits) courses. Consequently, students arrive in year 3 with a lot of ground to cover in core software engineering (e.g., software design) and computer science (e.g., operating systems) courses. In the final year, students are required to take a project management course, complete a design project, and choose from a list of elective courses that provide them an opportunity to pursue specific interests.

Thus, a curriculum designer faces the difficult choice of balancing between depth in software engineering and breadth in other topics. In essence, there are only a limited number of slots available for core software engineering and computer science courses and these are typically filled with topics recommended in standard curriculums.

Finally, a crucial issue that needs to be addressed is student buy-in for a course in performance evaluation. I have been an undergrad curriculum administrator for a decade, and I get to speak to students regularly. Most of them are anxious to make up for the time they have spent taking non-software courses. They prefer courses that will land them their dream jobs. Not surprisingly, courses on data science and game development, not performance evaluation, dominate their thinking. When I offered performance evaluation as an optional course for the first time several years ago, student feedback was overall lukewarm. They felt it was a very niche course and that it was "grad" material. They were somewhat put off by even the modest level math in the course. Somewhat disturbingly, they felt that the material was not relevant to them and that they do not see the course knowledge being useful to them as practicing software engineers.

Much to my satisfaction, a course in performance evaluation was made mandatory in the software engineering program at my university. My focus in the rest of the paper is the key strategies I used to foster student interest while teaching this course. Specifically, these strategies strive to convince students that performance evaluation is a critical topic that is going to be useful to them in industry. They seek to emphasize practical skills that students can use in their industrial practice.

3. COURSE CONTENTS

The course I teach is organized as follows. I start by motivating the importance of performance using several industrial case studies. Then, to coax students to come out of a functional mindset to a performance mindset, I review some of the previous courses they have taken with a performance lens. For example, I consider computer architecture and discuss how things such as cache hierarchy, processor, and memory organization impact performance. I talk about multi-core architectures and the importance of software level parallelism to take advantage of hardware parallelism. I discuss operating system level issues such as concurrency, synchronization, and threading models and how they impact performance.

Next, I switch to describing the software engineering lifecycle and how performance analysis fits there. I introduce different performance evaluation techniques such as analysis, simulation, and measurements and discuss how these techniques differ and complement each other. The rest of the course focuses on two main aspects namely, modeling and measurements. In modeling, I teach operational analysis and product form models. I also introduce some advanced modeling techniques, mainly Mean Value Analysis (MVA) extensions to study queuing for software resources. The final module emphasizes measurement concepts such as workload modeling, load testing, and performance monitoring. I also introduce experiment design and associated statistical analyses.

4. STUDENT ENGAGEMENT STRATEGIES

On reflection, the main strategy I used is to explicitly motivate performance analysis and its importance to a practicing software engineer. A particularly effective technique to obtain student buy-in is to narrate case studies or situations that I have encountered either in industry or in my industry-oriented research projects. One of the case studies I narrate early in the course was one I was personally involved in while I was at HP Labs. This was a study where there was a production Web site operated by a Fortune 100 company. The Web site had horrendous performance problems such as low throughput and high page load times. Initially, the Web site's engineers tried to throw hardware at the problem. They increased the system's horizontal scaling but to no avail. While space constraints preclude me from going into the details, at a high level, the problem occurred due to one method that had a very long critical section. When the method was modified to have a shorter critical section, the performance problems disappeared.

I use this case study to drive home several important points to students. First, performance problems are real, and they can have serious economic implications if left unchecked. Second, I emphasize that just conducting functional testing is not sufficient. The offending method in this case passed functional tests but scaled very poorly and this was not caught prior to deployment due to inadequate performance testing. Finally, throwing hardware at the system will not fix many performance problems. In this scenario, the system was suffering from a software bottleneck. This motivates the need to design systems that do not have such insidious performance problems.

The next strategy I used is to complement narration of case studies with active learning to promote student engagement. Specifically, I describe a certain performance problem and ask students to solve it in groups during the classroom. For example, one of these sessions is based on a paper we published on the scalability of a multi-core web server [2]. This exercise occurs during the part of the course where I establish linkages with computer architecture. In this session, I give groups snippets of information such as the architecture of the server we used, the concept of multiple sockets, multiple cores, local memory access, and socket inter-connect bottlenecks. I then tell them that we installed an Apache Web server on the system and noticed that the throughput scaled linearly with cores when only cores from a single socket were used. However, sublinear scaling was noticed when cores from both sockets were allocated to the application. Students are asked to brainstorm the source of the bottleneck and techniques to mitigate it. Students typically ask a lot of follow up questions but are in general able to figure things out, which in this case relates to poor operating system scheduling decisions that trigger a socket interconnect bottleneck. Many groups correctly suggested that running two Web server replicas with each replica pinned to

its own dedicated socket might mitigate this problem.

This exercise promotes student engagement with performance evaluation in many ways. For example, it shows that performance evaluation and debugging can be fun since it resembles forensics. Furthermore, it reinforces the real-life importance of performance analysis by highlighting the fact that even production grade software (Apache) can have poor scalability. Finally, it reiterates that throwing hardware (in this case cores) at a system may not be effective in solving all performance problems. Careful design of software components such as operating system schedulers combined with thoughtful application configuration is crucial for fully leveraging hardware level parallelism.

I also consciously prioritized applications over theory given the overall objective of establishing industrial relevance. Specifically, for the modeling part, I decided to cover techniques that are mathematically simple yet have good practical applications. So, operational laws and product form models are the main modeling topics I cover in the course. This is probably not an ideal choice. For example, product form models have obvious limitations, which might limit their applications in many real-world contexts. However, I decided to trade off mathematical sophistication for student buy-in and motivation. In keeping with this strategy, the course focus is always on coming up with or building a proper model for a given scenario rather than the mechanics of solving the model. I allow students to use formulae cheat sheets during exams and solvers for projects. The emphasis is not on solving the model but rather on coming up with the model. I also noticed that the wording of assignment and exam problems can influence student engagement. Problems should be worded to have a practical flavour so that they come across as plausible scenarios that could happen in practice.

The final strategy I emphasize is the building of strong linkages between the lab and lecture components. While I give students the freedom to propose their own performance evaluation project, I introduce activities that force them to apply concepts covered in lecture. For example, consider a project that involves load testing. Students are required to provide a rationale for the synthetic workloads selected. They are directed to use operational laws to ensure that the load test environment has been setup properly, e.g., verify that measurements of the number of concurrent users, response time, and throughput follow Little's law. Students are given exercises that show how operational laws can be used to derive performance model parameters such as resource demands. Students are encouraged to build a model of the system using the derived parameters and explore how the model can be used to answer various "what-if" performance questions. Students find this portion of the course enjoyable due to its hands-on nature. The key idea is to introduce activities within the project that demonstrate how modeling approaches can complement measurement exercises.

5. CONCLUDING REMARKS

While student feedback suggests that the strategies outlined here are effective, there remain several challenges that need to be addressed. For example, in my opinion, the way software engineering programs are typically structured leads to siloed teaching efforts. Specifically, many programs have standalone courses on requirements elicitation, design, architecture, coding, functional testing, and performance analysis with no link to one another. Since all these activities are inter-related, there is a need for more integrated teaching efforts. For example, requirements gathering should include performance requirements. Design should include building models from design to see if the designs are likely to meet performance requirements. Testing courses should also introduce load testing to discuss aspects such as concurrency and scalability.

There are other aspects that need a thoughtful discussion in the SIGMETRICS community. For example, given the prevalence of machine learning based performance models in industry, how should queuing theory be introduced and taught to undergrad students? How much emphasis should be placed on theory given many students are intimidated by the rigorous math involved in classical performance analysis?

6. REFERENCES

[1] C. Brooks. Documents show healthcare.gov couldn't handle 500 users before launch, Nov 2013.

[2] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson. Characterizing the scalability of a web application on a multi core server. Concurrency and Computation: Practice and Experience, 26(12):2027–2052, 2014.

[3] N. Statt. Amazon's website crashed as soon as prime day began, Jul 2018.

[4] The Joint Task Force on Computing Curricula. Curriculum guidelines for undergraduate degree programs in software engineering. Technical report, New York, NY, USA, 2015.

[5] U.S. Bureau of Labor Statistics. Software developers, quality assurance analysts, and testers, Feb 2023.