COMP 250RTS (Run-Time Systems)
Syllabus

Fall 2017

Contents

Introduction

Every high-level language is implemented by a combination of *translator* and *run-time system*. The translator targets low-level abstractions and instructions, many of which are provided by a *target machine.* Machine-level abstractions are typically supplemented by other abstractions which are implemented in the run-time system. Run-time abstractions can be high-level, but they are implemented by hand, not by a translator. Run-time abstractions may support such language features as procedure calls, exceptions, threads, dynamically allocated objects, and others.

The course provides a 50,000-foot view of the design and implementation of run-time systems, both classic and "modern."

What will we learn and why?

The purpose of the course is to help us all learn *how a run-time system should be designed and implemented.* This learning should serve the following goals:

- Build knowledge to use in research
- Become better able to understand systems that you may encounter in your research
- Get a starting point if you want to design and build your own run-time system
- Identify opportunities for research

What will we learn from?

A lot of the important stuff never seems to get written down. We will nevertheless prioritize written sources, of these kinds:

- Peer-reviewed publications
- Unpublished work such as technical reports and dissertations
- Books

To these typical academic sources we will add

- A case study of an actual run-time system

The sources will be drawn from three populations:

- To bring us up to speed on the foundations of run-time systems, I will draw from materials that I believe are clear, easy to read, and influential, if not at the state of the art. I will look for a combination that provides breadth.
- To approach the state of the art, I will scour the literature for the work of the best people in the field, and I will also dig through citation indices (and so forth) to try to discover what the best foundational work has led to.
- To learn about one system in greater detail, I will choose one run-time system for us to study as a group. My choice will be informed by students' "scouting reports" and by my own judgement and experience.

I would love to finish this seminar with a coherent and comprehensive view of run-time systems, but the topic is broad, and there are significant gaps in the literature. We will do our best.

How will you influence our learning?

A class like this is inevitably influenced by the instructor's biases. Here are some of mine:

- I value good design over novelty and popularity.
- I value original work over the latest variation.
- Although I have the most experience with systems that are trying to run at native-code speeds, I also honor interpreted systems.
- I love a dynamic compiler, but I am a bit skeptical about JIT compilation, dynamic decompilation, and other stuff that seems super-ambitious to me. But some of that work is work I respect very highly, and I hope at least to approach it.
- I spent ten years designing and building compiler infrastructure to support run-time systems. This experience biases me toward the basics and away from exotica.
- I have an irrational fear of memory models. (Actually, I think my fear of memory models is entirely rational.)
- My selection of materials is going to be influenced by my connections in the community and by my experience with existing software systems. Since I have contributed to code generators for both Standard ML of New Jersey and the Glasgow Haskell Compiler, you can expect a certain bias toward functional programming. That bias will be mitigated by my deep respect for Smalltalk and Self.
- I think the best way for me to meet my obligations to you is to expose you to work of very high quality. If that means that our study is less than comprehensive or is biased toward what is discoverable from what I already know, this is an outcome I can live with.

What kind of class is this? How is it different?

Most classes in computing are organized around lectures and homeworks (problem sets or programming assignments), perhaps with some laboratory experiences. This class is a *seminar*,[1] and as such it is organized around small-group collaboration. So that you understand the distinction, I compare the two kinds of classes.

In my mind, a good lecture course has these properties:

- The instructor provides challenging problems that students work on outside of class.
- The problems are the primary focus of the class.
- The experience is carefully scripted in advance.
- The pace of the course is set by the instructor, and once the train leaves the station, it does not stop for anybody.
- If as a student, you are not fully engaged with the class, you hurt nobody but yourself.

Our seminar will offer a dramatically different experience.

- It will not focus on problems or problem-solving. The experience will be broader and should revolve around three kinds of activities:
  1) Reading, understanding, and evaluating others' work
  2) Understanding how this work plays out in the context of a real run-time system
  3) Learning how members of a particular scholarly community think, talk, and write
- Our seminar is not scripted in advance. Instead, it has a general direction—a goal—and a plan for exploring the scholarly territory on that way to the goal.
- Our seminar's pace will allow for reflective pauses and side excursions. It will be flexible enough to allow the class to pursue interesting ideas as they arise.
- In a lecture class, you do the bulk of the work outside class (in problem sets or programming assignments), and the role of the class time is mostly to prepare you to do this work.

  In our seminar, we will do significant preparation work outside of class, but the key learning and understanding will be done in class, together. Your time outside of class is mostly to prepare you to take maximum advantage of class.
- A seminar succeeds only to the extent that everyone is prepared and energized. Therefore, if you're not fully engaged, the entire outcome of the course is changed—everyone loses. When you decide to take the course, you obligate yourself to be a full contributor to the group, not just a passive observer.

[1] According to the Oxford English Dictionary, a seminar is a select group of advanced students associated for special study and original research under the guidance of a professor.

One way to view what is going on is that we have a semester-long conversation that leads us to consensus on:

- What work is good
- What problems are important
- What techniques are useful
- How to exploit what we've learned

Why do we teach seminars? Primarily because this is the way real science is done: through extended, vigorous conversations about problems and ideas. When it goes well, a seminar is among the most rewarding experiences you can have in a classroom.

What will happen in the classroom?

Class periods will be spent on discussion questions grounded in outside reading (including reading of source code). Ideally, questions will be available two days before class. Discussions are intended not only to solidify our understanding of what we have learned from the outside material, but to call our attention to gaps and omissions in what we have read.

During discussion, we may often split into subgroups.

Study code? How will that work?

Almost everyone is keen to do a case study of a real run-time system. I'll find a way to make it work. We'll start out studying code in the same way that we study papers: by focusing on prepared discussion questions. Based on classes I have taught that have studied code, I expect to make a few adjustments, such as putting code up on the projector or asking students to bring portable computers to class. Also, when you're preparing discussion questions about code, I probably will ask a bit more of you than I would for a paper—you might become a mini-expert on some part of a system.

How much will we read and discuss?

In advanced CS seminars, it's typical to read 12-page conference papers and to take one or two class sessions per paper. But for whatever reason, information about run-time systems is rarely found in short papers—we are going to read longer-form sources. I cannot say in advance exactly how much time we might spend reading and discussing each source, and therefore I can't say how many sources we might grapple with over the course of one semester. I will balance these competing concerns:

- When reading a source, it is best to grapple with it until we understand it, or until we hit a point of diminishing returns.
- We want to get an overview of the design and implementation of entire run-time systems, so we must inevitably address some elements at a more shallow level than the element warrants. (Garbage collection, I'm looking at you.)

What is expected of me outside of class?

Outside the classroom, you will

- Read and analyze both text and code, to prepare for in-class work
- Take several turns developing questions that will help other students prepare for class
- "Scout" one software system
- Prepare a final deliverable
- Give feedback on a peer's final deliverable

These expectations are described in detail below. Reading and analysis are typical of other seminars. Developing questions is less common, and "scouting" is unique to this course. The final deliverable comes in one of three forms (a review, a system guide, or a white paper); you choose the form most suited to your interests. The final deliverable emphasizes analysis, thinking, and writing; it is not a "project," and it does not require project-style work.

How will I help others prepare?

With your classmates, you will take turns helping prepare discussion questions for class. Discussion questions for a Wednesday class will normally be ready by noon the preceding Monday; discussion questions for a Monday class will be ready by noon the preceding Thursday.[2] Questions for each class will be prepared by me with the help of a student:

- Both the student and I will read the assigned source, and we will develop discussion questions independently.
- The student will meet with me so we can agree on questions for class. I aim to include only questions that *both* of us agree are worthy, but if we cannot agree, I will decide.

  The meeting will take place sufficiently far in advance that I can deliver the questions at noon on the day promised.

One set of questions is prepared for each *class*, not one for each paper or for each software system. I will do my best to distribute the work equitably.

You can sign up to prepare questions by editing post 7 (a pinned post) on Piazza.

[2] Exceptions: If a Wednesday class is discussing the same paper or code as the previous Monday's class, then the preparation of Wednesday's questions will await the outcome of Monday's class, and the questions will be ready by noon Tuesday. And discussion questions for the Monday after Thanksgiving will be ready by noon the preceding Monday.

What do discussion questions look like?

In my classes, discussion questions typically range from very general questions, which could be asked about any paper, to very specific technical questions, which are almost like tiny homework problems. Overall, my questions skew deep and narrow. From past classes on advanced functional programming, dataflow analysis, and probabilistic programming, here are some examples:

1. Most papers answer a question or advance a claim. Just to be sure we're all on the same page, what is Hughes's question or claim?
2. Both C++ and Ada enable a programmer to overload functions, procedures, methods, and operators. So what's the big deal about type classes? Are there things type classes can do that you can't do in C++ or Ada? If so, list them and give an example of each.
3. Given a general control-flow graph and a label L, what model-theoretic idea would you use to characterize the set of all possible paths from the entry point to L? In general, is this set finite or infinite?
4. This question is about dominator analysis: Suppose that both D and D' dominate L. Prove that exactly one of the following three relations holds:
   - D equals D'
   - D dominates D'
   - D' dominates D
5. Using Church's `query` or `lex-query` operation, do you understand what sorts of prior/posterior distributions you can work with, and what sorts of observations you can make? Are they sufficient for the dice problems?
6. In the discussion of procedure values in section 2, is there a difference between an "ordinary procedure" and a closure?

How do I scout a software system?

You will "scout" one software system and judge how suitable it is for a case study. Scouting involves a somewhat cursory look at the source code and documentation (let's say at most a couple of hours) and a short, written report which answers these questions:

- What papers would help us understand the system?
- What are the components of the system?
- Measured in lines of code, how big is each component?
- What components appear to be especially interesting or well done?
- If we study this system, what should we focus on?
- Given a scale to be described in the system-scouting handout, how do you rate the system as a candidate for study?
- What are the reasons for your rating?

I plan to spend one class period, in the first week of October, discussing scouting reports and choosing a system.

Scouting reports must meet these requirements:

- I must approve the system you wish to scout. By September 11, I will have produced a list of pre-approved systems and recommended systems, but you may propose any system from which you think the class would benefit. You must get my approval by the end of class on Monday, September 18.

  Although I would prefer not to have much overlap, it is OK if more than one student wishes to scout the same system.
- Scouting reports should be complete and in my inbox by 11:59PM on Thursday, September 28. Your scouting report should be delivered as PDF or as Markdown with Pandoc extensions.

What's this about a final deliverable?

A final deliverable enables you to build knowledge that is deeper and more solid than is possible through discussion alone. To accommodate students with different interests and workloads, I have designed three potential deliverables:

- A review of a paper, dissertation, report, or chapter involving run-time systems
- A written guide, in the form of a short paper, to understanding and using an existing run-time system
- A "white paper" proposing some interesting problem or idea involving the design or implementation of run-time systems

What's expected of a review?

You may write a review of any paper, technical report, dissertation (or dissertation chapter), or other written text describing some aspect of run-time systems. The review should be organized as if for a journal such as *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *Software—Practice & Experience (SP&E)*, or possibly the *Journal of Functional Programming (JFP)*. Your review should not only address the material you read but should outline how you would like to see the material expanded to make a good journal article.

If you haven't had much practice writing reviews, this deliverable will give you some, for course credit. Beyond the typical challenges of reviewing technical work, the primary challenge here is to identify the important gaps and omission in the original work, which you would like to see remedied in a longer version.

To help you learn to write a review, I will pull together some "advice for reviewers" from *SP&E*, *JFP*, and other sources.

Your review must meet the following criteria:

- You may choose to review any related paper, including but not limited to those papers discussed in class. If you choose a paper discussed in class, I will expect something more than a recapitulation of the class's discussion.

- A review must be entirely your own work—it cannot be a team effort.

I'll judge your review based on how well it identifies interesting information that is either unclear or missing in the original work, and how clearly it identifies the information you want to appear in a revision.

The review is meant to be the easy option, but it will still take some time. I have written a lot of reviews, and it usually takes me about an hour and a half to read a good 12-page paper. Writing the review can take anywhere from another hour to another three hours, depending on quality. (A wise man once said to spend the most time reviewing the best papers.) Identifying good topics for a journal version might take me another half an hour. To estimate the amount of time you yourself might take, I recommend that you scale to the size of the paper you are reviewing, then (to account for inexperience) double or triple the number of hours.

What's expected of a guide to a system?

A guide to a software system should steer its reader toward the worthy parts of a real implementation. Such a guide should say

- What components are found in the system
- What components are worth special attention and why
- How to get the system
- Where to find the worthy components

A component might be worth special attention because it solves an interesting problem, embodies a new idea, was discussed in class, is especially well crafted, or has some other good property. If you think a component is interesting, others will probably also think it's interesting.

Using your guide, a reader should be able to pick up the system and quickly understand what is going on, appreciate some part of the code, and perhaps think about modifying it—or know what modifications not to attempt. Your readers should feel confident in studying the source code and should feel that you have told them where the bodies are buried (or where to find buried treasure).

I'll judge your review based on how confident I am that your classmates could use it as a starting point to learn something more about a system than was possible in class.

The system guide is meant to be an option for students who are focused more on software systems than on research, and who want to use this class as an opportunity to get deeper into a system than would be possible for the whole group.

Unlike a review, a system guide need not be individual work: you may write a system guide as part of a team of any size.

What's expected of a white paper?

A white paper should propose a question or problem for research. I'm looking for about two pages in ACM two-column format, but a white paper of one to four pages is acceptable. A white paper should answer some of the questions we would expect to be answered by a full research proposal:

- What's the problem or question?
- Why is it interesting or important?
- What reason do we have to believe that the problem can be solved?

If you want to exceed my expectations, you can enrich your white paper with more of the elements I would expect to find in a full research proposal:

- What's the shape of a solution or answer?
- Why will it work?
- How would you recommend to someone to demonstrate that it does indeed work?

I'll judge your white paper based on how clearly you state the problem or question and how much evidence you marshal in favor of claims that the problem is important, interesting, and solvable. If, in addition, your white paper contains an original idea, I will be very favorably impressed.

The white paper is meant to be an option for students who are considering research in run-time systems or who want practice writing a proposal or a thesis prospectus. It is the most open-ended and therefore the most challenging of the final deliverables.

Unlike a review, a white paper need not be individual work: you may write a white paper as part of a team of any size.

What's the schedule for final deliverables?

Final deliverables work on an "iterate until satisfied" model: as long as you submit early enough, I will look at as many drafts as you like. Only the final draft counts toward your grade. In order to help you produce your best work, I am suggesting several preliminary deadlines:

- You should let me know your plans by Tuesday, November 7.
- I encourage you to get a draft to me by Tuesday, November 14.
- I encourage you to get a draft to a peer reviewer by Tuesday, November 28.
- The absolute last day I will see a draft for the first time is December 4 (the Monday after Thanksgiving).
- Your final deliverable is due, in final form, on Thursday, December 14.

If you want to send your work to a peer reviewer first and then to me, that is OK, too.

That said, this is a graduate course, and I will not be evaluating or grading your ability to meet preliminary deadlines. I will evaluate only the work you deliver on December 14.

What kind of peer feedback do I have to give?

You are expected to be available to provide feedback on either a review or a white paper. (You need not, unless you want to, provide feedback on a system guide.[3]) You will read a draft and provide this sort of feedback:

- Do I understand what is written?
- Do I think it is interesting or useful?
- Do I have any suggestions for improvement?

Peer feedback is a lightweight way for you to help your fellow students. It need not be written; you can give oral feedback in person. While I hope you will give feedback of high quality, I will not evaluate your feedback—only your willingness to serve.

Depending on what choices your fellow students make about their final deliverables, you may or may not be called upon to provide feedback.

How long will my out-of-class commitments take?

To give you an educated guess about how many hours you might be committing to, here are some estimates:

- Your single biggest commitment will be to read papers (or code) and understand them well enough to contribute to class. How long this takes depends on the size of the paper. An ACM conference paper is usually 12 pages in two-column format in a 9-point font. In this format, I can read 5 to 10 pages per hour. It probably will take you longer. A journal paper may run 40–60 journal pages, which is roughly equivalent to 20–30 conference pages. I anticipate spending 1 to 2 classes on a conference paper and 2 to 4 classes on a journal paper (depending on how technical it is).
- Preparing discussion questions is hard intellectual work but not time consuming. Plan for an hour to prepare questions and a half hour to meet and refine them.
- Preparing to *answer* discussion questions shouldn't take more than an hour—probably less.
- Scouting a software system should take a couple of hours.
- If, as your final deliverable, you choose to review a paper, I imagine that might take ten hours or more, depending on the size of the paper you choose and your experience in reviewing papers. If you choose to write a system guide or a white paper, the sky's the limit.
- Giving feedback on a peer's final deliverable might take one to three hours: one to two to review the deliverable, and an hour to give feedback.

[3] Reviews and proposals are part of every project in our field, and every research student should know how to evaluate one. System guides are *sui generis*, and I feel that developing the skills needed to evaluate one is not worth your time.

How will everyone be evaluated?

My evaluation of your work, and your final course grade, will be based primarily on class participation and secondarily on your final deliverable. In detail, here is what I expect:

- In rotation with other students, you will prepare discussion questions.
- You will scout one software system.
- You will come to each class prepared to contribute.

  Being prepared does not necessarily mean that you have understood all the reading or all the code. It does mean that you can clearly articulate what you have understood, and where your understanding is incomplete, you can identify where the difficulty lies.
- In class, you will engage with your classmates in analysis, discussion, and design.
- You will not only contribute to discussions yourself, but you will also leave room for your classmates to contribute to the discussions.
- You will deliver a review, a system guide, or a white paper is clearly written and that demonstrates *either* technical depth or an original idea.
- You will be available to support one of your peers by providing feedback on a draft of a final deliverable.

How might these expectations relate to your grades? Well, you are an experienced student and well versed in computer science, or you would not be eligible to take a seminar like this one. You probably also know how to contribute effectively in a small-group setting, how to look at code, and how to write a short technical text. In my past experience with these kinds of classes, a large majority of students have earned straight A's or better, and almost every student has earned at least an A-minus. The occasional exception is the student who talks well in class but who delivers written work that shows no evidence of effort. Such students receive grades that are not sufficient for graduate credit.

Finally, here's how I expect you to evaluate me:

- You should expect that I have put thought and effort into the design and implementation of the class. In particular, you should expect that the learning goals I have identified will be enjoyable to explore and will be worth your time and effort.
- You should expect me to bias our study toward the best available papers and the most illuminating software systems. You should not, however, expect that all papers we read will be uniformly great—sometimes the great paper on an

important topic has not yet been written, and sometimes I make a mistake. You can expect that, at least once during the term, a paper I thought would be good works out badly.

- You should expect me to adjust the class schedule, as needed, to support everyone's learning.
- You should expect me to lead productive analysis and discussion in class. You should expect me to create and sustain an environment in which everyone has an opportunity to contribute.
- During class, you should expect me to guide you toward consensus opinions on the day's questions, paper, or system. I should acknowledge not only the majority opinion but any significant minority views.
- On some topics, you should expect me to compare your consensus conclusions with the consensus of scholars working in the field, as well as my own experience with run-time systems. On other topics, I may not be able to represent the community consensus—the field is too broad.
- You should expect me to steer you to suitable guidance for writing reviews or white papers.
- You should expect me to be willing to iterate with you on a draft of your final deliverable until it reaches a state in which we are both happy with it.

What challenges should I expect?

Any small-group discussion class poses predictable challenges. In addition, a class like this one, which studies a topic that is not well documented in the literature, poses special challenges. Here are some that I know about:

- For what we are trying to do, class time may be short. My experience shows that in a typical seminar of a dozen people, a good length of time to support deep class discussion is about 105 minutes. We have only 75 minutes, but fortunately, we are few in number. Even so, you can expect some awkwardness around scheduling.
- A lot of the most valuable information about run-time systems comes in longer works, including journal articles, book chapters, and dissertations. Because most work in programming-language implementation is explained in 12-page conference papers, all of us can expect challenges involved in reading the longer forms.

  The books present a special problem: morally, legally, and practically, I can neither copy books for you nor expect you to buy them. Some older books have been released from their copyright and can be found online; for others, we will do the best we can.
- While I have taught a couple of courses that have studied software systems in a classroom setting, I have done nothing at the scale I plan to attempt in this course. For us to undertake a successful case study of a software system, I may have to ask you to prepare in ways that are not enumerated in this syllabus.

What do I need to know coming in?

To succeed in this course, you must already be able to read research papers of the sort found in PLDI, OOPSLA, ICFP, ISMM, ESOP, Compiler Construction, and the like. You must also understand programming-language implementation at the level of COMP 40: You must understand the basic workings of registers, cache, memory, and the instructions that manipulate them, and you must be able to look at C code (or something similar) and explain what is happening at the machine level. You need *not* be able to explain how a compiler gets from here to there.

What else can I do to succeed?

Regarding a similar class, one of my former students contributed this advice:

- The learning you do at home is directly proportional to the amount of learning you will do in class.
- Read the assigned materials before class. Understand the paper completely. Re-read the paper. Take notes in the margins. Bring your annotated paper to class.

The same former student also had this comment about his experience in class:

- Even if it feels like you're learning stuff that will not be applicable to your future life, you will likely be surprised.