Learner-Sourcing in Engineering Classes at Scale

Elena L. Glassman
MIT CSAIL
elg@mit.edu

Chris Terman
MIT CSAIL
cjt@mit.edu

Robert C. Miller
MIT CSAIL
rcm@mit.edu

ABSTRACT

Teaching computer architecture as a laboratory course to approximately two hundred students per semester requires a large, dedicated teaching staff. This spring, a shortened version of the course will be deployed on edX to a potentially far larger cohort of students, with far less teaching support. We describe the development of three learner-sourcing systems that have been piloted on the residential computer architecture students, which we are now revising for deployment with the course on edX. The key design principle for our systems is that students should be directed to hints and explanations written by other students who have just completed the task themselves. Initial results are promising, and warrant follow-up work.

Author Keywords

engineering education; crowd-sourcing; learner-sourcing

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

One-on-one tutoring has been established as a costly gold standard in education [Bloom]. Engineering education is no exception. By opening up the virtual classroom to thousands of students, MOOCs make the teacher-to-student ratio drastically worse. Professors become even more distant from students' solutions because they can no longer walk around the lab or classroom interacting with students one-on-one to get a sense of common success and failure modes. Teaching assistants are still more plentiful but less knowledgeable.

We are developing systems which harvest and organize students' collective knowledge about engineering assignments. Students themselves are each becoming experts on their own bugs and solutions. The key design principle is that students write hints immediately after completing a task or fixing a bug. Later, other students can use these hints to help guide them to a correct solution or better design. While students do not have the pedagogical content knowledge to necessarily generate the optimal explanations, they also do not suffer from the curse of knowledge. These systems relieve some of the pressure on the relatively small teaching staff, and give students the valuable educational experiences of reflection and generating explanations [**?**].

We present three on-going case-studies deployed in an undergraduate computer architecture course. The residential course is taken by over two hundred students each semester, who are typically MIT sophomores and juniors majoring in Electrical Engineer and Computer Science (EECS). Over the course of the semester, students build entire simulated processors composed of logic gate primitives.

This coming Spring, the course will be offered for the first time on edX. These residential case-studies are in preparation for the online deployment of some or all of our developing tools. We continue to develop design principles from these experimental, evolving deployments.

RELATED WORK

In order to generate hints for others, students must reflect on their bug, their solution, their optimization, their design choice, etc. and communicate it to others. We review below some of the relevant literature at the intersection of learning theory and engineering education.

Reflection and Self-Explanation

Reflection and confusion are both treated at length in learning theory literature. Piaget theorized that cognitive disequilibrium, experienced as confusion, could trigger learning: the creation or restructuring of knowledge schema [Kibler]. However, D'Mello et al. point out that, for this learning to take place, it is important for confusion to be both appropriately injected and resolved [D'Mello]. Dewey theorized that reflection is a critical method for triggering that transformation from conflict and doubt into clarity and coherence [Dewey]. Reflection is part of the process of drawing meaning from an experience [Schon]. Turning that reflection into a self-explanation also improves understanding [Chi et al.].

Given the established value of reflection, Turns et al. argue that the absence of reflection in traditional engineering education scholarship is a hole that needs to be addressed. Susan Ambrose, Northeastern University's Vice Provost for Teaching and Learning asks, "Why, then, don't engineering curricula provide constant structured opportunities and time to ensure that continual reflection takes place?" [The Bridge, NAE]. The Consortium to Promote Reflection in Engineering Education, headquartered at the University of Washington's Center for Engineering Learning & Teaching (CELT), was established for this purpose. In this work, we aim to design scalable automated opportunities for students to reflect.

Peer Assessment

L@S'14, March 4–5, 2014, Atlanta, GA.
ACM 978-1-4503-2669-8/14/03
((DOI HERE))

While reflection is valuable in its own right, it is also a building block of larger frameworks that bring peers into the process. Peer instruction requires students to form educated guesses, and then discuss and reflect on their choices with peers. Peer assessment replaces peer discussion with peer evaluation. For example, in Peer Assessment Learning Sessions (PALS), students assessed each other's written calculations and sketches in numerical problem solving courses, i.e. Hydraulic Engineering and Reinforced Concrete Design. Reflecting on a peer's conceptual development or alternative solution may bring about cognitive conflict that prompts re-evaluation of the student's own beliefs and understanding [PALS, Reflecting on Reflection]. We intentionally pair students with other students' solutions that we have automatically determined to be distinct from their own. We try to trigger productive cognitive conflict that students can attempt to resolve through written reflection.

Generating Hints for Students

Many Intelligent Tutoring Systems (ITSs) incorporate mechanisms for students to get automated assistance when stuck. Barnes et al. and Gertner et al. generate hints and helpful questions automatically for students in the midst of solving logic proofs or physics problems within ITSs. These domains can be solved by students taking one of a family of sequential steps toward the final correct answer.

In contrast to these domains, Singh et al.'s AutoGrader generates automated feedback for a much more complex domain: introductory programming assignments. Students write freeform code that must pass tests for behavioral correctness. Automated hints can be revealed to the student if their code does not pass all the tests. However, the machinery depends on synthesizing and checking many permutations of the students' program, which limits its applicability to larger programs.

The processors constructed by students are far more complex than the introductory programming assignments handled by the AutoGrader. Hartmann et al.'s HelpMeOut system handles arbitrarily complex Java code by tracking and storing the changes programmers make when fixing a compilation error or runtime exception. Users, presumably students and teachers, can write helpful messages to accompany these automatically extracted bug fixes.

Our work also asks students to generate hints to fellow students who are struggling with the same task or bug. However, these hints are indexed by which overall processor submodule the student is trying to optimize, or which of hundreds of possible verification failures it will resolve.

REFLECTION: STUDENTS REFLECT ON FELLOW STUDENTS' ALTERNATIVE DESIGNS

In this computer architecture course, one of the early digital design assignments students complete is the construction of a Full Adder. A Full Adder adds two bits of information, plus an optional 'carry-in' bit, to produce a 'sum' bit and 'carry out' bit, which overflows into the next significant bi-

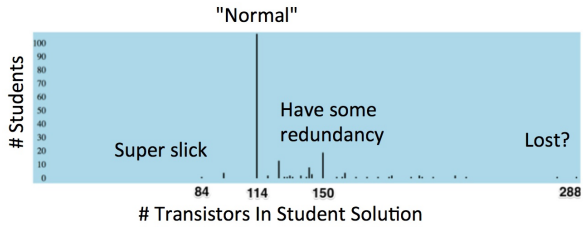
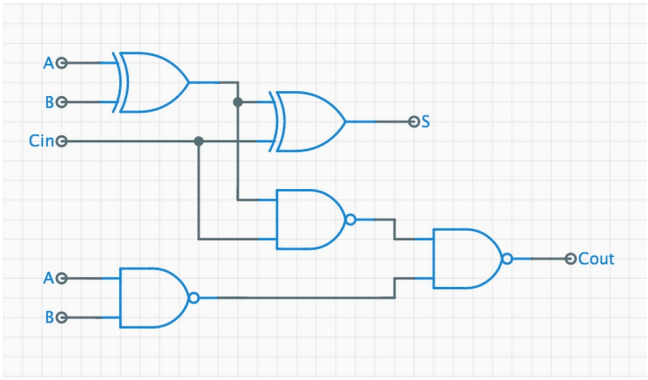


Figure 1. Distribution of the students' correct solutions (Full Adders), as a function of the number of trasistors.

nary digit.[1] Two possible solutions are shown in Figure 2. Since the assignment only requires a working Full Adder circuit and there is no pressure to optimize their solution, students create a variety of solutions that fit the behavior specification. Some are bloated, and some make use of one or more 'tricks' to use as few transistors as possible. In this scenario, few transistors will have better performance specs.

gates: 21, fets: 96



gates: 24, fets: 114 (expected solution)

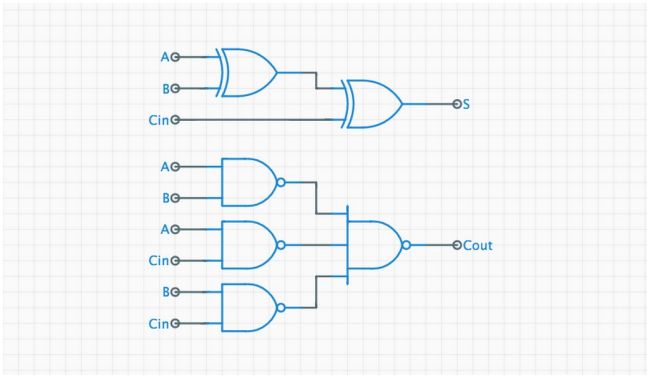


Figure 2. Two correct student solutions (Full Adders), which use different numbers of trasistors.

Students composed their Full Adders out of logic gates, e.g. NOR, NAND, and NOT. Each logic gate is composed of a particular number and arrangement of transistors. Through exploration of hundreds of previous students' solutions, we found that alternative correct solutions are nearly completely distinguishable by the number of transistors they contain. In

[1] In later labs, students cascade Full Adders together to make binary adders of arbitrarily many digits.

other words, we did not empirically find any morphological differences between correct solutions with the same number of transistors. This distribution is shown in Figure 1.

We picked a subset of representative distinct student solutions and rendered them graphically as a network of logic gates, so that students would not have to read any raw hardware description language code they did not write themselves. Each student was shown a pair of student solutions: their own and an alternative correct solution randomly chosen from the representative set. Figure 2 illustrates one possible pair of alternative solutions.

Students in the Spring '14 offering of this course were asked to give advice to a future student about how to improve the poorer of the two solutions, i.e., the solution with more transistors. If the students' own solution is the better solution, then the student can diagnose and describe, in a hint to this future student, what the poorer solution's creator had conceptually missed. For example, *You could replace the 'OR' of 'ANDs' with a 'NAND' of 'NANDs.'* If the students' own solution is itself the poorer solution, then they are challenged to understand how the other solution uses fewer transistors to achieve the same functionality, and then explain that to the future student that did not yet have that insight.

As discussed in the Related Work, reflecting on their own solution while comparing it to another solution that's potentially new to them, and then generating advice for fellow students, is pedagogically valuable on its own. In addition, students' explanations gave a rich window into their understanding, while serving as strikingly cogent advice to future students. **Table 1 includes several examples.** The online version of this course may afford us the opportunity to deliver these optimization hints back to students, when they complete a new, online-only adder optimization lab.

DEBUGGING: LEARNERS CROUDSOURCE HINTS INDEXED BY VERIFICATION FAILURES

In the same course, students design entire simulated processors composed of logic gate primitives. These designs, expressed as pages' worth of an in-house declarative programming language, can become so complex as to be very challenging to debug even with the one-on-one help of a seasoned teaching staff member. Given the complexity of students' simulated processors, students who have recently resolved a bug can be in a better position than a staff member to help a fellow student with the same bug.

Bugs are revealed by voltage verification errors. A set of simulated wires required of all designs are monitored during the simulated execution of a staff-provided sequence of instructions. The testing framework reports the first wire that fails verification, and the time of that failure, during the simulation. There are hundreds of possible unique verification failures. Some are more common than others.

While the course already has a vibrant Piazza forum, its generic forum structure was time-consuming to search for questions, and answers, about debugging a particular verification failure. We built *Dear Beta*, a website that serves as a central repository of debugging advice for and by students, indexed by verification failures. The processor in this class is called the Beta, and the system's name comes from the fact that it looks like a spreadsheet with an advice column, as shown in Figure 3.

As soon as student resolves a bug that was causing a particular verification failure, they can post an explanation of their bug on Dear Beta, indexed by the particular verification failure it caused. Providing the bug-resolving explanation is pedagogically useful, and students struggling with a particular verification failure can easily look up advice for their particular failure, if another student or staff member has added a hint. When students seek help for resolving a particular failed test case, and find one of their fellow students' hints helpful, they have the option of upvoting it.

A particular verification failure may be caused by one of several possible bugs. As the number of students and/or cumulative semesters becomes sufficiently large, crowd-sourcing students' own bug-fixes may generate hints about all the possible causes of a particular verification failure.

System usage statistics, interviews with teaching staff, and anecdotal evidence, including a student's unprompted class forum thank you note, suggest that many students find the system to be a consistently helpful tool. As one teaching assistant said, "Whenever I came to help a student, I first asked them if they'd checked *Dear Beta*."

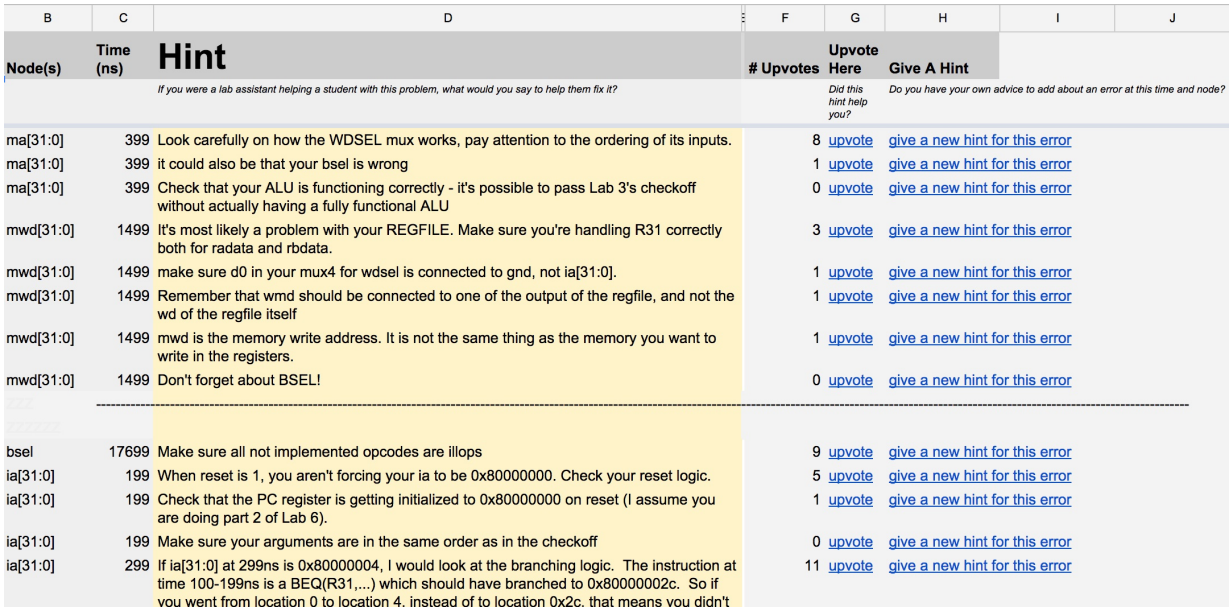
We hope to create a Meteor or Django-backed version of *Dear Beta* for both the residential and potentially much larger online cohort of students who take the course this Spring. We anticipate that students can cover a larger fraction of verification failures with hints that are just as helpful (as measured by upvotes) as the original staff-provided debugging hints collected before the creation of *Dear Beta*.

OPTIMIZATION: STUDENTS CROWDSOURCE OPTIMIZATION HINTS

As a final exercise, students optimize their own simulated processors. Students hear stories from classmates of the form, "I tried x and got y points!" One student heard that two classmates got significantly different outcomes from implementing the same two optimizations in opposite orders. This does not make sense, technically, but these kinds of optimization tales, and a long list of optimization hints written by staff, are all students have had to go on.

Our most recently deployed low-fidelity system is an adaptation of *Dear Beta* for this optimization portion of the course. Rather than indexing hints based on verification failures, we indexed hints based on whether they were intended to reduce a processor's size, minimum clock cycle time, or both. We also gave students the option of submitting the magnitude of the speed and size improvement they got from acting on a hint, so that future students could gauge which optimization hints were most beneficial.

Unfortunately, the effort of acquiring these statistics about the Beta was not trivial, and therefore very few students submitted their data to support this feature. It also could take hours



B	C	D	F	G	H	I	J
Node(s)	**Time (ns)**	**Hint**	**# Upvotes**	**Upvote Here**	**Give A Hint**		
		If you were a lab assistant helping a student with this problem, what would you say to help them fix it?		*Did this hint help you?*	*Do you have your own advice to add about an error at this time and node?*		
ma[31:0]	399	Look carefully on how the WDSEL mux works, pay attention to the ordering of its inputs.	8	upvote	give a new hint for this error		
ma[31:0]	399	it could also be that your bsel is wrong	1	upvote	give a new hint for this error		
ma[31:0]	399	Check that your ALU is functioning correctly - it's possible to pass Lab 3's checkoff without actually having a fully functional ALU	0	upvote	give a new hint for this error		
mwd[31:0]	1499	It's most likely a problem with your REGFILE. Make sure you're handling R31 correctly both for radata and rbdata.	3	upvote	give a new hint for this error		
mwd[31:0]	1499	make sure d0 in your mux4 for wdsel is connected to gnd, not ia[31:0].	1	upvote	give a new hint for this error		
mwd[31:0]	1499	Remember that wmd should be connected to one of the output of the regfile, and not the wd of the regfile itself	1	upvote	give a new hint for this error		
mwd[31:0]	1499	mwd is the memory write address. It is not the same thing as the memory you want to write in the registers.	1	upvote	give a new hint for this error		
mwd[31:0]	1499	Don't forget about BSEL!	0	upvote	give a new hint for this error		
22							
22222							
bsel	17699	Make sure all not implemented opcodes are illops	9	upvote	give a new hint for this error		
ia[31:0]	199	When reset is 1, you aren't forcing your ia to be 0x80000000. Check your reset logic.	5	upvote	give a new hint for this error		
ia[31:0]	199	Check that the PC register is getting initialized to 0x80000000 on reset (I assume you are doing part 2 of Lab 6).	1	upvote	give a new hint for this error		
ia[31:0]	199	Make sure your arguments are in the same order as in the checkoff	0	upvote	give a new hint for this error		
ia[31:0]	299	If ia[31:0] at 299ns is 0x80000004, I would look at the branching logic. The instruction at time 100-199ns is a BEQ(R31,...) which should have branched to 0x80000002c. So if you went from location 0 to location 4. instead of to location 0x2c. that means you didn't	11	upvote	give a new hint for this error		

Figure 3. *Dear Beta* **serves as a central repository of debugging advice for and by students, indexed by verification failures. The left two columns, wire name and time, uniquely characterize the verification failure. For example, there are three separate crowd-sourced hints for the first verification failure (**$ma[31:0], 399ns$**). The horizontal break indicates the break between verification errors detected by different verification files.** *Dear Beta* **is implemented with Google Spreadsheets, Forms, and Apps Scripts.**

to act on a single optimization hint, so activity on the system was already low. Finally, it did not support hints about verification failures created during the optimization process. As a result, this version of *Dear Beta* did not become an instrumental tool for processor optimization. This is our newest tool. We hope to take this feedback and redesign the tool so it can better support students through the optimization process.

CONCLUSIONS AND FUTURE WORK

This work in progress is the accumulation of several semesters of on-going development and deployment in an undergraduate computer architecture course with several hundred students each semester. We have found that students can write high quality optimization advice for simple digital circuits when their solution is paired with a solution that is different from theirs. We have also found that crowdsourcing students' debugging hints, indexed by the verification failures they're associated with is, anecdotally, a powerful way to help students help each other. We are preparing for deploying the two most successful systems, for reflection and debugging, in the Spring online course at scale, while we actively consider how best to measure the impact of these systems on the student learning experience.

ACKNOWLEDGMENTS

We appreciate the support of the NSF Graduate Research Fellowship, Quanta Computer, and the Amar Bose Teaching Fellowship for funding this work.

REFERENCES