Evaluating Student Understanding of Core Concepts in Computer Architecture

Leo Porter
Department of Mathematics and Computer Science
Skidmore College
Saratoga Springs, NY, USA

Saturnino Garcia
Department of Mathematics and Computer Science
University of San Diego
San Diego, CA, USA

Hung-Wei Tseng
Computer Science and Engineering Department
UC San Diego
La Jolla, CA, USA

Daniel Zingaro
Department of Computer Science
University of Toronto
Toronto, ON, Canada

ABSTRACT

Many studies have demonstrated that students tend to learn less than instructors expect in CS1. In light of these studies, a natural question is: to what extent do these results hold for subsequent, upper-division computer science courses? In this paper we describe our work in creating high-level concept questions for an upper-division computer architecture course. The questions were designed and agreed upon by subject-matter and teaching experts to measure desired minimum proficiency of students post-course. These questions were administered to four separate computer architecture courses at two different institutions: a large public university and a small liberal arts college. Our results show that students in these courses were indeed not learning as much as the instructors expected, performing poorly overall: the per-question average was only 56%, with many questions showing no statistically significant improvement from pre-course to post-course. While these results follow the trend from CS1 courses, they are still somewhat surprising given that the courses studied were taught using research-based pedagogy that is known to be effective across the CS curriculum. We discuss implications of our findings and offer possible future directions of this work.

Categories and Subject Descriptors

K.3.2 [**Computer Science Education**]: Computer and Information Science Education

Keywords

curriculum, assessment, computer architecture

1. INTRODUCTION

The CS1 research community continues to demonstrate that students do not learn what is expected in typical CS1 courses [5, 13]. Post-CS1, students continue to struggle with writing or explaining code that researchers and teachers feel should be well within reach of these students. By virtue of being the first CS course taken by many students, the reasons for this poor performance are unclear. Is it our inability to properly teach CS1? Is it related to fixed characteristics of successful and unsuccessful students? To what extent does this pattern continue through upper-level courses, in which the "strong" students are likely enrolled?

In this paper, we begin an investigation into what students learn in an upper-division introduction to computer architecture course. Through an examination of final exam questions, we developed questions designed to test student high-level understanding of core course concepts. The questions were also designed with expert architects in mind. What questions would an architect view as trivial and something "every student should get correct"? For example, from the view of an architecture instructor, *every* student should know that pipelining improves overall instruction throughput, not individual instruction latency.

Questions were designed for pre/post course usage. In one course, they were used as pre/post test questions in an online quiz. It was found that for many questions students were too unfamiliar with the material to hazard a guess on the pre-test (on which most students selected "Don't Know"). For a number of questions students learned from the course. Unfortunately, on some questions no statistically significant improvement was found between pre- and post-test results.

Following these perplexing results, we suspected students were not taking the test seriously. Therefore, we asked the same questions as a post-test in an in-class final exam study session. Again, students were seen to do quite poorly on a number of concepts. We then asked the post-questions, for credit, on the final exam of a small class and one large class. We again encountered perplexing results. Most notably, in this last large class taught with a slightly different focus, students showed improvement in previously troublesome questions, only to then do worse on other questions.

In this work, we provide a detailed investigation of the results from these high-level concept tests, provide intuition for the source of student confusions, and discuss potential implications for teaching of computer architecture. Our contributions include:

1. Pre/post analysis of questions from one term of computer architecture. We find statistically significant improvements for 56% of questions and no statistically significant improvements for 44% of questions.

2. Comparison of post-test results from four different classes with four different instructors and an identification of both common trends (e.g. students in all four classes struggled with understanding why tags are necessary in caches) and dissimilar trends (e.g. students in two classes did poorly answering a question on the impact of programmers on performance, whereas in two classes students did well on the same question).

3. Based on the results from this first evaluation of student understanding of basic concepts in computer architecture, we make recommendations for computer architecture instructors.

2. MOTIVATION

From our experience in teaching an upper-division computer architecture course, we began to suspect that students were missing the proverbial forest for the trees. In office hours and in class, students seemed to improve and even excel on "apply an algorithm" type questions. However, on core high-level questions like "how does pipelining improve performance," students curiously struggled. Initially, students misunderstanding these basic concepts appeared to be an aberration. However, after years of being involved in teaching students computer architecture, it became more clear that these misunderstandings might be more systemic than first imagined.

In reviewing final exams we found common trends. All the exams we reviewed had questions that asked students to provide pipeline diagrams and determine cache hits or misses for addresses given a cache configuration. While these questions require knowledge of important concepts, they are also testing students' ability to apply an algorithm from class. Higher level questions designed to test understanding of class principles were less common and, when present, were not answered well by students. In some cases, these questions could be viewed as "hard" as they asked students to apply understanding from multiple concepts to arrive at an answer. This led us to ask: how would students do on more basic concept questions? Are they missing the "hard" concept questions because they are struggling with understanding how basic concepts relate or do they not understand the basic concepts in the first place.

For example, when designing a cache, a *tag* is required for each cache line in order to determine the contents of that line. Although all computer architects understand this basic concept, we wondered how many students also understand.

```
Assuming your cache is smaller than your memory
space, your colleague, John, says a "tag" field
will be required.

Choose one answer.
 A. He's right, all caches require tags in order
    to make sure the correct data is in that line
 B. He's right, all caches require tags in order
    to know what line in the cache to access
 C. He's sometimes right, it depends on whether
    your cache is direct-mapped or associative
 D. He's partially right, it does require a tag
    BUT it also always requires an index
 E. He's wrong - only index bits are required
```

Figure 1: Question on the role of tags in caches. Answer A is the correct response.

Also motivating this question was our experience that students can often answer detailed cache questions about the number of "tag" bits required for a given cache configuration. Do they know why the "tag" is there in the first place? As such, an example of one of the questions from our pre/post test designed to test this concept is provided in Figure 1. After showing this question to a number of computer architects, the consensus was clear: all students should know the answer to this question. However, when we asked this question of our students in the post-test, the best course had 49% of students answer correctly and the worst course had only 28%. Our suspicions appear correct: students may be struggling to understand even this basic concept.

3. BACKGROUND

Considerable research suggests that CS1 students learn less than what instructors expect. For example, it is commonly believed that students should be able to explain a piece of code by giving its function as a whole, rather than its line-by-line description [5, 10]. Yet, students tend not to be able to do this: in one study, only 15% of undergraduates were able to explain a piece of code that determines whether an array is sorted. More recently, 952 students across two universities and three languages were provided equivalent, language-specific tests of fundamental CS1 knowledge [13]. On average, students scored 48.6%.

Parallel to the interest in student learning outcomes is a focus on the core concepts involved in CS courses (typically CS1). Such concept lists can be developed in a wide variety of ways. For example, Tew and Guzdial [12] used the contents of textbooks and the ACM's Computing Curricula 2001 to identify ten critical topics for CS1 courses. Schulte and Bennedsen [9] culled 28 topics from the literature and asked instructors to rate the difficulty and relevance of each. While their goal was not to develop an exhaustive concept list, Robins et al. [8] examined the distribution of "help calls" made by students in a lab setting to identify where and why students struggle when learning their first programming language. Each student question was classified according to its most salient concepts: control flow, loops, selection, etc. Goldman et al. [3] engaged instructors in the creation of a concept list for three areas (CS1, discrete math, and digital logic) using a structured approach for negotiating consensus among experts. The instructor comments leading to the list of digital logic concepts highlight particular difficulties involved in defining upper-division course requirements. For example, many instructors argued that number representations were not difficult because their students entered the course with this understanding; others indicated that the same topic was quite difficult because it was their students' first exposure to the material. In addition, these instructors argue that what is relevant for logic design has changed throughout the last few decades, so that what was once important (e.g. latches vs. flipflops) is no longer so.

The ultimate goal of generating these lists of concepts is to develop a Concept Inventory (CI) with which to test student understanding. Such tools are useful for comparing pedagogical techniques, comparing programming language of instruction, or, in general, obtaining a valid measure of student progress [13]. In addition to CS1, the development of other CS CIs is in progress [4]. The work closest to ours is the development of a digital logic concept inventory [4], however we are unaware of any work evaluating student understanding of basic concepts in computer architecture.

4. TEST DEVELOPMENT

In the summer of 2010, we asked instructors of computer architecture at our large research-intensive public university to provide prior final exams. Four instructors provided examples, one of whom is an author of the present paper. Two of these instructors analyzed those exams for common themes/concepts, having drawn from their experience as accomplished computer architecture researchers and instructors. We then developed eleven multiple-choice questions on basic high-level concepts for these common themes. Discussion among the four instructors subsequently led us to drop two of these questions due to poor/confusing wording. We provide brief descriptions of the remaining nine questions below.

Programmer Impact On final exams students were always asked to calculate the execution time of a program based on several key factors: instruction count (IC), cycles per instruction (CPI), and cycle time (CT). As a high-level basic concept, we asked which of these factors programmers could affect.

Execution Time (ET) Evaluation On final exams students were often asked to apply Amdahl's Law or the standard ET performance equation to calculate the potential speedup from a proposed improvement. To test their conceptual understanding of performance, we gave students four potential improvements and asked which would be the most valuable.

Single-Cycle Datapath On final exams students were commonly asked to modify the datapath of a single-cycle (SC) processor to support a new instruction or to reason about the impact of changes to control signals. We developed a question that provides students with the SC datapath from the course textbook [6] and asks them whether the datapath could support a new instruction. The "no" answers also include a reason why this instruction could not be supported, further testing their conceptual understanding.

SC/MC/Pipe Design On the final exams students were often asked to calculate the average or steady-state CPI for a multi-cycle (MC) or pipelined processor. However, in these exams students were rarely asked to reason about the performance impact of unbalanced stages, i.e. when different stages have different timing requirements. We developed a question where students are given unbalanced stages and asked whether the performance of an MC or pipelined processor with these stages would exceed that of a SC processor.

Pipelining Value On final exams students were asked to provide detailed pipeline diagrams that showed e.g. stalls and data forwarding. However, our experience indicated that students did not know why pipelining was valuable in the first place. To test their conceptual understanding of pipelining value, we developed a question on whether pipelining improves throughput, individual instruction latency, both, or neither.

Pipeline Depth On final exams students were often asked to provide a "pipeline diagram" for the five-stage pipeline discussed in class, augmented with additional pipeline stages (more pipeline depth). The basic corresponding concept is the impact of pipeline depth on performance. As such, our concept question asks students whether (and how) CPI or CT changes as pipeline depth increases.

Control Hazards On final exams students were frequently asked to simulate a dynamic branch predictor or provide the impact of static branch prediction on CPI. We developed a question to test conceptual understanding on the fundamental need for branch prediction. This question asks why branches pose a problem for pipelined processors and appears in Section 6.1.

Cache Tags On final exams students were almost always given a cache configuration and asked to determine the number of bits required for the cache's tag, index, and block offset. However, students were never asked a conceptual question about the utility of each of these fields. We developed a conceptual question that simply asks students why tags are necessary (Figure 1).

Cache Design As a follow-up to the cache tags question, final exams commonly asked students to determine the number of cache hits/misses based on a stream of addresses. The corresponding high-level concept involves understanding how block size, associativity, and cache size impact locality. We developed a question where students are given example code with solely spacial locality and asked which design decisions—bigger blocks, bigger cache, more associativity—would improve performance.

All questions were multiple choice with between four and six responses. As we will discuss in the next section, some variants of the test were given with an additional response choice of "Don't Know."

5. METHODOLOGY

5.1 Courses

Three of the four courses in our study (A-1, A-2, and A-4) were taught at a large, top-tier, research-intensive public university on the quarter system. The other course (A-3) was taught at a small, highly selective, private liberal arts college. All courses were taught using the same text [6], contained a laboratory component that involved some variant of small-scale processor design using the Verilog hardware description language, and used Peer Instruction (PI) pedagogy [2].

A-1: Summer 2011, n=32. This course was taught by a first-time graduate student instructor who had extensive experience as a teaching assistant for the course. The instructor is a published researcher in computer architecture. The PI materials were adopted from those written by one of this paper's authors.

Test Administration: The test was given both as a pre-test and post-test online using Moodle. Students were given credit for completing the test but were not graded on correctness. As such, a number of students completed the quiz rapidly with apparent random guessing. To remove these students from the study, we selected only those students who spent at least five minutes on the quiz (time was recorded and provided by Moodle). The pre/post test results evaluated in Section 6.1 are those from this course.

A-2: Fall 2011, n=74. This course was taught by an adjunct professor who was a full-time senior researcher in computer architecture and high performance computing. The professor had taught multiple classes (including this one) at this institution; however, this was his first time teaching using PI. Similar to A-1, this instructor adopted materials previously developed and used by one of this paper's authors.

Test Administration: Based on our experience with A-1, the authors aimed to help ensure the test was taken more seriously. To do this, the test was given as "final-exam preparation" by the course teaching assistant during the discussion section. The tests were anonymous and done on paper. After the tests were submitted, the teaching assistant led a final exam review session partially based on the test questions. Unfortunately, this tested only those students who attended the review session and tested their abilities before the rigor of final exam studying.

A-3: Spring 2012, n=10. This course was taught by an assistant professor who originally developed the materials adopted by the instructors of A-1 and A-2. The course differed slightly from the other classes in that it included background material on assembly programming and digital design that appeared in prerequisites for the other classes (A-1,A-2,A-4). However, as this course was a semester course rather than a quarter course, the same number of class sessions was spent on computer architecture as in the other courses. The professor is a researcher in computer architecture and had previously taught multiple classes, including computer architecture.

Test Administration: Based on our experience with A-2, the authors aimed to help ensure the test was taken at a point in time which best captured the culmination of student understanding. To this end, the test was given as part of the final exam, with students graded on the correctness of their responses. The only limitation of these results is the small number of students in the course.

A-4: Summer 2012, n=57. This course was taught by a first-time graduate-student instructor who had extensive experience as a teaching assistant for the course. The instructor is a published researcher in computer architecture, and developed his own materials loosely based on the materials provided by the instructor of A-3. More-so than the other courses, this instructor varied the focus of some portions of the materials. Further discussion of those differences appears in Section 6.2.

Table 1: Pre/Post-test (percentage correct) results for course A-1. Results include percentage of students responding Don't Know (D.K.) on the pre-test and the statistical significance of the pre-test to post-test improvement. Asterisks (*) denote statistical significance.

Question	D.K.	Pre	Post	p-value
Programmer Impact*	26%	15%	88%	< 0.01
ET Evaluation*	41%	30%	80%	< 0.01
SC Datapath	56%	5%	32%	0.22
SC/MC/Pipe Design	59%	25%	24%	1.0
Pipelining Value*	48%	10%	52%	0.02
Pipeline Depth*	67%	10%	68%	< 0.01
Control Hazards	33%	50%	44%	1.0
Cache Tags	59%	10%	28%	0.22
Cache Design*	70%	0%	40%	< 0.01

Test Administration: Similar to A-3, the test was given as part of the final exam, with students graded on the correctness of their responses. Changes in course content no longer allowed for the covering of multi-cycle processors, so the SC/MC/Pipe Design question was omitted.

5.2 Data Analysis

In course A-1, both administrations of the test were conducted using the same tool (Moodle), were not written anonymously, and included the "don't know" option on each question. Therefore, for this course, we were able to use the McNemar Chi-Square Test to determine statistically significant improvements from pre-test to post-test. For all classes, including class A-1, we then analyze post-test results.

6. RESULTS

6.1 Comparing Pre- and Post-Class Results

Table 1 provides the results from the pre/post-test in class A-1. Percentages reflect the percentage of students correct on each question, with an asterisk indicating statistical significance. Also provided in the table is the percentage of students responding "Don't Know" during the pre-test (denoted as D.K.). From this table, it is clear that while students improved on many questions, they continued to struggle on others. The ET Evaluation question and the Programmer Impact question showed significant improvements by the students with a respectable majority understanding the concept on the post-test. In contrast, students continued to struggle with topics such as SC Datapath, SC/MC/Pipe Design, and Cache Tags. Further discussion of post-test results appears below, following the results from other classes.

In terms of change from pre-test to post-test, we found that students did not know enough about the material of the course to do well on any of the questions pre-test. The

```
What is the primary problem when dealing
with branches (like beq and bne in MIPS) in
pipeline designs?

 A. Calculating if the branch should be taken or
 not taken complicates the design of the Execute
 (or ALU) stage
 B. You must fetch an instruction after the
 branch before knowing the branch outcome
 C. You can have a branch dependent on a value
 loaded from memory which delays completing the
 branch
 D. Trick question - branches pose no more
 problem than any non-branch instruction for a
 pipeline.
```

Figure 2: Control Hazards Question on the difficulty that branches pose to pipelining. Answer B was the correct response.

majority of students answering "Don't Know" to many of the questions and the generally poor pre-test results leads us to suspect that a pre-test is of limited value.

One particularly interesting result from the pre-test appears with the Control Hazards question in Figure 2. On the pre-test, a substantial 50% of students provided the correct result; on the post-test, this decreased slightly to 44%. It is unclear why students performed so well on the pre-test: perhaps students eliminated some of the other response choices since they contained unfamiliar terms. As for the poor performance on the post-test, perhaps students became so focused on the details of implementing solutions to control hazards (early branch resolution, need for branch prediction, role of a branch target buffer, etc.) that they missed the larger conceptual picture.

6.2 Post-Class Results

Table 2 provides the post-test results for all four classes. Focusing on the overall average across all classes (weighted equally by class, not student), an unfavorable trend appears. Recall that all of these questions were both designed and evaluated by experts to be questions that *every* student should understand. We were not so naive to think that all students would get the questions right, but did expect a vast majority of correct responses. The poor performance in the earlier classes (A-1 and A-2) caused us to suspect students were not taking the questions seriously. However, in the two courses (A-3 and A-4) where the questions were moved to be part of the final exam, no major improvement appeared.

In terms of student performance, students responded correctly more than 70% of the time for only two questions: ET Evaluation and Pipeline Depth. Students responded correctly between 60 and 70% of the time on three questions: Pipelining Value, Programmer Impact, and Control Hazards. More notably, less than half of the students (averaged by class) responded correctly on the remaining four questions. This poor performance was consistent across all classes, despite differences in course content and instructor experience. Either students are vastly under-performing our expectations on basic questions or the questions were overly difficult. We discuss potential limitations of our study in Section 7.

Table 2: Post-test results for all classes as well as average by question and by course. In class A-4, the SC/MC/Pipe Design was not included in the test.

Question	A-1	A-2	A-3	A-4	Avg.
Programmer Imp.	88%	26%	50%	87%	63%
ET Evaluation	80%	61%	80%	80%	75%
SC Datapath	32%	45%	30%	25%	33%
SC/MC/Pipe Des.	24%	39%	69%	N/A	41%
Pipelining Value	52%	53%	80%	69%	64%
Pipeline Depth	68%	79%	100%	60%	77%
Control Hazards	44%	71%	80%	67%	66%
Cache Tags	28%	45%	30%	49%	38%
Cache Design	40%	50%	40%	27%	39%
Average	51%	52%	61%	58%	55%

For most questions, the results by class are comparable, although with some variance. Two examples of between-class differences appear on the SC Datapath question where students perform better in course A-2 and on the SC/MC design question where students perform better in course A-3. Although we suspect these differences may relate to the underlying students and/or course, we found no apparent difference between sections.

We do, however, have some potential insight into the different results between classes for the Programmer Impact question. Although instructors in all courses asked a PI question on this very topic, students performed much worse in A-2 and A-3 than in A-1 and A-4. Post-class discussion between instructors identified that the instructor in A-4 had provided demonstrations to students of cache-aware and not cache-aware code and the resulting performance. We suspect this demonstration cemented student understanding of the impact of programmers on CPI. However, the strong result in A-1 without the demonstration informs us that further study is required before more significant conclusions can be reached.

7. DISCUSSION

7.1 Limitations

Although designed by subject matter and teaching experts, these questions have not yet gone through a verification process similar to that used in concept inventory development [13]. Despite this limitation, the authors believe the student performance on these questions motivates further study of the topic.

Another limitation of these results was the manner in which the results were collected. The poor results in A-1 motivated the authors to change the collection practice to ensure students took the test seriously. Unfortunately, these changes limit our ability to reason about results across classes. Also, the low number of students in A-3 (and the course being taught at a different institution) further limits comparisons between classes.

Although instructors were expected not to "teach to the test," there were no formal controls to ensure that instructors did not ask the same or similar questions in class or tell students to focus on learning the answer to such questions. To the extent that instructors did not adhere to this expectation, replication of these results may be jeopardized.

All four classes were taught using PI, which has shown promise at improving student understanding of course concepts in several disciplines including computer science [2, 7]. The extent to which PI impacted these classes for better or worse is unknown. Moreover, the use of PI limits the extent to which these results may be replicable in courses using different pedagogical practices.

7.2 Recommendations

One of the critical outcomes of a computer architecture class is that students understand that, although algorithm design often plays the largest role in program performance, full program optimization requires a deep understanding of the underlying system and architecture. This understanding becomes more critical as the memory wall persists and the parallelism crisis forces programmers to become increasingly system-conscious [1, 11].

As we speculated above, in-class code demonstrations may have led to some improvement on the Programmer Impact question. We suggest that such demonstrations are likely a worthwhile practice in all architecture classes. Further analysis of this practice as well as more extensive incorporation of demos into class sections is the focus of future work.

Students' struggling on basic course concepts is not new for computer architecture, but it does motivate further work on identifying the extent of student understanding of these concepts. Moreover, it further motivates instructors of computer architecture to focus on improving the course for student understanding of these concepts.

8. CONCLUSION

Based on previous teaching experience, multiple computer architecture instructors participated in developing basic concept questions for their course. The aim was that *every* student in the course should be able to answer the questions correctly. Pre/Post results in one class demonstrated that students improve on some questions considerably more than others, with statistically significant improvements occurring for only roughly half of the questions. Post-tests were performed in four classes of computer architecture, taught by four different instructors, at two different institutions, and under different conditions (including as part of the final exam). Across all four classes, we found that students underperformed expectations on these questions. Averaged across all four courses, the best performance by students on a question was 77% correct and the worst performance on a question was only 34% correct. Averaged across all courses and questions, only 56% of students answered correctly. These results motivate further inquiry into student understanding of core course concepts in computer architecture.

9. ACKNOWLEDGEMENTS

In memoriam of Dr. Allan Snavely. Thank you to the anonymous reviewers for their contributions to this work. This work was supported in part by NSF grant 1140731.

10. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, , and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. In *Technical Report Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley*, 2006.

[2] C. H. Crouch, J. Watkins, A. P. Fagen, and E. Mazur. Peer instruction: Engaging students one-on-one, all at once. In E. F. Redish and P. J. Cooney, editors, *Research-Based Reform of University Physics.* American Association of Physics Teachers, College Park, MD, USA, 2007.

[3] K. Goldman, P. Gross, C. Heeren, G. L. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Setting the scope of concept inventories for introductory computing subjects. *Trans. Comput. Educ.*, 10(2):1–29, 2010.

[4] G. L. Herman, M. C. Loui, and C. Zilles. Creating the digital logic concept inventory. In *Proc. of the 41st SIGCSE*, 2010.

[5] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE-WGR '04: Working group reports from ITiCSE*, 2004.

[6] D. A. Patterson and J. L. Hennessy. Computer organization and design: The hardware/software interface. In *Morgan-Kaufman*, 2008.

[7] L. Porter, C. Bailey-Lee, B. Simon, and D. Zingaro. Peer instruction: Do students really learn from peer discussion in computing? In *Proc. of the 7th ICER*, 2011.

[8] A. Robins, P. Haden, and S. Garner. Problem distributions in a CS1 course. In *Proc. of the 8th ACE*, 2006.

[9] C. Schulte and J. Bennedsen. What do teachers teach in introductory programming? In *Proc of the 2nd ICER*, 2006.

[10] J. Sheard, A. Carbone, R. Lister, B. Simon, E. Thompson, and J. L. Whalley. Going solo to assess novice programmers. In *Proc. of the 13th ITiCSE*, 2008.

[11] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. In *Dr. Dobb's Journal, vol. 30*, 2005.

[12] A. E. Tew and M. Guzdial. Developing a validated assessment of fundamental CS1 concepts. In *Proc. of the 41st SIGCSE*, 2010.

[13] A. E. Tew and M. Guzdial. The fcs1: a language independent assessment of cs1 knowledge. In *Proc. of the 42nd SIGCSE*, 2011.