End-to-End Automation of Feedback on Student Assembly Programs

Zikai Liu
ETH Zurich
Zurich, Switzerland
zikail2@illinois.edu

Tingkai Liu
University of Illinois at Urbana-Champaign
Champaign, IL, USA
tingkai2@illinois.edu

Qi Li
Purdue University
West Lafayette, IN, USA
qili8@illinois.edu

Wenqing Luo
University of Illinois at Urbana-Champaign
Champaign, IL, USA
wenqing4@illinois.edu

Steven S. Lumetta
University of Illinois at Urbana-Champaign
Champaign, IL, USA
lumetta@illinois.edu

***Abstract*—We developed a set of tools designed to provide rapid feedback to students as they learn to write programs in assembly language (LC-3, a RISC-like educational instruction set architecture). At the heart of the system is an extended version of KLEE, KLC3, that enables us to both identify issues and perform equivalence checking between student code and a gold (correct) version of each assignment. Feedback begins when students edit their code using a VSCode extension that leverages static analysis to perform a variety of correctness and style checks, encouraging students to improve their code quality. Each time a student commits code to their Git repository, our system triggers. Using KLC3 (KLEE), the student code is executed along with the gold version, and issues and behavioral differences are delivered back to the student through their Git repository as a human-readable report, test cases, and scripts. A queueing system allows students to monitor progress, but responses are generally available within minutes. We also extended the LC-3 simulation tools to support reverse debugging, making the process of finding complex bugs much more tractable for students, and used Emscripten to develop a browser-based interface for use in testing and debugging. Finally, our system maintains an individual regression test suite for each student and requires a submission to pass all previous tests before re-evaluation in KLC3, thus avoiding encouraging programming-by-guesswork. We deployed the system to provide feedback for the assembly programming assignments in a class of over 100 students in Fall 2020. Students wrote a median of around 700 lines of assembly for these assignments, making heavy use of our tools to understand and eliminate their bugs. Anonymous student feedback on the tools was uniformly positive. Since that semester, we have continued to refine and expand our tools' analysis capabilities and performance, and plan to deploy the system again in the near future (the class is offered every Fall).**

I. Introduction

Learning to program is difficult. As with all topics, students learn more quickly when given immediate feedback tailored to their efforts. However, while university staff—instructors and teaching assistants—are capable of providing such feedback, staff are not available 24/7 and lack the time needed to provide individual attention to each student's programs.

Automating feedback has been an important topic for years, and systems such as Web-CAT[1] are widely-used in classes. We address the need for rapid feedback by leveraging KLEE [2] to perform both symbolic analysis of student code as well as equivalence checking with a correct implementation of the given assignment. In our class, students first program in assembly language for the LC-3 instruction set architecture (ISA), which was invented for educational purposes in the textbook by Patt and Patel [3]. The symbolic equivalence checking that forms the core of our feedback system was one of the approaches explored by the early KLEE work [2]. Use of KLEE in providing rapid feedback on student programs written in C was pioneered by Gao [4], but the focus in that work is on C programs supported by the standard KLEE/LLVM infrastructure. In fact, most feedback systems focus on high-level languages [5], and few make use of symbolic analysis. This paper represents the first use of symbolic analysis to provide rapid feedback on LC-3 assembly programs through in-depth customization of KLEE, and describes the implementation and results of deployment as an end-to-end system.

We decided to make use of the low-level infrastructure provided by KLEE but to implement our own modules for preliminary LC-3 code analysis, LC-3 state execution, a 16-bit memory model, and search heuristics. We also generalize the idea of loop reduction [6] to accommodate the multi-entry-point loops often found in assembly language programs. Also, as we felt that the KLEE output might be difficult for novice programmers to understand and utilize, we developed issue filtering and human-readable report and script generation to help students through their first significant debugging efforts. We refer to our extended KLEE as KLC3.

We then extended the tools surrounding KLC3 to allow students to make use of more familiar interfaces, such as VSCode for editing and real-time static checks as well as Git to submit their code and receive the KLC3 analysis and feedback results. Finally, we added support for reverse (back-in-time) debugging to the existing LC-3 simulation tools and made them available through a browser interface by leveraging Emscripten [7] and minor modifications to the C implementations of the tools. As we extended the tools around KLC3, our system is able to to

cover the whole workflow of editing, testing and debugging when students work on their assignments.

After developing much of this framework, we deployed it to provide feedback to over 100 students taking our introductory assembly language and C programming course in Fall 2020, providing us with several thousand code samples as well as an opportunity to survey student opinions about the tools.

The remainder of this paper is organized as follows. In the next section, we provide additional details about our Fall 2020 deployment and student use as background. In Sec. III, we describe our LC-3 extension to the VSCode editor [8] that provides feedback as students edit their programs. Sec. IV then provides an overview of the system from when the student commits a new copy of an assignment through when they receive feedback. We follow with details of our extensions to KLEE in Sec. V. After deploying the system, we made a number of further optimizations, two of which we explain in Sec. VI. In Sec. VII, we discuss changes made to the LC-3 simulation tools (provided with the textbook) to support reverse debugging, and the browser-based interface that we developed. Sec. VIII provides timing information about the speed at which we are now able to offer feedback on student programs. Sec. IX describes a few other aspects of our system, including a summary of student feedback, and compares our approach with the commonly used Web-CAT[1]. Finally, Sec. X offers our conclusions.

II. Deployment Context

Prototype versions of most of the tools developed for this project were ready in the Fall 2020 semester, so we deployed them for use in providing feedback to students taking our introductory assembly language and C programming course. Conditions were fairly normal despite the COVID-19 pandemic: students and teaching assistants were resident on campus and made use of the usual classrooms and computer labs. The instructor, however, gave lectures live over Zoom, using multiple cameras to observe the students and to hear any questions. A video inset of the instructor enabled students to see gestures and facial expressions. While the interaction was not identical to a normal semester, we believe that it was as close as possible in the physical absence of the instructor.

A total of 109 students completed the course. Each student implemented three assignments over four weeks using LC-3 assembly language. Each assignment is built upon the previous one by including a student's previous code directly. In the first assignment, which we call SUBROUTINES here, students were required to write two subroutines to perform formatted output to the display. In the second assignment, SCHEDULE, students populated a weekly schedule with events, then printed the schedule to the display. Each event consisted of a name, an hour, and a bit vector of the days in a week on which the event occurred. Finally, in the third assignment, DFS, students implemented a depth-first search with backtracking to fit additional events with flexible hours (again specified by a bit vector) into an existing weekly schedule. Students wrote a median of 693 lines of LC-3 code for the three assignments. Since students received feedback each time they committed their code to the Github server, they were inclined to commit as they made progress, providing us with 1079 code samples for SUBROUTINES, 1474 samples for SCHEDULE, and 960 for DFS. For automating feedback, we also made use of correct implementations authored by the course staff.

The KLC3 feedback system was linked to the assignment submission system, so feedback was automatic for all students. We did not maintain a control group, as we felt it unfair to deprive students of the opportunity for feedback from KLC3. In this paper, we instead compare with a nearly identical version of the class held two years earlier, in which the same instructor presented the same material to students in person on the campus. Students and other course staff were different, of course. Students were not forced to make use of the LC-3 VSCode extension, and the survey results suggest that only about half did so. Students were also not forced to make use of our browser-based LC-3 tools, but about two-thirds did.

III. Editing in VSCode

Initially, we added basic static checking into KLC3 to identify problems such as dead code, but we wanted to give students feedback as early as possible. VSCode [8] is a popular editor amongst students in later programming classes, so we decided to encourage students to try it by developing an LC-3 extension that gives real-time feedback as students edit their code. By delivering appropriate warnings to students before they submit their code, we also enable them to fix potential bugs before they commit to their repositories and add load to our analysis server.

The VSCode extension reports three kinds of feedback: errors, warnings, and information. Errors imply that the code cannot assemble. Warnings indicate potential bugs or poor style. Information shows the results of analyses on the code, such as which registers in a subroutine are callee-saved (have their values preserved by the subroutine).

Our VSCode extension implements per-instruction analysis, control flow analysis, and procedure-based analysis. Feedback messages about potential issues in the code are conveyed via VSCode's squiggles and pop-up windows.

Although none of the extension's feedback is specific to any particular assignment, nor does the extension have any information about what constitutes correct behavior, we observe that the ability to convey meaningful feedback messages to students while they write their code may still increase their expected functionality grade in assignments. Functionality grade is the part of a student's grade allocated to correct behavior, as opposed to points obtained for demonstrating good coding style and including adequate comments. Using the SUBROUTINES program as an example, for which 770 code samples assemble, we used our grading script to calculate the functionality grade (out of 65) for each sample, then computed an average functionality grade among samples for which our extension reports the same number of warnings. The results appear in Tab. I. Code samples that generate no warnings often still fail to implement the assignment correctly, hence the

TABLE I
FUNCTIONALITY GRADES VS NUMBER OF WARNINGS

Number of Warnings	Count	Avg. Functionality Grade
0	492	46.9
1	169	36.0
2	42	40.9
3+	67	35.9

TABLE II
PERCENTAGE OF STUDENTS VS. NUMBER OF UNROLLED LOOPS

Number of Unrolled Loops	Fall 2018	Fall 2020
0	36%	14%
1	29%	31%
2+	35%	55%

```
1   LD R1, ADDRESS_1
2   BRz CHECK_1
3   STR R3,R1,#0
4   CHECK_1
5   LD R1, ADDRESS_2
6   BRz CHECK_2
7   STR R3,R1,#0
8   CHECK_2
9   LD R1, ADDRESS_3
10  BRz CHECK_3
11  STR R3,R1,#0
12  CHECK_3
13  LD R1, ADDRESS_4
14  BRz CHECK_4
15  STR R3,R1,#0
16  CHECK_4
17  LD R1, ADDRESS_5
18  BRz CHECK_FINISHED
19  STR R3,R1,#0
20  CHECK FINISHED
```

Fig. 1. A canonical example of a fully-unrolled loop typical of those produced by students not yet able to formulate iterative constructs.

average functionality grade for warning-free samples is not 65. Nevertheless, grades for warning-free samples averaged over 10 points higher than samples that contain warnings, indicating that feedback during editing can be helpful in guiding students to develop correct solutions.

In addition to helping with correctness, early feedback can also help students to avoid developing bad coding habits. For example, while extending the idea of loop reduction to assembly code (see Sec. VI-B), we found code samples in which students, unable to formulate loops properly, had instead written fully-unrolled loops. Fig. 1 shows a canonical example, in which five adjacent addresses containing pointers are checked for NULL (0), and a common value is written from R3 to the address referenced by each non-NULL pointer found. In the example, the branch instructions are independent, creating 32 possible paths. In some samples, however, students linked several such constructs, producing thousands of paths, many of which were impossible to execute due to correlations amongst the branches. Such style is not encouraged and a large number of paths undermines the performance of our symbolic analysis on the code. The extension identifies unrolled loops by scanning through the code with different strides and detecting repeated code segments that can potentially form a loop. When the extension finds an unrolled loop, it raises a warning to indicate that the code can be turned into a loop.

Interestingly, using the extension, we found that hand-unrolled loops are common in student code, and that their frequency depends strongly on the complexity of the particular loop that students are asked to write. To illustrate this idea, we compared the final versions of the DFS assignment of students in the Fall 2018 semester with those of the Fall 2020 students. The assignment was changed in minor ways to reduce the likelihood of sharing code between semesters. In particular, the days of the week were printed as three-letter abbreviations in 2018, but as full names in 2020. Also, the encoding of days in the bit vector for each event was reversed: in 2018, Monday was represented as 1, Tuesday as 2, and so forth. In 2020, Monday was 16, Tuesday was 8, and so forth.

The results are in Tab. II: failure to write loops is generally common in both classes, but is more common amongst the 2020 students. In terms of the assignments, the slight changes produced visible differences in the results by changing the complexity to conceptualize loops. Specifically, the variable-length weekday names complicate the process of finding the starting address of each string, and the reversal of bit vector ordering makes using these data more challenging because the LC-3 ISA makes left shift easy, but right shift difficult.

Fortunately, our VSCode extension is able to identify hand-unrolled loops and to raise warnings, as shown in Fig. 1 by the squiggles under the LD instructions—the first instruction in the loop body—to encourage students to think harder or to seek help for implementing a loop.

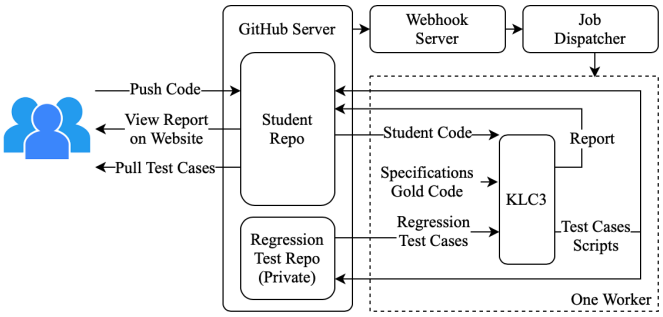

Fig. 2. Overview of dynamic code analysis system.

IV. DYNAMIC ANALYSIS OVERVIEW

An overview of our dynamic code analysis system appears in Fig. 2. The system consists of four main components: a GitHub server maintained by the university, a Webhook server, a job dispatcher, and the KLC3 execution engine. When a student submits a new version of a program by pushing code, the GitHub server immediately notifies the Webhook server, which is implemented as an HTTP server in Golang. The Webhook server filters out events other than modifications to the program residing in the master branch of the student's current assignment, then applies any policy decisions to the submission. Generally, we limited each student to one new evaluation every five to ten minutes in order to discourage students from attempting guesswork while debugging. The exact policy varied by assignment and is easily modifiable. Approved new submission events are wrapped up as KLC3 execution tasks and sent to the job dispatcher. The job

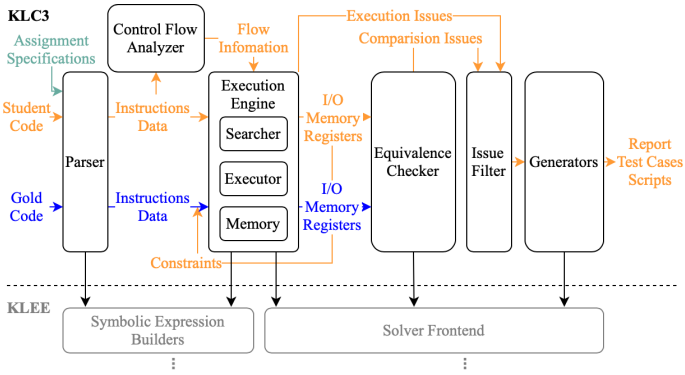

Fig. 3. KLC3 architecture and workflow.

dispatcher is also implemented in Golang, allowing us to leverage Go's channel abstraction to implement a queue and then to parallelize execution of multiple KLC3 executions on our server. The job dispatcher uses four worker threads that continuously fetch tasks from a single queue. For each submission drawn from the queue, a worker updates a local copy of the student's Git repository using the *go-git* library [9], extracts the submission, and forks off a instance of KLC3 to analyze the student's code. Once KLC3 has produced a report and associated files, the worker incorporates everything into a new directory in a "feedback" branch of the student's repository. The directory name indicates both the assignment and the date and time at which the student pushed the new version, allowing students to easily locate their feedback. The worker then returns to the queue to obtain a new assignment. Students can view the report on the GitHub websites or by updating their own local Git repositories. We also provide a web interface to our queue system to enable students to monitor queue status and to display KLC3 reports. For this purpose, we make use of the OAuth2 API of the Github server.

To encourage students to make use of test cases generated by KLC3 (see Sec. V) and to discourage programming-by-guesswork, we implemented a regression test system. A private class repository is used to maintain sets of regression tests for each student, each of which is initialized to the test cases provided with the assignment. Before analyzing code, KLC3 re-evaluates all regression tests. Only code that passes all such tests is then analyzed symbolically. Newly generated test cases are added to the regression test suite, so a student's next submission must pass the new tests as well.

V. KLC3 Architecture

KLEE offers a well-defined infrastructure for symbolic representations, optimizers, caches and interfaces with SMT solver backends, upon which higher-level modules such as the LLVM IR executor are built. KLC3 is similarly built upon the KLEE infrastructure (with a few modifications), but we replaced the higher-level modules with our own code, including an LC-3 symbolic executor, a 16-bit-addressable memory model, code flow analyzers, state searchers, and generators for test cases and human-readable reports. Fig. 3 illustrates the structure of KLC3, our modified KLEE architecture.

A. LC-3 Assembly Parser and Symbolic Executor

Rather than requiring instructors to learn the KLEE API, we developed a set of annotations on LC-3 assembly files through which instructors can specify symbolic variables as well as constraints, identify different types of memory regions (read-only, uninitialized but accessible, and so forth), and select types of output to be compared between student and gold codes to generate behavioral issues. Users can also override the default behavior, messages, and hints provided by the different types of issues tested by KLC3, and can to a limited extent define new issues. The input files are parsed along with command-line options to automatically generate the equivalent of the additional C code normally required for use with KLEE.

Our symbolic executor enables KLC3 to support direct symbolic execution of LC-3 code even when that code is questionable or obviously buggy. For example, a jump violating subroutine call/return semantics can be executed rather than immediately terminating the state (some students did so intentionally), although such operations are not encouraged and KLC3 does raise a warning by default. With precise control of each instruction, KLC3 is able to reproduce the exact behavior of the official LC-3 simulator (lc3sim), which is critical when students debug their code using the generated test cases in lc3sim. Detection of more subtle problems, such as using uninitialized registers, can also be performed in a more controlled manner.

B. Issue Detection and Equivalence Checking

KLC3 executes an LC-3 program and detects any improper operations by the code, such as out-of-bound memory accesses. When applied to a programming assignment and provided with a correct version of the assignment solution (a gold version), KLC3 not only detects the problems in the test program itself, which we call *execution issues*, but also performs equivalence checking between the test program and the gold version, thus identifying any *behavioral issues* between the two.

Execution issues typically indicate undefined or irreproducible behavior or the possibility of a crash when the program executes. For example, KLC3 can detect the use of uninitialized registers. When a program starts, all registers are considered to be uninitialized. If a test program state uses a register without first writing a value into the register, KLC3 raises an issue for that state. Most execution issues arise in the executor, but some rely on control flow analysis, such as identification of improper subroutine structure, in which a state executes a `RET` (return) instruction that does not return to the instruction after the most recent `JSR` (jump to subroutine) instruction.

The set of execution issues reported by KLC3 was developed based on experience with student code. Initially, we reported only a few common mistakes based on our own experience, such as reading uninitialized memory or registers.

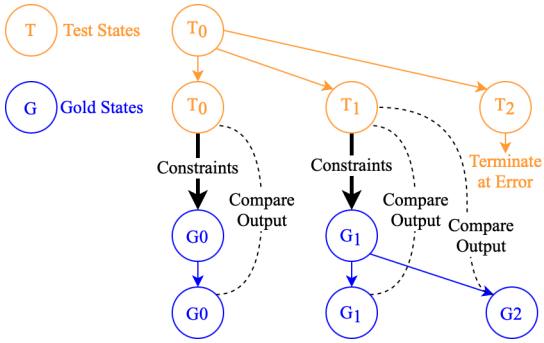

Fig. 4. Equivalence checking between test program and gold program.

Early versions of KLC3 were then tested on student codes from a previous semester (and later on students during Fall 2020 as they wrote their assignments), which helped us to identify additional issues through manual analysis, such as using a symbolic value for the program counter (PC), overwriting instructions, and broken subroutine calls. After testing several hundred student codes, we arrived at a stable set of issues, as detailed in the KLC3 manual (included in the replication package [10]), which we may extend in future semesters.

Behavioral issues signify differences between the output produced by a student's code and that produced by the gold code, for example, incorrect answers printed to the screen. These issues are identified by comparing symbolic equality for the display (an I/O device), memory, and registers after a program terminates, as shown in Fig. 4. Specifically, when a state of the test program terminates normally (instead of encountering a terminating issue), its final path constraints are used to launch a state of the gold program. After the gold state terminates, the equivalence checker module symbolically compares displayed output, memory, registers, and/or the last executed instruction of the two states. If the gold state forks to multiple states, all of them are compared with the test state. Assignments typically specify comparison for a subset of these possible outputs, so only those outputs relevant to the assignment are compared, as specified in the KLC3 input files. Divergence in the outputs raises behavioral issues.

C. Generation of Test Cases, Scripts, and Reports

Just like KLEE, KLC3 generates concrete test cases that reproduce detected issues. Issues are frequently triggered many times due to forked states and repeated execution of instructions. Ideally, each *bug* in a student's code should be reported exactly once. However, there is no easy way to localize bugs in the test code, particularly behavioral issues. Issues triggered on the same instruction may not necessarily result from the same bug, while a single bug may trigger a series of issues at different instructions. To avoid overwhelming students with failed test cases, we filter the set of issues produced by a student's code before reporting them to the student. In particular, we report only one instance of any given execution issue at any location in a student's program. In that way, for example, if 50 states access illegal addresses at a particular load instruction, the student sees only one report and one test case. For behavioral issues (such as incorrect output), which can't be localized in the test code, only one instance of each type is reported from a single run.

For each reported issue, a description, the instruction that triggers the issue (for execution issues), runtime information (such as the address accessed for memory issues and the output for incorrect output issues), and sometimes a hint about possible fixes for the issue, is provided to the student. Each reported issue is associated with a subdirectory containing a test case (one or more assembly files) and an LC-3 script. The test case contains concrete values derived from the symbolic subspace of the state that triggered the issue. The test cases are designed to be used with the LC-3 simulator (lc3sim), so students need understand nothing about KLC3 nor about symbolic execution in general. Ideally, a test case follows the same control path in lc3sim as it does in KLC3 and triggers the same issues (except for a small number of pitfalls, as described in the KLC3 manual [10]). The LC-3 script can be executed by lc3sim to help the student load the test case and reproduce the bug. Sample reports are available in the KLC3 manual [10].

VI. Optimizations

Timeliness is critical for effective feedback. Our goal is to provide feedback within 5 minutes after any submission. Static analysis in the VSCode extension is real-time, while achieving such an aggressive goal in symbolic analysis requires tuning of both the execution engine and input spaces. The raw speed of the execution engine dictates the number and length of paths that can be explored, and improving that speed enables exploration of larger input spaces. Tightly-constrained input spaces finish quickly, but may not expose bugs, while an overly general input space may make KLC3 run out of time exploring correct paths, thus again failing to expose bugs.

Most of our KLC3 optimizations had not been developed in time for the Fall 2020 deployment, forcing us to use fairly small input spaces, particularly for DFS, and thus to miss some bugs in our analysis. Using the code samples that we collected, we have been able to significantly improve KLC3's performance, and are now able to fully explore much larger spaces while meeting our 5-minute goal for most submissions.

In this section, we describe two of the more interesting optimizations: implementation of an additional cache within one of the lower layers of KLEE, and extension of the loop reduction algorithm [6] to assembly language in order to sidestep path explosions within loops.

A. IndependentElementSet Cache

The IndependentSolver module of KLEE removes irrelevant constraints from SMT queries before passing them to the next-level solver [2]. To identify the relevant constraints, the solver iterates through the query expression to construct an IndependentElementSet, a set of symbolic variables that are involved in each constraint and the query expression.

We noticed that student code samples produced large numbers of IndependentElementSets: for the 909 DFS samples that

TABLE III
TIMEOUT RATES FOR 960 DFS SAMPLES WITH AND WITHOUT THE INDEPENDENTELEMENTSET CACHE

Timeout in Minutes	**Cache**	**No Cache**
5	13.54%	65.10%
10	5.31%	22.08%

assemble and require 10 minutes or less to analyze, the average number of IndependentElementSets constructed is 2.20×10^8. Constructing such a large number requires substantial time. We also noticed that student code samples led to significant overlap in the constraints on queries issued by KLC3. For example, the constraints defining the input space are included in every query issued to the IndependentSolver. Given the cost of construction and the overlap in constraints, we decided to investigate adding a cache that maps symbolic constraint expressions to IndependentElementSets.

An effective cache must have a reasonably high hit rate and speed when results are cached. As KLEE infrastructure uses dynamic allocation for symbolic expression instances, cache comparisons can be either pointer-based or value-based. Pointer hashing and comparison are fast, but fail to match identical expressions if they are constructed separately (in different states, for example). Value-based hashing and comparison require walking through the nested expressions, which takes more time. We evaluated both approaches (and also a hybrid) and found that, for the purpose of KLC3, pointer-based comparison achieves a high hit rate and provides a substantial performance boost for most student codes.

For the 960 DFS samples that assemble, we measured the KLC3 analysis time up to 10 minutes with the cache enabled, and up to 15 minutes with the cache disabled (to capture performance information more accurately). In light of our 5-minute feedback goal, we summarize the fraction of samples that require more than 5 and 10 minutes of analysis in Tab. III. We then eliminated samples that timed out as well as those that finished within 5 seconds (to reduce measurement errors). For the remaining 803 samples, the fraction of baseline (no cache) analysis time required with the cache appears in Fig. 5. The geometric mean of the fraction is 0.288—an average speedup of $3.47\times$. The cache hit rate is over 99.9% for more than 98.5% of the DFS samples. Use of the cache for SCHEDULE samples shows a similar result: for 1239 samples that neither time out nor finish within 5 seconds, the geometric mean of the analysis time ratio is 0.273—a speedup of $3.67\times$.

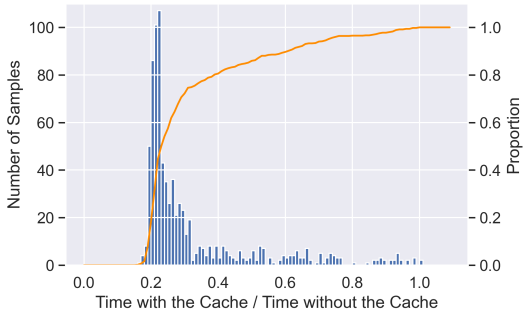

Fig. 5. Impact of the IndependentElementSet cache: distribution of analysis time reduction ratios for 803 DFS samples. The orange line shows the cumulative fraction of samples with reduction ratios below a given value.

B. Loop Path Reduction on LC-3 Programs

Loop reduction is effective in reducing KLEE execution time while maintaining high code coverage and bug detection for student C programs [6]. The key observation behind loop reduction is that even simple control flow within a loop body can produce an exponential number of paths over multiple loop iterations, but rarely are most such paths relevant to identifying bugs. Loop reduction identifies all paths through a loop body and prioritizes execution of states that cover previously unexplored paths through the loop body, while at the same time de-prioritizing or even avoiding execution of states that follow paths through the loop body that have already been covered by other states. The original loop reduction algorithm assumed that every loop had only a single entry point, which holds for any C program that doesn't use *goto* statements, and is also true in most C programs that do make use of *goto* (generally for exception handling).

In contrast, the single entry point assumption for loops fails to hold for many programs written in assembly language. Without C statements such as *if* *for*, and *while*, loops in assembly are constructed purely from branches and jumps, generally resulting in more complicated control flow and less clearly defined structure. As a result, we had to generalize the loop reduction algorithm to accommodate the more flexible forms of control flow used by our students (and in our own solutions to the assignments). We then implemented the generalized version of loop reduction and tested it on the samples collected from our class.

Loop structures in an LC-3 program must be identified and extracted automatically based on control flow analysis. The first step is to construct a control-flow graph (CFG), in which the nodes are basic blocks (sequences of instructions with a single entry point and no control flow except for the last instruction) and arcs connect each node to any other node that may follow it in dynamic execution. In this paper, we consider only the context of control flow within a single, well-defined subroutine (we do not apply the algorithm to codes in which subroutine structure is defined improperly, whether we detect such behavior statically or dynamically).

Starting with the CFG for a subroutine, we identify all strongly connected components (SCCs). Each SCC forms one or more of the outermost loops within the subroutine. For each SCC, we then identify each possible entry point—the CFG nodes at which arcs from outside the SCC arrive. For an SCC S, let's call one such entry point e. The node e forms one outermost loop, $L(e, S)$. Other entry points may form additional loops with the same code (the SCC S); often, these are used to implement similar but separate operations in the assembly code. A C compiler might produce such code if it found common subexpressions or common code sequences

within a single subroutine, for example.

For each loop $L(e, S)$, we identify all paths through a single iteration of L. In one iteration, loop L can either terminate or continue to another iteration. Continuing to another iteration we interpret as returning to e, and the Head to Head (H2H) paths are defined as all paths that form simple cycles within S, starting and ending at e. Similarly, the Head to Exit (H2E) paths are defined as all simple paths that start at e and exit S (without returning to e). We can then define the exit nodes for loop $L(e, S)$ as the set of CFG nodes outside of S at which one or more of the H2E paths of loop L terminate.

The nested loops (subloops) of a loop $L(e, S)$ consist of all loops in $S \setminus \{e\}$. In other words, a subloop has at least one H2H path in its parent loop that doesn't pass through the entry point of the parent loop. For each loop, we identify nested loops by executing recursively on the CFG induced by the nodes in $S \setminus \{e\}$, noting that the H2E paths of a subloop may also exit any number of containing loops as well.

During static analysis, we identify loop paths as follows. We define an H2H(H2E) segment of a loop as an H2H(H2E) path of the loop, but with each arc within any subloop replaced by a single virtual edge from the entry node to the exit node. In other words, the subpath in the subloop is invisible to the parent loop except for the entry and the exit nodes. The number of H2H(H2E) segments in a parent loop can be less than the number of H2H(H2E) paths, as multiple subpaths through a subloop are counted as one segment in the parent loop if the subpaths share the same exit node.

As loop analysis is static, dynamic jumps are not allowed. Subroutine calls are assumed to return and are summarized as a single edge from the call to the next instruction in memory (at the return address). Beginning with the most nested loops, loops and their segments are identified through depth-first-search (DFS) starting from each subroutine's entry point. The algorithm is shown in Fig. 6.

We implemented the sample coverage update algorithm from [6] to track loop coverage in KLC3. The algorithm uses a stack to record the current loop nest in each state and to update H2H and H2E coverage. We also implemented the *StatePruningSearcher* from [6], except that postponed states are selected randomly for reactivation, without checking constraint compatibility with the uncovered path. Unlike LLVM IR, LC-3 uses only the sign of the last operation's result to control conditional branches. Branch outcomes and constraints thus depend on both the preceding instructions as well as the control flow path into the branch, which is hard to determine without actually executing postponed states. Rather than making complicated speculations and incurring additional solver overhead, we chose to select a state randomly.

Among 1363 SCHEDULE samples, generalized loop reduction reduced analysis time for 307 (22.52%) of them. Excluding 26 samples that reported 0-second analysis, the average speedup for the samples that benefit is $9.67\times$ faster than the original DFS search heuristic. After acceleration, the analysis time of the 307 samples ranges from 0 to 16 seconds, with 91.21% (280) finishing within 5 seconds and an average of 2.30 seconds. Among 960 DFS samples, 69 (7.19%) samples were accelerated. Again excluding 4 samples which reported 0-second analysis, the average speedup is $37.89\times$, the resulting analysis times range from 0 to 63 seconds, and the average time is 4.55 seconds.

```
1: function AnalyzeLoopDFS(u, G, parent, path)
                ▷ G is the parent loop SCC excluding its entry node
2:     l ← none
3:     if u is the entry of a known loop l_0 then
4:         if l_0 ⊆ G then                    ▷ required for l_0 to be a subloop
5:             parent.subLoops ← parent.subLoops ∪ {l_0}
6:             l ← l_0
7:     else
8:         if u ∈ SCC S of G of size > 1 then      ▷ found a new subloop
9:             l ← new loop with entry node u and SCC S
10:            parent.subLoops ← parent.subLoops ∪ {l}
11:            AnalyzeLoopDFS(u, S\{u}, l, empty path)

12:    if l is not none then                  ▷ reached existing/new subloop
13:        E ← l.h2eEdg                           ▷ skip paths in the subloop
14:    else if u is a subroutine call then
15:        E ← {(u, u.next)}          ▷ skip subroutine and assume it returns
16:    else
17:        E ← u.outEdges
18:    for all (u, v) ∈ E do
19:        if v = parent.entry then           ▷ reach the the parent's entry
20:            parent.h2hSeg ← parent.h2hSeg ∪ {path + (u, v)}
21:        else if v ∉ G then                          ▷ exit the parent loop
22:            parent.h2eSeg ← parent.h2eSeg ∪ {path + (u, v)}
23:            parent.h2eEdg ← parent.h2eEdg∪{(parent.entry, v)}
24:        else                                        ▷ still in the parent loop
25:            AnalyzeLoopDFS(v, G, parent, path + (u, v))
```

Fig. 6. Analysis algorithm for generalized loop reduction.

We did observe one DFS sample for which loop reduction finished early but reported no issues, whereas issues were found without loop reduction. Considering how quickly the analysis finishes for the samples that benefit from loop reduction, we believe that we can simply execute the DFS search heuristic if loop reduction terminates without finding any issues in a student's code.

The fraction of LC-3 samples that benefit from loop reduction is lower than we had expected based on the C programs reported in [6]. While investigating this issue, we found that many of our samples contain unrolled nested loops, which produce large numbers of segments in the outer loops. Some of these segments may be difficult or impossible to cover with a limited input space. In fact, some samples contain provably uncoverable loop paths, sometimes due to poor style, such as consecutive branches based on the same condition. While we have found many such samples by hand, however, we have yet to identify fast and efficient ways to eliminate the impossible segments automatically, as most of the issues are more subtle than consecutive branches.

As another effect of samples with many paths, loop analysis can sometimes add significant time, on the order of tens of seconds. However, since the programs that benefit from the technique are those for which the analysis finishes quickly, we can set a limit of a few seconds on analysis time and thereby avoid any practical impact on the KLC3 response time.

VII. Testing and Debugging Environment

Programmers often rely on "cyclic debugging" [11], in which a program is relaunched to reproduce a single bug. However, restarting a program doesn't necessarily produce the same bug at the same place. Such difficulties are especially confusing for novice programmers. On the other hand, even when a bug can be reproduced, identifying its source often requires some kind of unexpected program behavior to be noticed, whereas the "reason" for the bug occurs earlier in the program's execution, forcing the programmer to use a combination of re-execution and reasoning to determine the cause. Enabling a debugger to support reversing execution back to the cause of the bug thus becomes attractive.

Students debug LC-3 programs by executing them under the control of the LC-3 simulator, lc3sim. In order to make debugging easier for our students, we extended this simulator to support reverse execution, sometimes called back-in-time or omniscient debugging. Two main approaches have been developed to support reverse execution: recording and reconstructing [11]. The recording method saves necessary trace information for each step of execution and uses this information to "undo" the effect of each step when executing in reverse. This method usually results in a large log and often requires hardware support for acceptable performance. The reconstructing method instead saves checkpoints (full state information) during forward execution. To perform reverse execution, the method reloads the closest checkpoint before the desired reverse execution stopping point, then executes in the forward direction to reach that stopping point. Reconstructing requires less log information and less hardware support, but checkpoint positions must be chosen carefully, making the approach less flexible than the recording method.

We chose to implement the recording method in the LC-3 simulator. LC-3 is a 16-bit ISA with 8 general purpose registers, so the architectural state is small, allowing state changes to be recorded in a compact log. And, as the LC-3 simulator uses software to simulate execution of LC-3 instructions, recording trace information at the ISA level does not add substantial overhead to instruction execution. We also felt that students benefit from the speed of reverse execution based on the recording approach, in which single instructions can be executed at approximately the same speed in both directions in time, allowing students to go easily back and forth in their code's execution trace.

The reverse execution functionality relies on two modules of the original LC-3 simulator: the first, the user interface module, handles three kinds of commands: information commands, management commands, and execution commands. Reverse execution is closely related to the execution commands that instruct the execution of the LC-3 instructions. By design, these commands are similar to the "step", "next", and "continue" commands available in most debuggers. The second module, the execution module, simulates the execution of LC-3 instructions.

We upgraded both modules with reverse execution functionality and documented the changes for students in a new LC-3 tool manual (provided as supplemental material). For the execution module, we added recording of state changes caused by each instruction's execution. An LC-3 instruction causes at most four registers and one memory location to change. We record the original values and the address of the changed memory location (if any) when executing an instruction. Each of these sets of changes is small and suffices to revert the effect of the execution step. The recording cache is implemented as a cyclic-array with enough space for all reasonable student codes.

In the user interface module, we added a new category of reverse execution commands. For every forward execution command, such as "step" and "continue," we implemented a reverse version, such as "rstep" (reverse step) and "rcontinue" (reverse continue). The semantics of each new command were selected carefully to be symmetric with the forward execution commands. For example, just as "finish" executes through the RET instruction at the end of the current subroutine and returns to the caller, "rfinish" executes in reverse back to the call site that entered the current subroutine. The LC-3 simulator also has a graphical interface version, in which simulator commands appear as buttons. We also upgraded this interface to include command buttons for reverse execution.

After finishing the implementation, we were pleased to realize that our extension can also simplify grading procedures. In particular, some information is lost when a program executes to completion in the simulator, making it difficult to test that information. For example, our SUBROUTINES assignment tests students' understanding of caller- and callee-saved registers, and register values are checked as part of grading. Doing so in certain cases requires asking students to add specific labels to their code so that grading scripts can check register values after setting a breakpoint at the labels. If a student fails to include the label, or puts the label in the wrong place, staff must fall back on time-consuming manual grading (with a minor penalty for the student). Using reverse execution, we can simply back out of the changes made when the student program terminates, revealing the final register values left in place by the student's code.

The LC-3 tools provided with the textbook assume the availability of a environments that novice programmers may not yet have learned to use, such as Unix or Cygwin. To make the tools more accessible to our students, we explored the automatic translation of these programs into JavaScript and WebAssembly for use through a web browser. We made use of Emscripten [7] to do so. We started by translating the LC-3 assembler and simulator into JavaScript modules, enabling our system to manage their use and execution lifetime. We then replaced the standard Unix filesystem used in the tools with an in-memory filesystem, adding import and export commands to the web interface so that instructors (and, eventually, our KLC3 feedback system) can populate the filesystem with a predefined set of files. Finally, we implemented a modern web interface to the tools to enable students to make use of them without the need to install a Unix-like platform and then to

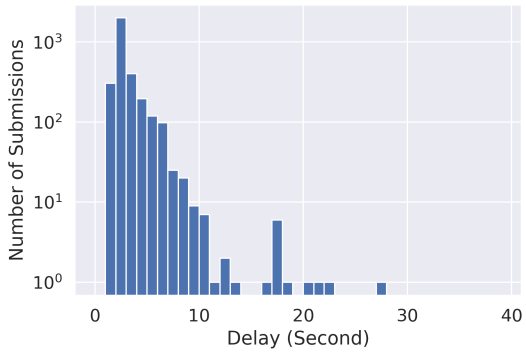

Fig. 7. Queueing delay for all Fall 2020 code samples.

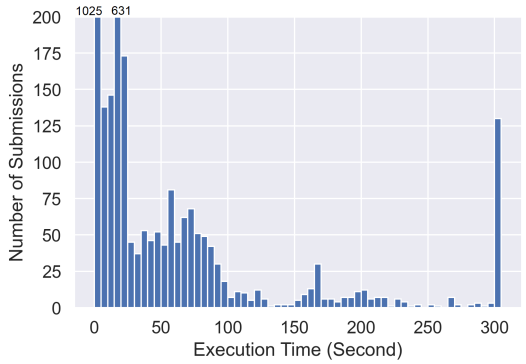

Fig. 8. KLC3 analysis time for the 3188 samples from Fall 2020 that assemble. We set a time limit of 5 minutes (300 seconds).

download and build the tools themselves. Supporting this interface from the original C code required a few modifications, as the textbook's GUI tool is based on Tcl/Tk and communicates through pipes with the simulator.

VIII. Feedback Timing

Our goal is to provide feedback on each student submission within 5 minutes. The time required from a student's point of view includes not only analysis by KLC3, but also queueing delays in the job dispatcher and other system components. Failures and downtime also contribute to perceived delay, and while the prototype deployed in Fall 2020 did suffer from several outages, most of the bugs have been tracked down and eliminated, leaving the system reasonably stable.

We examined queueing time as recorded in our logs for all submissions in the Fall 2020 semester to produce the histogram in Fig. 7. The vertical scale is logarithmic, and the distribution is dominated by delays of 10 seconds or less. Based on this data, we believe that our virtual server with four processors and 8 GB of DRAM more than suffices for a class of 100 students. While we expected more significant delays near deadlines, in fact the value of the feedback seems to have convinced students to work earlier, thus spacing out their submissions in a way that enabled them to digest and act on KLC3 feedback reports.

Looking forward, in Fig. 8 we show the distribution of KLC3 analysis times using all KLC3 optimizations currently implemented for the full set of 3188 samples that assemble (the remaining 360 require a negligible amount of time to determine that assembly fails). The input spaces are defined to be general enough to fully explore student code samples, and are substantially larger than those used during the Fall 2020 semester. For these data, we set a time limit of 5 minutes on KLC3, leading to timeouts for 130 (4.08%) samples. For each sample that timed out, however, KLC3 reported issues.

IX. Discussion

A. *Survey Results*

As the assembly assignments occupy only about the first third of the semester, ample time remained to survey students anonymously on their opinions about the tools. Roughly a quarter of the students answered the survey.

Few analytic features of our VSCode extension were ready in time for students, and our distribution method was private, requiring manual updates. Nevertheless, roughly half of respondents had used our VSCode and our extension rather than other editors to write their programs, and 80% of those users found the extension helpful. As described earlier, we have since extended both the syntactic and static analyses and have added style-specific features such as identification of unrolled loops, which we only realized were a major issue after analyzing student codes more carefully.

Survey respondents generally found KLC3 feedback to be useful in identifying and understanding their bugs, particularly when bugs were subtle. One student mentioned, for example, that KLC3, "...reminded me of an extreme case that I would otherwise neglect." As many bugs occur for corner cases, helping students to think more carefully about their programs is also a positive outcome. We were also surprised by student comments on the flow charts (visualizations of control flow) produced automatically for their code by KLC3. These were creating as a debugging aid for us, to help us to understand student code, but the students themselves also found them useful in identifying differences between the intended and actual control flow. We hadn't expected novice programmers to be able to make use of them, but the class does define and encourage the use of flow charts in understanding programs, and several respondents commented positively about their inclusion in the KLC3 reports.

Two-thirds of respondents made use of our new browser-based interface to the assembler and simulator. However, when making use of the KLC3 reports to debug their code, students preferred to use the traditional simulation and execution tools, perhaps because of the lack of scripting support in the browser interface at the time. We have rectified that lack, and plan to enable KLC3 reports to be pulled directly into the browser along with the scripts needed to reproduce any specific failure.

Although support for reverse debugging was available only in time for the third assignment, DFS, half of the students made use of it in completing that assignment. We were pleasantly surprised by this number and expect more students

and instructors to find the functionality useful. Although the idea has been around for decades [11], and reverse debugging is supported in open source debuggers such as GDB, it is turned off by default for performance reasons.

The survey also produced a number of interesting comments. First, although we had already kept response times from the server to minutes, students still wanted faster results. Some of this attitude may arise from instances in which our prototype system was unavailable, sometimes for several hours. However, as illustrated in Sec. VI, we have also significantly improved the time required for analysis since the class deployment. Second, although we implemented per-student regression testing and rate-limited new submissions to deter students from making guesses about their programs, students did seem to rely on KLC3 for finding bugs rather than developing their own tests, going so far as to express surprise when the tool (with a limited input space) missed bugs that showed up during grading. About half of the respondents admitted to relying completely on KLC3 for identifying bugs. Philosophically, providing more definitive guidance in an introductory class may be acceptable. Alternatively, by restricting the input space used by KLC3, one can leave certain aspects of testing to the students themselves. Either approach is easy to define and to use based on the input scripting language developed for KLC3.

B. Comparison with Samples from Fall 2018

To understand the effect of our system on student's ability to produce correct code, we compare code samples from 2020 with those produced by students in the Fall 2018 offering of the course, before the development of our tools. Many aspects of the two courses were the same: the same instructor presented the same material (albeit in person in 2018, rather than over Zoom as in 2020), and the assignments were nearly identical, with slight modifications to the ordering of inputs, the meanings of specific bits, and the exact output format required. As noted in Sec. III, the differences in assignments may introduce some difference in behavior, as might the differences in the student populations and the TAs.

For the Fall 2020 samples, we extracted the first and the last commits that assemble from each student's commit sequence of SUBROUTINES and DFS samples, then computed the fractions on which KLC3 reports any memory-related warnings (such as accessing uninitialized memory) or any error (such as incorrect output or timeout). Results are presented in Fig. 9 using blue triangles to indicate the percentage of students with each type of issue. Generally, student code improved from first to last commit, as expected.

In Fall 2018, students submitted their code once, after they had finished testing and debugging it. For comparison, we adjusted the symbolic input spaces to match the specification changes and executed KLC3 on each of these submissions. The orange lines in Fig. 9 show the percentage of student codes exhibiting each type of issue.

In the SUBROUTINES assignment, student subroutines were required to preserve the values of most registers. A code

Fig. 9. Percentage of student code samples on which KLC3 reports warnings or errors. Samples that fail to assemble are excluded.

fragment provided with the assignment (in both semesters) tested one register's value across a subroutine call to show students how such testing could be accomplished. To test their code, students needed to adapt that code to the other registers. In Fall 2018, this code fragment was the only method provided to help students test their programs, yet 19.28% of the students did not make use of the test provided, as shown in the top-left plot in Fig. 9. In Fall 2020, students also received feedback about incorrectly modified register values from KLC3. The fraction of the first commits that assemble from Fall 2020 with incorrectly modified registers is 30.21%, higher than that of Fall 2018, while the fraction of the last commits that assemble is much lower, only 1.04%.

The higher initial rate of errors in 2020 may indicate that students preferred the feedback system over performing their own testing, even in the early stages of development. Alternatively, since KLC3 executed automatically whenever students committed their code, students may have simply wanted to preserve a copy of their work before beginning to test. Regardless, the differences in the final submission show that the targeted feedback and test cases were more effective in identifying problems in their code as they made progress on testing and debugging. A similar pattern is observed for the other errors of SUBROUTINES and the errors of DFS.

Memory warnings show somewhat different behavior. For these, the rates among students in the Fall 2020 class for both assignments were lower in their first commits that assemble than in the final submissions (again, our only data point) for the students in 2018. We believe that this behavior results from changes in the assignment specifications: while students are never encouraged to use memory outside of the specific regions defined in the specifications, the 2018 specifications did not explicitly forbid such use, whereas the 2020 specifications did. This finding shows the potential for improving pedagogy with insights from students' submissions.

Although the fractions of erroneous submissions decrease, the overall impact of automatic feedback on student learning remains an open area for further investigation. Ideally, such impact should be evaluated through longitudinal assessment, in which students who have learned with and without such feedback are compared in terms of their proficiency in later

classes or even in their careers. We plan to deploy the system again in near future to further investigate this topic.

C. Comparison with Web-CAT

Many universities now use Web-CAT [1] rather than simple execution of test vectors in programming classes. The key idea behind Web-CAT is to have students generate their own tests, in a manner similar to Test-Driven Development. Like KLC3, Web-CAT relies on the availability of a gold version of an assignment—a solution.

Students are then evaluated based on a combination of the validity and correctness of their tests along with coverage of the gold version. In particular, validity evaluates the student-provided tests against the gold version of the code, while correctness evaluates them against the student version of the code. Coverage measures the degree to which a student's tests fully exercise the gold version, and can in practice be based on code coverage, branch coverage, or even on something like a full test set from symbolic execution.

Web-CAT's focus on encouraging and rewarding test development by students is an interesting and valid goal, but we believe that some of the more subtle errors and variations introduced by novice programmers are likely to be missed by such a system. For example, we note that roughly a third of students' final submissions of DFS still suffered from some type of memory error in Fall 2020 (see Fig. 9). While such errors may effect output, often they have no direct impact on program behavior. Some such errors are akin to out-of-bounds array accesses in languages like C, which may or may not cause a program to produce incorrect results. Often, the behavior ends up depending on the compiler, operating system, or even on the ISA. In the case of LC-3 code, for example, the simulator initializes most memory locations to 0, so student code with erroneous behavior often works fine, or works the first time, but not the second time, and so forth.

In our experience, and also as reported in [4] for C programs, using gold code coverage as the basis for student code evaluation is also limiting. Test generation using KLC3 with a symbolic input space on the gold program typically generates a superset of the tests required for code or branch coverage, but even those tests do not uncover all bugs in student code, even when combined with a similar set generated by KLC3 with the student code itself. In particular, while all possible code paths are covered by the tests generated by KLC3, the actual tests consist of specific input vectors, and the vectors that differentiate the student and gold versions may be missed. Equivalence checking, as described in this paper (and by the earlier work on C) specifically targets such vectors, and is thus better able to identify subtle behavioral differences.

Nevertheless, we recognize that Web-CAT may be better in avoiding the tendency of students to rely on the feedback system for testing. In the future, it may be interesting to combine the two approaches, using something like Web-CAT to provide initial feedback, but switching to KLC3 after students have surpassed some threshold with their own tests.

D. Alternative Designs

Early in our project, we considered an alternative approach to handling LC-3 programs with KLEE: translating LC-3 code to C (specifically, to an LC-3 virtual machine implemented in C), then compiling to LLVM IR and using KLEE. After some initial research, we decided on the more direct approach described in this paper. The LC-3 ISA differs from C in several noteworthy ways: LC-3 programs operate directly on registers and make explicit accesses to the 16-bit addressable memory, while registers and addresses are managed by compilers and hidden from C programmers. Also, LC-3 assembly can contain direct jumps to arbitrary memory locations (using `JMP` or `JSRR`), while C offers only the limited `goto` statement. Finally, subroutines calls in C rely on the concept of stack frames, which are not explicit in LC-3. We feel that the additional indirection implied by mapping LC-3 code through a virtual machine and then through a C compiler to LLVM would add too much overhead to analyzing the relatively simple programs produced by our students. The complexity of ensuring consistent behavior as well as inverting the mapping to explain issues found in the final version clearly to students in terms of their original code is also somewhat daunting.

E. Pushing Optimizations into KLEE

The authors of KLEE have also noted the possible need for an IndependentElementSet cache as a comment in the source code. To investigate the value for C programs, we translated one author's DFS solution into C, compiled it using Clang (-O2), and executed it with KLEE on the same symbolic input space. In that form, the code required construction of only 3.01×10^6 IndependentElementSets, $73.1\times$ fewer than did the LC-3 version. The gap may be due to differences between LC-3 and the LLVM IR: LC-3 lacks multiplication instructions, comparison instructions (use condition codes instead), and other features. Consequently, more LC-3 instructions are needed to implement the same functionality. Compiler optimizations also reduce the number of instructions. The average number of LC-3 instructions executed by KLC3 for 909 DFS samples is 2.41×10^8, while KLEE executed only 5.05×10^6 LLVM IR instructions for the C version. The impact of the cache is therefore less pronounced, although we believe that the cache may be useful in analyzing certain types of C programs.

X. Conclusion

Considering both student feedback and our success in using the system to deliver feedback based on symbolic evaluation of student submissions within our goal of 5 minutes, we plan to make continued use of these tools in future classes and to make them available to others using LC-3 to teach programming. Towards that end, we provide a replication package [10] with source code and manuals for all components, as well as the assignments and several sample solutions for each. Detailed implementations and discussions of the tools can also be found in the authors' theses [12][13][14][15].

References

[1] S. Edwards, "Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance," in *Proc. Int'l Conf. Education and Information Systems: Technologies and Applications (EISTA 03)*, Aug. 2003. [Online]. Available: https://web-cat.org/

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[3] Y. Patt and S. Patel, *Introduction to Computing Systems: From Bits and Gates to C and Beyond*, 2nd ed. McGraw-Hill Higher Education, 2004.

[4] J. Gao, "Use of symbolic execution as automated grading tool for introductory programming courses," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2019. [Online]. Available: http://hdl.handle.net/2142/105768

[5] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, and C. Szabo, "Introductory programming: A systematic literature review," in *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE 2018 Companion. New York, NY, USA: Association for Computing Machinery, 2018, p. 55–106. [Online]. Available: https://doi.org/10.1145/3293881.3295779

[6] J. Gao and S. S. Lumetta, "Loop path reduction by state pruning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 838–843. [Online]. Available: https://doi.org/10.1145/3238147.3240731

[7] A. Zakai, "Emscripten: An llvm-to-javascript compiler," https://emscripten.org/.

[8] "Visual studio code: Code editing. redefined." https://code.visualstudio.com/.

[9] "go-git: A highly-extensible git implementation library written in pure go," https://github.com/go-git/go-git.

[10] Z. Liu, T. Liu, Q. Li, W. Luo, and S. S. Lumetta, "Replication package of the lc-3 automatic feedback system," Aug. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5337117

[11] J. Engblom, "A review of reverse debugging," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, 2012, pp. 1–6.

[12] Z. Liu, "Using concolic execution to provide automatic feedback on lc-3 programs," 2021. [Online]. Available: http://hdl.handle.net/2142/110284

[13] T. Liu, "Improved feedback and debugging support for student assembly programming," 2021. [Online]. Available: http://hdl.handle.net/2142/110283

[14] Q. Li, "Vscode extension for lc-3 programming," 2021. [Online]. Available: http://hdl.handle.net/2142/110287

[15] W. Luo, "In-browser lc-3 toolchain and queue management for symbolic testing," 2021. [Online]. Available: http://hdl.handle.net/2142/110286