An Automated Feedback System for Computer Organization Projects

Peter M. Chen, *Senior Member, IEEE*

***Abstract*—This paper describes a system, built and refined over the past five years, that automatically analyzes student programs assigned in a computer organization course. The system tests a student's program, then e-mails immediate feedback to the student to assist and encourage the student to continue testing, debugging, and optimizing his or her program. The automated feedback system improves the students' learning experience by allowing and encouraging them to improve their program iteratively until it is correct. The system has also made it possible to add challenging parts to each project, such as optimization and testing, and it has enabled students to meet these challenges. Finally, the system has reduced the grading load of University of Michigan's large classes significantly and helped the instructors handle the rapidly increasing enrollments of the 1990s. Initial experience with the feedback system showed that students depended too heavily on the feedback system as a substitute for their own testing. This problem was addressed by requiring students to submit a comprehensive test suite along with their program and by applying automated feedback techniques to help students learn how to write good test suites. Quantitative iterative feedback has proven to be extremely helpful in teaching students specific concepts about computer organization and general concepts on computer programming and testing.**

***Index Terms*—Architectural simulator, automated grading, software testing.**

I. Introduction

COMPUTERS open many avenues for assisting teachers as they educate students. Computers can handle some existing tasks, such as grading, much more efficiently and accurately than humans can. They can also provide new capabilities, such as distance learning, to enhance the learning experience for students. The use of computers to improve education is especially appropriate when teaching computer science and education courses, because students in these courses are already accustomed to using computers and because much of the work being done is already online.

At the University of Michigan, Ann Arbor, the author began in 1995 experimenting with providing immediate automated feedback to students who were turning in programming projects for a class on computer organization (EECS 370). Since then, automated feedback systems have been developed and used for each of the four programming projects in this course, and over 2000 students have used the system. Instructors at the University of Michigan have since developed and used similar automated feedback systems in other courses in the mainstream curriculum for computer science and engineering majors (e.g., Introduction to Programming, Data Structures, and Operating Systems).

Providing automated feedback on student projects has resulted in many benefits to both the students and the teachers. Students submitted their program and received feedback within a few minutes. After receiving this feedback (which included the grade they would receive if this were their final submission), students were free to fix and resubmit their program. This iterative process encouraged them to find and work through their bugs, which resulted in a higher fraction of students with correct projects. While the feedback deliberately did not pinpoint the exact problem, it often helped point students in the right direction. The automated feedback system also replaced the need for human grading of project correctness, and this automation enabled course enrollments to scale to more than 200 students per semester. Using the automated system for grading was also more thorough, accurate, and consistent than using human graders. Finally, the automated feedback system made it possible to add some challenging parts to the project that would have been impractical without the system.

As professors at the University of Michigan gained experience with using the automated feedback system, they also encountered some inherent weaknesses of such systems. The most serious drawback was a seemingly irresistible urge by students to rely on the system to test their project (instead of testing their project themselves, as good programmers ought). This drawback ended up being a "blessing in disguise" because it forced the instructors to confront directly the students' need to learn and practice effective testing. This need was addressed by requiring students to submit test cases along with their program and by applying automated feedback techniques to help them learn how to write a good test suite. Thus, while automated feedback highlighted and worsened the problem of students' poor testing habits, it also provided the means to address the problem. A second drawback experienced was the heavy workload involved in building a robust feedback system that was clearly specified and that accepted all correct projects and rejected incorrect projects, especially for projects with some design freedom.

This paper describes the automated feedback system and the lessons learned over the past five years of using and refining the system. While the system is set in the context of computer organization projects, many of the lessons learned apply to any programming project.

Manuscript received August 27, 2001; revised April 3, 2003. This work was supported by the Dell Strategic Technology and Research Program and by NSF CAREER under Award MIP-9624869.

The author is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 USA (e-mail: pmchen@umich.edu).

Digital Object Identifier 10.1109/TE.2004.825220

II. Computer Organization Projects

The automated feedback system was developed for a junior-level course on computer organization. This course teaches students about the internal organization of a computer system, with particular emphasis on the central processing unit (CPU) and cache memory system. Students learn a simple instruction set architecture (named LC for "Little Computer"), including how to program in this instruction set, and several styles of implementing the architecture. The heart of the course is four substantial programming projects in which the students build an assembler and multiple simulators for the LC architecture. While writing these programs, students learn and express the detailed operation of several implementations of the LC architecture. Students often comment that most of their learning in the course comes from doing the projects.

This section describes the four course projects and the initial method used to generate automated feedback. The overall strategy is to evaluate a student's program by running it on a suite of test cases designed by the instructor (Fig. 1), and to e-mail a summary of the evaluation to the student. The automated feedback system uses black-box testing [1], which evaluates a program solely on its behavior rather than examining the internal code. The summary included a score for each test case and often contained some sketchy hint about the error. Students are not given access to the instructor's test cases because writing test cases in this course is considered an essential part of learning both computer organization and general programming.

A. Project 1: Assembler, Behavioral Simulator, Assembly-Language Program

For the first project, students write three programs (two programs in C or C++ and one program in the LC assembly language). The first program is an assembler; it takes as input an LC assembly-language program and produces as output the corresponding machine-language program. The LC assembly language uses eight instructions (add, nand, load from memory, store to memory, branch, jump and link, halt, and the null operation) and one assembler directive (fill with data). Some of these instructions specify register numbers as their operands; others specify memory locations. Memory locations can be specified by either a numeric value or a symbolic label. The assembler uses a two-pass strategy for resolving symbolic labels. In the first pass, it finds all symbolic labels and associates them with instruction addresses. In the second pass, it computes the machine-language program by translating the instructions, register operands, and memory references. The assembler must recognize a few specific errors, such as undefined instructions, undefined and duplicate labels, and out-of-bound memory addresses. To evaluate a student's assembler, the system runs it against a suite of 17 assembly-language programs, four of which test the assembler's ability to detect erroneous assembly-language programs. For each test case, the system compares the machine language generated by the student assembler against the machine-language generated by a solution assembler. For each test case, students are told whether their program generated incorrect machine language, exited with an error, or failed to catch an erroneous assembly-language program.

The second program of this project is a behavioral simulator for the LC architecture. The simulator takes as input an LC machine-language program (such as one generated by the assembler above) and simulates the execution of that program on the LC architecture. The simulator carries out the data transformation specified by each instruction. For example, an add instruction computes the sum of two registers and stores the result in a third register, and a branch instruction changes the value of the program counter. As the simulator runs, it prints the architectural state after each instruction simulated. This architectural state consists of the program counter, the general-purpose registers, and the contents of memory. Students are given a function to format the output of architectural state to make the output easier to grade and therefore students can concentrate on the substantive parts of the project. To evaluate a student's behavioral simulator, the system runs it on a suite of eight LC machine-language programs. For each test case, the system compares the instruction-by-instruction output of state generated by the student simulator against the output generated by a solution behavioral simulator. Students are told if their output is incorrect and which variable of the architectural state is incorrect.

For the third part of this project, students write a simple LC assembly-language program that multiplies two numbers. This program takes its input from two memory locations and places the product in a specific register. To evaluate a student's assembly-language multiplier, the system assembles it using the solution assembler, then simulates the resulting machine-language using the solution behavioral simulator and checks the answer in the register. The system runs four different sets of input by modifying the values of the memory locations specified in the student program. Students are told if their program generated the wrong result, had an assembly-language syntax error, or did not conform to the specification in certain ways (e.g., how it stores input values, the maximum number of instructions executed, and program length).

The following is a partial example of the feedback students receive for Project 1's simulator.

```
----0----
perfect!
----1----
master file and student file differ at
  student file line 59
student line:  pc 2
----2----
master file and student file differ at
  student file line 305
student line:     reg[ 4 ] 0
----3----
master file and student file differ at
  student file line 113
student line:     mem[ 7 ] 12 779 527
```

B. Project 2: Recursive Assembly-Language Program, Finite-State Machine

In the next project, students write a more complicated assembly-language program and a more detailed simulator for the

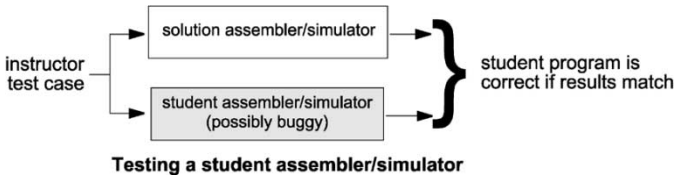

Fig. 1. Experiments run by the initial automated feedback system. The box labeled as solution assembler/simulator represents a correct program written by the instructor. The lightly shaded box indicates a program that may have a bug. The system tests a student's assembler or simulator by running it on test cases written by the instructor.

LC architecture. The assembly-language program computes $\binom{n}{r}$ (the number of r combinations that can be chosen from a set of n elements), using the following recursive definition: $\binom{n}{0} = \binom{n}{n} = 1$; $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$. Writing this program gives students experience with function-calling conventions, including the management of stack-based activation records and callee-save/caller-save registers. As with the assembly-language program in Project 1, this part is evaluated on multiple (six) sets of inputs using the solution assembler and behavioral simulator. Students are told if their program generated the wrong result, had an assembly-language syntax error, or did not conform to the specification in certain ways (e.g., how it stores input values or the maximum number of instructions executed).

For the second part of this project, students are given a data path and are asked to design a control unit that implements the LC architecture on this data path. The control unit is a finite-state machine that specifies a sequence of data path operations for an arbitrary sequence of LC instructions. Data path operations may transfer data between micro-architectural registers or memory, or they may perform an arithmetic computation on micro-architectural registers. Each operation in the finite-state machine is expressed as a C/C++ statement that manipulates a state variable in the data path. The next-state function in the finite-state machine is expressed as a "goto" statement. To make the finite-state machine's operation visible to the feedback system, students call an instructor-supplied function (printState) to print the current state of the data path. The following example shows a simple state that uses the system bus to transfer the program counter's value to the memory address register, then goes to a state that reads the memory at that address:

```
fetch:
  printState(&state, "fetch");
  memoryAddress=programCounter;
  goto readMemory.
```

The "state" variable in the above code is a C structure that encapsulates all parts of the data path state, including the architectural state (program counter, general-purpose registers, and memory contents) and the micro-architectural state (memoryAddress, memoryData, instrReg, aluOperand, and aluResult).

Conditional next-state functions are expressed as C/C++ "if" statements with a "goto" in the body of the if statement; this style of programming is equivalent to a Moore finite-state machine (i.e., the inputs to the state machine affect only the next-state control signals). The following example shows a simple state whose next state depends on the instruction opcode (stored in instrReg).

```
decode:
  printState(&state, "decode");
  if ((instrReg≫22)==ADD) { goto add;
}
```

Students integrate this control unit into the Project 1 simulator to form a lower level, more detailed simulator of the LC architecture.

The feedback system runs the finite-state machine on ten test machine-language programs. On each test program, the system evaluates the finite-state machine according to several criteria. First, each transition between states must be legal; that is, the data path must be able to change between these states in a single cycle. For example, it is legal in a single cycle 1) to read the value of one of the general-purpose registers and transfer that value to the aluOperand register, and 2) to read the memory location specified in the memoryAddress register. However, it is not legal in a single cycle to use the bus to transfer two different values, because the hardware does not support this combination of operations in a single cycle. Checking the legality of each state transition is accomplished by comparing the starting and ending values of each data path variable and seeing if this transition was possible given the starting state and the transitions being made by other data path variables. If the system detects an illegal state transition, it tells the student which starting and ending state label was involved in the illegal transition. Unfortunately, illegal operations that control the next state in the sequence are difficult to detect solely by examining each state transition. For example, a finite-state machine may resolve a branch comparison before reading the registers being compared, and this illegal resolution cannot be detected without understanding the structure of the finite-state machine. For these cases, an illegal operation is detected if the student's program simulates a test machine-language program in fewer cycles than the instructors believe to be possible.

Second, after running a test machine-language program to completion, the feedback system checks to see that the final architectural state is correct. One cannot check the architectural state after each instruction because a clever finite-state machine may overlap the execution of several instructions.

Finally, the feedback system evaluates how efficiently the student's finite-state machine executes each test machine-language program. The system counts how many datapath cycles a student's finite-state machine takes to execute each machine-language program and gives full credit only when the cycle count matches the solution finite-state machine. The system tells students how close their cycle count was to the optimal. Grading students on efficiency results in several complications. Because certain illegal state transitions are detected by seeing if a finite-state machine takes fewer cycles than possible, one cannot simply require a finite-state machine to run in the same or fewer cycles than the solution finite-state machine; students must be required to achieve exactly the same cycle count as the solution. Under this scheme, a student may be penalized unjustly if he or she invents a more clever finite-state machine than the solution.

This situation occurred several times over several semesters as creative students went through the course. It was resolved as follows: in the semester, the new optimization was discovered, the feedback system was modified to give full credit to solutions with or without the optimization; in future semesters, the new optimization was required, but students were given a very explicit hint to help students discover the optimization. Students were given an explicit hint because it did not seem fair to require students to invent (without help) an optimization tricky enough to be overlooked by the instructors. In hindsight, a better solution may be to require finite-state machines to run in the *same or fewer* cycles than a certain minimum and simply leave the illegal next-state transitions undetected.

The following is a partial example of the feedback students receive for Project 2's simulator.

```
----0----
Finite-state machine ran for more than
  5000 cycles. Your program probably
  entered an infinite loop
----1----
perfect!
----2----
Wrong final answer
----3----
Illegal state transition from jalr1 (cycle
  14) to jalr2 (cycle 15)
----4----
Right final answer; no illegal state
  transitions (that the auto-grader could
  pinpoint); but your FSM executed 23 more
  cycles than needed.
----5----
There was an illegal state transition
  that the auto-grader couldn't pinpoint.
  Project 2 Tips on the course home page
  has some suggestions on what the error
  might be.
```

C. Project 3: Pipeline Simulator

For the third project, students write a program that simulates a pipeline implementation of the LC architecture. Students are given the pipeline data path from [2, Ch. 6] and specific directions on how to resolve control and data hazards. Each stage of the pipeline takes its input from the prior pipeline register, computes the values for the next pipeline stage, and then stores the results into the next pipeline register. The pipeline uses simple branch prediction (assumes branches are not taken) to resolve control hazards and uses a combination of data forwarding and stalling to resolve data hazards. As the program runs, it prints the architectural and micro-architectural state of the pipeline data path. The micro-architectural state consists primarily of the values in the pipeline registers.

To evaluate a student's pipeline simulator, the feedback system runs it on a suite of 25 LC machine-language programs. The large number of test cases are needed to exercise the simulator on the wide variety of possible pipeline conditions. For each test case, the system compares the cycle-by-cycle output of architectural and micro-architectural state generated by the student simulator against the output generated by the solution pipeline simulator. Students are told if their output is incorrect, which variable of the architectural or micro-architectural state is incorrect, and in which cycle the error occurred.

The following is a partial example of the feedback students receive for Project 3.

```
----0----
perfect!
----1----
solution file and student file differ in
  print State before solution cycle 7
  starts
error occurred in IDEX
suspect line:     readRegA 0
----2----
solution file and student file differ
  in printState before solution cycle 18
  starts
error occurred outside a pipeline register
suspect line:     dataMem[ 39 ] 0.
```

D. Project 4: Caching Simulator

The final course project simulates the behavior of adding a CPU cache to the Project 1 behavioral simulator. Students simulate a unified instruction/data cache that uses a write-back policy on stores and an LRU (least-recently used) replacement policy. Other cache parameters (e.g., cache size, line size, and set associativity) are given as arguments to the simulator. As the simulator executes an LC machine-language program, it services memory loads and stores by looking for data in the cache and issuing loads and stores to memory to transfer data to/from the cache if needed. While the simulator runs, it prints output describing the movement of data between the memory, the cache, and the processor. This output helps the student understand how caches work, and it also exposes the behavior of the cache to the feedback system. Earlier versions of this project had students write a traditional cache simulator that took as input a trace of memory references, but students seemed to learn more about caching processors by actually simulating an LC program that generated memory references as it ran.

To evaluate a student's caching simulator, the feedback system runs it on a suite of 11 test cases. Each test case specifies an LC machine-language program and a cache configuration. For each test case, the system compares the movement of data generated by the student simulator with the movement of data generated by the solution caching simulator.

The following is a partial example of the feedback students receive for Project 4.

```
----0----
master file and student file differ at
  student file line 24
student line: @@@ transferring word [4-4]
  from the cache to the processor
----1----
perfect!
```

III. Experience

Students' initial reaction with automated feedback was very positive. They liked finding out their project grade right after submitting, and they considered any additional feedback a bonus. In addition, with the feedback system, nearly all students were able to build programs that had no functional errors. In contrast, before using the feedback system, only slightly more than half the students were able to build programs that had no functional errors. Providing automated feedback also enabled more rigorous enforcement in what students were expected to build. Without feedback, students can easily make trivial errors that result in large grading penalties. These trivial errors result in either inappropriate penalties or enormous expenditures in human grading time (because the grader must fix these bugs before regrading). With feedback, however, students find out immediately if their program has a small bug that results in a large penalty, and they can then fix the bug and resubmit. The feedback system thus allows students to be held to a higher standard and enables students to meet this standard. The feedback system also cut down dramatically on the human effort required to grade projects. This savings helped greatly during the rapidly rising enrollments of the late 1990s.

However, in later semesters, as students gained experience with the system, a few negative trends began to emerge. Students began to view the automated feedback more like an entitlement than a bonus, and as this occurred, their expectations of the system began to change and they began to depend on the system in ways for which it was not intended. Specifically, they tried (usually in vain) to use the feedback system as a replacement for their own testing and debugging. The feedback system was never intended to replace the need for student testing and debugging. In fact, the task of testing and debugging is a valuable way of learning the material on computer architecture and is a necessary part of any programming project (and indeed any design project).

In hindsight, obviously some students could be expected to try to abuse the system in this manner. However, one is surprised at how many students abused the system and how steadfastly they claimed to be justified in this usage. In particular, students were outraged that the system would run their programs against test cases, tell them that their programs had errors, but *not* give them the test cases that exposed their errors. This outrage resulted from students trying to use the feedback system in lieu of their own debugging. One clearly has considerable difficulty debugging a program without having on hand a test case that generates the error. The author naively expected that just as students knew they needed to construct their own test cases without the feedback system, they would see the need for constructing their own test cases even with the feedback system. Unfortunately, despite many attempts to change students' perceptions of how to use the feedback system, students continued to try, in vain, to use the system as their main debugging tool and to complain vociferously when this procedure proved impractical. In fact, they complained so much about the system that the course instructors came very close to turning off the automated feedback system and using it only as a tool for grading. Fortunately, a better way than turning off the feedback system can address students' perceptions of testing; the next section describes this solution.

IV. Teaching Students the Importance and Practice of Testing

The biggest drawback to using the automated feedback system was the tendency of students to cease testing and debugging their own programs. Without the automated feedback system, students were uncertain about what grade their program would receive, and this uncertainty motivated them to test their programs (though not thoroughly enough). With the feedback system in place, students lost this motivation and ceased constructing and using test cases to debug their programs. They then complained about how impractical it was to test their programs without any test cases.

The initial attempt to solve this problem was to limit the number of times students could submit their programs. Before limiting submissions, some students submitted programs as often as once per minute, some of which did not even compile. Several policies were tried, such as taking points off for each submission and only allowing one submission per day. These policies had mixed results. On one hand, they did have the intended effect of forcing students to do some testing on their own. On the other hand, these policies did *not* change their overall viewpoint that testing and writing test cases was a waste of their time and that it was cruel to force them to write test cases when the system could have easily given them the instructor's test cases.

In the fall semester 2000, the author decided to address head-on the problem of students viewing testing as a nonintegral activity. Others have also argued for emphasizing testing as a first-class activity, both by integrating testing into existing courses [3] and by teaching testing in a separate course [4]. The author's approach was to integrate testing into this course by requiring students to submit test cases as a graded part of each project. The hope was that this requirement would have the following benefits:

1) making test case construction a graded part of each project would elevate their view of testing as a first-class activity (although many students end up working on the testing side of software projects rather than the designing and building side, most computer science curricula emphasize designing and building to the near exclusion of testing);
2) writing a suite of test cases that comprehensively tested the project functionality would help them learn the course material;
3) writing a good suite of test cases would help them debug their own programs;
4) the experience they gained from writing a good suite of test cases for computer organization projects would improve their general ability to test computer programs.

A good test suite systematically exercises all aspects of the project specification. For example, to write a good test suite for Project 3, a student should think through all the possible types of pipeline hazards and construct test cases that exercise each hazard. To write a good test suite for Project 4, a student should

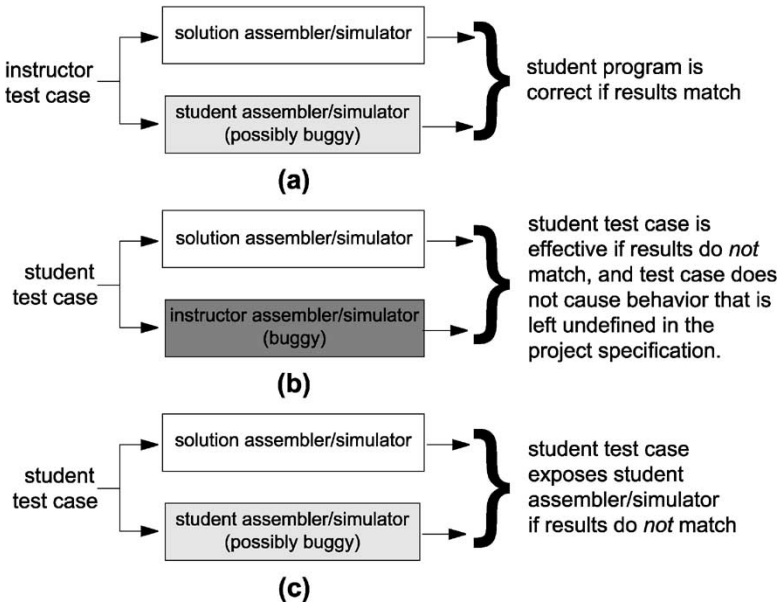

Fig. 2. Experiments run by the revised automated feedback system. Lightly shaded boxes indicate programs that may have a bug; dark shaded boxes indicate programs that definitely have a bug. (a) (The same as Fig. 1.) The system tests a student's assembler or simulator by running it on test cases written by the instructor. (b) The system evaluates the effectiveness of a student test case by using it as input to both a known-good program and a known-buggy program. (c) The system helps the student by identifying which of their test cases exposes their own program as having a bug.

think of the possible states of a cache line (e.g., empty, clean, or dirty), think about how different events in the system cause the cache lines to change state, and write test cases that cause a cache line to transition through systematic sequences of states.

Having decided to require testing as a first-class activity, the next issue was how to help students learn to write a good suite of test cases. Just as with writing programs, testing programs is learned largely by doing. A key insight was that one could apply automated feedback to the testing part of a project and that the iterative feedback process could help students learn how to write good test suites. Ironically, a problem that arose because of automated feedback (that of students not testing their programs) ended up being solved by the same mechanism.

The system was enhanced, as follows, to provide feedback for student test suites [Fig. 2(b)]. The first step was to write a number of buggy programs (e.g., several buggy simulators for Project 1). Students would submit their own test suite (a set of test cases) with their program. The system would grade a student test suite based on how many of the instructor's buggy programs were exposed as incorrect by at least one of their test cases. To expose a buggy program, a test case must cause the program to execute incorrectly, i.e., differently from the behavior specified in the project assignment. As an example, consider a buggy Project 1 simulator that performs a subtraction instead of addition for the add instruction. A successful test suite would include at least one test case that exposed this program by adding two numbers with the second operand being nonzero, so that subtraction and addition could be differentiated.

Providing automated feedback for a student test suite is essentially the inverse problem of providing feedback for a student program. To provide feedback for a student program, the system runs the student program against the instructor's suite of test cases and grades them based on how many of the test cases produce correct results on their program. To provide feedback for a student test suite, the system runs a buggy set of instructor programs against the student test suite and grades them based on how many of the buggy programs produce incorrect results on at least one of the student test cases. The infrastructure for the automated feedback system was easily extended to handle this inverse case.

As an added bonus for the students, the system also runs a student's test suite against the student's own program and notifies the student if any of the test suite exposes the program as buggy [Fig. 2(c)]. Strictly speaking, this service should not be needed; students can certainly run their own programs on their own test suite. However, this feature often helps students who misunderstand the project specification, thus failing to recognize certain executions as being faulty. This feature helps alert these students to their misunderstanding before they waste considerable time looking for implementation errors in their programs. This feature also adds incentive to students to write good test cases. After they are told which of their test cases exposes their own program as buggy, students can focus on these test cases and use them for productive debugging.

The following is an example of the feedback students receive on their test suite.

```
----grading student test case tc0.as---
student test case tc0.as exposed the fol-
  lowing instructor buggy simulators:
  A D G L
-----grading student test case tc1.as---
student test case tc1.as exposed the fol-
  lowing instructor buggy simulators:
  A B D E H I
-----grading student test case tc2.as---
student test case tc2.as executed too many
  instructions
```

```
-----grading student test case tc3.as---
error: offset 32 769 out of range
illegal assembly-language program
overall student test suite exposed 8 of
  the 12 instructor buggy simulators
Hint: the following student test cases ex-
  posed the student simulator as buggy:
  tc1.as.
```

The experience of requiring students to submit test suites was very positive. Students were forced (often for the first time) to think through how to test systematically that a program met a specification. Their attitude toward testing improved as well, because they could see how useful a good suite of test cases was when they were debugging their programs. Providing automated feedback for student test suites had similar benefits to providing automated feedback for student programs; this feedback helped students learn what constituted a good suite by giving them quantitative feedback, and it encouraged them to persevere until their test suite received full credit.

Some students did struggle to understand what it meant to expose a program with bugs in its error-handling code. For example, the Project 1 assembler was required to handle the following errors: undefined instructions, undefined and duplicate labels, and out-of-bound memory addresses. If one of the instructor's buggy programs contained a bug that prevented the program from recognizing a duplicate label, exposing this bug would require a test case that contains a duplicate label. Instead, many students mistakenly provided test cases that contained errors for which the project assignment did not specify correct behavior. For example, the project assignment did not specify what the assembler should do if it encountered an out-of-range register number. Because correct behavior is undefined, to say that a program incorrectly handled a test case with an out-of-range register number is meaningless. Since such test cases cannot expose any buggy programs, the automated feedback system was modified to detect them and prevent them from exposing any programs.

V. Other Lessons Learned

This section describes some of the lessons that were learned from five years of providing automated feedback. The first lesson was an awareness of how much work is needed to design and build a robust correct feedback system. To build a system that can handle all the mistakes and new solutions generated by 200 creative students is difficult. This fact was especially evident for the part of Project 2 that challenged students to optimize the finite-state machine. Students invented new optimizations that the instructors did not anticipate; in fact, students kept inventing new optimizations even after the system had been refined over several semesters of 200-student classes. To make the workload of creating the feedback system more manageable, projects from prior semesters were reused heavily. This practice also improved the quality of the feedback system and clarity of the projects because it takes several semesters to create a system that works well. Unfortunately, cheating is always an issue whenever one reuses projects from prior semesters, and this situation was the case at the University of Michigan as well. Over the course of a semester, 5% of the class were regularly caught turning in prior solutions as their own work. The feedback system uses several automated correlators to prioritize which programs to investigate by hand. Unfortunately, a steady stream of cheaters have continued to be caught every semester, despite very explicit warnings about the number of students who have been caught before (this experience differs from the one reported in [5]). Despite the problems with cheating, instructors at the University of Michigan have chosen to continue reusing projects and posting solutions afterwards. This choice was made because the highest priority was to create the best learning experience for honest students, and this priority required reusing projects that had been refined and validated over several semesters.

Other lessons were learned about how to write test cases to evaluate student programs and how to write buggy programs to evaluate student test suites. Because the instructors use the feedback as their actual grade, the grade needs to be roughly proportional to the degree of correctness in their programs. Two steps were required. First, the assignment requires little work to get the trivial functionality correct. For example, instructors provide the parser code for the Project 1 assembler, because writing a parser has nothing to do with the concepts in the class. Instructors always provided code to print the required program output. These provisions enabled students to comply easily with the requirements on output format. Also the instructor was relieved from writing a sophisticated output parser that ignored unimportant differences in formatting [6]. Second, the system uses test cases constructed with a variety of difficulties. Some test cases (and, hence, some points awarded) were very simple, such as one-line assembly-language programs. Later test cases required more functionality to be correct [6]. Writing good test cases took a substantial amount of thought and care.

Examining statistics on how students used the automated feedback system is interesting. In the Fall semester of 2002, the average number of submissions across all students was 3.7 for Project 1's simulator, 4.2 for Project 2's simulator, 5.6 for Project 3's simulator, and 4.6 for Project 4's simulator. As expected, the harder the project, the more times students submitted before completing the project. Fig. 3 shows how scores for Project 1 assemblers increased as students debug and resubmit their program. The graphs for other projects showed similar trends.

Other practical lessons were learned about how to build the submission and grading infrastructure. The submission mechanism uses e-mail to submit student projects. The feedback was, likewise, carried back to the students via e-mail. Using e-mail allowed asynchrony between the students and the feedback system, and this asynchrony had some benefits and some problems. The main benefit of the asynchrony of e-mail was that it allowed students to submit their programs even if some of the networks and computers (name servers, routers, and mail servers) that needed to communicate to the grading system were temporarily broken (which happened fairly often). E-mails are dated by college-administered computers, which allows instructors to grade projects fairly that were submitted on time but received later. E-mail responses normally reached students

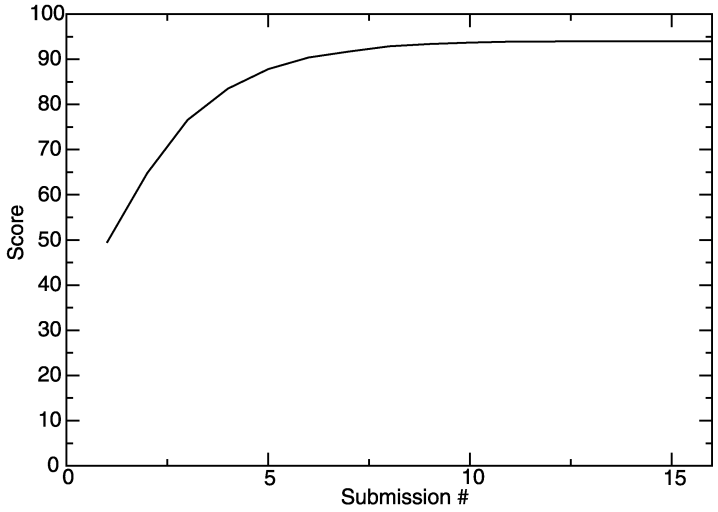

Fig. 3. Progression of scores as students submit programs. This graph shows how student scores increase as they fix and resubmit their Project 1 assembler. The submission patterns for other projects are similar.

in a few seconds; however, overloaded or malfunctioning mail servers or networks occasionally delayed the feedback for several hours (which naturally worried students). Overall, using e-mail as our means of communication has worked well. It is ubiquitous, and it tolerates computer/network faults well. An alternate approach is to establish a direct, synchronous connection between the student's computer and the grading computer [7], but this approach disallows submissions when the network or grading machine is down.

A small lesson learned was to make the submission process as easy as possible for students to use correctly. An early system had them simply execute "mail < program.c". This command led to disastrous consequences when students mistakenly used "mail > program.c" which overwrites the program rather than submitting it.

Finally, experience taught lessons about how to make the feedback system robust in the presence of student programs that are incorrect or malicious. These programs can cause problems because the system executes unaudited code. This concern was addressed by running student programs on an unprivileged user account, by limiting the processing and memory resources available (using the Unix setrlimit call), and by saving a copy of each submission on a different computer to allow analysis should it be malicious. More stringent mechanisms exist to sandbox student programs, such as disallowing certain system calls [8]. However, to date, no real problems with malicious students have occurred.

VI. Related Work

Prior literature contains several descriptions of systems that analyze student programs automatically [5]. Examples include the TRY system [6], ASSYST [9], and PSGE [10]. Some systems provide immediate feedback to the students [6], [11]. Many of the mechanisms used in the present paper for improving security and robustness have been used before. The TRY system runs student programs in a low-privileged user ID, and Arnow's homework checker enforces run-time resource limits.

The system described in this paper differs from prior work in several ways. The most significant difference is the present system's emphasis on testing as a first-class activity. Some prior systems provide the student with the test cases used by the system; the author believes that constructing good test cases helps students learn about computer organization and about computer programming in general. The present system addresses head-on what the author considers the primary danger of automated feedback systems, which is the tendency of students to depend on the system as a replacement for their own debugging and testing. ASSYST [9] is the only automated program analyzer known to the author that also requires students to submit test cases. ASSYST evaluates a student test case based on how many lines of the student's program are executed by running the test case. This approach (white-box testing) measures how effective the test case is on the specific student's program; in contrast, the approach used by the present system (black-box testing) measures how effective the test suite is on evaluating any program written to implement the project specification. While both approaches are useful, students will find valuable thinking through how to exercise a specification thoroughly and systematically, in part, because they are forced to think through the specification (and thus the course concepts) in a new way. ASSYST also differs from the present system because it serves only as an aid to the graders; it does not provide immediate feedback to the students.

A second difference from prior systems is that the present automated feedback system takes place in the context of a computer organization course and uses domain-specific knowledge when evaluating student programs. For example, the Project 2 finite-state machine has an infinite number of correct (but not necessarily optimally efficient) solutions, and the feedback system requires knowledge about hardware constraints to distinguish between correct and incorrect solutions.

VII. Conclusion

The author experimented for five years with a system to analyze student programs automatically and provide limited

feedback to the students. The system provided many benefits. It improved students' learning experience by allowing and encouraging them to improve their program iteratively until it was correct. It enabled the addition of challenging parts to each project, such as optimization and testing, and enabled students to meet these challenges. It significantly reduced the grading load of large class sizes and helped instructors handle the rapidly increasing enrollments of the 1990s.

Initial experience of the feedback system exposed an inherent difficulty in automated feedback systems, which is that students depend inappropriately on the feedback system as a substitute for their own testing. This problem was addressed head-on by requiring students to submit a comprehensive test suite along with their program, then applying automated feedback techniques to help students learn how to write good test suites. Quantitative iterative feedback was found to be extremely helpful in teaching students specific concepts about computer organization and general concepts on computer programming and testing.

ACKNOWLEDGMENT

The author would like to thank M. Brehob, M. Papaefthymiou, and G. Tyson, who have used the automated feedback system while teaching the computer organization course at the University of Michigan helped refine the projects, the automated feedback system, and this paper. The first two course projects were derived from projects designed by Y. Patt.

REFERENCES

[1] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, CA: Morgan Kaufmann, 1994.

[3] E. L. Jones, "Integrating testing into the curriculum—Arsenic in small doses," in *Proc. 2001 SIGCSE Tech. Symp. Computer Science Education*, Feb. 2001, pp. 337–341.

[4] D. Carrington, "Teaching software testing," in *Pro. 1996 Australasian Conf. Computer Science Education*, July 1996, pp. 59–64.

[5] D. G. Kay, T. Scott, P. Isaacson, and K. A. Reek, "Automated grading assistance for student programs (panel presentation)," in *Proc. 1994 SIGCSE Symp. Computer Science Education*, 1994, pp. 381–382.

[6] K. A. Reek, "The TRY system -or- how to avoid testing student programs," in *Proc. 1989 SIGCSE Tech. Symp. Computer Science Education*, 1989, pp. 112–116.

[7] (1999, Aug.) Enhanced Automated Grading System: Student Guide. Dept. Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA. [Online] Available: http://ei.cs.vt.edu/eags/EAGS.html

[8] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications," in *Proc. 1996 USENIX Security Symp.*, July 1996, pp. 1–13.

[9] D. Jackson and M. Usher, "Grading student programs using ASSYST," in *Proc. 1997 SIGCSE Tech. Symp. Computer Science Education*, 1997, pp. 335–339.

[10] E. L. Jones, "Grading student programs—A software testing approach," *Proc. 14th Annu. Consortium on Small Colleges Southeastern Conf.*, pp. 185–192, 2000.

[11] D. Arnow, "When you grade that: Using e-mail and the network in programming courses," in *Proc. 1995 ACM Symp. Applied Computing*, 1995, pp. 10–13.

Peter M. Chen (M'94–SM'00) received the B.S. degree in electrical engineering from Pennsylvania State University, University Park, in 1987 and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1989 and 1992, respectively.

He is currently an Associate Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His research interests include operating systems, fault-tolerant computing, computer security, and distributed systems.