SPIMbot: An Engaging, Problem-based Approach to Teaching Assembly Language Programming

Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
zilles@cs.uiuc.edu

ABSTRACT

This paper describes SPIMbot, an extension to James Larus's widely-used MIPS simulator SPIM, that allows virtual robots to be controlled by writing programs in the MIPS assembly language. SPIMbot was written to provide an engaging environment to motivate students to learn assembly language concepts. The SPIMbot tool allows the development of scenarios—in which students must program the robot to perform certain tasks—and provides the means to compete two robots against each other.

In our sophomore/junior-level class, we structure the programming component as a collection of structured assignments that produce sub-components for the robot; these sub-components are then used in a final open-ended programming assignment to produce an entry for a SPIMbot tournament. In our experience, this has been an effective means of engaging students, with many students investing time to aggressively optimize their implementations. SPIMbot has been effectively used in large classes and its source code is freely available [8].

1. INTRODUCTION

As one of their "Seven Principles for Good Practice in Undergraduate Education", Chickering and Gamson [1] list **emphasizing time on task** as number 5. They state:

> Time plus energy equals learning. There is no substitute for time on task.

Thus one of our chief tasks as undergraduate educators is to develop activities that encourage our students to spend time on the course concepts and approach them with desire to master them. This paper describes one such set of activities, focused on teaching concepts related to assembly language programming.

In the remainder of this section, we describe the motivation for this work (Section 1.1) and abstractly how we use SPIMbot to achieve our pedagogical goals. After discussing the capabilities of the software (Section 2), we discuss, in detail, how it was used in the Spring 2004 semester (Section 3). We conclude, in Section 4, with a discussion of student feedback that supports our assertion that SPIMbot is an engaging way for students to learn assembly language programming concepts.

1.1 Motivation

In teaching assembly programming in our Computer Science curriculum[1], we have two primary goals: 1) to provide students a mental model of how a computer executes their high-level language (HLL) programs, and 2) to provide the background knowledge necessary for later courses on compilers and operating systems. To this end, we teach the students about instruction sets, stacks and their management (including recursion), calling conventions, floating

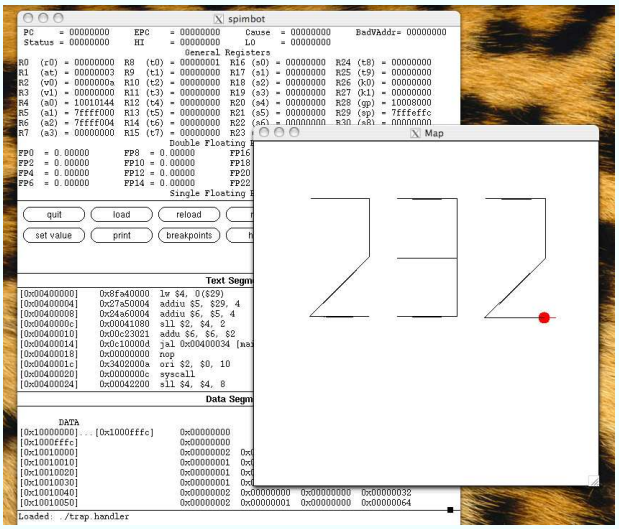

Figure 1: Example SPIMbot screen shot. *The* map *window shows the robot's current location, orientation, and virtual environment; in this scenario, SPIMbot can turn on/off a paint trail allowing it to write out messages. Behind the map window is the main window (unmodified from xspim) that shows the MIPS processor's machine state.*

[1]Assembly programming is taught in the context of the second semester-long class in a required two-class sequence in computer architecture. The first class in the sequence teaches digital fundamentals: the digital abstraction, combinational logic, finite-state machines, and basic architecture concepts (*e.g.*, a single-cycle implementation). The second class covers three main topics: assembly programming, machine organization, and memory and I/O systems; each topic receiving roughly a third of a semester. As our undergraduates predominantly pursue software-oriented (rather than hardware-oriented) careers, the goal of this second class is to provide the practical understanding of computer hardware necessary to be an effective programmer. Most students continue their architecture sequence, taking a third course in either high-performance architecture or embedded systems.

point arithmetic, instruction encoding, I/O interfacing, and interrupt handling.

If one is not careful, these topics can come across as dry. The students' limited programming experience (this class is early in the curriculum) coupled with the inherent inefficiency of assembly programming can limit the scope of programming assignments. Furthermore, the demands of grading, especially in large enrollment classes where some form of automation is necessary, require most assignments to be rather structured. Examples of common assembly programming assignments found at many universities include: producing the Fibonacci sequence, string manipulation (reversing a string, `toupper()`, etc.), and sorting arrays. In many cases, HLL source is provided, reducing such assignments to somewhat mechanical translation.

The goal of SPIMbot was to produce an environment for teaching assembly programming that was fun and interesting, to motivate students to want to learn the material. While there is a long history of using robots for instruction (*e.g.*, [5]), the author's inspiration came from Patricia Teller's presentation [7] at the 2003 Workshop for Computer Architecture Education. In their semester-long course on assembly programming concepts, students program 68HC11-based robots to escape from mazes and chase other robots. Pedagogically, programming robots has three appealing features: 1) it is visceral: students like seeing their code control motions and actions of objects in the physical world, 2) it is cognitively challenging: debugging requires mapping robot behavior back to the behavior specified in the code, and 3) it provides a non-contrived way to expose students to I/O programming.

The problem with (physical) robots is one of logistics; in a high enrollment class—we have 100-150 students per semester—acquiring, maintaining, and scheduling sufficient resources is prohibitive. In contrast, virtual robots are cheap, plentiful, take-up no space, require no maintenance, yet (for students accustomed to interpreting computer-rendered virtual realities) still provide the fundamental qualities of physical robots.

1.2 How we use SPIMbot

The central part of our implementation is the SPIMbot tournament, a friendly competition between the programs that the students write. The contest presents a challenging, multi-part task for the robots to perform. We use this concrete task to motivate the presentation of the desired assembly language concepts and the problem solving/design process.

As most of our students have not been exposed to assembly language previously, the SPIMbot tournament is the last activity in our assembly language segment. We work up to the contest by solving isolated sub-problems as programming assignments. We start with small structured assignments and then move onto larger structured assignments before attempting the contest (a large open-ended assignment). This structure lets us provide the students with early, motivating successes.

Although it is the last assignment, we present the contest first, because it allows us to model a problem solving process: a top-down design, followed by a bottom-up implementation. In class, we brainstorm approaches to the contest task, making it clear that there are multiple approaches. Then, we identify sub-tasks necessary for accomplishing the contest goal; these sub-tasks make up the structured programming assignments leading up to the contest. The contest itself challenges students to figure out what they need to implement and requires them to integrate the components they've completed in previous assignments.

When it comes to covering the desired course material, the fact that SPIMbot exists only in a virtual reality can be an advantage, as we can structure that reality to include those concepts that we want to teach. For example, two concepts that we cover in the course are recursion and the implementation of linked-data structures. To incorporate these concepts into our programming assignments, our Spring 2004 contest (see Section 3) involved an I/O device that returned its output as a tree, requiring students to write a recursive procedure to traverse the nodes of the tree.

After the students have submitted their contest entries, we use one class period to hold a tournament. With each competition lasting about 15 seconds, a double-elimination tournament for 32 teams can easily be held in a 50-minute class period. While this class time could be used for other purposes, we believe that it successfully motivates students to be actively engaged with course material *outside of class* achieving our objectives.

A Note on Competition: As competition can be demotivating if not handled properly [2, 3], we take a number of steps to alleviate the potential downsides of competition: 1) performance in the competition is responsible for a minimal fraction (about 1 percent) of student's final grade, 2) students compete as teams, reducing the pressure on individuals, and 3) teams select team names allowing students to compete anonymously.

2. SPIMBOT SOFTWARE

SPIMbot is an extension of James Larus's widely-used MIPS simulator SPIM [4]. SPIMbot involves three major enhancements: 1) a framework for simulating robots and their interactions with a virtual world, 2) a 2-D graphical display to visualize the robots and their environments, and 3) support for concurrently simulating multiple programs—each on their own virtual processor—allowing multiple robots to be simultaneously active in a single virtual world.

Simulating the virtual world requires tracking and updating the state of the robots and other objects in the simulated world. In addition to location, orientation, and velocity, we have to keep track of the state of any I/O devices. Updating the world involves computing new locations for objects based on their current velocities. Collision detection is performed to update an object's velocity/orientation (*e.g.*, when a robot runs into a wall) and to allow interaction between robots and simulated objects (*e.g.*, when a robot picks up an object or pushes a button). Events in the virtual world can also trigger events in the MIPS processor, either updating the state of an I/O device and/or triggering an interrupt.

To interact with the virtual world, SPIMbot provides the robot programmer an (extensible) array of input/output devices. These virtual I/O devices, like real I/O devices, have their I/O registers mapped to memory addresses and, thus, are accessed using normal loads and stores. Simple examples include "sensors" that tell SPIMbot its or the opponent's (X,Y) coordinates and "actuators" to control its orientation. The SPIMbot code is structured so that the collection of I/O

devices can easily be extended for a particular scenario. Furthermore, SPIMbot includes a programmable interrupt controller (PIC) that allows individual device interrupts to be enabled/disabled. Standard interrupts include the "bonk" interrupt (raised when SPIMbot runs into something) and timer interrupts (SPIMbot includes a programmable timer). The collection of interrupts can also be extended.

To achieve a tight coupling between the virtual world and the simulated MIPS code, we interleave the simulation of the virtual world with that of the MIPS code. Every *cycle* we execute a single instruction for each robot and update the physical world based on the actions of the robots. Simulating multiple concurrent robots required eliminating the use of global variables in SPIM's parsing and simulation of MIPS code; while currently we only simulate two robots, this could easily be extended to any number. As there can be interactions between the robots, we alternate each cycle which robot is simulated first in an attempt to be fair.

The graphics are currently decidedly low tech—XWindows drawing primitives are used to draw geometric shapes (lines, boxes, circles, etc.)—but this appears to actually have two advantages: 1) it is very simple; a minimal amount of development time is required to add the rendering code for a new scenario, and 2) it is not distracting; students can focus on what the graphics represent instead of the graphics themselves. Because the graphics are not demanding, smooth animation can be achieved without state-of-the-art hardware. In part this is because the graphical display need not be rendered every cycle. Currently, we re-draw every 1024 cycles and can achieve a refresh rate over 60 Hz on a 1GHz laptop.

3. EXAMPLE SCENARIOS

In this section, we discuss one scenario in detail to demonstrate how we organize the competition and the assignments that lead up to it and, then, discuss two other competitions more briefly to demonstrate the expressiveness of SPIMbot.

3.1 Spring 2004: Token Collection

In the Spring 2004 semester, the competition revolved around collecting "tokens": 15 tokens were randomly placed on a square map, tokens could be collected by driving over them, and the location of tokens can be divined by using an I/O device called the "scanner." The winning robot was the one that collected the most tokens by the end of competition.

Writing a program to compete in the contest involved: 1) allocating memory for the results of a scan, 2) communicating with the scanner to initiate a scan, 3) handling the scanner's interrupt, 4) searching the tree-like data structure returned by the scanner for the location of tokens, and 5) repeatedly orienting SPIMbot toward a token and recognizing when it has arrived, until all tokens have been collected. As this represents a relatively difficult programming assignment for students at this point in the curriculum, we broke out major components of the program as individual programming assignments. Below is a list of the structured assignments that led up to the contest:

1. A SPIMbot introduction: write a simple interpreter that reads a string of commands (*e.g.*, turn, wait, paint on/off) and invokes provided functions that perform these actions. *Introduces students to SPIM/SPIMbot and exposes students to loops, arrays, calling functions, control flow and I/O interfacing.*

2. Arctangent Approximation: given the (x,y) location of 2 points, compute the angle to drive from one to the other using a Maclaurin series expansion. *Exposes students to computing in floating point.*

3. Tree Traversal: SPIMbot's scanner returns the location of the tokens embedded as leaves of a tree-like data structure. Students write a recursive function that traverses the tree. *Exposes students to linked data structures and recursive functions in assembly.*

4. Interrupt Handler: write an interrupt handler for the timer interrupt that commands SPIMbot to turn 90 degrees and resets the timer, resulting in SPIMbot driving in a square. *Introduces students to writing interrupt handlers.*

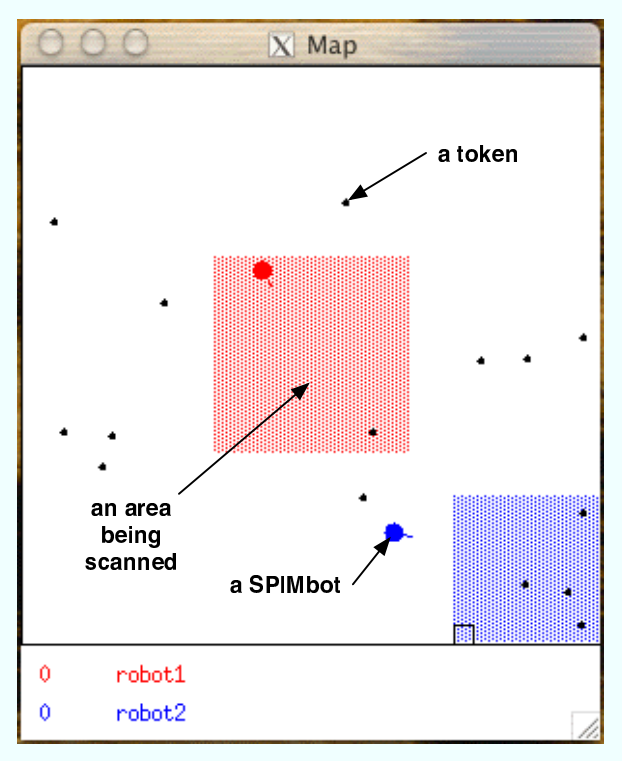

Figure 2: SPIMbot token collection competition.

While the solutions to these assignments can be integrated into a working contest entry, designing a competitive entry requires a little more effort. Three activities dominate the execution time of most of the robots: scanner latency, tree traversal, and collecting tokens. In a straight-forward implementation, which scans the whole map at once, these activities are performed completely sequentially (Figure 3a).

A higher performance implementation can be developed which pipelines the scan/traversal/collection process. The scanner can be programmed to scan only a portion of the map at a time, and its latency is largely a function of the area scanned. Once a small portion of the map has been scanned, the robot can begin collecting tokens from that

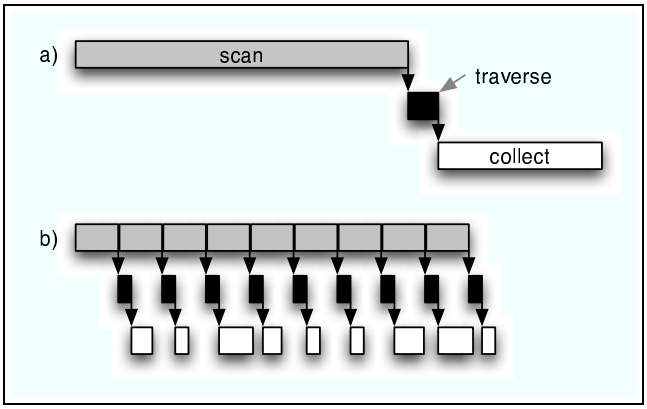

Figure 3: Pipelining the three sub-tasks reduces the latency of the task. *By scanning one-ninth of the map at a time, the pipelined version (b) overlaps the collection of tokens with the scanner latency, completing the task significantly before the non-pipelined version (a).*

portion while it requests the scan of the next region. In this way, much of the scan latency can be overlapped with the latency of tree traversals and token collection. Students found that breaking the map into 9-36 pieces and pipelining the processing of those pieces resulted in good performance. Another enhancement that students developed was driving to the center of the region currently being scanned after all known tokens had been collected.

Developing such a pipelined solution requires managing concurrent activities and demonstrates the importance of interrupts. The students learn first hand that their interrupt handler must avoid clobbering the applications registers, because it could be called at any time. It also demonstrates that pipelining—a concept we introduce in the machine organization portion of the class—is not a concept that is restricted to hardware.

3.2 Fall 2004: Block Pushing

In the Fall 2004 semester, the contest revolved around pushing blocks onto your side of the map (see Figure 4). The contest had a fixed running time and the winner was the one with the most blocks when time ran out. Elementary physics were implemented so that robots could push blocks, which in turn could push other blocks. An I/O device was provided that could be queried to provide the location of each of the blocks.

Like the token collecting contest, we integrated a computational challenge into the contest. Initially, a most of the blocks are "locked" to one or both of the robots. When a robot runs into a locked block, an interrupt is triggered and the robot receives a six character string. This string is a scrambled version of common english word, which, if unscrambled, can be used to unlock the block for this robot so that it can be pushed. As machine problems leading up to the contest, students wrote a string compare function, a function that would do a binary search of a sorted dictionary looking for a given word, and a recursive function that produces every permutation for a 6 character word. These functions can be integrated to unscramble the scrambled clues.

Because we provided the dictionary to the students ahead

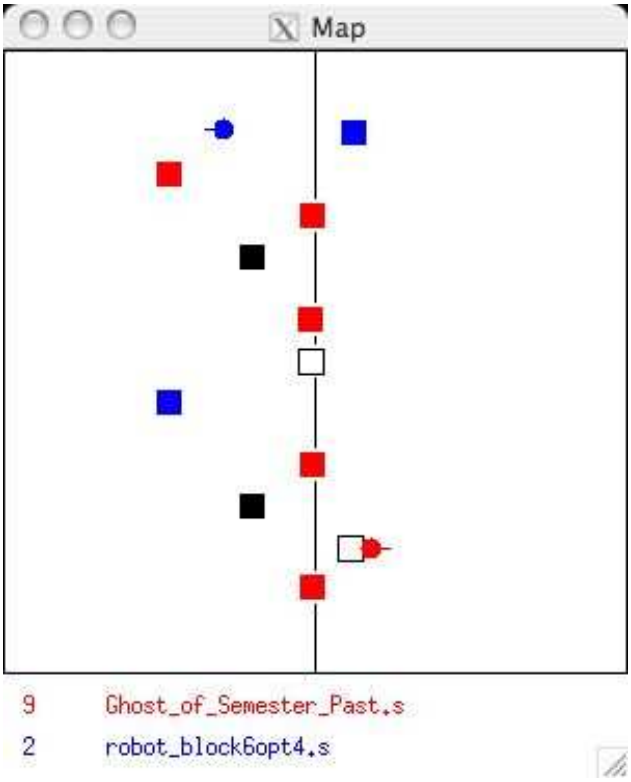

Figure 4: SPIMbot block pushing competition.

of time, there was a significant opportunity to optimize the unscrambling function by offline computation. The following is representative of what the winning robots did: 1) sort the characters in the scrambled word into a canonical order (*i.e.*, alphabetical order), 2) as only the 26 lower case letters were used, each ascii character could be represented in 5 bits; use this insight to translate the 6 char string into an integer (6 * 5 bits = 30 bits), 3) do a binary search on a precomputed table that maps these canonical integers to the strings they encode.

3.3 Spring 2005: Maze Traversal

This Spring semester our contest goal was to completely traverse a maze without being able to see the walls (see Figure 5). Since the mazes we generate are *unicursal* (*i.e.*, there are no isolated islands), the "right-hand rule" (*i.e.*, never letting your right hand leave the wall) can be used traverse the whole maze. Alas, SPIMbot does not have arms, much less hands, but the right-hand rule algorithm can be implemented with two interrupt handlers, by periodically checking to see whether a wall is still to the right of you, as follows: 1) request timer interrupts at a period so that roughly one is received for each square visited; when a timer interrupt is received, turn right and request another timer interrupt, and 2) when you run into a wall (which triggers a "bonk" interrupt), turn left. This was assigned to the students as a machine problem.

The computational challenge for this contest was to sort an array of double precision floats to find the Nth highest number (for varying N). Each time the correct value was identified, the SPIMbot was provided additional "energy"; energy could be used to drive faster, or, in large amounts, to

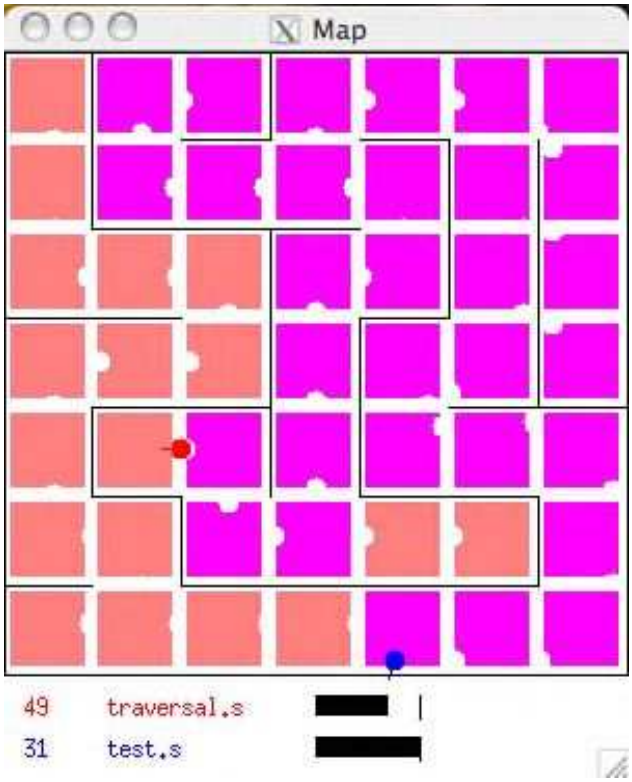

Figure 5: SPIMbot maze traversal competition. *The red squares are those that have only been visited by the red robot; the purple squares have been visited by both the red and blue robots (red + blue = purple). The black bars below the map indicate energy.*

drive through walls for short periods of time. Incorrect answers were penalized, so that the expected value of random guessing would be negative.

As a machine problem early in the semester, the students implemented a bubble sort, but there is clearly much opportunity to do better. A number of students implemented quicksort with the optimization of, at each stage, only sorting the partition that contains the Nth number. A few groups recognized that because a small error rate could be tolerated, the computation could be done on the integer pipeline only loading the top word of the doubles; this optimization saves one cycle on each of the load, because lw.d is translated into two instructions. The winning group realized that, because multiple guesses were allowed, the penalty for incorrect guesses was low enough that it was more efficient to guess an expected range for the Nth value (based on the properties of our random number generator) and perform a single pass over the array guessing any number in that range. In this way, their robot could maintain full energy while constanting driving through walls; their run time was minimized by finding the shortest path that visited every square.

Scenario Implementation Time: After the Spring 2004 semester, we re-factored SPIMbot's implementation to decouple the scenario-specific aspects from the core of SPIMbot's implementation. With these changes in place, it is rather straight-forward to implement new scenarios, by implementing a collection of functions for supporting scenario-specific initialization, physics, drawing, and I/O devices. The Fall 2004 scenario required about a day to prepare; this accounts for the time to implement both the SPIMbot code, as well as the MIPS code to test the scenario (which includes solutions to most of the structured assignments). The Spring 2005 scenario took longer to implement (perhaps a 40-hour week of programming time), but was largely completed by an undergraduate.

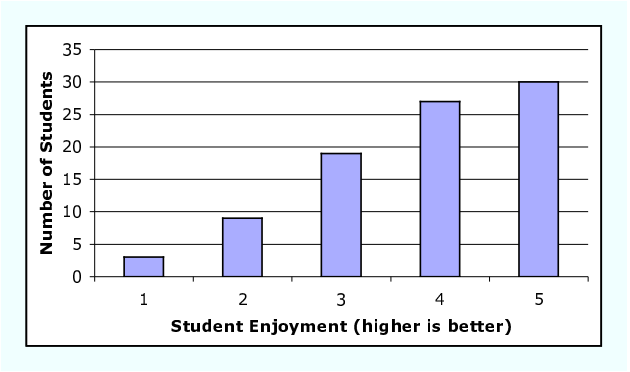

Figure 6: SPIMbot achieved high-level of student enjoyment. *Data shown for the 88 (out of 99) responding students for the Spring 2004 semester.*

4. STUDENT REACTION

The Spring 2004 students had a quite positive opinion of the SPIMbot assignments and student anecdotes suggest that they found it engaging. Students were asked in an anonymous electronic survey to rate their enjoyment of the SPIMbot assignments on a 5-point scale (5: "very much so" to 1: "not at all"). Of the 88 out of 99 students that responded, the mode was a 5 and the mean was just under 4 (see Figure 6).

In the course evaluations, six students commented specifically about SPIMbot when asked "What do you like about this course?", including the following quotes:

> "I really liked the SpimBot Tournament. That was the coolest thing I have done in a class. It makes it a lot more fun".
>
> "I liked the MP's, especially the SPIMbot Tournament and how the MP was designed to make us think of optimizations for ourselves."
>
> "... I also really liked the SPIMbot tournament"

The feedback was not uniformly positive, suggesting that there remain opportunities for improvement. One student mentioned SPIMbot in response to the question "What do you NOT like about this course?", giving the following response:

> "Spimbot. Pointless, difficult, and closed source, so hard to see exactly what was happening, so it's not entirely useful".

We have addressed this comment in more recent contests by providing the SPIMbot source to the students when the contest is assigned. By having the source, students can run

SPIMbot inside a debugger which helps them debug problems relating to interrupts, which are challenging to identify from SPIM's built-in debugger. We encourage students to inspect the code by stating that they are free to exploit any bugs they find[2]. A number of students do inspect the source; in the Spring 2005 contest, we received many comments about an unused "SPIMBOT_CHEATER" `#define` statement that was left in the code from when we were developing and testing the scenario. As the ability to efficiently read source code is a skill that comes with practice (one not emphasized early in our curriculum), organizing the contest in this way motivates some students to study the code.

Another measure of student engagement is the effort they expended. Along with their source code, students handed in a short write-up describing any noteworthy aspects (generally optimizations) of their program. Of the 30 teams, over 3/4's of the teams attempted optimizations with half completing significant optimizations:

- 15 teams (50%) described aggressive optimizations like segmenting the scan and the aforementioned pipelining,
- 5 teams (17%) described modest optimizations like greedily picking up the closest known token at any time,
- 7 teams (23%) reported attempting no optimization, and
- 3 teams (10%) reported attempting aggressive optimizations, but failed to get them working, requiring them to submit unoptimized versions.

Some of the teams that aggressively optimized their code reported trying a variety of techniques or parameterizing their code and tuning those parameters. Here are two student comments:

> "We tried many different strategies, including sorting the nodes in order of increasing distance from the spimbot, using an algorithm which heads toward the closest node to spimbot each time spimbot moves toward a new token, rescanning token locations to determine if they have been picked up, and breaking up the scans into different sizes. After trying all of these, we found that the only one which sped up the collection of tokens was breaking down the scan."

> "Our program does scans of size 25 thus giving 36 scans. We found this to be optimal because we started out with scans of size 5 doing 900 scans and found it to speed up as we approached 36. We even went down to 16 and found it slowed down as the scan sizes got bigger. Thus we have an optimal scan size."

In the Fall 2004 semester, we had students report the number of hours they contributed to the development of their SPIMbot programs. While there was some variability, most students spent 10-20 hours each, working in teams of 2-3 students.

A final metric of effort that students expended on their contest entry is the number of lines of code. While lines of code is a metric of little practical utility, it outlines the of the work and the amount of effort the students put into it. The assignments that the students handed in ranged from 186 to over 608 lines of code and data segments (not counting blank lines and those containing only comments), with most in the 200-400 lines of code range. For comparison, there were about 130 lines of code provided in solutions that most students incorporated into their designs.

In light of the age-old challenge of teaching a student body with a diversity of aptitude (*i.e.*, "How can we teach so that all of the students learn the fundamentals, while still pushing the best students?"), perhaps the SPIMbot tournament's best use is providing the best students a challenge that pushes them.

[2]Interestingly, the first thing that many students look for is a way to write to the memory image of the other robot, which provides a nice segue to discussing virtual memory, also covered in the course.

5. FUTURE WORK

As it stands, SPIMbot is derived from SPIM which is only a functional simulator: each instruction takes a single cycle. Given that our course teaches pipelining and cache fundamentals, it would be desirable to enhance SPIM (as was done for CLSPIM [6]) to model pipeline and cache stalls. In this way, the course material would be unified in this final project and students would be exposed to a more realistic optimization scenario.

6. ACKNOWLEDGMENTS

This work was supported by NSF CAREER award 434 CCF 03-47260.

7. REFERENCES

[1] A. W. Chickering and Z. F. Gamson. Seven principles for good practive in undergraduate education. *American Association for Higher Education Bulletin*, 39:3–7, 1987.

[2] B. G. Davis. *Tools for Teaching*. Jossey-Bass, San Francisco, CA, 2001.

[3] E. M. F. III and L. Silvestri. Effects of Rewards, Competition and Outcome on Intrinsic Motivation. *Journal of Instructional Psychology*, 19:3–8, 1992.

[4] J. Larus. SPIM: A MIPS R2000/R3000 Simulator. http://www.cs.wisc.edu/~larus/spim.html.

[5] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

[6] A. Rogers and S. Rosenberg. Cycle level SPIM. Technical report, Department of Computer Science, Princeton University, Princeton, NJ, October 1993.

[7] P. Teller, M. Nieto, and S. Roach. Combining Learning Strategies and Tools in a First Course in Computer Architecture. In *Workshop on Computer Architecture Education, held in conjunction with the 30th Annual International Symposium on Computer Architecture*, June 2003.

[8] C. Zilles. SPIMbot. http://www-faculty.cs.uiuc.edu/~zilles/spimbot.