ICOS: Support for "Bare Metal" Computer Architecture Assignments

Zachary Kurmas
School of Computing
Grand Valley State University
kurmasz@gvsu.edu

Abstract

We present `ICOS`*, a platform for running code on a CPU's "bare metal" (i.e., without a traditional operating system). Our Computer Architecture students use* `ICOS` *to carefully time code in order to observe the performance effects of a CPU's branch predictor and superscalar design. Running code using* `ICOS` *avoids the noise introduced by context switches, interrupts and other traditional operating sytem features. This increase in consistency provides a clearer picture of the CPU's performance, which can be especially helpful for students who have not had much experience interpreting experimental results.*

1. Introduction

While reviewing our Computer Organization and Computer Architecture courses, we noticed that our students spend almost no time at all interacting at a low level with actual hardware. During the units on combinatorial and sequential circuits, we ask our students to build simple circuits using breadboards. After this initial introduction, however, we have students use tools like JLS [1] and Logisim [2] to simulate more complex circuits and CPUs; use the MARS MIPS simulator [3] to practice assembly language programming; and use Simplescalar [4] to investigate branch prediction and cache behavior. Worse yet, we cover superscalar and I/O at a conceptual level only.

Our retreat into simulation was not intentional; but, is understandable: In order to increase performance, CPUs have become considerably complex. As a result, it has become increasingly difficult to isolate the effects of one specific component (e.g., the branch predictor or cache) on a program's overall performance. Real processors — especially x86-based processors — are *much* more complex than the pedagogical CPUs presented in textbooks. Asking a student to work at a low level on an x86 processor is like introducing wind resistance into a physics problem.

As a result of this complexity, performance data gathered by a standard user process on a traditional operating system can be noisy: Interrupts, page faults, and context switches are just a few of the many system features that affect measurements. Minimizing and/or accounting for this noise can be difficult for both students (because it adds complexity to assignments), and system administrators (because many performance measurement techniques require root access).

`ICOS` is our first step toward addressing these challenges and re-introducing real hardware to our Computer Organization and Computer Architecture assignments.

`ICOS` is a platform for running code on a CPU's "bare metal" (i.e., without a traditional operating system). In some sense, `ICOS` is an ultra-simple operating system that allows students to insert code directly into its kernel. `ICOS` does not enable virtual memory, interrupts, or processes; therefore, the resulting measurements are very consistent. In addition, the lack of these features, as well as a lack of device drivers, makes the `ICOS` code very understandable: Students can reasonably follow what the CPU is doing from the moment it boots until it halts. This ability to follow the code from start to finish is where we get the name `ICOS`: "*I See*" OS (pun intended).

1.1 Motivation

Our original motivation for `ICOS`'s simplicity (i.e., lack of standard operating system features) was to maximize the consistency of performance measurements. For example, by not creating or enabling interrupts, we need not account for them (page faults, incoming network traffic, etc.) when interpreting results.

In hindsight, the most significant benefit of `ICOS`'s simplicity may be that students can thoroughly understand how the tool and their code interact with the CPU. We do not want students treating `ICOS` as a magic "black box"; we want them to read the code and understand what the CPU is doing from boot to halt. Furthermore, we want students to achieve this level of understanding in one or two study sessions — even students who have not yet taken an Operating Systems course.

Table 1. URLs for the labs presented in Section 3

Branch Prediction:	`http://www.cis.gvsu.edu/~kurmasz/NiftyAssignments/BP_ICOS`
Superscalar:	`http://www.cis.gvsu.edu/~kurmasz/NiftyAssignments/Superscalar_ICOS`

2. Related Work

In some sense, `ICOS` is an ultra-lightweight operating system. There are several operating systems designed for use in the classroom, including Minix [5], Xinu [6, 7], Nanvix [8], and Nachos [9]. Although they all have an educational focus, they are designed primarily for Operating Systems students and, therefore, have many more features than `ICOS` (interrupts, processes, virtual memory, etc.). In fact, Minix and Xinu have so many features that they now have non-pedagogical uses as well. Although these operating systems are simpler and more accessible to students than a traditional operating system like Linux or Windows, they are not "digestable" by students in one sitting. They are also designed to be functional operating systems. Thus, inserting code into the kernel (as is done with `ICOS`) would be awkward and error-prone.

Before building `ICOS`, we also considered using analysis tools, such as Intel VTune, for our Computer Architecture labs. We chose `ICOS` because the existing tools appear to be primarily focused on measuring overall application performance and/or finding performance bottlenecks as opposed to precisely measuring small segments of code. They are not intended to be simple, and are often proprietary. Therefore, they tend to be "black boxes" (i.e., it is not always possible or practical for students to examine precisely how these tools work).

3. Example Assignments

We present two `ICOS`-based labs. (See Table 1 for the URLs.) Students begin these assignments during a dedicated two-hour lab period, then complete the writeup on their own (which typically takes no more than 30 minutes). Lab sections have 20 students working in pairs. Our Computer Architecture laboratory contains 10 Intel Core i7-4790 processors running CentOS Linux.

3.1 Branch Prediction

This lab has students write code that (1) verifies the presence of a branch predictor on the CPU, then (2) investigates how long of a repeated branch pattern the predictor can correctly predict. We promote this as *Observing Branch Predictors "In the Wild"*.

During the first stage of the lab, students create three arrays: The first, "`always`", is filled with 0's; the second, "`never`", is filled with 1's; and the third, "`random`", is filled with a random sequence of 1's and 0's. The students then time the code shown in Figure 1 for each pattern.

```
uint32_t start = ic_rdtsc();

long sum = 433;
for (int i = 0; i < pattern_length; i++) {

  // We provide a non-trivial body for the
  // if statement so that the compiler's
  // optimizer does not remove the branch.
  if (pattern[i]) {
    sum *= 5910;
    sum += 17;
  } else {
    sum *= 4317;
    sum += 19;
  }
}
uint32_t end = ic_rdtsc();
```

Figure 1. The performance of this code is dominated by the behavior of the branch predictor.

The code being timed contains two branch statements: One in the `for` loop, and one in the `if` statement. The branch in the `for` statement is almost always taken and is highly predictable. The branch in the `if` statement is either always taken, never taken, or randomly taken, depending on the contents of `pattern`.

The measurements for `always` and `never` are consistently much lower than the measurements for `random`. Because the only difference between runs is the branching pattern, students can clearly see that the CPU handles a predictable branch pattern more quickly than a random pattern.

During the second part of lab, students attempt to determine how long of a pattern the branch predictor can recognize. For example, the branch predictors in many desktop CPUs can correctly predict the behavior of a branch following this repeating pattern of length 5.

11010 11010 11010 11010 11010 ...

In fact, these predictors can correctly predict much longer branch patterns. There is, of course, a limit: At some point, the pattern will be too long for the predictor to represent precisely using its available state, which will cause mispredictions. This lab has students generate and evaluate repeating patterns of increasing length and determine the length at which the code in Figure 1 becomes as slow as when the branch pattern is completely random.

Students typically obtain results like those shown in Fig-

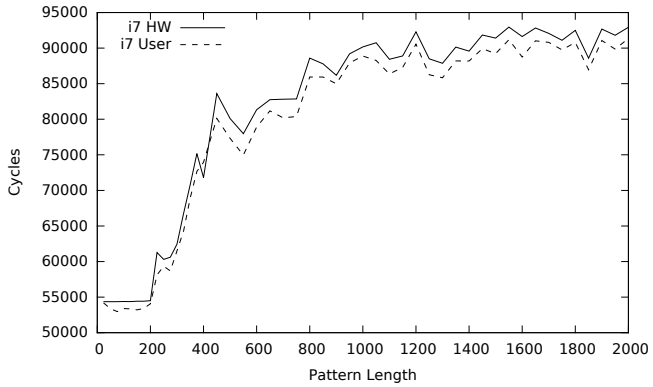

Figure 2. Effects of all branch pattern length on performance

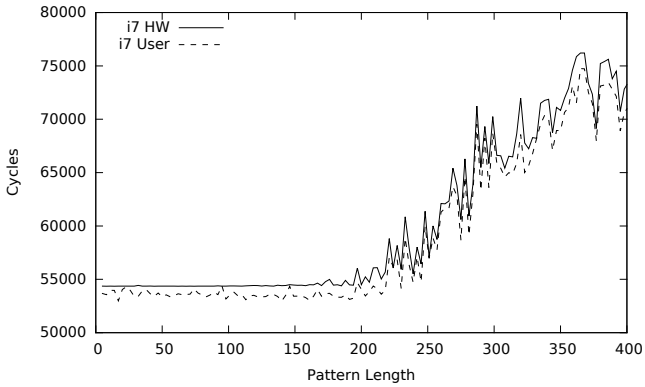

Figure 3. Effects of short branch pattern lengths on performance

```
time_n_instructions:
  push %ebx       # %ebx is a preserved register

  rdtsc           # read the starting cycle count
  push %eax       # push to avoid clobber later

  addl $1, %eax
  addl $1, %eax
  ...             # repeat until there are n adds
  addl $1, %eax
  addl $1, %eax

  rdtsc           # read the ending cycle count
  pop %ebx        # retrieve starting cycle count
  subl %ebx, %eax # compute elapsed num. cycles

  pop %ebx
  ret
```

Figure 4. Assembly function measuring the number of cycles taken to execute n instructions

ures 2 and 3. These graphs show that the branch predictor on the Intel i7 processors in our lab can completely handle patterns with lengths up to about 200. Longer patterns cause mispredictions that degrade performance.

3.2 Superscalar

This lab has students write code to verify the presence of multiple functional units within the CPU. Students also design an experiment to estimate the number of functional units.

A code segment's instructions per cycle (IPC) provides an approximate lower bound on a CPU's number of functional units. For example, a CPU that executes 100 instructions in 25 cycles must have at least four functional units. (It could have more functional units that went unused.) At a high level, this lab has students write code that maximizes the observed IPC.

We estimate a code segment's IPC by executing the instructions and using a pair of `rdtsc` instructions to count the number of elapsed cycles. For example, the code in Figure 4 estimates the number of cycles required to execute n consecutive `addl` instructions. However, it is not sufficient to simply choose and evaluate a single value of n because we must also account for the overhead of the beginning and ending `rdtsc` instructions as well as any `addl` instructions that the CPU schedules before, after, or in parallel with, the `rdtsc` instructions.

We expect the effects of overhead and overlap on the number of cycles measured to be independent of n: The overhead should be constant; and, any overlap should affect only the first few instructions. Therefore, the lab has students determine the IPC for values of n that increase from 100 to 5000. (We provide a Ruby script that automatically generates the different versions of `time_n_instructions`.) As n increases, the observed IPC should trend toward the optimal IPC (at least until the code grows large enough to cause instruction cache misses).

In Figure 4, each `addl` instruction depends on the previous instruction; therefore, no instructions can run in parallel, resulting in an expected IPC of 1. Students typically obtain results like those shown in Figure 5: The observed IPC varies for small values of n; but, once n reaches 500, the results quickly trend toward 1.06. (We don't know why the IPC for this code segment is greater than 1. We have previously obtained similar results on other Intel processors.)

To demonstrate the presence of additional functional units, we have students generate a pattern of instructions that use alternating registers:

```
add $1, %eax
add $1, %ecx
add $1, %eax
add $1, %ecx
...
```

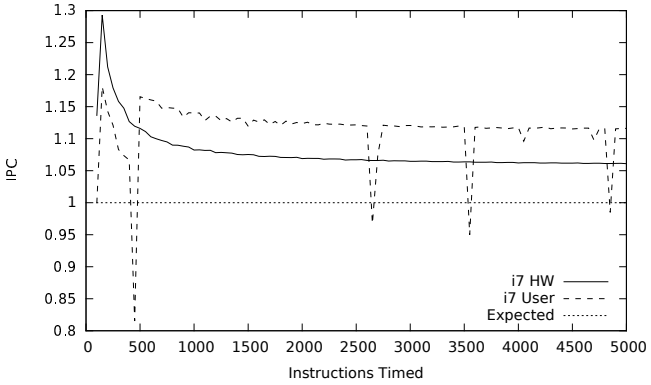

Figure 5. IPC for sequences of instructions with no parallelism (all lengths)

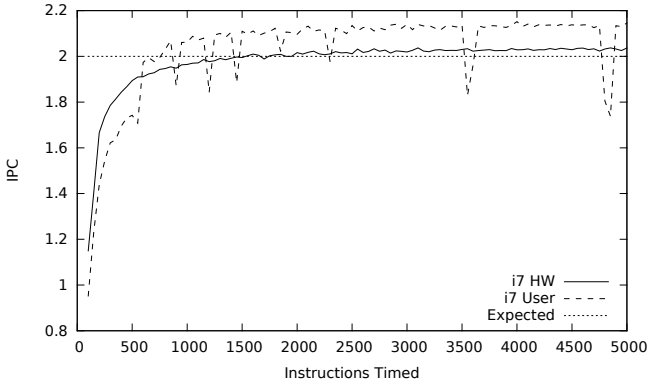

Figure 6. IPC for sequences of instructions where every pair can run in parallel (lengths up to 400)

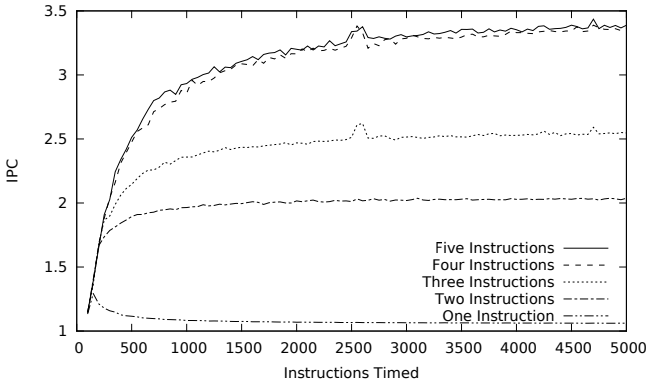

Figure 7. Finding the maximum IPC using bare hardware

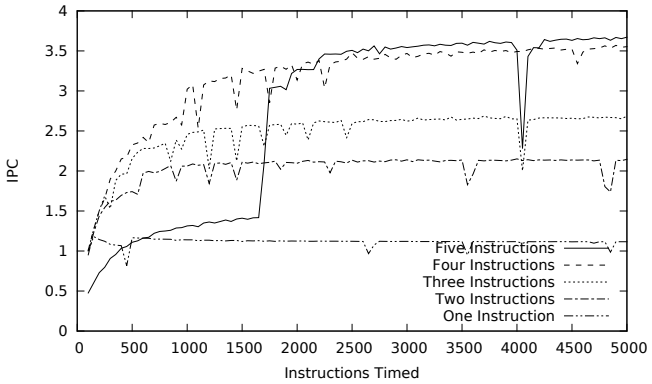

Figure 8. Finding the maximum IPC using a user process

Because no instruction depends upon a neighboring instruction, pairs of neighboring instructions can execute in parallel. As a result, the IPC tends toward 2, as shown in Figure 6.

Finally, students are asked to add additional registers to the instruction pattern and determine the maximum number of instructions that can be run in parallel. As additional registers are used, more independent instructions are available to be scheduled in parallel. This increase in parallelism increases the IPC until the code exhibits the maximum amount of parallelism allowed by the CPU.

Students typically obtain results like those shown in Figures 7 and 8: IPC does not increase when students increase the number of independant instructions from four to five. Although it would be nice if we could simply conclude that the CPU contains four functional units, further reading (including the Intel Developer Manuals) suggests that the observed limit is a result of the CPU's issue width rather than the number of functional units. Nevertheless, the exercise does clearly demonstrate that the CPU can execute multiple instructions per cycle and provides students with practice designing and critiquing experiments.

4. Experimental results

One motivation for running timing code directly on the "bare metal" is to avoid the noise induced by interrupts, context switches, and other background operating system activity. Early versions of the labs presented in Section 3 ran as user processes only and generated noisy results. Although the big picture was clear to the instructors, some students found the "bumps" in the graphs distracting. Students' frustrations with inconsistent results motivated us to develop `ICOS`.

To evaluate the benefit of running code on bare metal instead of as a user process, we ran the key steps of the sample labs in three environments:

- a user process on an Intel i7
- on a `vmware` virtual machine running on an Intel i7
- directly on the Intel i7 hardware

4.1 Branch Prediction

Table 2 shows the results of the first stage of the branch predictor lab from Section 3.1. This portion of the lab asks students to time code where the branches are always taken, never taken, or taken at random. Each measurement is repeated 25,000 times.

The differences between running the test code as a user process and using ICOS to run the code directly on the i7 hardware are minor, but noticeable. Both results clearly show that fewer cycles are required to execute code with a predictable branch pattern. In both cases, the average was very close to the minimum, suggesting that the vast majority of measurements are similar. However, running the experiment as a user process generated nosier data. The variance was 4 to 8 times larger. When we examined individual data points, we found that increased variance is generally caused by additional outliers. When run directly on the hardware, all observations were at most 110% of the average, while when run as a user process, up to 0.27% of the observations were greater than 110% of the average. Similarly, the maximum value observed on a user process was up to 400% of the average.

Although both environments lead to the same conclusion, our experience has shown that students are more comfortable with more consistent data. Some students will focus on the outliers and fail to see the big picture.

Interestingly, the cycle counts when running the ICOS image on the virtual machine were much higher than those of either the user process or the ICOS image run directly on the hardware. We find this especially surprising because `vmware` is not an emulator, and the code being timed does not contain system calls, device accesses, or any other operations that we would expect the virtual machine to intercept.

Figures 2 and 3 show the results of the second part of the Branch Predictor lab. This portion of the lab has students generate branch patterns of increasing length and examine the resulting time trend. These graphs show the difference between running the experiment in user space, or directly on the hardware using `ICOS`. Each data point in the graph is the average of 25,000 observations.

The results of running the experiment as a user process and directly on the hardware tell the same story: The branch predictor can completely handle patterns with lengths up to 200. At this point, increasing the pattern gradually degrades performance until performance is similar to that of a completely random pattern. Again, the variance among each set of 25,000 observations is reasonable; but, the trend is slightly more obvious when the experiment is run directly on the hardware. Notice, specifically, how the measurements for patterns with lengths up to 200 are very consistent when run directly on the hardware, but vary noticeably when run as a user process.

4.2 Superscalar

As with the Branch Prediction lab, running the Superscalar experiments on bare metal produced results that were similar to, but less noisy than running the experiments within a user process.

The first step of the Superscalar lab is to time sequences of dependent instructions. These sequences range in length from 100 to 5000 instructions. Each measurement is re-

Table 2. Measuring effects of branch predictor in different environments

	Always				Never				Random			
	avg.	max.	var.	pct. out.	avg.	max.	var.	pct. out.	avg.	max.	var.	pct. out.
i7 User	51,927	285,120	4.4×10^6	0.12%	52,033	279,957	5.7×10^6	0.27%	88,211	326,565	8.6×10^6	0.12%
i7 HW	54,776	58,236	1.4×10^6	0.0%	54,360	58,236	1.2×10^4	0.0%	95,365	100,269	1.1×10^6	0.0%
i7 VM	77,134	1,241,814	2.5×10^8	5.33%	126,345	223,618,379	4.8×10^{12}	6.07%	160,218	726,528	9.5×10^9	7.97%

Table 3. Measuring IPC in different environments

	Variance				
	min.	avg.	max.	max. vs. avg.	pct. out.
i7 HW	7	41.8	72	1.15	0%
i7 User	2.3	2.3×10^6	7.4×10^7	93	44%
i7 VM	8	1.1×10^9	4.4×10^8	364	92%

peated 1000 times. As shown in Table 3, when run on bare metal, the variance over each set of measurements ranged from 7 to 72 with an average of 41.8. The maximum measurement over the 1000 trials for a given n was never more than 1.15 times the average. In contrast, when run as a normal user process, the variance ranged from 3 to 7.4×10^7 with an average of 2.3×10^6. The maximum measurement was more than twice the average 44% of the time (i.e., for 44% of the values of n tested), reaching as high as 93 times the average. As with the Branch Predictor Lab, the virtual machine produced very inconsistent results.

In spite of the high variance, Figures 5 and 6 show that both techniques tell the same story: When all the instructions depend on each other, the IPC is close to 1, and when pairs of instructions run in parallel, the IPC is close to 2. Similarly, Figures 7 (bare metal) and 8 (user process) show that running experiments as a user process finds the same maximum IPC as running the experiments on bare metal; although, in this case, the data generated by the user process appears much more noisy.

5. Usage and Limitations

`ICOS` is freely available on GitHub: `https://github.com/kurmasz/ICOS/`

5.1 Usage

Limiting complexity was a primary concern when designing `ICOS`. To build an `ICOS` image to run their custom code, students need only:

1. Clone the `ICOS` git repository.
2. Place their code in the `usr_src` directory.
3. Place the entry function in a C file with the same name. (For example, place a function named `hello_world` in `usr_src/hello_world.c`.)
4. Run `make hello_world.img`

The file `hello_world.img` is a disk image that can be placed on a bootable device such as a USB drive. Running `make hello_world.debug` will generate an executable file that can be run as a user process. Both versions will run some setup code then execute the student's code beginning with the `hello_world` function.

5.2 Limitations

`ICOS` code may not use the standard C library. Instead, `ICOS` provides a much smaller library containing functions for:

- writing text to the screen,
- writing data to the output buffer
- generating random numbers, and
- gathering timing data (i.e., executing the `rdtsc` instruction).

There are no input functions; and, output is limited to (1) an 80x25 VGA text terminal (which wraps around when full), and (2) a data buffer that is dumped back to the boot image immediately before `ICOS` halts.

For simplicity, `ICOS` uses BIOS routines to perform all I/O. BIOS disk I/O routines are accessible only when an x86 CPU is in *real* mode. The boot sequence places the CPU in 16-bit real mode; but, like most operating systems, `ICOS` switches to 32-bit *protected* mode as soon as the OS is loaded into memory. Switching between real and protected mode is complex and error-prone; therefore, `ICOS` performs disk I/O during the startup and shutdown phases only. Writing to devices while in protected mode (i.e., while running the student's code) would require device drivers. Our aversion to the complexity of device drivers and switching between real and protected mode is why the data buffer is always written back to the boot device, and only written immediately before `ICOS` halts.

Similarly, because all disk I/O is performed while in real mode, the user code and data buffer are limited to a combined 1MB. (Memory addresses are limited to 20 bits while in real mode.)

6. Future work

We look forward to preparing more assignments that use `ICOS`. As we add more assignments, we expect that we will also add more features to the standard library. (`printf`-style output is currently at the top of our list.) We are also considering adding a 64-bit version of `ICOS`.

We personally find working close to the hardware more intellectually satisfying than working entirely in simulation. We look forward to gathering student feedback and learning how working closer to the hardware benefits the typical Computer Science student (e.g., a student who is anticipating a career in software development as opposed to CPU design).

We also see potential use for `ICOS` in Operating Systems courses. Although simpler than Linux and Windows, existing pedagogical operating systems like Minix, Xinu, and Nachos are still quite complicated. We don't intend for `ICOS` to replace these tools; but, we do believe that studying `ICOS`'s implementation could reduce the effort needed to understand the implementation of a more-full featured operating system. To this end, it may be helpful to add a few more simple, minimally functional features like interrupts and virtual memory so that students can see how these features are incorporated into a simple operating system before they attempt to understand a more complicated one.

7. Conclusions

`ICOS` is a simple, understandable means for students to run code on "bare metal". The low-level access does reduce noise when measuring performance; however, the amount of reduction for the example assignments presented was less than expected.

We are particularly excited that `ICOS` gives Computer Architecture and Computer Organization students the satisfaction of working directly with hardware (instead of simulators). We are optimistic that providing students a tool that allows them to follow the operation of a computer from the moment it boots until it halts will generate greater interest and enthusiasm for hardware-based Computer Science courses.

Acknowledgements

When I first began discussing my ideas for `ICOS`, several people warned me that an operating system is one of the most difficult pieces of software to write. They were right. I would not have been able to get `ICOS` working without a lot of help from many different people including Lawrence O'Boyle (our system administrator), Neil Dickson (creator of `PwnOS`), Greg Wolffe (professor at Grand Valley State University), Ed Gehringer (professor at NC State), Pedro Penna (creator of Nanvix), Dennis Brylow (Xinu), former students David Beerens and Kyle Niewiada, and countless members of the OS Dev and Stack Overflow communities.

References

[1] D. A. Poplawski, "A pedagogically targeted logic design and simulation tool," in *WCAE '07: Proceedings of the 2007 workshop on Computer architecture education*, pp. 1–7, ACM, 2007.

[2] C. Burch, "Logisim: A graphical system for logic circuit design and simulation," *J. Educ. Resour. Comput.*, vol. 2, pp. 5–16, Mar. 2002.

[3] K. Vollmar and P. Sanderson, "MARS: An education-oriented MIPS assembly language simulator," in *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pp. 239–243, ACM, 2006.

[4] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, pp. 13–25, June 1997.

[5] A. S. Tanenbaum, *Operating Systems: Design and Implementation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

[6] D. Brylow, "An experimental laboratory environment for teaching embedded operating systems," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, (New York, NY, USA), pp. 192–196, ACM, 2008.

[7] D. Comer, *Operating System Design: The XINU Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[8] P. H. Penna, "Nanvix an operating system for manycore platforms."

[9] W. Christopher, S. Procter, and T. Anderson, "The nachos instructional operating system," tech. rep., University of California, Berkeley, 1992.