

Experiences Integrating Research Tools and Projects into Computer Architecture Courses¹

William Tsun-yuk Hsu
Department of Computer Science
San Francisco State University
1600 Holloway Avenue
San Francisco, CA 94132

hsu@tlaloc.sfsu.edu

Abstract:

Hands-on program instrumentation and simulation projects are good tools to teach computer architecture to students who may have limited backgrounds in hardware design. Through working with toolkits such as Atom [SE94] and Shade [CK94], students are able to become comfortable with the concrete behavior of complex hardware and software structures, and be prepared for more advanced research projects. This paper summarizes my experiences developing hands-on program instrumentation and measurement projects for students in a sequence of computer architecture courses.

1. Introduction

In the Department of Computer Science at San Francisco State University, we have been running a fairly typical two-course architecture sequence. Prior to the first course, students have taken classes in digital logic design, assembly language programming and operating systems (this may change soon, but that is beyond the scope of this paper). In the first architecture course (which is primarily for upper division undergraduates), we use the Patterson and Hennessy Computer Organization text [PH98]; in the second course (primarily for beginning graduate students), we use the Hennessy and Patterson Computer Archi-

itecture text [HP96]. Both courses are supplemented with a number of hands-on programming and measurement projects, which I will describe in this paper. (The second course is also supplemented with a number of readings, which I will not be describing here.)

Since our academic program is more software-oriented, my emphasis in the architecture sequence is more to teach students the complex interactions between hardware and software and how they affect the design and performance of computer systems, rather than detailed hardware implementation. Also, our students generally have not had a detailed compiler design course; in the context of the course sequence, students make a number of standard compiler optimizations by hand, but we avoid the details of compiler construction.

When I started teaching the two-course sequence, I initially thought that many students were able to appreciate the abstract concepts; but when some of them started doing research in the area under my supervision, it seemed that they did not have a good grasp of the complex interactions in a real system. Hence, I felt that my main goal was to make the principles and insights covered in a typical computer architecture course sequence as concrete as possible to students who were not experienced in detailed hardware design. I decided that a good way to achieve this might be through hands-on projects. With proper tools, students can have the means of

1. Partially supported by an NSF-ILI laboratory equipment grant NSF DUE-9751724

observing and measuring how software choices and transformations influenced hardware events. The insights and experience cited in the texts become "real" when students are able to recreate the results. Quite often, unexpected results are observed, which lead to further investigations and better understanding of relatively complex behavior.

Initially, we worked briefly with pixie, then switched to DEC (now Compaq)'s ATOM binary instrumentation toolkit [SE94]. This obviously gave us much more useful information than running microbenchmarks and looking at execution times. The toolkit allowed us to focus our measurements on simple events of interest at some particular point in a course, such as instruction count, instruction frequency, data dependences, etc, without being distracted by certain kinds of interactions. While students are required to verify measurements with handcounts for simple code fragments, it is not practical to make handcounts for even smaller non-trivial programs. Using ATOM and similar toolkits cuts down on the boring and repetitive work involved in handtracing numerous lines of assembly code. Also, students are able to make measurements of SPEC and other widely-used benchmarks, albeit in scaled-down versions, and compare their results with experiments reported in their textbooks.

In Fall 1998, for practical reasons (our Alpha died), we switched to running Sun's Shade toolkit Version 5 on Sparcs [CK94]. Shade is a dynamic compiler with efficient and extensible trace generation capabilities. From the point of view of most of our projects, after adapting to the different API and the lack of symbolic information in Shade, there is little difference between working with either Shade or Atom.

2. Student Projects

The framework of our projects through a semester generally runs like this:

1. handcode code fragments (at source or assembly level) to demonstrate types of code transformations
2. compile code fragments before and after transformations
3. run instrumented code fragments to measure relevant events such as instruction counts, cache misses, etc; variations in microarchitecture may be simulated as well
4. compare with handcount estimates, analyze results, evaluation

For the lower division class, representative projects include:

1. comparing instruction count and instruction frequencies of code resulting from applying different compiler optimizations (loop unrolling, register allocation etc)
2. comparing instruction count and instruction frequencies of code generated by different compilers
3. measuring data dependences and pipeline stall behavior
4. evaluating basic cache configuration parameters (cache size, block size, set associativity, write policy etc)

For the upper division class, representative projects include:

1. measuring the behavior of branch prediction schemes, such as static schemes and McFarling's gshare predictor [Mc93]
2. development of a simple multicycle pipeline simulator with multiple functional units
3. exploring hardware configurations for multiple instruction issue, including

different types of instruction windows
functional unit limitations
branch prediction limitations
register renaming limitations

4. hardware and software prefetching schemes [Jo90]

5. victim caches [Jo90]
6. write miss handling (allocate vs. no-allocate)

In addition to coding the simulators and making measurements, students are also required to present their measurements clearly in tables or graphs, analyze their data and summarize the insights learned from the experiments.

2.1 A Detailed Example: Comparing Branch Predictors

The branch prediction project I designed for the upper division course in Spring 1999 is a good example of a simple, self-contained assignment that is manageable by students with only limited experience with the Shade toolkit. Prior to this project, students in the course had run Shade analyzers that were provided, but had not done any programming using the toolkit.

The full text of the problem statement is available at [EXCL]. Students are required to compare the performance of three branch prediction schemes:

1. static: forwards not-taken and backwards taken
2. dynamic predictor with 2-bit counter array, indexed with PC of branch [HP96], number of entries varied from 256 to 4096
3. McFarling's gshare predictor [Mc93], number of counter entries varied from 256 to 4096, number of PC bits used varied from 8 to 10

Implementation details of the three schemes are made very explicit; otherwise perfectly reasonable variations in implementation can make verification a nightmare. Students are required to make measurements of the branch characteristics of two contrasting benchmark programs: the SPEC95 Lisp interpreter *li* with a scaled-down data set, and *smooth*, an image-smoothing program that averages each element of an integer matrix with its neighbors. In general, I try to locate two or three benchmarks that behave very differently with respect to the archi-

tectural features under consideration. Smooth is mostly very predictable loop-based code. The static predictor is excellent for smooth and horrible for *li*. The 2-bit predictor is better than the static predictor for both benchmarks. Gshare actually performs worse for smooth, but much better for *li*.

Students are given a Shade analyzer that extracts from each executed conditional branch the branch address, branch offset, target address, and whether the branch is taken/not-taken. This Shade API call allocates space in the trace record for the information to be captured for each branch:

```
shade_trctl_it (IT_BICC | IT_FBFCC, 1, 0, TC_I | TC_IH
| TC_PC | TC_TAKEN | TC_EA);
```

The main Shade analyzer loop traces each executed instruction and displays the relevant information:

```
for (; tr = shade_step(); )
    if (ih_isbicc(tr->tr_ah) || ih_isbfcc(tr->tr_ah)) {
        printf("cond br at %x disp22 = %x target = %x
taken = %d\n", tr->tr_pc,
            tr->tr_i.i_disp22, tr->tr_ea, tr->tr_taken);
    }
```

Students would replace the printf statement with code that simulate the various branch predictors. At this relatively inexperienced stage, they concentrate on implementing the prediction schemes and interact only in a minimal way with the Shade API. I generally provide less "infrastructure" in later projects.

3. Findings

We have found that students are able to learn quickly the ATOM and Shade APIs, provided that adequate examples are given. Also, templates can be provided so that students can ignore the more complex details of the toolkits for the time being, and focus on the essential parts of monitoring the relevant hardware events or developing the subsystem simulator.

For example, the Shade toolkit's API for extracting instruction text information can be rather intimidat-

ing for a first-time user who has limited exposure to the Sparc instruction set. I felt that it was necessary for me to write a parser for the instructions so students can concentrate on dependence checking and other aspects of a project on measuring instruction level parallelism. Given a Sparc instruction parser, students were able to implement multiple-issue simulators in a relatively short period of time (usually three to four weeks, as part of a larger assignment).

4. Lessons learned

One of the first observations we made was that verification is very time-consuming. It is of course necessary to provide students with test-suites for the more detailed programming projects. Aggregate measurements often hide inaccurate implementation details.

Compiler optimizations must be controlled carefully. It is often not clearly documented what transformations a compiler tries to make at a specific optimization level. Language details or other subtle factors may prevent a compiler from making a transformation that seems “obvious” to a programmer (for example, loop unrolling is generally not done when loop bounds are unknown at compile time, even if loop bounds might be determined to be a constant through simple interprocedural analysis). The default compiler optimizations are often surprisingly inadequate, even with a commercial compiler shipped with the manufacturer's own hardware. For example, some compilers will default to performing integer division in software even if hardware integer division is available, if the correct flag is not explicitly specified. Or simple pipeline scheduling may not be performed until the highest level of optimization is specified.

When I got down to defining an actual RTL-level implementation of even a small subsystem such as a branch predictor, I often found that textbooks and research papers do not clarify all the implementation details necessary for a simulation. An experienced research student can obviously work out a lot of the details independently; in a coursework situation,

with students who are exposed to this type of work for the first time, it is often necessary to write a very explicit specification that spells out all the details clearly.

It is important to control the scope of each project carefully. We generally isolate and focus on a relatively small part of the system, for example the pipeline or the cache. Otherwise students easily become overwhelmed with trying to make sense of a potentially huge set of measurements and interactions. Since the kind of detailed simulations and measurements for some of the projects result in huge slowdowns of benchmark runs, it is crucial to reduce runtimes by scaling down benchmark input data sets or by other means; students are often frustrated when it takes hours to run a benchmark for a project that they have only two to three weeks to complete.

Limiting the scope of a project carefully also helps to reduce unpredicted effects which might be confusing to students (and do not help to illustrate the points one wants to make in the project!) For example, system calls may cause the Sparc register windows to overflow, and are easily avoided with address range limiters in Shade. A seemingly regular and predictable code fragment like a matrix-multiply inner loop may not see much benefit from data stream buffers, if there are not enough stream buffers.

5. Conclusions

We feel that hands-on projects involving programming, simulation and measurement are very helpful in teaching computer architecture concepts to students who may not have advanced hardware design experience. Our students have been able to grasp quickly the essentials of research toolkits such as ATOM and Shade, perform simple simulations and experiments, and measure and analyze the results. They are thus well-prepared for further research work in the area.

This hands-on approach we have taken in the architecture courses is typical of both the systems area

and software development courses at the Department of Computer Science at San Francisco State University. More information about our NSF-funded Experimental Computer Science Laboratory can be found at [EXCL]. Shade is available as a free download from Sun Microsystems [Shade]. The ATOM toolkit is distributed free with Digital Unix.

6. References

[CK94] B. Cmelik and D. Keppel, Shade: a fast instruction-set simulator for execution profiling. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1994.

[EXCL] <http://userwww.sfsu.edu/~whsu/EXCL/>

[PH98] D. Patterson and J. Hennessy, Computer Organization and Design, Morgan-Kaufmann, 1998.

[HP96] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, Morgan-Kaufmann, 1996.

[Jo90] N. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, 21st Annual International Symposium on Computer Architecture, May 1990.

[Mc93] S. McFarling, Combining branch predictors, WRL Technical Note TN-36, June 1993.

[SE94] A. Srivastava and A. Eustace, ATOM: a system for building customized program analysis tools. 1994 Programming Language Design and Implementation, pp. 196-205, ACM, June 1994.

[Shade] <http://www.sun.com/microelectronics/shade>