Combining Object-Oriented Design and Computer Architecture into a Single Senior-Level Course

David R. Kaeli
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA

Abstract

Undergraduate computer architecture course work typically covers some standard topics, including pipelining, memory hierarchy design and their relationship to performance. While there are a number of available tools that can be used to exercise the functionality of these concepts, students will many times only learn how to use the tool and will not gain an appreciation for the underlying technology or design.

To insure that students understand many of the design tradeoffs encountered with computer design and computer architecture, they can construct software models of these designs. The students must have a very clear understanding of the design for them to be able to model it correctly. By developing a model, they gain insight into the real tradeoffs associated with the design of such features.

To insure that students are able to develop well-structured, reusable simulation models, they must also have sufficient background in formal programming methods. Object-oriented design provides this structure and C++ provides a reasonable implementation language for developing object-oriented simulation models.

This paper describes a senior-level course offered in the ECE department at Northeastern University that encompasses both of these subjects. As a final project for this course the students must construct a simulation model of some architectural feature. This paper describes the motivation for this course, and the associated course work.

1 Introduction

Too many graduating computer scientists and engineers complete their undergraduate course work without gaining a full appreciation for computer architecture principles. They are able to describe the concepts associated with architecture, but they are not able to produce a functional implementation of the concepts. While the concepts are extremely important, without experience of how to implement architectural features (e.g., a caches, pipelines), the concepts become lost in the theory. The students need some experience with implementation.

Actual physical design and fabrication of a design, while may be becoming more feasible, usually is too costly and time consuming to be offered in an undergraduate curriculum. A reasonable alternative is to develop software simulation models of the architectural concepts can be constructed. Only if the students have a full understanding of the architecture feature that they are modeling will they be able to code up their model.

While most undergraduates receive some amount of programming education (typically in C), many students are not "good programmers." They have no appreciation for proper coding conventions, algorithms, or abstract data types. How can we expect they to develop well-designed simulation models if they have not had the necessary programming education.

This paper describes a course that addresses both of these needs in a single course. In section 2 the motivation for offering this course is provided. In section 3 the object-oriented design content of the course is covered. In section 4 the architecture portion of the course is described. Finally in section 5 the course project is discussed and feedback from students is also presented.

2 Motivation For This Course

Northeastern University offers a Computer Option in the Electrical and Computer Engineering department. The option allows students to obtain a concentration of courses in computer-related subjects. The curriculum has recently been under review and some areas that needed to be addressed included programming design and advanced computer architecture. Previously, undergraduates received only one quarter of instruction in C programming and one quarter of instruction in the fundamentals in computer architecture. This led to the formation of a new course.

There are two main themes of this course:

1. Advanced Computer Architecture
2. Object-Oriented Design and Programming

The current computer architecture course covers the topics presented in the undergraduate Hennessy and Patterson text [1]. This gives a good overview of many of the basic architecture features present in today's computer systems and serves as a good foundation. The advanced architecture portion of the new course focuses on many of the new advances in the research literature, as well as some of the new microprocessors coming on the market.

The object-oriented design portion of this new course begins by presenting OOD principles. The goal is for students to learn design before they start programming. Then C++ is presented as a simple extension of the C language. Next, the course content will be described.

3 Advanced Computer Architecture

Since a majority of the students taking this course already have a good understanding of the basic computer structures present in today's architectures, we can begin by presenting some of

the more recent advances in the field. Much of this material is taken from the proceedings of ISCA, ASPLOS, MICRO and HPCA, (I also discuss much of my own research).

The students are taught to question the results presented in the articles. To encourage them to become more critical of what they read, the students are asked to review 10-15 articles handed out in class and provide a one-page critical review. The review must use one paragraph to summarize the article and then spend the rest of the page to comment of the validity of the work, as well as on the presentation of the material. This teaches the students to become better reviewers, which should in turn, make them more critical of their own work.

Some of the topics covered in this portion of the course include:

- Hazard Detection
- Branch Prediction
- Novel Cache Organizations
- Hardware and Software Cache Prefetching
- Compiler Techniques to Enable Pipelining
- SuperScalar Design
- VLIW
- Memory Coherency and Memory Consistency
- Bus Architecture
- RAID Technology

The required text for this portion of the course is the beta copy version of the 2nd edition of Hennessy and Patterson's "A Quantitative Approach" [2]. This is an excellent text for introducing many of the above topics. Homework assignments are selected from the problem sets found at the end of each chapter. Many of the problems are thought-provoking exercises, which are very appropriate for a course at this level. Next the software portion of this course is discussed.

4 Object-Oriented Design and Programming

The feeling of this educator is that too many computer engineering and computer science students graduate without a solid foundation in programming. This skill-set should include design, analysis and style. This phase of the course attempts to give the students a crash course in design and analysis (it is generally too late to correct their style).

First, the students are presented with the concepts associated with an Abstract Data Type (ADT). Many of the concepts presented here can be found in the data structures and algorithms text by Aho, Hopcraft and Ullman [3]. Another good text used in this part of the course is the Baase text [4]. Some ADT's that are discussed include:

- lists
- queues
- arrays
- trees
- directed graphs
- stacks
- heaps

Along with these ADT's, discussion of various algorithms is presented. Complexity analysis of these algorithms, when operating on various ADT's, is discussed. From this portion of the class the students come away with a richer appreciation for data analysis and computational complexity.

Following an introduction to ADT's, a discussion of objects begins. An object is just an ADT with a lot more personality (i.e., behavior). The students get a good dose of OOD that is based on the Wirfs-Brock approach of responsibility-driven design [5].

As soon as the students have a grasp of the OO concepts, a C++ tutorial is handed out and homework assignments are provided that parallel the tutorial. Each of the OO concepts is presented and illustrated with a programming example, including:

- encapsulation
- inheritance
- polymorphism
- reuse

The students come away from this part of the course being able to write truly object-oriented C++ code that exhibits some intelligent choices for data structures and algorithms.

5 Course Project

It is always a useful exercise for students to practice much of the knowledge they have acquired during a course and produce a finished project. This exercise should also provide the implementation level design details described above. To accomplish this, the students are assigned to develop a trace-driven model of some architectural feature. They must develop their models using object-oriented C++.

The students' project grades are heavily weighted on the quality of their OO designs (40%). They are asked to develop their code using the DEC Alpha tool ATOM [7]. ATOM allows users to perform execution-driven simulation. Traces are generated dynamically and are directly

passed (using a procedure call) to the analysis routine. The analysis routines contain the C++ models. In addition, using a user-defined instrumentation file, the user has the ability to only trace selected events (e.g., branches). This provides for an efficient simulation platform. The tool also provides sample models so that students can quickly get up to speed.

The students are asked to find a paper in the literature that describes some architectural feature. They then attempt to reproduce (or invalidate) the results. Available to them are the SPEC92 benchmark suite, ready to be instrumented with their analysis files.

Appendix 1 provides a listing of a portion of the C++ analysis code for one project that modeled the *Column-Associative Cache* described by Agarwal and Pudar [6]. Notice the reusability of the base class Cache in the direct mapped and two-way set associative class declarations. The students are able to reuse the base class Cache without respecifying the behavior of the Cache class. They use polymorphism to override the member functions of the base class.

The only difficulty encountered with this project was the use of the g++ compiler to compile the analysis files. The DEC libraries needed to be specially linked to enable ATOM to successfully link the analysis routines. If the DEC C++ compiler was available for the course, this problem would not have occurred.

In summary the students found this course to contain many of the topics that they were looking for, while combining two very different course topics into a single class project. A majority of the comments on the course were very positive. The students suggested even a stronger emphasis on programming.

6 Summary

This paper described a new course offered in the undergraduate ECE program at Northeastern University. The goal of the course was to effectively combine object-oriented programming and advanced computer architecture in one course.

The course provides background in both software design and computer architecture and culminated in a course project that extensively used both of these subjects. This course will continue to be offered once a year during the spring quarter.

References

[1] D. A. Patterson, “Computer Organization and Design: The Hardware/Software Interface,” Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[2] J. L. Hennessy and D. A. Patterson, “Computer Architecture: A Quantitative Approach”, 2nd edition beta copy, Morgan Kaufmann Pub., San Mateo, CA, 1995.

[3] A.V. Aho, J.E. Hopcraft and J.D. Ullman, “Data Structures and Algorithms,” Addison-Wesley, Reading, MA.

[4] S. Baase, “Computer Algorithms: Introduction to Design and Analysis,” Addison-Wesley, Reading, MA.

[5] R. Wirfs-Brock, B. Wilkerson and L. Wiener, “Designing Object-Oriented Software,” Prentice-Hall, Englewood Cliffs, N.J..

[6] A. Agarwal and S.D. Pudar, “Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches,” Proc. of the 20th International Symposium on Computer Architecture, May 1993, pp. 179-190.

[7] “ATOM User Manual”, *Digital Equipment Corporation*, Maynard, Massachusetts, March, 1994

7 Appendix A

```
//
// The following defines must be set before the #include file
// #define BLOCKSIZE X X in bytes
// #define CACHESIZE Y Y in bytes
//      #define MEMACCESS Z     Z is # bits in address

#include <stdio.h>
#include "instrument.h"
#include "derived.H"
#include "stats.H"

//++++++++++++++++++++++++++++++++++++
// Cache() is the base class for all
//   types of caches.  Note that the
//   LookupEntry() functions are void
//   requiring definition in the child
//   classes.
//++++++++++++++++++++++++++++++++++++
class Cache {
protected:
  long BlockSize;                       // # bits in block mask
  long IndexSize;                       // # bits in index mask
  long TagSize;                         // # bits in tag mask
 unsigned long BlockMask, IndexMask, TagMask;  // bit masks for ease of use
 long GetBits(long num);          // Finds the log2() of a 2^n number
public:
  Cache();
```

```
  virtual void LookupEntry(long Address) = 0;
  virtual void LookupEntry(long Address, ITypeType IClass) = 0;
};

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
long Cache::GetBits(long num)
// class DMCache
// This class is an implementation of a direct mapped cache.
// It inherits some of its functionality from the Cache
// base class.
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

class DMCache: public Cache
{
private:
  DM_Stats DMStats;              // An object to compute statistics
  Line cache[(CACHESIZE/BLOCKSIZE)]; // This is the number of entries
public:
  DMCache();
  virtual void LookupEntry(long Address);
  virtual void LookupEntry(long Address, ITypeType IClass);
};


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// class TWSACache
// This class is an implementation of a Two-Way Set Associative
// cache.  It inherits some of its functionality from the Cache
// base class.
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

class TWSACache: public Cache
{
protected:
  TWSA_Stats TWStats;
  TwoWSA_Index cache[(CACHESIZE/BLOCKSIZE)/2L]; // This is the number of entrie
public:
  TWSACache();
  virtual void LookupEntry(long Address);
  virtual void LookupEntry(long Address, ITypeType IClass);
};
```