Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

THE DESIGN OF AN EFFICIENT SIMULATOR FOR THE PENTIUM PRO PROCESSOR

David A. Sykes

ObjectKnowlogy, Inc.
11 Oakview Drive
Greenville, SC 29605–1816, U.S.A.

Brain A. Malloy

Department of Computer Science
Clemson University
Clemson, SC 29634–1906, U.S.A.

ABSTRACT

The increasing size and complexity of computer systems has created the need to develop new techniques to facilitate the design and evaluation of developing architectures. The most common technique applied to the development of new architectures is simulation, which permits detailed accurate modeling of the architecture. However, there are several problems associated with using simulation in developing computer systems. First, traditional simulation approaches are not flexible enough to permit easy extension or modification of the model. While most simulators are parameterized to provide flexibility, parameterization alone is not adequate to permit the modification and extension required for developing architectures. Second, traditional simulation approaches are computation intensive, slowing development and prohibiting simulation of large programs. To address the first problem, we use an object-oriented design for our simulator to produce a flexible, extensible model. To address the second problem, we incorporate the *decode* step of processor simulation into the state of each instruction object, saving the time to simulate the decode during execution. We have implemented a prototype of our design and initial experiments with the object-oriented prototype, coded in C++, ran twice as fast as another prototype that we implemented using the traditional approach to simulation where the simulator was written in C.

1 INTRODUCTION

The increasing size and complexity of computer systems has created the need to develop new techniques to facilitate the design and evaluation of developing architectures. The most common technique applied to the development of new architectures is simulation, which permits detailed accurate modeling of the architecture (Malloy 1993; Philip 1992). The most common simulation approaches for computer systems are *trace-driven* and *execution-driven* simulation. Both of these approaches have been used successfully in developing and testing new architectures.

However, there are several problems associated with using simulation in developing computer systems. First, traditional simulation approaches are not flexible enough to permit easy extension or modification of the model. While most simulators are parameterized to provide flexibility (Malloy 1993; Philip 1992), parameterization alone is not adequate to permit the modification and extension required for developing architectures. Second, traditional simulation approaches are typically computation intensive, which slows development and prohibits simulation of large programs. Third, to provide accurate results and to avoid design errors, simulators must be validated; validation of simulators, especially simulators for developing architectures, is difficult and time consuming.

In this paper, we address the first two problems associated with traditional approaches to simulation of developing architectures. To address the first problem, we use an object-oriented design for our simulator to produce a flexible, extensible model. To address the second problem, we incorporate the *decode* step of processor simulation into the state of each instruction object, saving the time to simulate the decode during execution. We have implemented a prototype of our design and initial experiments with the object-oriented prototype, coded in C++, ran twice as fast as another prototype that we implemented using the traditional approach to simulation where the simulator was written in C. However, almost twice as much memory was required to execute the object-oriented simulator over the traditional simulator. Our current focus is only on execution-driven simulation. Further work will address the reusability of our design and the components it produced in the construction of trace-driven simulators.

The remainder of this paper is organized as follows. In the next section, we provide background about trace-driven and execution-driven simulation

together with information about the target architecture of our simulator: the Pentium Pro Processor. In section 3, we present the object model for our simulator together with discussion about the advantages of our design. In section 4 we describe our prototype and in section 5 we discuss our experiments and give concluding remarks.

2 BACKGROUND

In this section, we provide background about simulators and processors. We begin by discussing two techniques applied to processor simulation: *trace-driven* and *execution-driven* simulation. We then overview recent trends in processor design with special emphasis on the Pentium Pro processor by Intel, selected because of its complexity and the challenge it introduces to simulator development.

2.1 Processor Simulation

The classic approach to simulating existing or developing architecture is *trace-driven simulation* where the simulation is based on a predetermined instruction sequence or *trace*. Trace-driven simulation requires information about branch targets and memory references made during the execution of the program. This technique is used extensively to evaluate uniprocessor cache performance and techniques have been developed for capturing trace information to include references from system calls and the operating system. However, the problem with applying the trace-driven approach to simulating developing architectures is that the simulated architecture must be similar to the architecture on which the trace was obtained; this is a significant problem when simulating an architecture that is in the development stage (Koldinger, Eggers, and Levy 1991). For a new architecture, the order of events, the latency of communications, and/or the number of processors might be completely different from the trace host, rendering the traces useless when evaluating timing-sensitive applications. Moreover, using traces obtained from a sequential system may be completely inappropriate for use in simulating a parallel system since the order of execution is usually dependent on the order in which different parts of a parallel program complete execution (Mukherjee and Bennett 1990). Previous research has shown that traces obtained on a multiprocessor can produce erroneous results when used to simulate a different multiprocessor (Dahlgren 1991). Even when the trace-driven approach is applied to uniprocessor simulation, traces are difficult to obtain and trace files are frequently prohibitively large (Ball and Larus 1992).

Because of the drawbacks of trace-driven simulation, recent trends reveal a proliferation of the use of *execution-driven simulation*, also referred to as instruction level simulation, register-transfer simulation or cycle-by-cycle simulation. In the execution-driven approach, the simulator actually executes an input program. Thus, the main actions of the simulator are expressed with a large **case** statement enclosed in a **while loop** that runs for the length of the simulation; on each iteration of the loop, an instruction is executed and each case option is an `op code` for the processor to decode and execute. Execution-driven simulation permits development of the software together with the hardware because the simulator isn't bound by the traces.

Although recent research has produced reliable execution driven simulators (Malloy 1993; Philip 1992), the classic, time-honored approach to processor simulation is trace-driven simulation.

2.2 Trends in Processor Design, The Pentium Pro

In the earliest stored-program computers, hardware was expensive so that early processor designers produced a machine with a single register for arithmetic instructions. Since all operations would accumulate in a single register, it was called an *accumulator*; the EDSAC computer was a single accumulator machine. This early design philosophy was replaced by more elaborate schemes, since hardware became progressively less expensive. Recent processor designs combine a lush supply of registers with multiple instructions issued on each execution cycle and elaborate memory hierarchies.

An extremely efficient microarchitecture, the Pentium Pro Processor by Intel, has recently been presented that combines innovations resulting in processor speeds that significantly exceed the 100 MHz Pentium Processor, though it is manufactured with the same semiconductor process that produced the Pentium Processor. The Pentium processor is superscalar, using five stages to extract high throughput. However, the Pentium Pro processor is capable of speeds in excess of 200 MHz. To achieve this, several design innovations were incorporated into the new processor including *dynamic execution*, *associative memory*, *branch prediction*, *branch recovery*, a *branch target buffer* and an *instruction pool* that permits three independent processor units to communicate. The instruction pool is implemented as *content addressable memory* called the *reorder buffer* (ROB).

Dynamic Execution technology can be summarized as optimally adjusting instruction execution by predicting program flow, using data flow analysis to

choose the best instruction to execute and then executing the instructions speculatively, in a preferred order. To do this, the processor uses an in-order *fetch/decode* unit, an out-of-order *dispatch/execute* unit, an in-order *retire unit* and a *bus interface* unit that communicates with the off chip cache supporting as many as four concurrent cache accesses.

Figure 1 illustrates the important design features of the Pentium Pro Processor. At the top of Figure 1, is the system bus and the off chip cache or *L2 Cache*. The *Bus Interface* connects the processor with the system bus and the off chip L2 cache. The processor has an on-chip cache with an 8K byte instruction cache, shown in Figure 1 as *L1 I-Cache*, and an 8K data cache, shown in Figure 1 as *L1 D-Cache*. The *Fetch/Decode* unit fetches instructions from the L1 I-Cache while the *Dispatch/Execute* unit and the *Retire* unit fetches (stores) instructions from (to) the L1 D-Cache. All three units access the *Instruction Pool*. Figure 1 is an extremely simplified description of the Pentium Pro Processor, which requires detailed simulation to capture subtle but important nuances to facilitate processor development. A simulator for the Pentium Pro Processor must permit easy extension and modification together with efficient execution.

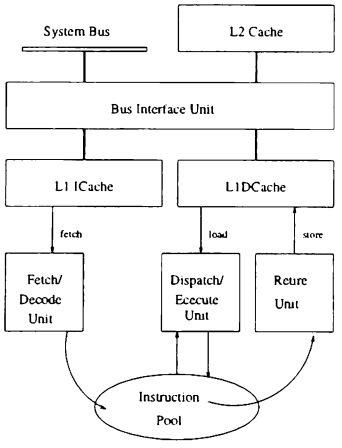


Figure 1: The Pentium Pro Processor Design

3 DESIGN OF THE SIMULATOR

In our brief description of the Pentium Pro Processor, provided in the previous section, we have tried to capture the flexibility and power of the processor. However, development and testing of such a processor requires the use of a simulator that is also powerful and flexible. The processor simulator that we present in this paper was motivated by the following goals:

- *efficiency*, particularly with respect to execution time since traditional simulators are CPU-intensive;
- *extensibility*, to permit addition of new kinds of instructions or simulation of new hardware components to study their impact without requiring large changes to existing hardware or software;
- *flexibility*, to permit easy adaptation to new algorithms; for example, adaptation of a new caching strategy or pipeline schedule to determine their effects; and
- *reusability*, such that we can build simulators for other computer architectures using existing software components.

We have chosen to use an incremental development approach. In starting our work, we have focused on simulators for simple architectures and are building on that work to address computers that have increasing amounts of sophistication. Our plans for future work include the development of an object-oriented framework for simulators. We are first focusing on building a set of components that can be reused to simulate a wide range of architectures—for example, an architecture containing multiple levels of cache memory, register arrays, and instruction pipelines including those that support out-of-order processing and in-order retirement of instructions. Our ultimate goal is a simulator for the Pentium Pro. The design that we present in this paper comprises the fundamental architecture. From that and its refinement over a series of increments, we will identify a more general design that will be captured in a framework from which we can design many simulators. We have already determined that framework will include both abstract classes and templates.

To address the goals listed above, we begin with the object-oriented programming paradigm to exploit its strengths for supporting extensible, flexible designs and reusable components. For reasons of efficiency, we adopted C++ as our implementation language, although our design is virtually language-independent.

3.1 Analysis of Computer Architectures

We began our development effort with an analysis of the domain of computer architecture. The object model for our initial design is illustrated using "unified method notation" in Figure 2. Although this model is in the development stage, we consider it sufficient for investigating a design for simulating a variety of computer architectures. In the model, we attempt to capture major object classes within the domain so that when we consider a specific computer architecture—for example a Motorola 68040 or the

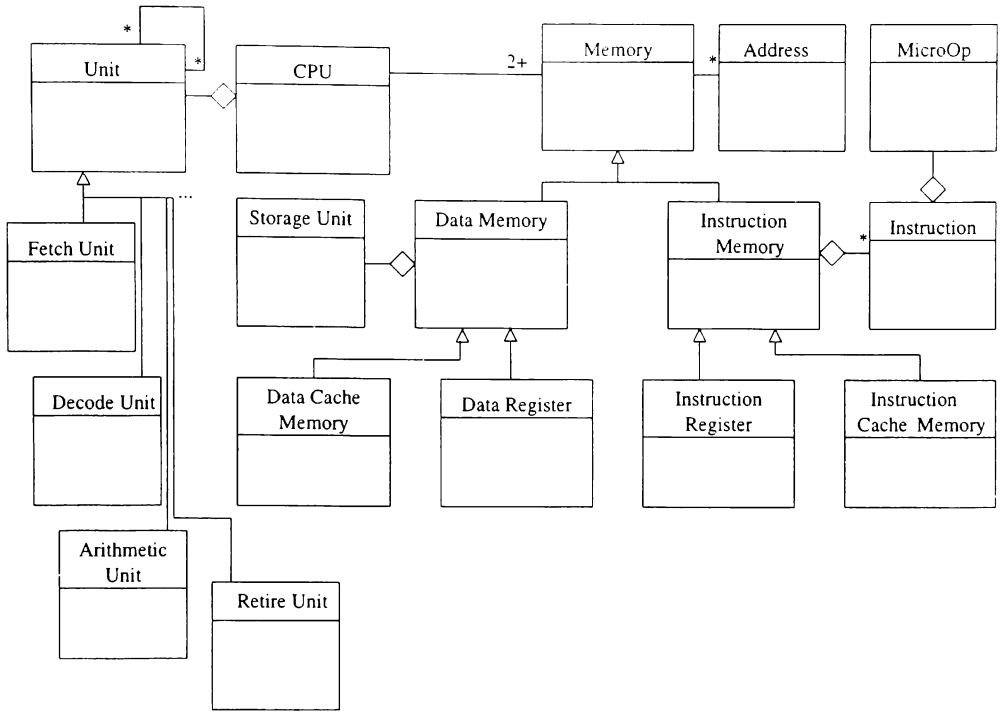


Figure 2: Analysis Object Model for the Computer Architecture Domain

Pentium Pro Processor—we will refine the classes in this model and likely introduce new relationships.

The object model is based on an assumption that data and instruction memories are separate, although there is no assumption as to whether this separation is logical or physical. Implicit in this assumption is that instructions can be executed only from instruction memory.

3.2 Design Overview

Our simulator architecture is based fundamentally on the separation of instruction memory and data memory. Instruction memory contains only instructions—never data—that we treat as objects. That is, instead of representing instruction memory as an array of bytes (or words) that are fetched and assembled into instructions that are then decoded, we represent instruction memory as an associative array of instructions (associated with addresses), each of which knows how to "execute" itself given a current processor environment.

There are at least three implications that arise from a design based on such assumptions:

1. The "decode" step of simulation is implicitly represented in the state of each instruction object, saving the steps in the simulator that interprets the bit pattern of a sequence of bytes retrieved from the memory array. A set of experiments we ran on a design prototype shows that the execution of a simulator using this technique is approximately twice as fast as another prototype we built in C using a traditional design, although about twice as much memory is required to run the simulator. We note that the prototype uses a very simple implementation of instruction memory: an array of pointers to instruction objects that yields fast access to instructions, but at a cost of more space. (the prototype is discussed further in Section 4.)

2. Writing into instruction store introduces inefficiency. In most circumstances, runtime modification of code is not performed by application programs so this is not a problem. Some processor architectures separate code and address spaces physically, making runtime modification of instructions impossible. However, some processors do allow writes to instruction memory and some algorithms—most notably, those that perform "bit block transfer" (BITBLT) in animated computer graphics applications—optimize their efficiency by, in essence, assembling a customized segment of code that executes its function faster than a data-driven approach would yield. Since, in our design, we essentially *predecode* instructions in instruction memory, then any write into a byte of that memory must redecode the instruction stored there. This is pos-

sible in our design. We can treat the associative memory as a cache, such that any write to a byte at an address invalidates the instruction associated with that address. If it can be assumed that a sequence of writes to instruction memory will result in replacement of one or more whole instructions before they are executed, then construction (by decoding bytes) of the new instruction objects can be accomplished straightforwardly. Algorithms used by interpreters in compiling language constructs into a cache to achieve faster execution—for example,the caching of methods by a Smalltalk interpreter—can be adapted for use here in a way that is transparent to all objects outside those representing instruction memories. If that assumption is not valid, then a byte image of instruction memory must be maintained by the simulator and instruction objects in memory be reconstructed from that. [This would certainly be expensive in terms of both time and space.]

3. If a transfer of control is made to an address that corresponds to a byte that lies within—that is, after the first byte of—an instruction object, then a situation similar to the previous one arises and the equivalent of a byte image must be maintained and used to reconstruct instructions in memory.

We believe that these three situations are rare and have decided not to support them until a later version of our simulators, although we are keeping them in mind as we design.

4 DESIGN PROTOTYPE

We developed a prototype as a proof of concept for our design. Our prototype implementation supports simulation of a simple computer architecture that has byte addressing and single-word (two 8-bit bytes) instructions. The basic architecture is illustrated in Figure 3. All addresses are real and comprise twelve bits. Instruction and data memory are physically separate. Transfer instructions contain instruction memory addresses, while all other instructions contain data memory references. There is a single sixteen-bit accumulator (AC), a twelve-bit stack pointer register (SP) used for subroutine calls and returns, and a status register that describes characteristics of the contents of AC.

The prototype was built in C++ to prove the design concept and to collect rough performance characteristics. We purposely chose a simple architecture for our initial experiments so that we could evaluate our object-oriented design, devoid of nonessential considerations. We also built a corresponding simulator in C based on traditional simulator design so that we could directly compare the designs and their performance.

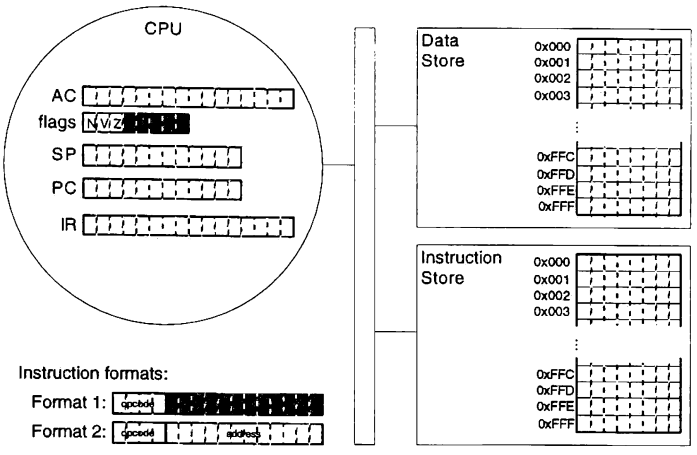
The design object model for the prototype is shown in Figure 4. It includes some details about the class specifications. In that model, Environment represents the computer configuration, including the registers in the CPU. We have not included processing units that perform arithmetic, for example, in the prototype, but future designs will include such components. We are currently evaluating the design vis-a-vis the benefits of this change as well as defining a CPU class. We expect the interface for the environment to contain operations that gain access to all system components and handle requests such as fetching instructions from code store and reading and writing registers.

In terms of an algorithm, the most significant change in the object-oriented version from the procedural version is the way in which instructions are fetched from code store, decoded, and executed. The high-level algorithms may be described as:

```
do {
  instPtr =
    this->fetchInstruction(PC.read());
  PC = PC + instPtr->size();
  instPtr->execute(*this);
} while (not done);
```

In a procedural version, all data used in the simulation is typically global to the main simulation loop and to all functions called by that loop. In the object-oriented version, PC is the program counter and **instPtr** is a pointer to the current instruction [object] to be executed, and **this** is the environment that is responsible for running a simulation (member function go). All data used in the simulation is defined in the environment object, but not to other objects it messages, thus the need to store return values—for example, into **instPtr**—and pass parameters to instructions—for example, ***this** to an instruction being **executed**.

Experiments with our prototype show that the object oriented simulator, coded in C++, ran twice as fast as another prototype that we implemented using the traditional approach to simulation where the simulator was written in C. Figure 5 illustrates these results. The table of Figure 5 lists three programs having complexity $O(n)$, $O(n^2)$, and $O(n^3)$. Conceptually, these correspond to programs that compute factorial, perform a sort, and perform a fast Fourier transform, respectively. On average, these programs executed 262,145, 16,787,457 and 67,110,145 instructions respectively. The $O(n)$ program required 97 clock units to execute using the traditional C simulator and only 56 clock units to execute using the



Instruction Set	
LDA addr	*load AC with data[addr]*
ADD addr	*add data[addr] to [AC], set flags*
STA addr	*store [AC] in data[addr]*
JSR addr	*jump to subprogram at address addr and save return address on stack*
RTS	*return from subprogram*
HLT	*stop execution*
BRZ addr	*branch on [AC] zero to addr*
BRN addr	*branch on [AC] negative to addr*
BRV addr	*branch on [AC] zero to addr*
JMP addr	*jump unconditionally to addr*
CMP	*boolean complement [AC]*
XOR addr	*bitwise xor data[addr] with [AC]*
AND addr	*bitwise and data[addr] with [AC]*
NEG	*negate [AC]*
LDR addr	*use the lower twelve bits of AC as a data address and load contents into AC*
NOP	*no operation*

Figure 3: Architecture for the Design Prototype

object-oriented C++ simulator. The other two programs experienced similar speedups. The graph in Figure 5 highlights this result.

From this data we have concluded that the storage of instructions as objects, which by their existence makes them "pre-decoded," speeds up execution. However, we can realize a benefit only for programs that involve looping because initialization of the object-oriented simulator requires more time.

5 CONCLUDING REMARKS

We have presented a design that exploits object technology to produce a flexible, extensible, efficient simulator for developing architectures. We have proven the concept of our design by the development of a prototype for a simple processor. We have compared the execution characteristics of that prototype implementation with the implementation of a simulator in C to determine the costs and benefits of our approach.

We are currently extending the prototype in two ways. First, we are modifying the analysis model to address the Pentium Pro Processor. Second, to further demonstrate the extensibility of our model, we are looking at our design to build an object-oriented framework. As we extend our design over a series of more and more complex processor architectures, ending at a simulator for a Pentium Pro, we will validate our design in terms of its reusability and extensibility.

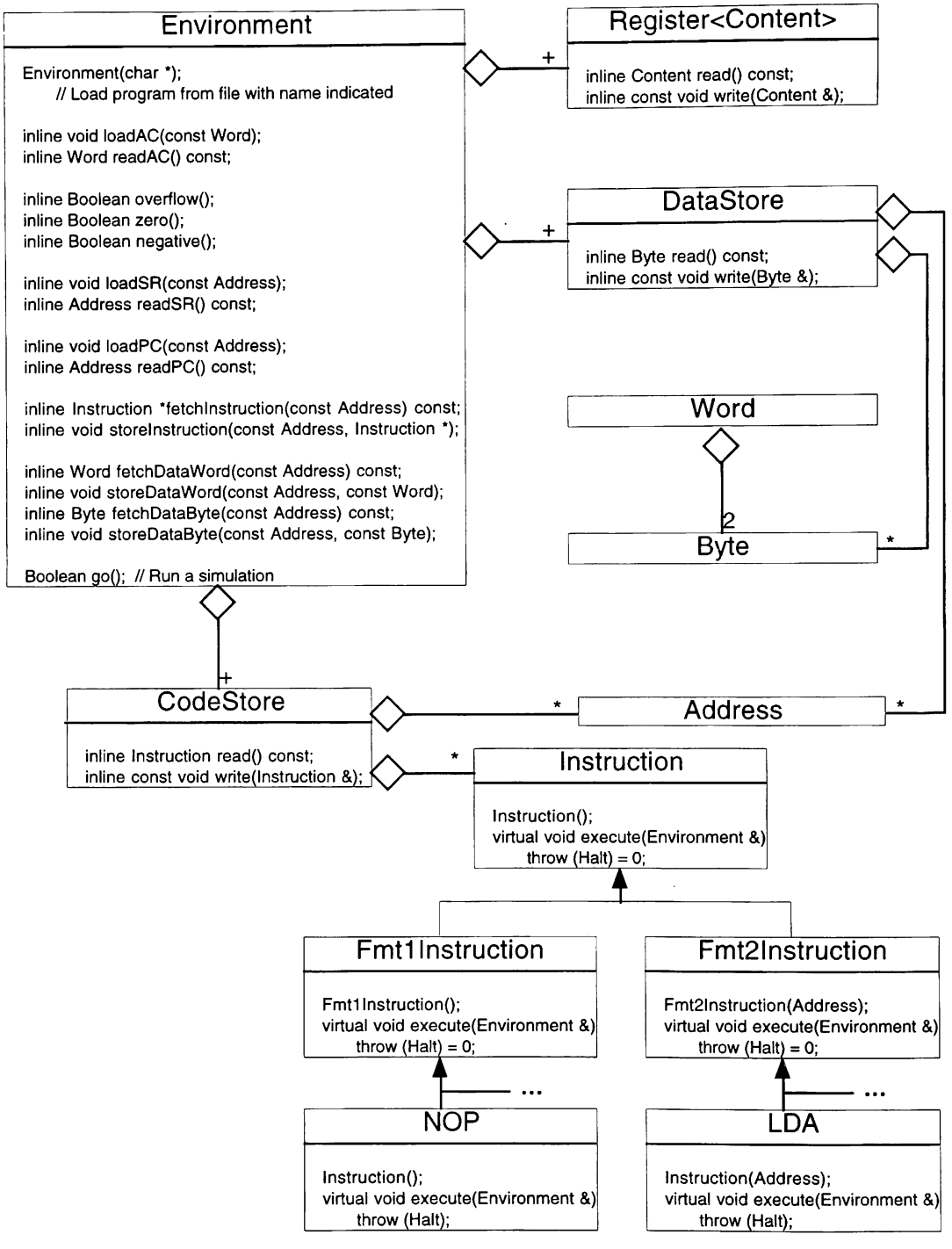
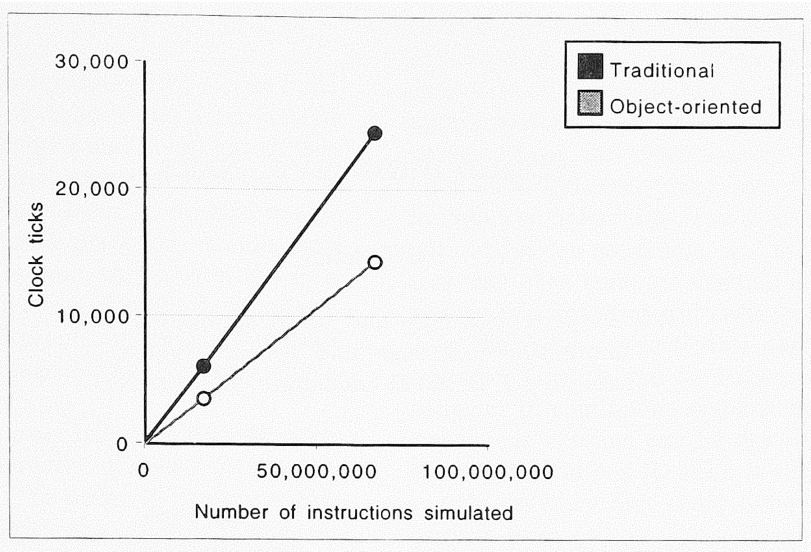


Figure 4: Design Object Model for Our Prototype



Program complexity	Instructions executed	Traditional time	Object-oriented time
$O(n)$	262,145	97	56
$O(n^2)$	16,787,457	6122	3602
$O(n^3)$	67,110,145	24556	14397

Figure 5: Experimental Results

REFERENCES

Ball, T. and Larus, J. R. January 1992. Optimally profiling and tracing programs. In *Proceedings of Symposium on Principles of Programming Languages,* 59–70.

Dahlgren, F. 1991. A program-driven simulation model of an MIMD multiprocessor. In *24th Simulation Symposium,* 40–49.

Koldinger, E., Eggers, S., and Levy, H. 1991. On the validity of trace-driven simulation for multiprocessors. In *Proceedings of 18th Annual Symposium on Computer Architecture,* 244–253.

Malloy, B. 1993. The validation of a multiprocessor simulator. In *Proceedings of the 1993 Winter Simulation Conference,* 625–631.

Mukherjee, R. and Bennett, J. 1990. Simulation of parallel computer systems on a shared memory multiprocessor. In *Proceedings of 23rd Hawaii International Conference on System Science,* 1:242–251.

Phillip, M. J. 1992. Performance issues for the 88110 RISC microprocessor. In *Proceedings of IEEE COMPCON,* 163–168.

AUTHOR BIOGRAPHIES

DAVID A. SYKES is a Principal Investigator with ObjectKnowlogy, Inc. He received a B.S. degree in computer science from Purdue University in 1972, an M.A. degree in computer sciences from the University of Texas at Austin in 1975, and a Ph.D. degree in computer science from Clemson University in 1995. His research interests focus on the design and testing of object-oriented software systems.

BRIAN A. MALLOY is an Associate Professor in the department of Computer Science at Clemson University. He received an M.S. and Ph.D. from the University of Pittsburgh in 1984 and 1991. His research interests focus on simulation techniques that exploit object technology and parallelization.