Extending SIMx86 to Include Prefetching, Segmentation, Virtual Memory Addressing and Protection Mode

Brian A. Malloy
malloy@cs.clemson.edu
Dept. of Computer Science
Clemson University
Clemson, SC 29634

Sudarshan A. Chitre
sudarc@microsoft.com
Microsoft Corporation
26N/1053 One Microsoft Way
Redmond WA 98052

ABSTRACT

The complexity of modern processors together with the high cost of their development requires that processor construction be paralleled by the construction of a simulator to guide design decisions. The major obstacle in using simulation to facilitate development of new architectures is that traditional simulation approaches are not flexible enough to permit easy extension or modification of the underlying simulation model. In this paper, we exploit object technology to present the design and implementation of `Sim286`*, an execution-driven simulator that emulates much of the functionality of the 80286 processor. The design and implementation of* `Sim286` *is accomplished through extension and modification of an existing simulator,* `SIMx86`*.*

1 INTRODUCTION

The complexity of modern processors together with the high cost of their development requires that processor construction be paralleled by the construction of a simulator to guide design decisions. In addition to facilitating the development of the processor, a simulator can also facilitate the development of compiler techniques to exploit the increased speed and functionality of the new processor. Since successful processors are frequently succeeded by newer and better processors, an established simulator can be invaluable in aiding the development of the successor processor. However, the advantages that a simulator provides to a developing architecture require that the simulator be easy to extend and modify. Thus, an important aspect of a simulator for a developing architecture is ease of extension and modification.

In this paper, we exploit object technology to present the design and implementation of `Sim286`, an execution-driven simulator that emulates much of the functionality of the 80286 processor. Object technology facilitates construction of a simulator for a processor by permitting each component of the processor to be represented directly in the simulator by an object. The relationships between these objects in the software reflect the logical relationships between processor components.

Previous approaches to processor simulation include [BAB96, CUL89, CHKW86, CK94, FC88, NMST96, May87] and [VF94]. Two of these efforts include a set of tools to facilitate processor simulation, *SimpleScalar*[BAB96], and a toolkit that executes on the Intel Architecture, *Augmint*[NMST96]. The *SimpleScalar* tool set emulates the MIPS instruction set. A derivative of *SimpleScalar*, *SimpleScalarx86*[BAB96], was developed to demonstrate the extensibility of *SimpleScalar* to other instruction sets; *SimpleScalarx86* emulates the x86 instruction set. The difficulty with the *SimpleScalar* tool set is that, although its instruction set is extensible, the actual simulator design is coded in C and is not easy to extend or modify[AB96]. The *Augmint* tool set inserts calls to the simulator in the input assembly code at compile time. These calls generate events to simulate different actions of the processor. The *Augmint* simulator makes it easy to simulate other memory hierarchies but the simulator does not simulate the actions of any particular processor and modification or extension of the C-coded simulator to other processors is non-trivial.

The remainder of this paper is organized as follows. Section 2 presents background about `Simx86` and the Intel family of processors. Section 3 presents the design of `Sim286` including a detailed description of extensions and modifications to the `Simx86` model. In Section 4 we investigate the performance of `Sim286`

and we draw conclusions in Section 5.

2 BACKGROUND

In this section, we provide background for the Intel family of processors, ranging from the 8086 to the 80386 processor. We also provide background for a previous version of `Sim286`: `SIMx86`[SMS97].

2.1 The 80x86 Processor Family

The origin of the 80x86 family of processors began in 1978 with the introduction of the 8086 processor. Shortly thereafter, the 8088 processor was added to the family. Both processors have 16-bit registers and use 20 bits (little-endian) to address memory; this permits addressing of a megabyte (M) of memory. Instructions provided for three type of operands: memory, register, and immediate. Instructions could combine these operand types in any manner, except that two memory operands could not be included in the same instruction. The important distinction between the 8086 and the 8088 is that the 8086 processor had a 16-bit external data bus and a 16-bit internal data bus where as the 8088 processor had an 8-bit external data bus and a 16-bit internal data bus.

The 80186/80188 architectures, like the 8086 and 8088, are nearly identical. The only difference between the 80186 and 80188 is the width of their data buses. The internal registers of the 80186/80188 are identical to the 8086/8088. The only difference is that the 80186/80188 contains additional reserved interrupt vectors and some very powerful built in I/O features.

The 80286 introduced several new features as compared to the 8086. The 80286 has two different types of operating mode: real address mode and protected address mode. The real address mode was introduced in the 80286 to permit backward compatibility with the 80186 and the 8086 processors. In real mode the 80286 uses 24 bits to address up to 16 megabytes of memory. In the protected mode the 80286 uses 32 bits to address up to 1 gigabyte (G) of memory. The advanced architectural features and full capabilities of the 80286 are realized in its native protected mode. Among these features are sophisticated mechanisms to support data protection, system integrity, task concurrency, and memory management, including virtual storage.

The 80386 added memory paging and introduced 32-bit processing. The size of data registers were increased to 32-bits.

No substantial changes to memory addressing and registers have been made in processors that followed the 80386. Rather, subsequent 80x86 processors have concentrated on fine tuning the micro-architecture of the processor to increase performance. The 80486 introduced pipelining and integrated the CPU and FPU (*Floating Point Unit*) on one chip.

2.2 SIMx86

Figure 1 shows the basic model for the `SIMx86` simulator. The essential entities of a processor are represented by classes in the object model and their relations are shown by the lines joining them. These entities are part of each processor in the x86 family. Processors improve their performance by adding to the functionality of these basic entities. Using this basic model, we intend to evolve our simulators, as the processors of the Intel x86 family evolved, by applying Object Oriented techniques such as inheritance, genericity and polymorphism.

3 THE DESIGN OF Sim286

In this chapter we present our design for the `Sim286` simulator. This design is extended from the design for the `SIMx86` shown in Figure 1. We believe that the model for the `SIMx86` can be extended to design simulators for Intel's 80286, 386, 486 and Pentium processors. We demonstrate the extensibility of the `SIMx86` model by implementing `Sim286` (`Sim286` simulates some of the features of a 80286 processor).

We have added 3 instructions to the instruction set of the `SIMx86`. These are the far `Call` (for immediate and memory addressing modes), the far `Jump` (for immediate and memory addressing modes), and the `Ret` instruction for inter-segment return. `Sim286` simulates most of the features of a 80286 processor. The new features that are added to the simulator for the 8086, to extend it to a simulator for the 80286 are, virtual memory addressing, protection mechanism, segmentation, prefetching of instructions, and the two modes (real and protected) of operation. Functionality is added to use COM file as input in protected mode. Functionality is added to collect cycle count information for a executing program. The cycle counts for the individual instructions and memory accesses are defined as constants. Accurate cycle counts for the execution of the program may be calculated if cycle counts for the individual instructions and memory accesses are available. We currently do not present cycle count information.

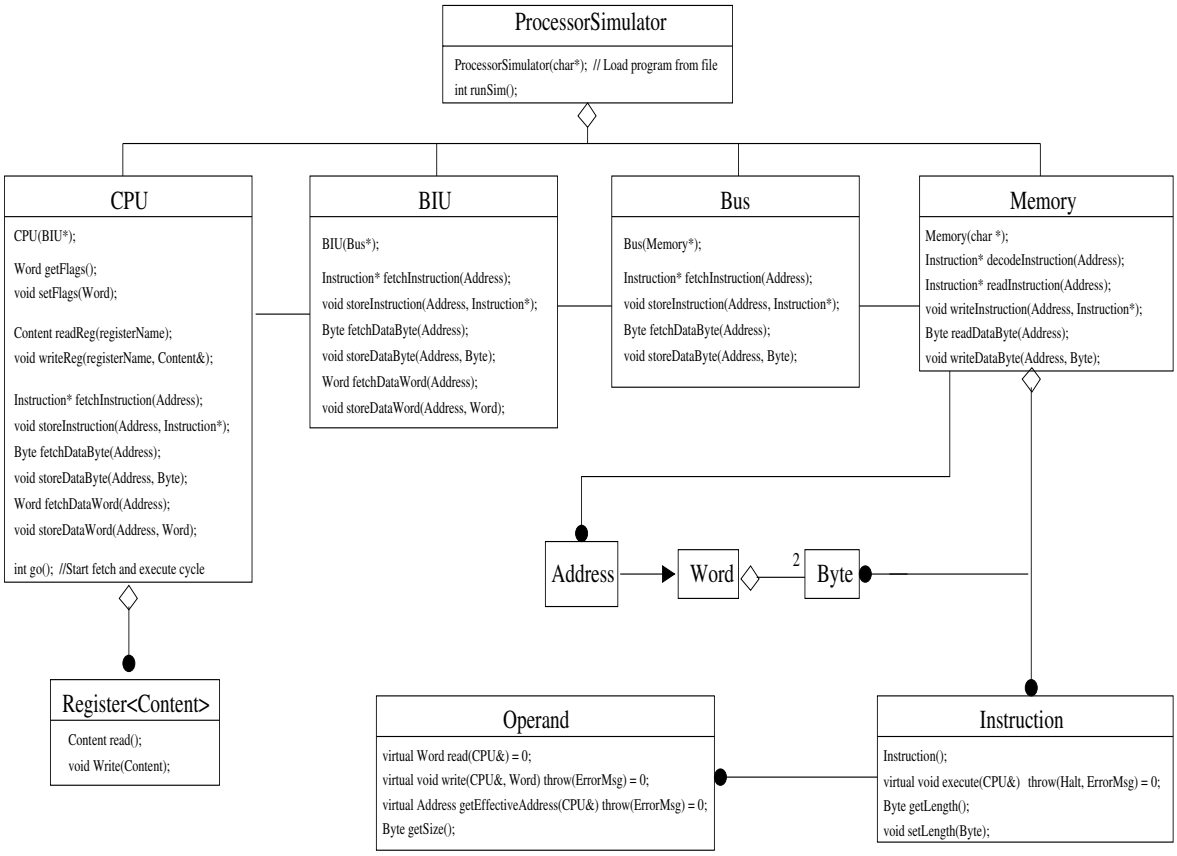

Figure 1: `SIMX86`: the original model for the 80x86 architecture.

`Sim286` does not implement simulation of task management and interrupts. We believe that our model for the `Sim286` could be extended to incorporate these features. These features may be added in the future simulators extended from this model.

In designing the `Sim286`, additions, extensions and modifications are made to the `SIMx86` design. We will discuss each of this in detail in the following subsections. In designing `Sim286`, whenever a tradeoff between extensibility and performance was evident, our primary consideration was always extensibility. Our secondary consideration was to construct the simulator model so that entities in the simulator resembled entities in the actual architectural model.

We have designed the `Sim286` with capability of simulating *prefetching*. In prefetching, the instructions are prefetched and predecoded in parallel with other operations of the CPU that do not use the BIU or the decode unit respectively. To simulate the parallel execution of events, we added an event list; the detailed design of the classes in the *Simulation Architecture* component is shown in Figure 2. The operations of the processor are partitioned into events and each event is triggered at appropriate time.

The `Event` hierarchy of Figure 2 has 15 different events that can occur. When an event is generated it is provided with a trigger time, a priority, and possibly another event on which it depends. When the system time becomes the trigger time, the event occurs (is triggered). Events occur in the order of their trigger time and for events with the same trigger time, the events with higher priority are triggered before the ones with lower priority. The two key methods in the abstract base class `Event` are `canOccur` and `execute`. The implementation of these methods depends on the particular event in the hierarchy.

An event is triggered by invoking the `execute` method of the `Event` class; the `execute` method calls another method `canOccur` to see if the event can occur at this time. In case the event has all the resources that it needs to occur, the actions associated with the event are simulated. The `canOccur` method checks to

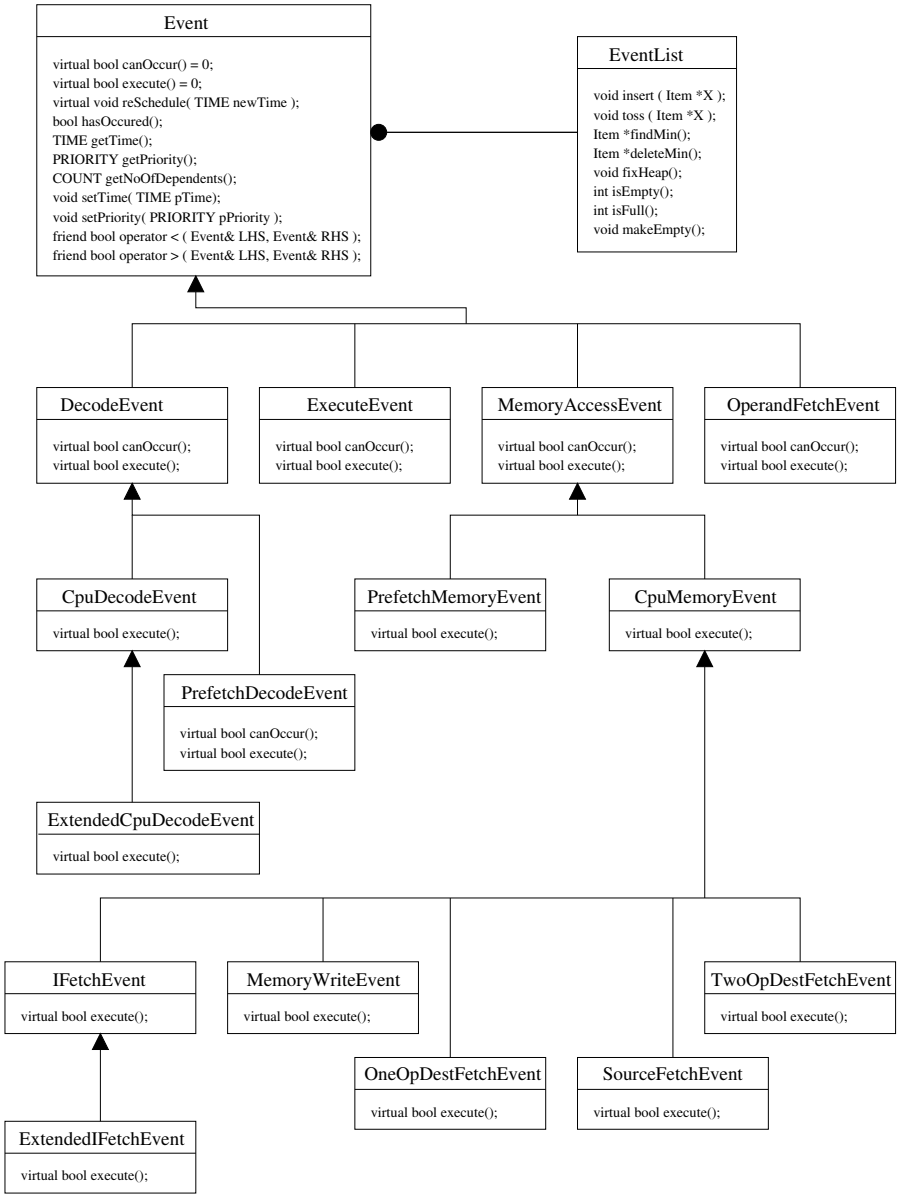

Figure 2: This figure summarizes the events that are simulated in Sim286.

see if the resources are available for the execution of that event. For memory events, `canOccur` checks if the BIU is idle; for decode events, `canOccur` checks to see if the decode unit is idle. Apart from the resources, `canOccur` also checks for dependencies between events. An event may have a list of events that it depends on. An event can occur only when all the events that it depends on have occured. The methods operator< and operator> in the `Event` class are used to decide the order in which the events have to be triggered.

The `EventList` class in Figure 2 is a binary heap where the events are inserted as they are generated and then they propagate up towards the root as their trigger time comes near. The event at the root is always the one with the smallest trigger time and the one with the highest priority if there are more than one with same trigger time.

Program	*No. of Insts.*	*Sim286Real (seconds)*	*Sim286Protected (seconds)*	*Sim8088 (seconds)*	*Sim286Real/ Sim8088*	*Sim286Protected/ Sim8088*
1. `fibbk`	797	0.22	0.23	0.01	22	23
2. `gauss`	9,336,673	2,815.69	2,944.73	44.78	62.86	65.76
3. `isort`	50,609	14.88	15.57	0.21	70.86	74.14
4. `livermore`	8,815	2.68	2.82	0.04	67	70.5
5. `matmult`	5,290,788	1,443.26	1,509.7	22.73	63.5	66.42
6. `normal`	3,613,580	1,024.75	1,067.7	15.08	67.95	70.8
7. `qsort`	3,609,661	1,011.89	1,057.5	14.09	71.82	75.05
8. `sieve`	224,961	73.44	76.58	0.98	74.94	78.12
9. `tiling`	4,506,527	1,321.92	1,381.01	19.02	69.5	72.61
10. `transpose`	2,641,436	779.13	810.27	11.51	67.69	70.34

Figure 3: Performance Results for the Test Suite of 10 programs.

4 PERFORMANCE

In this chapter we compare the execution times of `Sim286` to its predecessor `Sim8088`. We present the statistics collected by `Sim286`. We then go on to show some test scenarios run on `Sim286` to test its functionality and in the end we discuss some ways in which we could improve the performance of our simulator.

In chapter 3, we presented our model of a simulator for the 80286 architecture, `Sim286`, that exploits object technology to produce a design that is easy to extend and easy to modify. When making design decisions, our primary consideration was extensibility and ease of modification over performance. Our secondary consideration was for design to closely reflect the actual architecture. To illustrate how we effected these decisions, `Sim8088` decoded each instruction the first time it was encountered and saved the results, thus saving time when encountering instructions repeatedly during loops. However, `Sim286` decodes each instruction as part of the execution cycle; thus, instruction execution in `Sim286` closely reflects instruction execution in the 80286 architecture. These two design considerations can erode performance.

To provide some measure of the performance of `Sim286`, we compare execution of a test suite of ten programs on both `Sim286` and `Sim8088`. The problem with this comparison is that we are comparing two different architectures. `Sim286` implements much more functionality than `Sim8088`. However, both simulators are derived from the same base model, they both take COM files as input, and both simulators simulate the same core instruction set.

All experiments with the test programs were conducted on a Gateway 2000 with a 133 MHz Pentium processor running the Solaris 2.5 operating system. The programs were executed ten times and the execution times reported in this chapter are averages over these ten executions.

Our testsuite is composed of ten C programs that include both scientific and general purpose workloads. To create binary executables for both `SIMx86` and `Sim286`, we use a C compiler for the PC to produce 8086 assembly code. The 8086 assembly code is assembled using Wolfware Assembler, or WASM[Tau85], to create an executable COM file. A COM file is a command processing file that contains executable commands or statements.

Figure 3 presents in tabular form, the results of our experiments. The first column of the table lists our test programs, the second column lists the number of instructions executed, the third and fourth columns list the execution times for `Sim286` in Real and Protected mode respectively. Column five lists the execution times for `Sim8088`. The final two columns lists the ratio of execution times for `Sim286` to `Sim8088`.

The test programs listed in column one of the table in Figure 3 include a program to compute Fibonacci numbers, `fibbk`; a program that uses Gaussian elimination without pivoting, `gauss`[Wol96]; an insertion sort, `isort`; the first Livermore loop, `livermore`[McM72]; matrix multiplication, `matmult`[Wol96]; a program to transform a matrix into *Hermite normal form*, `normal`[Wol96]; the quicksort program, `qsort`; the sieve of Erosthothenes, `sieve`; a program that uses *tiling* to optimize data cache references, `tiling`[LRW91]; and a program to perform matrix transposition, `transpose`.

To illustrate our results, the second row of Figure 3 presents performance results for the gauss program. `Sim286` required 2,198.59 seconds to simulate execution of the gauss program in real mode and 2,815.69 seconds to simulate execution of the gauss program in protected mode. `Sim8088` required only 44.78 seconds to simulate execution of the gauss program; thus, `Sim8088` is approximately 62 times faster than `Sim286` when `Sim286` executes in real mode, for this program.

5 CONCLUDING REMARKS

In this paper we have presented the design and implementation of `Sim286`, an execution-driven simulator that emulates much of the functionality of the 80286 processor. We have shown how object technology facilitates construction of a simulator for a processor by permitting each component of the processor to be represented directly in the simulator by an object. We have shown the extensibility of the `SIMx86`[SMS97], the predecessor of `Sim286`, by extending the design of `SIMx86` to include much of the functionality of the 80286 processor. The design of `Sim286` exploits polymorphism, overloading and genericity to produce a model that is easy to extend and modify. We have compared the performance of `Sim286` to its predecessor, `Sim8088`. Finally, we have outlined how our model for `Sim286` can be extended to simulate some of the functionality of the 80386 processor.

References

[AB96] Todd Austin and Doug Burger. Personal communication, October 1996.

[BAB96] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report 96-1308, Wisonsin University, July 1996.

[CHKW86] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a risc compiler system. *IEEE COMPCON*, March 1986.

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1994.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *Proceedings of OOPSLA '89*, October 1989.

[FC88] Richard M. Fujimoto and William B. Campbell. Efficient instruction level simulation of computers. *Transactions of the Society for Computer Simulation*, 5(2), 1988.

[LRW91] M. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of Fourth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[May87] Cathy May. Mimic: A fast s/370 simulator. *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, June 1987.

[McM72] F. H. McMahon. FORTRAN CPU performance analysis. *Lawrence Livermore Laboratories*, 1972.

[NMST96] A. T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. *Proceedings of the International Conference on Computer Design*, October 1996.

[SMS97] A. R. Shealy, B. A. Malloy, and D. A. Sykes. Simx86: An extensible simulator for the intel 80x86 processor family. *Proceedings of the 30th Annual Simulation Symposium*, pages 157–166, April 1997.

[Tau85] Eric Tauck. *WASM 1.0: Wolfware Assembler for the IBM Personal Computer.* Wolfware, 1985.

[VF94] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. *Proceedings of the Second International Workshop on Modeling and Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, first edition, 1996.