Abstract

We report on our progress in developing a family of simulators, Sim86, for the Intel 80x86 family of processors. Our functional simulator covers most of the 80386, 80486, Pentium and Pentium Pro instruction sets. The extensible framework of classes that we present includes the important components required for simulating a high performance processor. We have already extended the framework to include demand paging and we are extending the framework to include cache memory. The simulator is amenable to further extension using subtyping, as described in this paper. Our current implementation provides the basis for studying the use of object technology to construct simulators, for gathering profile information to guide compiler optimizations and to examine coverage of the instruction set exercised by a program.

1 INTRODUCTION

The trend in the development of high performance processors is that each new processor is soon replaced by a newer and more powerful processor. The Intel family of 80x86 processors is illustrative of this trend: the Pentium has been replaced by the Pentium Pro and the Pentium Pro has been replaced by the Pentium II. The Pentium II is soon to be succeeded by the Merced.

To facilitate processor development, the design and implementation of a processor is typically paralleled by the design and implementation of a simulator that can be used to avoid errors in the development process. The simulator can also be used to guide design decisions, since software is more flexible than hardware. The ideal is that the family of processors should be accompanied by the design and implementation of a family of simulators where each successive simulator can be derived from the previous by an incremental change in both the design and implementation of the simulator. All too often, the family of simulators ideal is not achieved and the construction of a high performance processor must begin with the construction of a processor simulator.

In this paper, we report on our progress in developing a family of simulators, Sim86, for the Intel 80x86 family of processors. We begin by reviewing previous simulators in the family, Sim8086 (Shealy et al. 1997) and Sim286 (Malloy & Chitre 1998). We then describe our effort in building Sim386, the successor to Sim286. The construction of Sim386 involved several important extensions over its predecessor. First, Sim386 performs both 16-bit and 32-bit processing; the predecessor of Sim386 performed only 16-bit processing. Second, Sim386 can accept both COM and ELF binary file input; the predecessor of Sim386 accepted only COM file input. The first extension exposed an important drawback in Sim286: the extension of Sim8086 to Sim286 involved implementation extension but not a corresponding design extension. Thus, the inclusion of 32-bit processing into Sim386 required a major redesign effort; however, the redesign of Sim386 empowered easy extension to wider word architectures, including 64-bit processing. The extension to ELF binaries makes Sim386 a more viable tool, since ELF binaries are more widely accessible than COM files.
Sim386 is a functional 1A-32 simulator covering most of the 80386, 80486, Pentium, and Pentium Pro instruction sets. The construction of Sim386 is based on an extensible framework of classes that includes the important components required for simulating a high performance processor. We have already extended the framework to include demand paging (Utami 1998) and we are extending the framework to include cache memory. Sim386 is amenable to further modification and extension using subtyping, as described in Section 3.

Our current implementation provides a basis for studying the use of object technology to construct functional simulators, comparing design alternatives for their ability to provide extensibility versus efficiency. Furthermore, our simulator can be instrumented to gather profiling information to be used as feedback to guide decisions about compiler optimizations such as branching behavior, inlining behavior and value locality. Finally, our simulator can be instrumented to examine coverage of the instruction set to study instruction patterns and addressing modes exercised by programs or categories of programs.

In the next section we provide background about simulators including Simx86, the family of simulators for the Intel 80x86 architecture, together with background about ELF binaries. In Section 3, we describe our implementation of Sim386 to include 32-bit processing and to accept ELF binary input. In Section 4 we report the results of experiments comparing the COM file executions of the test suite described in reference (Malloy & Chitre 1998) using COM and ELF binaries as input. Finally, in Section 5 we draw conclusions.

2 BACKGROUND

In this section, we provide background about the Intel family of processors that relate to this work. We overview the processors ranging from the 8086 to the 80386, with particular emphasis on the 80386. We also provide background about previous versions of Simx86 simulators including Sim8088 (Shealy et al. 1997) and Sim286 (Malloy & Chitre 1998).

2.1 The 80x86 Processor Family

The origin of the 80x86 family of processors began in 1978 with the introduction of the 8086 pro-

![Figure 1: Overview](figure.png)

Figure 1: Overview. This figure illustrates an overview of the Intel family of architectures. The solid underlined features are implemented in Sim286, the dash underlined feature is implemented in Sim386. Sim8088 and Sim286 accept COM input files as input; Sim386 accepts both COM and ELF input files.

cessor. 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 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 whereas the 8088 processor had an 8-bit external data bus and a 16-bit internal data bus.

The 80286 introduced several new features into the x86 including two different types of operating modes: real address mode and protected address mode. The real address mode was introduced in the 80286 to permit backward compatibility with previous 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 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
registers have been made in processors that followed the 80386. Rather, subsequent 80x86 processors have concentrated on fine tuning the microarchitecture of the processor to increase performance.

2.2 Simulating the x86 Architecture

In this section we overview the simulators that preceded Sim386. We begin by overviewing Simx86, the framework of classes that describes an architecture for most modern processors. We then discuss Sim286, a simulator for the Intel 80286 processor. A discussion of Sim8088 can be found in reference (Shealy et al. 1997).

Figure 2 shows the basic model for the Simx86 simulator. The essential entities of a processor are represented by classes in the class diagram and their relations are shown by the lines joining them. These entities are included in 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.

The simulator for the Intel 80286, Sim286, can be partitioned into two class frameworks where the first framework is an extension of the framework for Simx86 and the second framework incorporates an event list to simulate the execution of events in quasi-parallel fashion. We adopt the naming convention used in reference (Malloy & Chitre 1998) to refer to these two frameworks as the Architecture Framework, captured in Figure 3, and the Simulation Framework. The extensions to Sim286 that we report in this paper focus on the Architecture Framework. The interested reader may consult reference (Malloy & Chitre 1998) for a description of the simulation framework.

The CPU class for Sim286 is extended to include a global descriptor table register, a local descriptor table register, an interrupt descriptor table register and segment registers. The CPU class is shown on the left side of Figure 3 with classes DescriptorTableReg, DescriptorTable, and SegRegister drawn beneath class CPU. These classes, together with class Register, are the components of our representation of the Intel processor; thus, they form an aggregation relationship with Sim286 from 16-bit processing to 32-bit processing.

The Sim286 simulator extended Simx86 to include both real mode and protected mode. This extension is illustrated in Figure 3 with classes RealModeBIU and VirtualModeBIU derived from the BIU class; the arrow connecter in the figure represents the inheritance relationship (Rumbaugh et al. 1991). The figure also illustrates that the BIU is related to class PrefetchQueue through aggregation. An important feature of Sim286 is the ability to simulate the prefetch and decode of instructions in parallel with other CPU operations.

2.3 The Executable and Linking Format (ELF)

The executable and linking format (ELF) was originally developed by Unix System Laboratories and is rapidly becoming the standard in file formats (Standards 1998). The ELF standard is growing in popularity because it has greater power and flexibility than the a.out and COFF binary formats (L 1998).

The scope of this paper precludes a presentation of our technique for incorporating ELF binary input into Sim386; the presentation can be found in reference (Haungs 1998). Reference (Standards 1998) presents an overview of the ELF file format including a detailed description of each of the five section types that an ELF file might include. Reference (L 1998) describes the representation of data in an ELF file is described.

3 THE IMPLEMENTATION OF SIM386

In this section, we present our design and implementation of the Sim386 simulator, an extension of the Sim286 simulator. Through subclassing, Sim386 increases extensibility over Sim286 and facilitates extension of Sim386 to simulators for other architectures. Sim386 simulates most of the features of the 80386 processor. These features are those inherited from Sim286 (virtual memory addressing, protection mechanism, segmentation, prefetching of instructions, and the real and protected modes of operation) and those added by the extensions in the current work (32 bit processing and ELF binary input). We also modified the user interface of Sim286
to take additional command line parameters to eliminate recompiling the simulator for different modes of operation.

First, in section 3.1, we describe the modifications made to the class framework of Sim86 (shown in Figure 3) necessary for the components in Sim386. Next, in section 3.2, we describe the modifications and additions that were necessary to convert Sim86 to 32-bit processing. In Section 3.3, we summarize our approach to extending Sim86 to accept ELF binary input.

### 3.1 Extending the design of Sim86

To incorporate ELF binary input, we exploited subtyping to extend the Memory class and the ProcessorSimulator Class, illustrated in Figure 3. Both of these classes were designed to accept input in COM file format and to simulate execution in an MS-DOS type environment. The Memory and ProcessorSimulator classes are redesigned so that they are easier to extend to alternate input file formats and operating environments. The Memory class is made into a base class, shown on the left in Figure 4, that embodies a common interface to different types of input files. Two new classes, both derived from Memory class, are added to the framework. The first class that is added is COMMemory, which manages COM input files exactly as done in Sim86. Thus, Sim86 is backward compatible with Sim86. The other class that is added, ELFMemory, manages ELF input files. Figure 4 shows an additional class, COFFMemory, derived from Memory; this class is dashed to illustrate that it is not currently implemented in Sim86 but may be added using subtyping.

We extended the ProcessorSimulator class, also illustrated in Figure 4, in a similar manner. The ProcessorSimulator class is made a base class that performs simulator initializations that are common to all platforms. Initializations that are particular to specific platforms are incorporated into subclasses of ProcessorSimulator. Two such subclasses are DOSProcessorSimulator and LinuxProcessorSimulator; these classes perform
platform dependent simulator initialization. Figure 4 also shows SolarisProcessorSimulator class as a subclass of ProcessorSimulator; this class is dashed to illustrate that it is not currently implemented in Sim386 but may be added using subtyping.

The class framework that we have incorporated into Sim386 empowered a change to the command line interface of the simulator that obviated recompilations for different modes of operation. We can now use dynamic binding to instantiate the exact simulator that we require.

3.2 Adding 32-bit processing

Converting from 16-bit processing to 32-bit processing required the following additions to Sim386: (1) 64 new addressing forms, (2) the addition of SIB byte to the instruction format, (3) adding additional instructions, (4) modifying current instructions to perform 32-bit calculations, (5) modifying segmentation, and (6) increasing the size of the bus and registers. These modifications are described in the sections that follow.

3.2.1 Adding additional addressing modes and the SIB byte

The instruction format for the Intel386 is shown in Figure 5 (Corporation 1998). The major change from the format for Sim386 is that a new byte, the SIB byte, is part of the instruction format. The SIB byte encodes 32 additional forms of addressing available for each instruction. Figure 6 describes and shows the layout of the SIB byte (Corporation 1998).

3.3 Adding ELF binary Input

One of the most important contributions of this work is the extension to permit Sim386 to accept ELF object files as input to the simulator. The ability of Sim386 to accept ELF files enables additional functionality to be incorporated into Sim386, for example paging and multiprogramming. The ad-
Figure 4: Extending the class framework. The classes shown in bold are new classes in Sim386. The dashed classes show possible extensions. The triangle indicates an inheritance relationship, with Memory and ProcessorSimulator as base classes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address size</th>
<th>Operand size</th>
<th>Segment Override</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0 or 1 bytes)</td>
<td>(0 or 1 bytes)</td>
<td>(0 or 1 bytes)</td>
<td>(1 or 2 bytes)</td>
<td>(0 or 1 bytes)</td>
<td>(0 or 1 bytes)</td>
<td>(0.1, 2 or 4 bytes)</td>
<td>(0.1, 2 or 4 bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: General Instruction Format. This format consists of one or two opcode bytes, a MOD byte, a SIB byte, an address displacement (optional), and immediate data (optional). Prefix bytes can also precede the instruction in order to override the default segment, operand, and address size used. Prefix bytes are optional.

dresses used in a COM file undergo a logical to linear transformation and these linear addresses are then assumed to be the physical addresses. However, to simulate paging, linear addresses must be translated into physical addresses by utilizing page table lookup. Thus, they cannot adequately test paging or multiprogramming capabilities in, for instance, a Solaris 2.XX or Linux operating environment. Further, compilers and assemblers that can generate object files in the COM format are not readily available so that it becomes difficult to generate test programs for the simulator. Since most executables today are in ELF format, existing test suites can be used as input to the simulator.

4 PERFORMANCE OF SIM386

In this section, we report the results of some experiments that gather timings for Sim386, our simulator for the Intel 80386 processor. Sim386 can accept either COM or ELF binary input. Since both Sim286 and Sim386 accept COM files as input, we compare timings for these two simulators using a test suite of nine programs with COM binary input. We also report timings for Sim386 using ELF binary input. All experiments with the test programs were conducted on a Gateway 2000 with a 200 MHz Pentium Pro processor running the Linux Red Hat 5.0 operating system. The programs were executed ten times and the execution times reported in this chapter are averages over these ten executions.

To create COM binary executables for both Sim286 and Sim386, we use the Borland 4.5 C compiler to produce 8086 assembly code. The 8086 assembly code is assembled using Wolfram Assembler, or WASM(Tauck 1985), to create an executable COM file. To create ELF binary executables for Sim386, we use the gcc C compiler version 2.7.2.3 with O2 optimizations.

The test programs listed in column one of the table in Figure 7 include a program to compute Fibonacci numbers, fibbk; a program that uses Gaussian elimination without pivoting, gauss(Wolfe 1996); an insertion sort, isort; the first Livermore loop, livermore matrix multiplication, matmult(Wolfe 1996); a program to transform a matrix into Hermite normal form, normal(Wolfe 1996); the sieve of Eratosthenes, sieve; a program that uses tiling to optimize data cache references, tiling(Lam et al. 1991); and a program to perform matrix transposition, transpose.

Our experiments indicate that for COM file input, Sim286 is, on average, 2.06 percent faster than Sim386 when using the test suite of nine programs. Additional logic is included in Sim386 to check for operand size during processing. Sim386 now in-
Figure 6: *SIB byte*. The SIB byte consists of a 2 bit scale field, a 3 bit index field, and 3 bit base field. It specifies the based indexed and scaled indexed forms of 32-bit addressing.

includes an additional check for 32-bit operands; this check is not a part of Sim286.

The table in Figure 7 also indicates that six of the nine programs slow down for ELF binary input and the three others are faster. For example, gauss, matmult, normal, sieve, tiling and transpose slow down when ELF binary input is used. Of these six programs, sieve uses a one-dimensional array and the other five programs use two-dimensional arrays. When performing array computations the SIB byte is used heavily by the gcc compiler. Sim386, in determining the additional addressing modes provided by the SIB byte, requires more time when simulated in software but is more efficient when executed in hardware. Thus, for the test suite, those programs that make heavy use of array computations are slower when they accept ELF binary input.

5 CONCLUSIONS

In this paper we describe the design and implementation of Sim386, a partial simulator for the Intel 80386 processor. Our simulator performs both 16-bit and 32-bit processing and accepts both COM and ELF binary input. Sim386 incorporates most of the components to enable simulation of high performance processors. We report preliminary results of experiments with nine test programs.

References


<table>
<thead>
<tr>
<th>Program</th>
<th>No. instr 286 (COM input)</th>
<th>No. instr 386 (ELF input)</th>
<th>286 COM (seconds)</th>
<th>386 COM (seconds)</th>
<th>386 ELF (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibbk</td>
<td>797</td>
<td>135</td>
<td>0.14</td>
<td>0.15</td>
<td>0.03</td>
</tr>
<tr>
<td>gauss</td>
<td>9,336,673</td>
<td>13,436,169</td>
<td>1806.9</td>
<td>1828.5</td>
<td>2875.28</td>
</tr>
<tr>
<td>isort</td>
<td>50,609</td>
<td>52,982</td>
<td>9.22</td>
<td>9.33</td>
<td>10.63</td>
</tr>
<tr>
<td>livermore</td>
<td>8,815</td>
<td>5,218</td>
<td>1.67</td>
<td>1.7</td>
<td>1.07</td>
</tr>
<tr>
<td>matmult</td>
<td>5,290,788</td>
<td>8,900,880</td>
<td>930.89</td>
<td>941.41</td>
<td>1905.74</td>
</tr>
<tr>
<td>normal</td>
<td>3,613,580</td>
<td>8,633,740</td>
<td>651.57</td>
<td>660.57</td>
<td>1889.9</td>
</tr>
<tr>
<td>sieve</td>
<td>224,961</td>
<td>186,875</td>
<td>46.06</td>
<td>46.5</td>
<td>43.12</td>
</tr>
<tr>
<td>tiling</td>
<td>4,506,527</td>
<td>6,872,184</td>
<td>829.48</td>
<td>841.03</td>
<td>1449.97</td>
</tr>
<tr>
<td>transpose</td>
<td>2,641,436</td>
<td>3,056,643</td>
<td>489.48</td>
<td>495.04</td>
<td>645.49</td>
</tr>
</tbody>
</table>

Figure 7: Performance results for the test suite of 9 programs.