Efficient Instruction Level Simulation of Computers

Richard M. Fujimoto
William B. Campbell

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF UTAH

UUCS-87-022

EFFICIENT INSTRUCTION LEVEL SIMULATION OF COMPUTERS

Richard M. Fujimoto[1]
Computer Science Department
University of Utah
Salt Lake City, UT 84112
(801) 581-4845

and

William B. Campbell
Tektronix Corp.
P.O. Box 500 M/S 39-222
Beaverton, OR 97077
(503) 627-1583

Keywords: simulation techniques; simulation of computer systems; instruction level simulation; discrete simulation; direct execution.

[1]This work was supported in part by IBM under a faculty development grant and from ONR contract number N00014-87-K-0184.

ABSTRACT

A technique for creating efficient, yet highly accurate, instruction level simulation models of computers is described. In contrast to traditional approaches that use a software interpreter, this technique employs *direct execution* of application programs on the host computer. An assembly language program for the machine to be modeled is *decompiled* to a high level language, instrumented, and then recompiled and executed on the host computer. A prototype implementation modeling the Motorola MC68010 microprocessor is described, and the efficiency and accuracy of this prototype is reported. It is demonstrated that the direct execution technique can be used to produce accurate simulation models which are orders of magnitude faster than traditional, register transfer level simulators.

1. INTRODUCTION

The simulation of computers using other computers is an important but computationally expensive process that continues to challenge the fastest available machines. Computer manufacturers often rely on simulation of benchmark programs to analyze new and existing machine instructions in order to evaluate their impact on performance. The high computation costs of these simulations often force designers to limit their benchmarks to small programs that are not representative of typical machine usage.

This problem is especially acute when simulation of large multiple processor systems containing hundreds or thousands of CPUs is required. As multiple processor systems become more prevalent, simulations of even small programs becomes problematic. New techniques are required to improve the efficiency of simulators for such large systems.

This paper will focus attention on instruction level simulation of computer systems, i.e., simulation that emulates the execution of application programs to the extent that the simulator performs the same computations that the application program would execute and generates the same numerical results. Instruction level simulation is used extensively to evaluate the performance of single and multiple processor computers (e.g., see Tamir 1981 and Fujimoto 1983b). Such evaluation often takes place during the design of the machine before a physical implementation has been realized, so performance evaluation based on *measurement* techniques are not always possible (Ferrari 1978). Instruction level simulation is also sometimes used to evaluate the performance of software systems, e.g., to ensure that real time constraints are satisfied.

Traditional techniques for instruction level simulation use a software interpreter to laboriously simulate the fetch, decode, and execute cycle of each instruction. Such techniques require the simulator to execute hundreds or thousands of instructions in order to reproduce the behavior of each instruction in the application program (VanTuyl 1973, Tamir 1981, and Cragon 1983). These high overheads cannot be tolerated when simulating large programs or large multiple pro-

cessor systems.

The *direct execution technique* greatly enhances the efficiency of instruction level simulation. When used to simulate multiple processor systems, it is most easily used in conjunction with object-oriented simulation methodologies in which different hardware components are modeled by distinct simulator objects (for example, see Fujimoto 1985). A simulator for, say, a large MIMD (Multiple Instruction stream, Multiple Data stream) computer is constructed by combining processor models using the direct execution technique with models for other components of the system, e.g., the switching network. Interactions between components are modeled by time-stamped messages. For example, the processor simulation model will generate a message when the application program running on the simulated processor invokes an interprocessor communication primitive.

The direct execution technique may be used in *parallel* discrete event simulation programs as well as conventional, uniprocessor-based, simulation environments. In other words, the technique may be used to simulate a parallel processor *on* a parallel processor. The aforementioned simulation methodology is frequently used in distributed simulation strategies such as those proposed by Chandy and Misra (1979 and 1981) or Jefferson (1985).

Throughout this paper, the *host* machine refers to the computer on which the simulator is executing. The *target* machine refers to the processor being modeled. We distinguish between the *behavioral model* which ensures that the computation performed by the application program is faithfully reproduced by the simulator, and the *performance model* which estimates the execution time of the program on the target machine. It is assumed that both the host and target machines are general purpose von Neumann processors. Finally, a *basic block* is a block of sequential instructions such that control enters only at the beginning of the block and leaves only at the end.

Key features of the direct execution technique that distinguish it from others include:

- **High performance.** Direct execution of application programs avoids the overhead associated with interpretive execution.
- **High accuracy for a large class of processors.** Exact timing statistics can be derived for processors in which execution time is not dependent on dynamic phenomena such as pipeline turbulence or cache behavior. Extensions of the technique to accurately model these phenomena will be discussed.
- **Generality.** The technique requires few restrictions on the machine to be modeled. Only the instruction set for the target machine must be specified. A physical realization need not exist.
- **Portability.** An assembler program for the target machine is decompiled to a high level language (C was used in the prototype implementation), and recompiled for execution on the host. This process may take place on a wide range of hosts, assuming the appropriate high-level language compiler and data types of sufficient precision are available on the host machine. The prototype implementation was demonstrated on Sun™ workstations as well as VAX™ computers.
- **No interference.** Measurement tools which *monitor* the execution of programs through the insertion of software probes may alter the behavior of the machine, invalidating the information that is collected. The direct execution technique *simulates* the target processor rather than measuring an existing processor's performance, so no interference can occur.
- **Flexibility.** The technique is a software-based simulation tool. No special purpose hardware is required.

The central limitation of the approach lies in the detailed modeling of sophisticated processors in which execution time is highly dependent on dynamic phenomena such as pipeline turbulence and cache memory performance. In situations where the execution time of a sequence of instructions is dependent on complex interactions among them, e.g., contention for hardware

resources, exact execution times can only be derived by modeling the internal structure of the CPU. It is possible to extend the approach to perform detailed modeling of such phenomena, but this may incur a significant performance penalty. This problem can be alleviated if the host is a *multiprocessor* system — separate processors can be used to model specific components such as the cache or pipeline. Even if multiple processors are not available, however, we note that (1) the modeler is afforded the flexibility to trade off between simulator efficiency and accuracy by selecting either efficient, approximate performance models or slower but highly accurate simulation models; (2) detailed simulation models will still be more efficient than a register transfer level simulation because the entire processor need not be simulated in such great detail.

The remainder of this paper is organized as follows: In section 2 we review related work. The proposed direct execution technique is introduced in section 3. Details of the approach are described by example in section 4 where a prototype implementation of the MC68010 microprocessor is described. Section 5 contains an evaluation of the prototype. Both the accuracy and efficiency of the implementation for several benchmark programs are reported. Finally, extensions of the technique to model dynamic phenomena and areas of future work are discussed.

2. RELATIONSHIP TO PREVIOUS WORK

Register transfer level simulation models are frequently used for instruction level simulation (Lipovski 1977). Registers, condition codes, and other forms of system state are represented by program variables, and the fetch-decode-execute cycle is implemented with a programmed loop. The direct execution method also uses program variables to model system state, but eliminates the software loop for interpreting instructions. In effect, this loop is implemented by the instruction interpretation circuitry of the host machine. This accounts for the improved efficiency of the technique.

Hardware and firmware emulators essentially perform register transfer level simulation but utilize special purpose hardware to speed up the simulation (Drummond 1973 and Svobodova

1976). Although execution time can be significantly reduced by this approach, there are several important disadvantages. Most importantly, extension of this technique to model parallel computer systems is not straightforward because the special purpose hardware must be replicated or shared among several distinct simulation models. The latter requires expensive multitasking hardware to avoid significant context switching overheads. Also, special purpose hardware and software support is expensive to develop and maintain, and firmware emulators are difficult to debug. Finally, the resulting simulator is not portable.

Variations of the direct execution technique proposed here have been described. The most simple form involves the insertion of timing probes into a high level language (HLL) application program. The number of required probes can be optimized by simple control flow analysis of the program structure (Oldehoeft 1983). Several "software performance systems" or "performance profilers" using this strategy have been reported in the literature (e.g., see Booth et al. 1984, DePrycker 1982, and Fishwick 1984). However, it is difficult to tune these models for specific machine architectures because the timing model is derived by estimating the execution time of HLL primitives on the target machine. Such estimates can only be rough approximations unless specific assumptions are made about the machine code generated by the HLL compiler. Further, it is difficult to extend the technique to model the effects of pipeline turbulence and cache memory because the analysis is conducted at such a high level representation of the program. Finally, an implementation of the model is only applicable to a single HLL and compiler.

A variation of this approach is to compile an HLL application program to the host machine and insert a timing probe into each basic block of the resulting assembler (Fujimoto 1983a). This approach is sometimes used as a measurement technique to benchmark existing processors. The timing may be based on weighted instruction frequency counts or periodic examinations of a real time clock, provided a clock of sufficient precision is available. The central disadvantage of this approach is the poor accuracy of the timing statistics. Since timing information is based on execution times on the *host* machine, these must be converted into the corresponding times on the

target machine. But in general, no tractable relationship exists between execution times on the host and those on the target, so timing statistics will be inaccurate. Even if the host and target machines are the *same*, some inaccuracy may be incurred when real time clocks are used due to interference — the probes affect cache and pipeline behavior. The approach is useful only if approximate timing statistics are adequate.

Recently, new direct execution techniques have been developed independently of the work reported here. These techniques use a direct assembler-to-assembler translation of target machine code to that on the host (Huguet et al. 1987, May 1987). Although this approach can lead to good performance, direct assembler-to-assembler translation requires detailed analyses of both the host and target architectures, and can be difficult if the machines are very dissimilar. For example, the condition codes in the target may not map very easily to those on the host. More importantly, the resulting translator suffers from a lack of portability. Huguet et al. (1987) also proposes compiling an application to both target and host machine code, and establishing a correspondence between the basic blocks of the two. This approach relies, however, on both compilers generating code in which both the same number of basic blocks are created, and the *order* in which the blocks are executed is the same. As the authors point out, this is not the case for many compilers.

3. THE PROPOSED STRATEGY

A new strategy was developed to address the deficiencies associated with existing techniques. In the proposed approach, the target assembler is first translated to a standard intermediate representation. During this translation, code for the performance model is also inserted into the program so that timing statistics can be compiled as the program executes on the host. The resulting program is then translated into machine code for the host machine where it is executed.

We assume in this discussion that the original application program to be simulated is written in a high level language, and that a compiler for the target machine already exists. Alternatively, the application program may be an assembler program for the target machine. In this case,

we simply skip the first step in the procedure outlined below.

It is convenient to use a high level language (HLL) for the standard intermediate representation since this eliminates the need to develop a new compiler for the host machine. The translation of the target machine code to the intermediate representation is therefore a *decompilation* process. The HLL used for the intermediate representation need *not* be the same as that in which the original application program was written, however.

The strategy consists of the following steps (see figure 1):

(1) Compile the HLL program into assembler for the target processor. This step is omitted if the HLL program is already in target assembler.

(2) Translate, i.e., decompile the target assembly language program into a high level language program that performs the same computation.

(3) Analyze the assembly language code to construct the timing, i.e., performance model, and insert timing probes into the program.

(4) Compile the instrumented program for execution on the host processor.

This strategy allows timing analysis and instrumentation to be done at the granularity of the target machine instruction, thereby eliminating many of the problems encountered earlier. Use of a standard intermediate representation enhances portability. These positive factors led us to explore this approach in greater depth. A prototype implementation for the 68010 microprocessor was developed (Campbell 1985). The next two sections describe this prototype and report results concerning execution efficiency and the accuracy of the timing model.

4. MODELING THE MC68010: A CASE STUDY

A prototype implementation of a simulation model for the MC68010 was developed to evaluate the proposed approach. The C programming language was chosen as the intermediate language because of its rich set of low-level operators. One could obtain greater efficiency by

decompiling to a lower level intermediate form, e.g., a three-address format like that commonly used in compilers. Therefore, the performance results derived from the prototype should be viewed as conservative. In this study, the original application programs to be simulated were either written in C or assembler for the target machine.

The model is divided into two distinct parts, one describing the processor's behavior and the other its performance. The behavioral model is obtained by decompiling MC68010 assembler into a functionally equivalent C program. This C program is then instrumented by the performance modeling program, compiled, and executed on the host processor.

A machine architecture is defined in terms of the data types it supports, the machine state visible to the assembler program, addressing modes, and the operations that are provided. The HLL to which the assembler program is decompiled must provide constructs to support each of these aspects of the target machine architecture. Implementation of these facilities for the 68010 architecture is described next, followed by a discussion of performance models.

4.1. Data Types

The 68010 supports the data types byte (8 bits), word (16 bits), and longword (32 bits) (Motorola 1984a). These must be mapped to appropriate data types on the host. The same mapping must be done in conventional (i.e., interpretive) instruction level simulation programs. To simplify the discussion, we will assume that the corresponding data types with the necessary precision are available in the host. In general, abstract data types and operators must be defined if this is not the case.

The data types *b*, *w*, and *l* are defined in C to correspond to these types of the 68010:

```
typedef unsigned char b;        /* byte data (8 bit) */
typedef unsigned short w;       /* word data (16 bit) */
typedef unsigned long l;        /* longword data (32 bit) */
```

4.2. Machine State

The target machine state consists of data storage locations and registers. Data storage is modeled as a single, contiguous block of memory. Registers are modeled by program variables.

Data directives in the assembler program reserve blocks of memory and may optionally specify initial values. Label identifiers are associated with the reserved blocks, and used in the decompiled program when assembler instructions reference memory locations. These data labels are modeled as constant offsets into a single block of dynamically allocated memory.

For example, assume an assembler program contains the following data directives:

```
L0:     .byte 'b', 1, 2, 3        ; four bytes (8 bits each).
L1:     .word 10, 12, 0xff        ; three words (16 bits each).
L2:     .long 0                   ; one longword (32 bits).
```

The following C code would be generated by the decompiler:

```
#define L0 (vars + 0)
#define L1 (vars + 4)
#define L2 (vars + 10)

        char *vars;
init()
{
        vars = (char *) malloc( STACKSIZE );
        *(char *)(vars + 0) = 'b';
        *(char *)(vars + 1) = 1;
        *(char *)(vars + 2) = 2;
        *(char *)(vars + 3) = 3;
        *(short *)(vars + 4) = 10;
        *(short *)(vars + 6) = 12;
        *(short *)(vars + 8) = 0xff;
        *(long *)(vars + 10) = 0;
}
```

By convention, program variables are stored in low addresses of the runtime stack. The stack grows from high addressed locations to low. The *init* routine is invoked prior to the execution of the instruction modeling code. *STACKSIZE* indicates the amount of memory allocated for the stack.

A single structure variable models the MC68010 address and data registers and condition codes. The *m_state* structure, defined in figure 2, contains the necessary definitions. The condition codes are modeled as distinct integer variables because packing them into a single word would require time consuming bit extraction and insertion operations and would save only a negligible amount of storage. Two "scratch" registers *t0* and *t1* are used to hold intermediate results necessary for the modeling of some instructions and condition codes. Member names in the *d_reg* and *address* union structures correspond to the byte, word, and longword register modes of the MC68010. The stack pointer variable is initialized to point to the top of the simulated runtime stack.

4.3. Addressing Modes

Addressing modes are easily modeled in C. Table 1 gives examples of 68010 addressing modes and the corresponding C code. The '*' operator in C is used extensively to implement indirect addressing.

4.4. Representative Instructions

We shall briefly describe the implementation of a few key instructions, summarized in table 2. A more complete description is described by Campbell (1985). As can be seen from table 2, many frequently used 68010 instructions map to a single C statement. For example, the MOVE instruction is implemented as a simple assignment statement and the ADD as an addition.

The CMP (compare) instruction affects only the condition codes. Condition code settings were omitted from the previous examples to simplify the explanations. The assembler instruction "cmpl d2, d0" which compares the D0 and D2 registers decompiles to the following C statements:

```
{
reg.t1.l = reg.d0.l - reg.d2.l;

/* CCV */
reg.V = 0x80000000 & ~reg.d2.l &  reg.d0.l & ~reg.t1.l ||
        0x80000000 &  reg.d2.l & ~reg.d0.l & reg.t1.l;

/* CCC */
reg.C = 0x80000000 &  reg.d2.l & ~reg.d0.l ||
        0x80000000 &  reg.t1.l & ~reg.d0.l ||
        0x80000000 &  reg.d2.l &  reg.t1.l;

/* CCZ */
reg.Z_bar = reg.t1.l;

/* CCN */
reg.N = 0x80000000 & reg.t1.l;
}
```

The boolean expressions used to set the condition codes are, for the most part, translated directly from MC68010 documentation (Motorola 1984a).

The unconditional jump instruction is modeled by a goto statement and conditional jumps by a conditional goto: The conditional *GE* constant in table 2 is macro expanded to the expression:

```
reg.N && reg.V || !reg.N && !reg.V
```

corresponding to the definition given in the MC68010 reference manual. Similar C expressions are defined for each of the 16 MC68010 conditional mnemonics. Here, we assume that branch target operands generated by the HLL compiler are always labeled instructions. Were this not the case, the target address could easily be computed and a jump table used to reach the target instruction.

Modeling the JSR (jump to subroutine) instruction requires some reflection. In the current implementation, the JSR instruction is implemented as a parameterless function call. The stack pointer is used for passing function parameters and returning values. Similarly, the RTS (return from subroutine) is modeled by the C *return* statement.

4.5. Subroutines and System Calls

The original HLL application program may contain several HLL procedures. The decompiler must recognize the beginning and end of each so that the proper code segments may be encapsulated as C functions. This is easily accomplished by constructing a list of subroutine names and locating the corresponding labels in the assembler program.

Calls to external procedures such as printf require special handling by the decompiler. The strategy currently in place assumes the decompiler does not distinguish between calls to system and user defined procedures. Instead, an interface procedure is provided for each system procedure that moves parameters from the simulated runtime stack to local variables and then calls the system procedure using these variables in the appropriate parameter positions. Similarly, returned results must also be placed on the simulated runtime stack.

Finally, since the code for the system procedure is not instrumented, the interface procedure must also increment the simulation clock by a value indicative of the time required to execute the procedure. In general, the amount of time that is required is not deterministic, so an appropriate model must be developed.

4.6. Modeling Processor Performance

In order to model the *performance* of the MC68010, additional code must be inserted to advance the simulation clock. Each basic block of the target assembler program is analyzed, and an estimated execution time is derived. A statement to increment the simulator clock is then inserted. The timings for the MC68010 opcodes and operands are available in published documentation (Motorola 1984a), so a simple table lookup suffices for most instructions.

A few instructions require a slight modification of this approach. For example, the Bcc (conditional branch instruction) has a different execution time depending on whether or not the branch is taken. This was easily handled because there is a close correspondence between the decompiled code and target assembler instructions. For other instructions, e.g., block moves,

execution time is data dependent. This can be modeled by inserting code that computes the run-time dynamically.

4.7. Implementation

The prototype decompiler was implemented in three parts:

(1) a program that strips comments and separates the assembler data directives from the assembler program text,

(2) a program that generates C code to model MC68010 data areas, and

(3) a program that generates C code to model the assembler instructions.

The first two parts were written using AWK (Aho et al. 1984). AWK was chosen because it allows rapid prototyping of moderately complex string and text handling algorithms. The third part was implemented by a C program that invokes a lexical analyzer generated by LEX to parse the assembler text (Lesk and Schmidt 1984).

The prototype required approximately 1.5 man months to develop. Approximately one month was required to develop the behavioral model, and half a month to develop the performance model. The prototype was developed to evaluate the approach for a reasonably sophisticated microprocessor. It was not designed as a general purpose tool in which arbitrary instruction sets could be easily specified and decompilers generated automatically. This is the next logical step in refining the technique.

5. MEASUREMENTS OF THE MC68010 MODEL

Three test programs were modeled and measured. The first program, called *simple*, is a tight loop that is executed 10,000 times. Since the loop is of minimal length, this program represents a real challenge to the technique insofar as the simulation overhead is inversely proportional to the basic block length. The program *tour* is a recursive solution of the Knight's Tour problem (see Grappel and Hemmenway 1981). Execution timings were measured for solutions of

a five by five and a six by six board. The *H* program is a linked list insertion routine. This application program was originally coded in 68010 assembly language, so compilation to target assembler was not necessary. The *simple* and *tour* programs were coded in C and compiled into 68010 assembler language using the C compiler on a Sun workstation. The decompiled code was again compiled by the portable C compilers on the Sun and Vax host systems to create executable files.

Performance measurements were made using the Sun's time command which samples program execution every one hundredth of a second, as described in Unix™ documentation (1984). System overhead such as paging is excluded from the reported statistics. The programs were executed with no other user processes running on the workstation to minimize interference.

5.1. Validation of the Performance Model

Since precise instruction execution times are available, an exact performance model could be derived. To verify that the timing model yielded correct information, the original programs were first executed and timed on a 10 Mhz, zero wait-state, MC68010 Sun workstation. The timings are given in table 3.

Simulation models of each program were then generated, and the resulting program executed on the host machine. The predicted execution time, measured in machine cycles and scaled to the 10MHz clock rate is reported in table 3. The execution time estimated by the simulator differs from the actual measured time by less than 4%, within the accuracy afforded by the measurement program on the host.

5.2. Efficiency

Simulation overhead is defined as the host CPU time required to execute the simulation model divided by the execution time of the original program on the target machine (taken from table 3). In order to factor out the speed of the host processor, the same processor was selected as the host and target, a 10 MHz 68010. Overhead ratios for the benchmark programs are given in

table 4.

Simulation of the MC68010 on a 68010 host was from six to eight times slower than real time, comparing favorably to techniques using software interpreters. The experiments were also repeated on a VAX 8600. It was found that the 68010 could be simulated in approximately real time on the VAX. Of course, one can always buy a bigger machine to improve the performance of any simulation program, regardless of the technique that is used. The experiments were repeated on the 8600 to demonstrate the portability of the method, as well as to provide an additional point of comparison.

The manner in which condition codes are managed has a dramatic impact on the efficiency of the model. An initial implementation of the program used a brute force strategy in which all condition codes are set on every instruction. However, many machine instructions affect the condition codes, but only a few instructions examine them. Simple data flow analysis techniques were applied to optimize the program by eliminating "superfluous" condition code settings. This improved performance by a factor of two to three over the original naive approach. The overhead ratios in table 4 reflect performance after this optimization is applied. This removal of unneeded condition code settings is counter to the philosophy of most register transfer level simulators where all of the machine state is modeled in great detail. Because such simulators are often used to verify a machine architecture, it is paramount that the machine state is modeled as precisely as possible.

6. EXTENSIONS TO THE MODEL

The direct execution technique can be extended to model more sophisticated processors such as the MC68020. The MC68020 is an architecturally compatible successor to the MC68010. It provides an expanded instruction set, extended addressing modes, instruction execution pipeline, and an on-chip instruction cache (Motorola 1984b).

Extending the behavioral model is relatively straightforward. The performance model is more problematic, however, since exact instruction timings for the MC68020 cannot be derived statically. Only best case and worst case figures can be derived without actually modeling the dynamic behavior of the cache and pipeline.

A wide range of possibilities exist to extend the model to include pipeline effects. A simple approach that minimizes performance degradation of the simulator at the expense of accuracy is to statically analyze each basic block of instructions to evaluate pipeline turbulence. Fixed penalties may be associated with branch taken and/or not taken decisions to model incorrect decisions made by the instruction prefetch policy.

An alternative approach is to perform a more detailed simulation of the internal operation of the processor. One way to accomplish this is to provide a cleaner separation of the behavioral and performance models than that used in the 68010 prototype where the two were intimately intertwined. The performance model now becomes an autonomous functional simulation of the internal operation of the CPU which is driven by the behavioral model. The simulation then operates in much the same fashion as a trace driven simulation such as those reported by Peuto and Shustek (1977) and Kumar and Davidson (1978). Rather than using a tape containing the sequence of instructions executed by the benchmark program to drive the simulation, the behavior model generates the instruction stream. Decoupling the behavioral and timing models in this way facilitates parallel execution if the host machine is a multiprocessor. If only a single processor is available, however, this approach may incur a significant performance degradation depending on the complexity and level of detail that is required, but as noted earlier, the performance will still be better than a full register transfer level simulation because most or all of the processor will not have to be simulated at such a fine level of detail. Detailed timing information concerning the internal operation of the MC68020 was not readily available, so a pipeline model was not implemented in the prototype.

Similarly, the MC68020 instruction cache requires some dynamic modeling of program behavior. Here again, one may trade off efficiency with accuracy. Simple timing models based on apriori assumptions regarding hit ratio may be used if simulator performance is of critical importance, or a more detailed model for the cache may be used to achieve high accuracy at the expense of efficiency. Since the proposed strategy models the application program at a relatively low level, incorporation of such models is straightforward.

7. CONCLUSIONS AND FUTURE WORK

The direct execution technique for modeling processors at the instruction level has been presented. Overhead ratios measured from a prototype implementation demonstrate the feasibility of using this technique for simulating moderately complex microprocessors at a speed within an order of magnitude of real-time assuming the host and target processors provide equivalent performance. This compares favorably with traditional techniques that usually require two or three orders of magnitude degradation. Optimization of condition code settings proved worthwhile, yielding performance improvements ranging from a factor of two to three. Our measurements indicate that a VAX 8600 could simulate the MC68010 in approximately real time. The cost of using the proposed technique lies in the additional time required to compile the program.

Highly accurate performance models can be obtained when instruction execution times are not dependent on complex interactions among instructions. Accurate performance models of processors with sophisticated pipelines and/or caches require simulation of the dynamic behavior of the CPU, or the use of approximation techniques. The strategy can be easily extended to model such behavior, and is well suited for parallel execution if multiprocessor hardware is available. Some performance penalty will result for uniprocessor simulations, but the resulting simulator will still be more efficient than register transfer level simulation.

Several areas of research remain to be pursued. This study was intended to demonstrate the feasibility and evaluate the expected performance of the technique rather than to develop a general purpose tool. Convenient means of specifying the target machine architecture must be developed, as well as programs to automatically generate decompilers. Methods to easily specify and efficiently model dynamic phenomena such as pipelines and caches are also needed.

REFERENCES

Aho, A. V., Kernigan, B. W., and Weinberger, P. J., ''Awk - A Pattern Scanning and Processing Language,'' in Unix User's Manual, Supplementary Documents, University of California, Berkeley, CA (1984).

Booth, T. L., Kim, M., Qin, B., and Albertoli, C., ''PASS: A Performance Analysis Software System to Aid the Design of High Performance Software,'' *International Conference on Computers and Applications*, pp. 317-323 (June 1984).

Campbell, W. B., ''The Efficient Modeling of Processor Behavior and Performance,'' Master's Thesis, University of Utah, Salt Lake City, Utah (September 1985).

Chandy, K. M. and Misra, J., ''Distributed Simulation: A Case Study in Design and Verification of Distributed Programs,'' *IEEE Transactions on Software Engineering* **SE-5**(5) pp. 440-452 (Sept. 1979).

Chandy, K. M. and Misra, J., ''Asynchronous Distributed Simulation via a Sequence of Parallel Computations,'' *Communications of the ACM* **24**(4) pp. 198-206 (April 1981).

Cragon, H., ''Executable Instruction Set Specification,'' *Computer Architecture News* **11**(1) pp. 25-43 (March 1983).

DePrycker, M., ''On the Development of a Measurement System for High Level Language Program Statistics,'' *IEEE Transactions on Computers* **C-31**(9) pp. 883-891 (Sept. 1982).

Drummond, M. E., ''Evaluation and Measurement Techniques for Digital Computer Systems,'' Prentice-Hall, Englewood Cliffs, New Jersey (1973).

Ferrari, D., ''Computer Systems Performance Evaluation,'' Prentice Hall, Englewood Cliffs, New Jersey (1978).

Fishwick, P. A., ''Profgen: A Procedure for Generating Machine Independent High Level Language Profilers,'' *Performance Evaluation Review*, pp. 27-31 (Spring-Summer 1984).

Fujimoto, R. M., ''Simon: A Simulator of Multicomputer Networks,'' Electronics Research Laboratory Report No. UCB/CSD 83/137, University of California, Berkeley, CA (1983).

Fujimoto, R. M., ''VLSI Communication Components for Multicomputer Networks,'' Ph. D. dissertation, Electronics Research Laboratory Report No. UCB/CSD 83/136, University of California, Berkeley, CA (1983).

Fujimoto, R. M., "The Simon Simulation and Development System," *Proceedings of the 1985 Summer Computer Simulation Conference, Chicago, Illinois* **17**(1) pp. 123-128 (July 1985).

Grappel, R. G. and Hemmenway, J. E., "A Tale of Four Microprocessors: Benchmarks Quantify Performance," *Electronic Design News,* pp. 179-265 (April 1, 1981).

Huguet, M., Lang, T., and Tamir, Y., "A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements," *Proceedings of the Sigplan '87 Symposium on Interpreters and Interpretive Techniques, St. Paul Minn.* **22**(7) pp. 14-25 (July 1987).

Jefferson, D. R., "Virtual Time," *ACM Transactions on Programming Languages and Systems* **7**(3) pp. 404-425 (July 1985).

Kumar, B. and Davidson, E. S., "Performance Evaluation of Highly Concurrent Computers by Deterministic Simulation," *Communications of the ACM* **21**(11) pp. 904-913 (Nov. 1978).

Lesk, M. E. and Schmidt, E., "Lex - A Lexical Analyzer Generator," in Unix User's Manual, Supplementary Documents, University of California, Berkeley, CA (1984).

Lipovski, G., "Hardware Description Languages: Voices from the Tower of Babel," *Computer* **10**(6) pp. 14-17 (June 1977).

May, C., "MIMIC: A Fast System/370 Simulator," *Proceedings of the Sigplan '87 Symposium on Interpreters and Interpretive Techniques, St. Paul Minn.* **22**(7) pp. 1-13 (July 1987).

Motorola, Inc., "M68000 Programmer's Reference Manual," Prentice Hall, Englewood Cliffs, New Jersey (1984).

Motorola, Inc., "MC68020 32-bit Microprocessor User's Manual," Prentice Hall, Englewood Cliffs, New Jersey (1984).

Oldehoeft, R. R., "Program Graphs and Execution Behavior," *IEEE Transactions on Software Engineering* **SE-9**(1) pp. 103-108 (Jan. 1983).

Peuto, B. L. and Shustek, L. J., "An Instruction Timing Model of CPU Performance," *Proceedings of the 4th Annual Symposium on Computer Architecture,* pp. 165-178 (March 1977).

Svobodova, L., "The Role of Emulation in Performance Measurement and Evaluation," *Symposium on Computer Performance Modeling, Measurement, and Evaluation,* pp. 126-135 (March 1976).

Tamir, Y., "Simulation and Performance Evaluation of the RISC Architecture," Electronics Research Laboratory Memorandom No. UCB/ERL M81/17, University of California, Berkeley, CA (March 1981).

University of California, Berkeley, "Unix User's Manual Reference Guide," Computer Science Division, Berkeley, CA (1984).

VanTuyl, W. H., "An Engineering View of Performance," *SHARE Computer Measurement and Evaluation Selected Papers,* pp. 816-829 (August 1973).

Table 1: Decompiled Code for Addressing Modes		
Mode	68010 Assembler	Decompiled C Code
Register Direct	d3	reg.d3.w
Register Indirect	a3@	*reg.a3.w
Postincrement	a3@+	*reg.a3.w++
Predecrement	a3@-	*--reg.a3.w
Displacement	a3(12)	*(w *)(reg.a3.b + 12)
Register Indexed	a3(12, d1:L)	*(w *)(reg.a3.b + 12 + reg.d1.l)
Immediate	#27	27
Normal (data)	LL0	*(w *)LL0
Normal (code)	L24	L24

Table 2: Code Generation Examples		
Instruction	Example Assembler	Decompiled C Code
Move longword (D1 <- D0)	movl d0,d1	reg.d1.l = reg.d0.l;
Add word (D1 <- D1 + D0)	addw d0,d1	reg.d1.w += reg.d0.w;
Unconditional Jump	jmp LE24	goto LE24;
Branch if Greater than or Equal	bge LE25	if (GE) goto LE25;
Jump to Subroutine	jsr _simple	--reg.sp.l; _simple();
Return from Subroutine	rts	++reg.sp.l; return;

Table 3: Actual and Predicted Execution Times for 10 Mhz MC68010 (Timings in Seconds)				
	simple	tour (5x5)	tour (6x6)	H
Actual	11.1	4.3	122.5	5.1
Predicted	11.2	4.2	118.7	5.3
% Deviation	0.9	2.3	-3.1	3.9

Table 4: Simulator Overhead Ratios				
host	simple	tour (5x5)	tour (6x6)	H
MC68010	7.9	6.1	6.2	7.9
VAX 8600	1.0	0.82	0.82	1.1

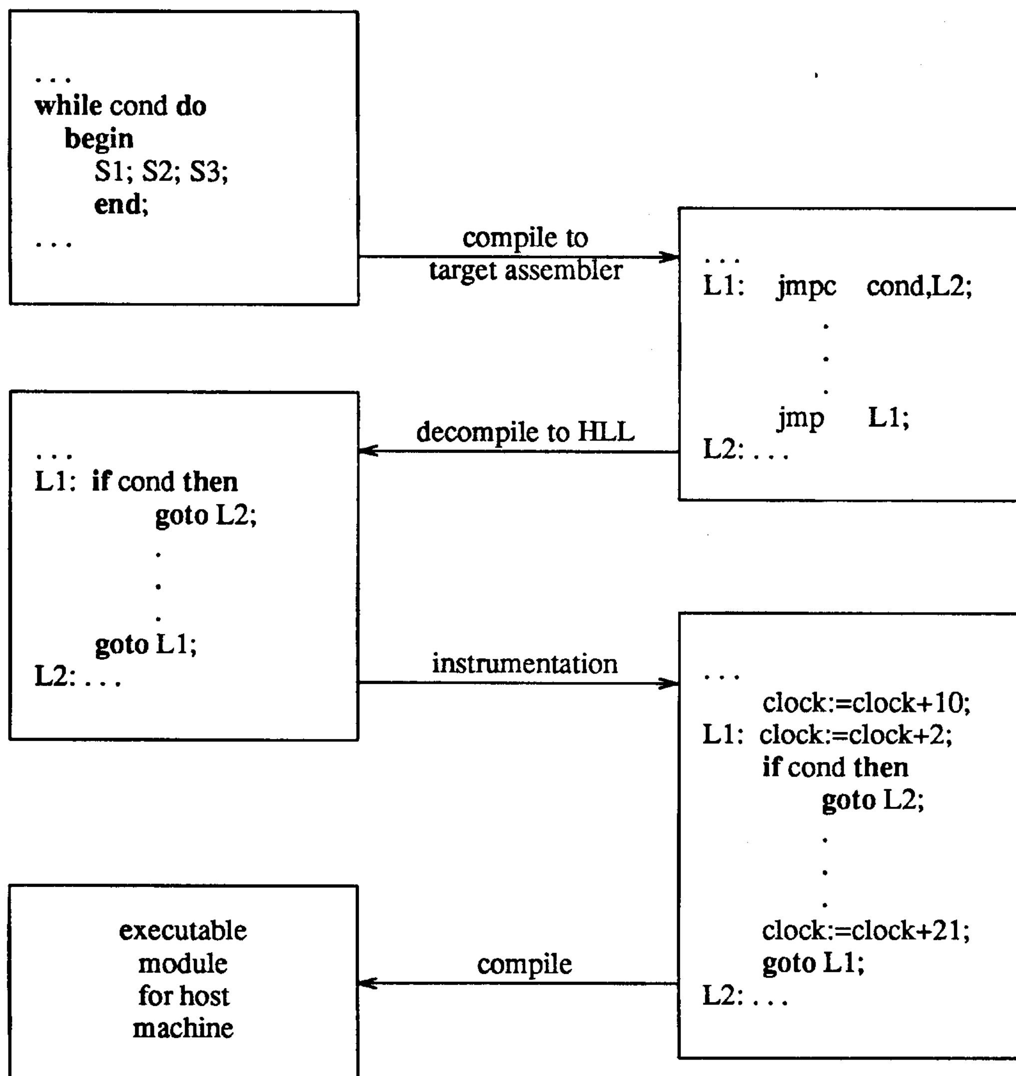

Figure 1. The proposed strategy.

```
typedef union {
        b *b;                           /* Pointer to byte */
        w *w;                           /* Pointer to word */
        l *l;                           /* Pointer to longword */
} address;

typedef union {
        unsigned char b;                /* byte register */
        unsigned short w;               /* word register */
        unsigned long l;/* longword register */
} d_reg;

typedef struct {
        address a0;                     /* Address registers 0-6 */
        address a1;
        address a2;
        address a3;
        address a4;
        address a5;
        address a6;
        address sp;                     /* Stack pointer register */
        d_reg d0;                       /* Data registers 0-7 */
        d_reg d1;
        d_reg d2;
        d_reg d3;
        d_reg d4;
        d_reg d5;
        d_reg d6;
        d_reg d7;
        d_reg t0;                       /* temporary registers */
        d_reg t1;

        /* Condition Codes: */
        long N;                         /* Negative Flag */
        long Z_bar;                     /* Zero Flag (complemented) */
        long V;                         /* Overflow Flag */
        long C;                         /* Carry Flag */
        long X;                         /* Extend Flag */
} m_state;

m_state reg; /* reg is structure variable of type m_state */
```

Figure 2. Machine State Type Definitions.