Emulation—Tool for software development

by N. F. SCHNEIDEWIND
Naval Postgraduate School
Monterey, California

INTRODUCTION

A major computer operations problem is the conversion of programs from one language to another when a replacement computer is acquired. Emulation was developed as one solution to the conversion problem. Emulation allows the machine instructions of the emulated (target) machine to be executed on the emulating (host) machine. Thus permanent program conversion is avoided. Frequently, emulated programs execute several times faster than their non-emulated counterparts. This occurs because the host is faster than the target or its microinstructions are more efficient than the target’s machine instructions, or for both reasons. Other methods of program conversion which compete with emulation are: recompilation, reassembly, translation and simulation (interpretation).

Although “translate” is frequently used as a general term to denote all types of language conversion, including assemblers and compilers, here it will be used more specifically to mean a conversion from language A to language B where A and B are not machine languages. It should be noted that recompilation, reassembly and translation effect a permanent conversion, i.e., the result of the process is a program in another language, whereas emulation and simulation employ another language, microinstructions and interpretive routines, respectively, only during the period of program execution. Another way of viewing this difference is that execution of a converted program using recompilation, reassembly or translation is a two step process—conversion followed by execution—whereas emulation and simulation are one step processes; as far as the user can discern, conversion and execution take place simultaneously in a single computer process.

Another application of program conversion techniques is in a software development organization, where a multiplicity of hardware and software systems are produced. This occurs, for example, in organizations which are developers of tactical software systems for ships and aircraft. Typically, many computers comprise a single system. The variety of computers is even greater across the many systems with which the software development organization could become involved. It would be quite expensive for the software organization to acquire all the required computers and programming languages, particularly in view of the transient nature of the development process. In addition, the computers and languages which are needed for testing purposes are frequently unavailable at test time. If the organization were to wait until all hardware and software is designed and implemented, system integration would be delayed to the point of causing delivery slippage. Thus a procedure is necessary for executing and testing programs prior to the availability of the hardware which will be used in the delivered system. In addition to the hardware being unavailable, it is possible that compilers to be used with the hardware are also unavailable at test time. This situation would require a method for producing programs which are not dependent on the use of machine language generated by the operational compiler. Here and subsequently, “operational” will refer to the software and hardware which will be used in the field after the system is delivered to the customer.

This paper will describe and evaluate different alternatives for providing a software development organization with the capability of writing and testing programs prior to the availability of the operational hardware and compilers. In this context we will describe the concept of a software production and test facility which could be used for the development and implementation of a variety of software and hardware systems in the absence of full operational capability. We will also address some of the problems which arise in the execution of this concept, such as the difficulty of faithfully representing I/O operations.

PROGRAM CONVERSION ALTERNATIVES FOR A SOFTWARE DEVELOPMENT ORGANIZATION

We now explore various alternatives for achieving a multihardware, multilingual capability for a software development organization. First, some terminology and notation are in order for the purpose of identifying the various types of languages and machines which may be involved in the operation of the software facility. The terms “target” and “host” are frequently used to refer to the “executed on” and “executed by” machines, respectively. In order to describe the software development environment referred to above, it is necessary to expand the terminology to include two types of languages and two types of machines as shown below. It should be noted that the case described below differs from the situation where an organization converts
from system A to system B. In this case A and B represent
the target language/machine and host language/machine, re­
spectively. In the software development organization case,
the target language/machine is the operational system and
the host language/machine is the system used in the software
development facility for testing and integration.

- Source language: Language used for coding the pro­
gram initially.
- Host language: Language used for executing the
source language program at the soft­
development facility.
- Host machine: Machine on which the source pro­
gram is executed in the software de­
vlopment facility.
- Target machine: Operational computer.

These relationships are shown in Figures 1 and 2. Figure
1 shows the general concept of the software development
facility wherein two major choices exist for producing soft­
ware for test and integration purposes. One method in Figure
1 involves producing host machine language which is exe­
cuted on the host machine. This method tests problem logic
in terms of host machine language. This method could be
implemented by selecting one of the following paths in Fig­
ure 2:

- 1, 2 Interpretation
- 1, 3, 9, 10 Translation
- 1, 9, 10 Compilation
- 5, 9, 10 Assembly

The second method shown in Figure 1 involves producing
target machine language which is executed in the host. This
method tests problem logic in terms of target machine lan­
guage. This method could be implemented by selecting one
of the following paths in Figure 2:

- 1, 4, 7 or 1, 6, 7 Compilation followed by Simulation
- 4, 7 Simulation with Assembly Language Input
- 6, 7 Simulation with Machine Language Input
- 4, 6, 8 Assembly followed by Emulation
- 1, 6, 8 Compilation followed by Emulation
- 6, 8 Emulation with Machine Language Input

For tactical system development, the use of either method
would be limited to testing the logic of individual programs
and the logic of integrated programs. Tests of system timing,
storage utilization and hardware dependent tests—performance
testing—would be deferred until the target system is
available. However, a substantial portion of the testing of
most systems involves the checking of sequence of opera­
tions and the correct production of outputs for given inputs.
The correct functioning of many of these operations is not
speed or hardware dependent. This approach is also com­
patible with the top-down method of system design which
emphasizes the early development of the system frame­
work—calling sequences and communication paths. Thus
logic testing performed in advance of performance testing
will facilitate the entire test process.

In Figure 2, many alternatives are listed for the sake of
completeness; some of the alternatives would be impractical
for a software development organization to implement. Al­
though many alternatives are shown in Figure 2, only one
or two would actually be used in a single organization. We
now examine each of the alternatives shown in Figure 2;
before doing this, the languages indicated in Figure 2 will be
identified.

Item 1. HLL1: A host compiler exists for this high level
language.
Item 3. HLL2: A host compiler and translator exist for
this high level language. A host compiler
does not exist for HLL1.
Item 4. AL1: Target assembly language.
Item 5. AL2: Host assembly language.
Item 6. ML1: Target machine language.
Item 9. ML2: Host machine language.

**Interpretation**

- One step process.
- Slow because of software approach used in interpreta­
tion.
- Does not execute target machine language.

**Translation**

- Two step process.
- One hundred percent translation is not feasible.
- Does not execute target machine language which is
necessary for testing equipment oriented operations.

**Compilation and Assembly**

- Two step process.
- Does not execute target machine language.
  (see Translation above)

**Simulation**

- One step process.
- Slow because of software approach used in simulation.
  About one-half the speed of the target system.¹
- Executes target functions but by using host machine
instructions to simulate target machine instructions:
  modified target execution.

**Compilation followed by Emulation**

- Two step process.
- Relatively easy to program.
- Execute microinstructions in host which carries out
target machine instructions.
- Fast execution relative to simulation (at least as fast as
target systems).¹
Figure 1—Software development concept

Figure 2—Alternatives for testing target machine programs

HLL: High Level Language
AL: Assembly Language
ML: Machine Language
Emulation with Machine Language Input

- One step process.
- Difficult to program.
- Execute microinstructions in host which carries out target machine instructions.
- Fast execution relative to simulation (at least as fast as target system).

EVALUATION OF EMULATION FOR A SOFTWARE DEVELOPMENT FACILITY

Some of the considerations involved in deciding whether to employ microprogramming (emulation) for software development are the following:

Word Length

Is the host machine microinstruction length an integral multiple or divisor of the target machine instruction?

Program Size

Is the program so large that target assembly or machine language programming would be prohibited—necessitating the use of a HLL and its compiler for producing machine language which can be emulated?

Control Store Speed

How fast is the control store of the host relative to its main memory speed? Unless the control store is several times the speed of the host’s native mode, the speed penalty will be excessive and the target machine’s timing characteristics become distorted.

Type of Control Store

Will the emulator be stored in Read Only Memory (ROM) or Writable Control Store (WCS)? The latter is clearly preferable for a software development organization. One computer could suffice for emulating a variety of target machines by changing the contents of WCS. An on-line library of target machine emulation programs, accessible to the WCS, can be envisioned. Another advantage of WCS is that it provides a user microprogramming capability.

Internal Data Code

Does the internal data code in the host differ from that used in the target (byte versus word orientation)? Does one machine use variable and the other fixed word length?

Arithmetic Operations

Do the arithmetic operations differ (decimal versus binary instructions)?

Instruction Length and Format

What are the differences in instruction length and format (size of operation code, number and length of addresses)?

Nature of I/O Operations

Are the I/O operations of target and host vastly different, such as the use of programmed I/O in one and DMA in the other? Even a common device like a card reader may cause problems if the number of columns read, data code or error processing differ between machines. Another example of the complexity of the microprogram which can ensue from attempting to emulate another common device are the control signals of a typewriter terminal—carriage return, margin release, backspace, paper advance, etc.

Microprogram Preparation Tools

How easy will it be to prepare emulation programs? Since microprogramming by conventional means requires a high degree of programming skill, lacks good self-documentation procedures and is difficult to debug, research has been under way to develop high level languages for producing microcode.

Despite some of the problems of using emulation cited above, it can provide a powerful capability for the representation of a variety of systems in a software development organization because of its fundamental property which allows the host architecture to be changed in accordance with different target machine requirements. Rather than provide n sets of hardware in a software development organization, n ROMS can be provided or, better, n emulators can be recorded in a library for loading into WCS. Another alternative is to use a combination of ROM and WCS, where the former is used as a backing store to hold emulator programs and the latter is the working storage where the current emulator program is stored; programs are swapped between ROMS and WCS. This capability would allow testing of a multiple computer target system.

Examples can be cited of the successful employment of emulation in software development. However, in order for emulation to be considered a general solution to the software development problem, its limitations must be assessed and circumvented. Suggestions for dealing with the limitations are discussed in the next section.

LIMITATIONS OF EMULATION

The following discussion addresses certain limitations and issues which are crucial to the success of an emulator oriented software development facility.
Input/output operations

As stated previously the objective of advanced testing in the software development facility is limited to checking for correct operation and sequence of various functions; testing of timing relationships would be deferred until the target hardware, including I/O, is available. In other words, advanced testing checks for correctness of logic; later testing checks for correctness of performance. Using this approach, the following emulation control of I/O could be employed during advanced testing: Upon decoding an I/O command in the emulator program, control would be transferred to the host operating system where a timer would be set to simulate the duration of the I/O operation. Control would then be returned to the emulator. At the expiration of the timer interval, a timer interrupt would occur. For a read, the operating system would load the input buffer from the test data area in memory after the interrupt and prior to returning control to the emulator. For a write, the emulator would transfer data from the test data area to the output buffer; when the write command is decoded, control is passed to the operating system as described previously. In neither case is data transferred to or from a peripheral unit. Only the movement of data in memory, to or from the I/O buffers, is simulated. The time interval which is set in the timer would be drawn from a probability distribution which is representative of the characteristics of the target I/O device.

Speed relationship between host and target machines

It is important to be able to predict the speed of the target machine based on the speed of the host emulator. The speed analysis would be separated into two parts: CPU and I/O. In the case of the CPU operations a time ratio would be established based on sample measurements of time taken in the host to emulate specified target commands and the known time required for executing the target commands. This ratio would be applied to the number of commands to be executed by the target machine in order to estimate its CPU time. Several ratios might be needed, one for each type of command. The I/O timing analysis would be treated differently. Measurement of emulator time would not apply, since the emulator is not involved in I/O operations. Input/output time would be estimated from target I/O equipment specifications and problem specifications; numbers of file accesses, records read/written; blocking factor, etc.

Validation of emulator program

There is no way to guarantee total correctness of emulator operation during advanced testing because all target equipment is not available and performance characteristics involving timing dependencies and storage utilization cannot be totally tested during this period. The best that one can do, as with any test, is to compare the observed operation with system and program specifications. In the case of advanced testing, the comparison is limited to checking of: expected outputs for given inputs and expected sequence of function execution. The time of these events and efficiency of resource utilization cannot be evaluated.

Figure of merit for fidelity of emulation

It is desirable to have a simple-to-compute figure of merit which provides an overall measure of the closeness between emulator and target machine operations. A single comprehensive measure may not be possible. If closeness is to be the criterion, speed ratio is not a good measure, because we desire the host to operate several times faster than the target machine. This would be advantageous for advanced testing because the time and cost of testing would be reduced. A more appropriate measure would account for host deviations from target operations. Examples would be the frequency of simulating I/O. A ratio could be computed of software simulated commands to total number of target commands processed in the host. This would provide a measure of lack of fidelity between emulator and target machine.

Simulation versus emulation

A natural competitor to emulation is simulation. Simulation has an advantage of being less limited in the types of target operations which can be performed. This is because, in general, software provides a flexibility for performing complex operations which is unattainable with microinstructions. An example is the operating system task management and resource allocation operations which are necessary for executing the microprogrammed application functions. However, if all target functions were performed by simulation, there would be a lack of validity in target program testing because the simulator executes host machine instructions. This amounts to executing in pseudo target machine language. In contrast, although emulated CPU instructions are not pure target machine language, the use of microinstructions for representing target machine control and execution sequences comes closer to imitating operational conditions. In addition, emulators are reported to have a performance of five to 10 times more than simulators. The best approach is to blend emulation and simulation so that the characteristics of each are used to advantage.

CONCLUSIONS

Alternatives for providing an advanced testing capability in a software development organization have been discussed. The need arises because some organizations are responsible for developing a series of systems each of which may be comprised of several software and hardware systems. It would be too expensive for the organization to acquire all the hardware and software systems necessary for test and integration. In addition many of the systems would not be available at test time. Emulation, combined with simulation, has been discussed as one solution to this problem. Its
advantages relative to compilation/assembly and translation have been examined. Although emulation is an attractive alternative, it should not be pursued unless the following conditions are satisfied:

- An easy way of producing microprograms is needed. This requires a high level language. Research has demonstrated that it is practical to use high level languages for producing efficient microcode.\(^9\)
- The host should have a writable control storage for allowing multiple emulator programs, representing a variety of target machines, to be utilized.
- Related to the preceding items is the need for a user microprogramming capability. In a study of user microprogrammable systems it was concluded that this capability can improve both the space utilization and execution time performance of a computer system.\(^10\)
- The host should not differ greatly from the target machine in terms of CPU and memory characteristics: word length, number of registers and memory size. Where differences do exist, the host should have the greater capability.

REFERENCES