AD-A051 835 STANFORD UNIV CALIF DIGITAL SYSTEMS LAB F/G 9/2
EMULATION ORIENTED SOFTWARE FIRST DEVELOPMENT.(U)
AUG 76 L W HOEVEL, W A WALLACH DAAG29-76-G-0001
UNCLASSIFIED DSL-TN-95 ARO-12958.9-M NL

| OF |
AD
A051 835

END
DATE
FILMED
4-78
DDC

EMULATION ORIENTED SOFTWARE FIRST DEVELOPMENT

by

Lee W. Hoevel and Walter A. Wallach

August 1976

Technical Note No. 95

DIGITAL SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford Electronics Laboratories
Stanford University
Stanford, CA 94305

The work described herein was supported in part by the U.S. Army Research Office-Durham under Grant DAAG-29-76-G-0001.

Digital Systems Laboratory
Stanford Electronics Laboratories
Stanford University

Technical Report No. 95

August 1976

EMULATION ORIENTED SOFTWARE FIRST DEVELOPMENT

by

Lee W. Hoevel and Walter A. Wallach

ABSTRACT

"Software First" is the design philosophy whereby applications software is developed to solve specific problems prior to the availability of applications hardware. We propose the use of an interpretive computing facility, designed around a high performance microprogrammable host machine, to support and enhance Software First in the following manner:

1) Applications programs are initially converted into a high-level intermediate text (DEL) by a straightforward "one-plus" pass compiler. The intermediate text so generated is executed interactively via a microcoded interpreter. This assures that diagnostics can be generated at the source level (e.g., "dumpless debugging"), and allows the exploitation of the host machine's inherent capabilities to attain speedy interactive response.

2) The intermediate text surrogates for applications programs, having been verified by interactive debugging, are then processed by a simple generator to produce applications-hardware compatible code. This "hard" code is then checked out on the development system by redefining the microcode running in the host machine so that it becomes an image of the projected applications hardware.

Advantages of this approach, as compared to the conventional approach, accrue from the directness with which the source language and applications hardware are mapped into the development facility. System integrety is increased by using the intermediate text produced in phase one to generate the hard machine code emulated in phase two. The phase two testing does not require the delivery of actual hardware, and can be used in the evaluation of proposed systems and the selection of a final applications system.

The work described herein was supported in part by the Army Research Office-Durham under Grant DAAG-29-76-G-0001.

Emulation Oriented Software First Development

We are interested in promoting "Software First" system design, in which development and check-out of applications software preceeds acquisition of applications hardware. While analytic simulation can provide considerable insight into the proper design of complex systems, modeling tools alone do not constitute an adequate environment for systems implementation. Cross-compilation in an alien "development environment" (i.e., using a host system other than the intended applications hardware for initial programming work) can introduce non-trivial verification problems during debugging, and may be prohibitively expensive in many applications [1-6].

In this paper, we discuss possible application of an "emulation-oriented" host system to obtain a straightforward Software First development path. It is believed that recent technological developments have brought the cost of such a host system to a currently competetive level vis-a-vis general-purpose development systems. Increased productivity of enhanced Software First design, development, debugging appears to justify further investigation in this area.

2.0 What is Software First?

"Software First" is the technique whereby applications software is developed prior to or concurrent with the procurement of applications hardware. This results in cost effective system implementation by:

1) Minimizing lag between delivery of applications equipment and final verification by providing maximum lead time for software development and verification.

2) Minimizing mis-specification of hardware requirements by providing detailed engineering data on which selection (or design) of applications equipment may be based.

Software First is most effective when applications software must be developed for hardware that is currently unavailable. The requisite applications hardware might be backordered, under construction, or even under development. Given (1) a host system in which applications software may be written and exercised (as far as practical) and (2) a method of predicting the behavior of the final applications system, however, software development need not be delayed by hardware procurement delays. In the case where hardware is being developed or has not yet been specified, software developers can provide insight as to actual system requirements (this is especially relevant to OEM and military applications [8]).

2.1 Current Software First Strategies

Existing realizations of this design methodology rely on the use of a general purpose computing facility to simplify program development. Non-trivial mappings are required to relate the central processor of such a facility to either the applications software or the applications hardware. Typically, a special "debug" compiler is used to translate applications programs into the native language of the central processing unit within the development facility. This compiler inserts auxilliary machine instructions not directly related to the semantics of the original applications program that will evoke an enveloping "run-time monitor" to perform "trace" and "debug" functions. The output of run-time traces is usually processed by an analytical modeling program to simulate the behavior of the prospective applications hardware and provide feedback during the design process [5].

Once the "debug" surrogate for an applications program is verified functionally correct, the original source program is translated by a second "production" compiler into an (optimized) program in a language accepted by the intended applications hardware. Output of the production compiler must subsequently be checked for correctness on a true applications system, after the appropriate hardware has been delivered [see figure 2.1]. Obtaining accurate instruction/data traces requires either the use of hardware monitors on applications systems or a

comparatively expensive simulation on the general purpose host facility [1-2].

Hence, the use of a general-purpose central processor for program development constrains the effectiveness of Software First design in at least the following ways:

1) The complexity of the "debug compiler" and run-time support required to convert a general-purpose host processor into a source language executor limits its effectiveness, transparency, and apparent performance.

2) Discrepancies between the program development facility and the actual applications hardware are significant obstacles during check-out of applications software.

3) Sampling run-time data is comparatively expensive, and may alter the normal behavior of the system.

Implicit in this discussion is the assumption that a "general purpose" facility is not easily tailored to the specific tasks of program development and debugging. In general, the central processors for such systems, as well as their software support, must be configured to handle "batch" computations efficiently. The computational nature of production runs differs radically from the text manipulation common to program entry and compilation.

3.0 Interpretive Software First Strategy

Emulation is the implementation of an Image machine by mapping the states of this Image machine into substates of a given Host machine, then programming the Host machine to perform state transitions over this substate space as required by the architecture of the image machine. Emulation offers a new approach to Software First development through a two phase design procedure. The first phase will accomplish development and verification of higher level language source programs through the emulation of a "virtual" machine that directly executes source text [4]. The second phase will allow the checkout of actual applications code through the emulation of the applications hardware on the development system [see figure 3.1].

During phase one, applications source text is transliterated into a high-level intermediate text directly reflecting source language semantics by a straightforward one-plus pass compiler. The intermediate text so generated is then executed interactively via a microcoded interpreter. Trace and debug functions are embedded in the interpreter, rather than in the intermediate text itself. This assures that diagnostics can be generated at the source level (e.g., inherent "dumpless debugging"), and allows the exploitation of the host machine's (assumed) interpretive capabilities to attain speedy interactive response.

The intermediate text surrogates for applications text, having been verified by interactive debugging, are subsequently processed by a specific transliterator to produce code compatible with the intended applications hardware. This "hard code" is then checked-out on the development system by redefining the microcode running in the host machine so that it becomes an image of the applications system. By emulating the actual applications environment, it is possible to determine the performance of the combined applications software/hardware system, as well as predict the effect of changes in architecture or configuration. During phase two, a number of alternative applications systems can be evaluated, and insight into new system design(s) can be obtained with greater accuracy and at less cost than on a general purpose system.

3.1 Advantages of the Interpretive Approach

The key advantages of this approach, as compared to conventional program development, accrue from the directness with which the source language and applicatons hardware are mapped into the development facility. System integrity is increased by generating a relatively high level "Directly Executed Language" (DEL code) text in phase one for the initial functional check-out of applications software. Debugging need occur only at the source level in phase one, and the "transliterator" required to generate the hard machine code emulated in phase two is of

minimal complexity. This also improves cost-effectiveness by reducing compilation overhead in both phases.

"In environment" phase two debugging, which does not require delivery of actual applications hardware, is also an important contribution to the Software First design strategy. The same types of software probes inserted into the phase one interpreter for debug purposes may be inserted into the phase two emulator to obtain detailed memory-utilization data. Such data is valuable input for program/data placement algorithms, determination of applicatons peripheral requirements, etc. Firmware probes are also useful in "pseudo real-time" emulation, in which applications system timing is formulated by updating an internal counter once each interpretation cycle. Such figures are useful in selecting appropriate applications hardware; this type of analysis is far more difficult and time-consuming to perform accurately using traditional program development tools.

4.0 Applicability

An interpretive computing facility can support Software First design methodologies in a number of ways, depending on the type of product being developed. Figure 4.1 outlines some software/hardware advantages offered by such a soft development facility for two general markets of application. OEM/Military users include manufacturers of special purpose and process-control systems, while Commercial users include general

purpose computing machinery manufactures and software developers.

4.1 OEM/Military

Consider a typical OEM/Military contractor who intends to use a mini- or micro- processor as a real-time control device. Undoubtedly, weight and power constraints dictate that the final system be as efficient as possible in both time and space. Using Software First on an emulation-oriented development system, the contractor can start by developing source-level code to control the application process before having to evaluate various alternative hardware systems, including off-the-shelf as well as new designs. The effect of architectural changes can be measured quickly and acccuately. An actual production system can be configured and evaluated before hardware is delivered (or ordered) to ensure program verification and hardware suitability.

4.2 Commercial

Typical general purpose computers are not designed for compilation and program verification. The use of an interpreter in program development offers a simpler verification procedure in that the compiler is far simpler than a compiler used on a general purpose machine, and, since the interpreted text is a direct representation of source text, program tracing and debugging is greatly simplified.

Once a program has been verified using an interpretive facility, the directly executed text can be used to drive code generators for any number of applications systems. In this way, software portability is enhanced, compiler complexity is distributed over several development phases (reducing computing costs, since only the first pass must be repeated each time source text is modified), and shops utilizing multiple computing resources can generate code for any and all of their machines. Compilation load is reduced, increasing through put of the general prupose machines.

Manufacturers of general purpose computing machinery can use such a facility to investigate the usefullness of thier proposed designs and tune processor/peripheral configurations. Software can be executed both directly and in the machine code of a proposed machine to determine the efficiency of the new machine.

4.3 Interpretive Computing Facilities

Figure 4.2 outlines the needs of various interests and market applications utilizing Software First. All areas require an Interpretive Engine - a dynamically microprogrammed processor which is easily microcoded. In addition, some form of I/O control processor such as a programmable terminal, is required to provide interactive capability.

A commercial software interest might also employ code generating engines for a number of hardware systems, including general-purpose systems used for normal simulation processing, in-house accounting, etc. Code generation and DEL interpretation are easily isolated functions; by extracting these functions from an enveloping general-purpose operating system/host, the complexity of these systems can be reduced. In installations where compilation occurs with almost the same frequency as production execution, emulation oriented processors capable of efficiently interpreting "Compiler-DEL's" would certainly be cost-effective "front end" machines for a powerful central processor.

Hardware developers would require a variety of perpheral devices to allow the emulation of configured systems in "real" production environments. The interpretive engine can be included in a general purpose facility, where I/O requests are mapped through the larger machine, utilizing its peripheral devices and main memory. In this way, channel programs can be interpreted by the general purpose machine concurrent with target code execution by the interpreter with little performance degradation.

Several such interpretive computing facilities are currently under development. The Stanford Emulation Lab is configured around EMMY, a vertically organized dynamically microprogrammed processor [7,9]. We are presently doing research in languages for direct execution, inter-architectural comparison through

emulation, as well as software first techniques. TRW is assembling facilities around the Nanodata QM-1 and QM-2, which feature writable control store, two level microprogramming (vertical microinstructions interpreted by horizontal "nanoinstructions"), and a flexible bus structure. Seven CDC 5600 series micro-processors have been tied together in a "multi-microprocessor" Emulation Laboratory at the United States Army Electronics Command, Ft. Monmouth, N.J. [8]. The University of Maryland is also establishing an emulation and direct execution facility built around the Burroughs B-1700.

References

1. Balzer, R.M., An Overview of the ISPL Computer System Design, CACM February, 1973, pp. 117-122.

2. Chu, Y., Methodology for Software Engineering, Maryland Univ. Computer Science Center, TR-256, 1973.

3. Graham, Robert M., Clancy, Gerald, J., and De Vaney, David, A Software Design and Evaluation System, CACM February 1973, pp. 110-116.

4. Hoevel, L.W., Languages for Direct Execution, Proceedings of the 7th Annual Workshop on Microprogramming, (ACM/SIGMICRO), Palo Alto, California, September 1974, pp. 307-16.

5. Lee, J.B., A Survey of Programming Methodologies, Texas Univ. Computer Science Dept. TR-047, 1975.

6. Levy, J.V., A Simulation Package for Computer Design, Stanford Univ. SLAC Computation Group, CGTM-113, 1971.

7. Neuhauser, C., An Emulation Oriented, Dynamic Microprogrammed Processor (Version III), Stanford Univ., Stanford Electronics Lab, Digital Systems Lab TN-65, 1975.

8. Mattson, Roy, The Microprogrammable Multi-Process (MMP) System for Simultaneous Emulation of Interoperating Computer Systems, Proceedings of the 7th Annual Workshop on Microprogramming, (ACM/SIGMICRO), Palo Alto, California, September 1974, pp. 290-96.

9. Flynn, M.J., and McClure, R.M.,An Integrated Facility for Emulation Research, presented at the USA-Japan Computer Conference, August 1975.

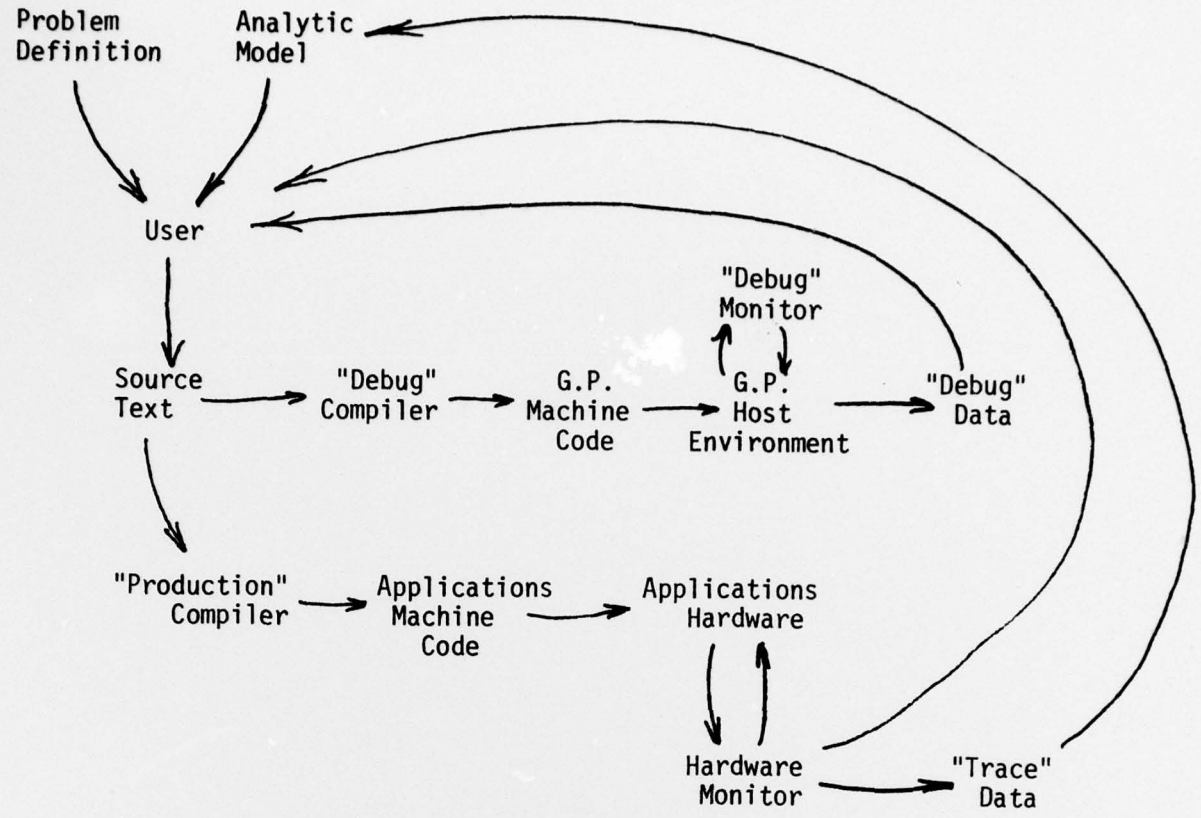

Figure 2.1 Traditional Software First

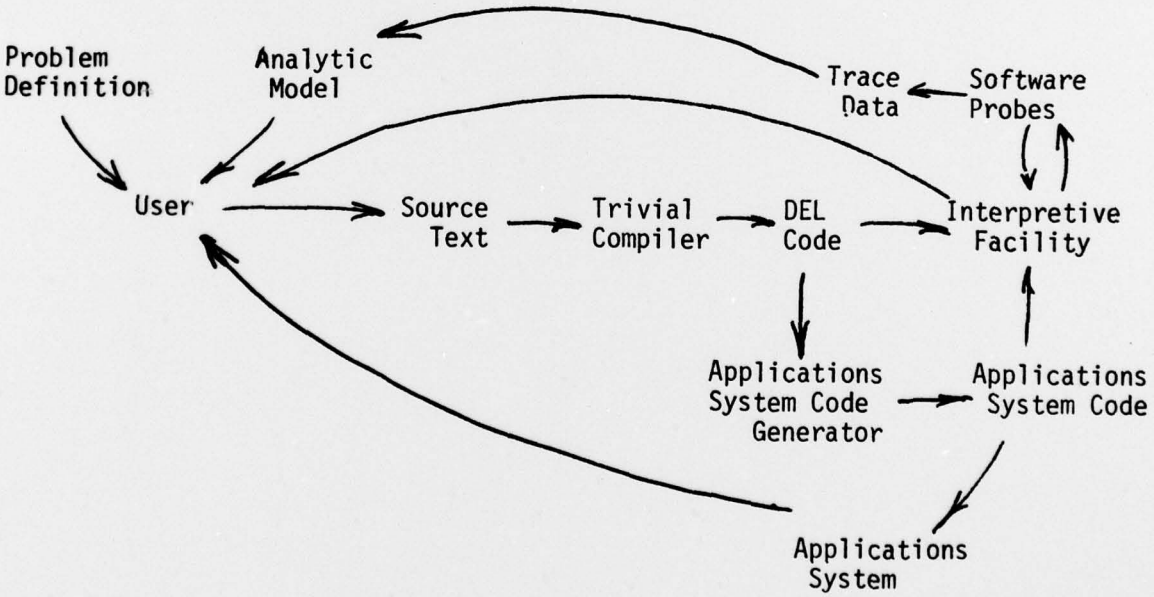

Figure 3.1 Interpretive Software First

Market Application Area

Design Interest	OEM/Military	Commercial
Software	Verification (independent of hardware) Lead Time Portability Component Simplification	Verification (independent of hardware) Lead Time Portability Multimachine Compilers DEL's
Hardware	Hardware Tuning (Configured Emulation) Hardware Procurement Hardware Design	Hardware Selection Hardware Design (Emulation Lab) Peripheral Selection/Tuning

Figure 4.1 Applications Areas for Interpretive Software First

Market Application Area

Design Interest	OEM/Military	Commercial
Software	Interpretive Engine Writable Control Store Interactive Capability General Purpose Computing Facility	Interpretive Engine Writable Control Store Interactive Capability Various Code Generators General Purpose Computing Facility
Hardware	Interpretive Engine Writable Control Store Interactive Capability Variety of Peripherals I/O Processor or General Purpose Computing Facility	Interpretive Engine Writable Control Store Interactive Capability Variety of Peripherals

Figure 4.2 Facilities Required for Interpretive Software First

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

(18) ARO 12958.9-M

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER ~~Technical Note # 95~~ 12958.9-M	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER (14) DSL-TN-95	
4. TITLE *(and Subtitle)* (6) EMULATION ORIENTED SOFTWARE FIRST DEVELOPMENT.	5. TYPE OF REPORT & PERIOD COVERED (9) Technical Note
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Lee W. Hoevel and Walter A. Wallach	8. CONTRACT OR GRANT NUMBER(s) (15) DAAG-29-76-G-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Digital Systems Laboratory Stanford Electronics Laboratories Stanford University, Stanford, CA 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 18p.
11. CONTROLLING OFFICE NAME AND ADDRESS (11) U.S. Army Research Office-Durham	12. REPORT DATE August 1976 13. NUMBER OF PAGES 14
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)* 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

"Software First" is the design philosophy whereby applications software is developed to solve specific problems prior to the availability of applications hardware. We propose the use of an interpretive computing facility, designed around a high performance microprogrammable host machine, to support and enhance Software First in the following manner:

(1) Applications programs are initially converted into a high-level intermediate text (DEL) by a straightforward "one-plus" pass compiler. The intermediate text so generated is executed interactively via a microcoded interpreter. This → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

408 071

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

assures that diagnostics can be generated at the source level (e.g., "dumpless debugging"), and allows the exploitation of the host machine's inherent capabilities to attain speedy interactive response. and

(2) The intermediate text surrogates for applications programs, having been verified by interactive debugging, are then processed by a simple generator to produce applicatons-hardware compatible code. This "hard" code is then checked out on the development system by redefining the microcode running in the host machine so that it becomes an image of the projected applications hardware.

Advantages of this approach, as compared to the conventional approach, accrue from the directness with which the source language and applications hardware are mapped into the development facility. System integrety is increased by using the intermediate text produced in phase one to generate the hard machine code emulated in phase two. The phase two texting does not require the delivery of actual hardware, and can be used in the evaluation of proposed systems and the selection of final applications system.