CMU-CS-79-118

Specification, Evaluation, and Validation of Computer Architectures Using Instruction Set Processor Descriptions

Mario R. Barbacci
William B. Dietz
Leland Szewerenko
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

13 April 1979

The development of ISPS is part of the research on the Symbolic Manipulation of Computer Descriptions effort at CMU and is sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The evaluation and specification of computer architectures was performed in support of CENTACS' MCF project under ARO contract DAAG29-77-C-0034.

Table of Contents

1. Introduction

Since early 1975, the Center for Tactical Computer Sciences (CENTACS) of the U. S. Army Electronics Command has been supporting an effort to develop a family of military computers based upon a common instruction set architecture. This **Military Computer Family (MCF)** will make available to DoD projects a series of computers which feature both an established software base and current hardware technology.

The fundamental premise of the MCF project is that software compatibility should be achieved by the adoption of an existing, proven computer architecture for the MCF, thereby minimizing the risks inherent in the design of a new computer architecture and permitting the "capture" of an existing and evolving software base. In this context, computer architecture is distinguished from implementation considerations, and is defined as the structure of a computer which a machine level programmer needs to know in order to write any time independent programs which will run correctly on the computer.

As part of this effort, Carnegie-Mellon is employing an ISP description and simulation facility for use in specifying, evaluating, and controlling these architectures. The overall goal is to provide the ability to precisely specify the selected architectures and to validate the correctness of hardware implementations.

2. The ISPS Facility

The evaluation of a candidate architecture is based on the simulation of a formal description of the instruction set. This description, written in ISP [Bell71, Barbacci77], is compiled into an intermediate code which is then interpreted by the ISP simulator. During the interpretation phase, the simulator collects statistics on the activities of the registers and functional units declared in the ISP description. These statistics are then processed (off-line) and used to derive the architecture parameters used to compare and rank the architectures.

Figure 2-1 depicts the process followed from the writting of the ISP description, the creation of test programs, the simulation of the machine executing the test programs, and finally, the collection of statistics.

The process starts with the creation of an ISP description, as shown at the top of the figure. The ISP compiler verifies the syntactic correctness of the description and generates code for an artificial machine. This machine is dubbed the Register Transfer Machine (RTM) and its order code was selected to suit the syntax and semantics of ISP (e.g. there is one RTM operation for each ISP data or control operation). The ISP simulator is simply a software implementation of the RTM machine.

The RTM code appears as a set of tables to be interpreted by the simulator. For convenience, it is generated as a PDP-10 Assembly Language programs, but it must be remembered that it does not contain PDP-10 code, only binary data. Using the standard PDP-10 operating system utilities, the RTM code can be merged with the simulator proper. The result is a PDP-10 program, a simulator for a given architecture.

The simulator contains an interactive command language interpreter which allows the user a fine control over the simulation and data collection. Thus, a user can arbitrarily start, stop, trace, and count events during the execution of the simulator.

A typical simulation session involves the loading of programs and data in the simulated memory and registers of the architecture. This initialization phase is followed by the setting of tracing flags and breakpoints (optional). Finally, the user can issue a command to start the simulation at some point in the ISP description, not necessarily the instruction interpretation cycle.

The latter aspect requires some explanation. It is clear that from the point of view of the programmer, the machine fetches, decodes, and executes instructions in a loop. From the point of view of the architect however, the operation of the machine is more complicated and there are many procedures describing other aspects of the machine (e.g. virtual memory translation, interrupt and trap detection, condition code setting, etc.) During the development

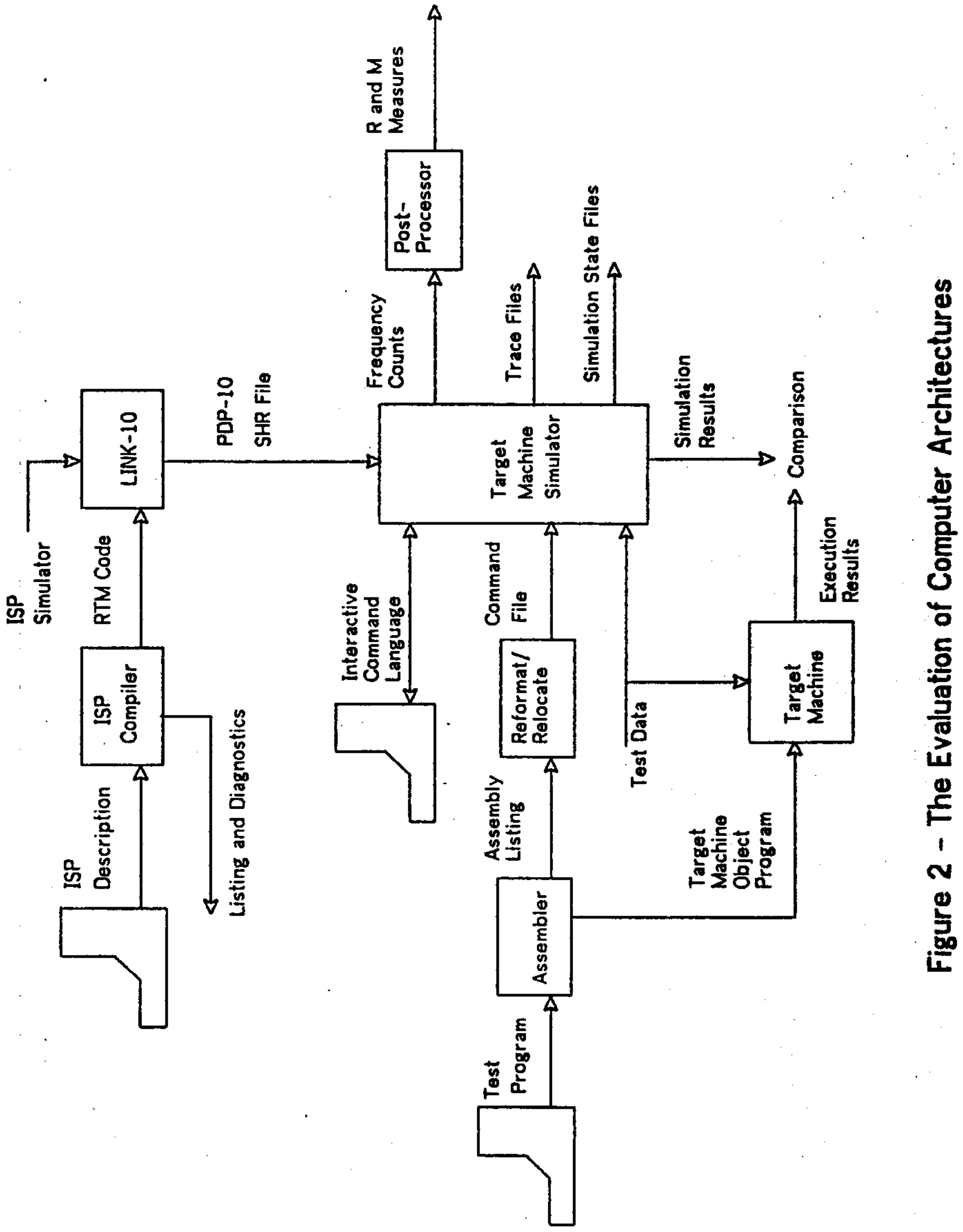

Figure 2-1: The Evaluation of Computer Architectures

of the ISP description, one might wish to conduct experiments without having to wait for the completion of the description. The ISP simulator allows the ISP writer to start the simulation at any arbitrary point in the description, provided of course, that the proper values have been stored in the machine registers and the main memory. By allowing this variable level of detail, the simulator is a facility that can be used by both programmers and machine architects alike.

The simulator allows the user to specify command files which can be created off-line and executed during the simulation session as if the user had typed them directly. This facility provides the mechanisms to load large programs and is also the means for interfacing the simulator to compilers and cross-assemblers capable of translating programs into the simulator command language. Alternatively, the state of the simulation can be dumped as a command file for later use thus allowing the user to reinitialize the simulator and continue from a previous session.

When a program runs to completion, the simulator returns to the user for additional commands. Typically, one would examine the contents of selected memory locations or machine registers to verify the operation of the program. By comparing these results with known or expected results, one can decide whether an error in the program or in the description exists (in Figure 2-1, this is indicated by comparing the results of the simulated program with those obtained from executing the program on the real machine.) If errors are suspected, the simulation can be restarted and by suitable setting of breakpoints and traces one can localize the source of the error. Debugging can occur at the instruction level (executing one instruction at a time), at the machine cycle (executing one ISP procedure at a time), or even at the register transfer level (executing one RTM operation at a time). The level at which debugging takes place depends on the nature of the error, the experience of the user, or even the level of confidence on portions of the machine description or the program. The important fact is that all of these activities can take place simultaneously, using the same set of simulation commands.

Throughout the execution of the RTM code, the simulator keeps count of ALL activities. These can be categorized into three classes:

1. Counting the bits read from each register or memory declared in the ISP description.

2. Counting the bits written into each register or memory declared in the ISP description.

3. Counting the number of times each procedure or labelled statement declared in the ISP description has been encountered (i.e. executed).

Under control of the user, these counters can be written onto a file for post-processing and analysis. The simulator itself does not perform any data reduction.

3. Specification of Instruction Set Processors

The specification of a computer architecture should unambiguously define all of the features of the architecture while leaving unspecified the details of the implementation. Specification of the MCF architectures in this manner is complicated by the fact that they are defined by existing machines with (more or less) incomplete principles of operation manuals. This encourages the programmer to consider the hardware as defining the architecture, and to relegate the manual to a role as a secondary reference. The behavior of a given machine is of course never ambiguous. Thus existing software may make use of "peculiarities" which are undocumented in the existing machine manuals.

The problem of specification is further complicated by the fact that there may be several different implementations (as realized by different models or simply by "engineering changes") in current use. Software generated by programmers working on different implementations may assume conflicting behavior of the architecture.

Finally, the use of the machine as the ultimate reference manual can result in undesirable over-definition of the architecture. It is frequently desirable in specifying an architecture to leave some aspects of the behavior undefined, in the sense that the programmer cannot rely on the result of some actions. Some examples of things which could be desireably left unspecified are:

1. Value and function of "reserved" bits. Most architectures contain control or status registers in which some bits are unused. Specifying such bits as unpredictable provides a means of adding new facilities at a later time. Dependance by programs on the behavior of these bits can make such an extension impossible.

2. Implementation of architectural registers. Registers visible to the programmer are sometimes implemented as memory locations or locations in the I/O space in order to achieve cost savings. If the software is made aware of this mapping, it will force all future implementations to mimic it. This may result in a greater cost in future implementations.

3. Ordering of non-primitive operations. If the exact execution order of a multi-word operation is specified, future implementations may be constrained to access in the same order, even though some other order may be desireable (in a virtual memory environment, for example).

The art of computer architecture requires carefully walking the thin line between overspecification and imprecision. The architecture specification which is produced must resolve these conflicts and ambiguities in a manner which will balance software capture with implementability.

In order to develop a vendor-independent specification of a computer architecture, the

MCF project decided to use an ISP description as a part of the formal specification. The ISPS description and the verbal description serve complimentary functions. The ISPS description is a precise software implementation of the architecture. As such, it can unambiguously define the behavior of the architecture. Further, the simulation facility allows potential implementers to actually run programs and examine the actions of the architecture. Thus, it serves a tutorial as well as a documentary purpose.

The verbal description compliments the ISP by providing both a less formal overview and a quiker reference ability. It is also particularly suited to specification of those areas of the architecture which contain undefined or ambiguous actions.

3.1 Specification Accuracy

One advantage of specifying an architecture with an ISP description is that the description can be mechanically checked for accuracy. The simulator facility provides the ability to execute programs written for the original implementation directly on the ISP description. In this way, diagnostics and various test programs written for the original machine are used to verify the accuracy of the description. [Barbacci78]

4. Evaluation of Instruction Set Processors

Selecting a computing architecture is not a well established science. The problem is particularly severe when one tries to choose an architecture meant to serve a broad range of users whose present and future requirements are poorly understood. Recent work within the Department of Defense has resulted in the development of a methodology to specify, evaluate, and select candidate computer architectures for tactical applications [Burr77, Wald77].

The ISP compiler and simulator, together with a a number of auxiliary programs (e.g. cross-assemblers, cross-compilers, data reduction routines, etc) have been used in several architecture evaluation projects [Fuller78,Dietz78].

4.1 The MCF project

Three computer architecture comparison studies have been carried out to provide data that would lead to the selection of a Computer Family Architecture (CFA) which could serve as the basis for a family of architecturally compatible computers spanning a wide range of performance levels. The first study was conducted under the direction of the Army/Navy Computer Family Architecture (CFA) Selection committee. The Selection Committee was charged with the evaluation of a set of candidate computer architectures, independently of any implementation features. Towards this end, a set of criteria were applied to screen an initial field of nine architectures and reduce it to the three most promising candidates. These candidates (IBM System/360, DEC PDP-11, and Interdata 8-32) were then subject to a more rigorous analysis to obtain a relative ranking. Based on a complete life cycle cost analysis which included architecture efficiency and software base (among other things) the committee recommended the PDP-11 as the CFA. In the second study, the CFA "elect" was compared with four military computer architectures which are now used in a number of systems (the AN/UYK-7, the AN/GYK-12, the AN/UYK-19, and the AN/UYK-20). The third study compared the PDP-11 (designated the AN/UYK-41) with three architectures which are currently being developed for use in military systems (the DG Eclipse C/330, the AN/AYK-14 and the AN/AYK-15A).

The evaluation of the candidates was based on their performance while executing a set of test programs, as well as the evaluation of their existing software base, and their life cycle cost. For the purposes of this paper, we will briefly describe how the test program evaluation phase was performed using an architectural research facility driven by ISP descriptions.

The concept of writing benchmarks or test programs is not a new idea in the field of computer performance evaluation. The main difference in the approach used in the MCF project was the departure from the traditional measures gathered from typical computer performance studies, namely the execution speed of a test program. A computer architecture does not specify the instruction execution times and the following alternative measures were developed:

1. S -- Number of bytes used to represent a test program.
2. M -- Number of bytes transferred between primary memory and the processor during the execution of a test program.
3. R -- Number of processor cycles required within a canonical processor for the execution of the test program.

The S measure is an indication of the amount of memory needed to represent the programs running on a computer. For the same technology, two architectures that require different amounts of memory would have different costs. A small S measure is a good feature of a computer architecture.

The M measure characterizes the bandwidth of the data path between the central processor and the primary memory. For the same technology, a higher M measure implies a larger volume of information that has to be transferred, thus implying a slower instruction rate or a costlier implementation.

The R measure provides an indication of the amount of processing required of the processor to execute the program. A higher R measure implies a slower instruction rate or costlier implementation.

Techniques such as memory interleaving, cache memories, and pipelines could be used to improve the performance of a system. However, these are implementation details and are not part of the architecture. That is, two architectures with different M (or R) measures could be implemented using these and other speed-up mechanisms so that their time performance on the tests would be similar. However, the relative ranking given by their M (R) measures would not change.

The S measure is a static parameter and is easily computed by hand. The M and R measures are however, indicators of the dynamic behavior of the test programs and could not be easily and accurately computed manually. Hardware monitoring devices that could gather the data by measuring the performance of the test programs on real machines would have to be tailored to specific hardware implementations and would therefore be costly to build and of little general use. Moreover, implementation details of a particular machine could opaque or hide some of the necessary information. By constructing an ISPS description of each of the candidates and running the test programs under the resulting simulators, the execution statistics necessary to derive the M and R measures were easily obtained from the counting facilities of the simulator. By having the test programmers debug their programs on the simulator, the problem of limited (or non-existent) access to hardware implementations of the candidate architectures was eliminated.

4.2 Results of the Evaluations

The results of the three studies are displayed in Figures 4-1, 4-2, and 4-3. The numbers shown in each figure are relative to the average of the machines in that study. The log scale for the results is shown on the right of each figure and the linear scale is on the left. Since S, M, and R are measures of cost, a lower number is more desirable. The most efficient machines appear closest to the bottom of the figure. Thus, a machine with a normalized S measure of .80 on the linear scale will require only 80% of the memory used by the average of the machines tested.

The calculated confidence intervals were used to determine which machine effects were significantly different at the 95% confidence level. The dashed boxes in the figures enclose architectures whose performance differences were not significant at the 95% level. The size and placement of the boxes are such that if any other architecture's corresponding measure were to fall within the box, it too would be indistinguishable at this level of certainty.

Three of the architectures included in the third study are either similar or identical to architectures from the second study. Their relative performances, between the two studies, are consistent within the statistical accuracy of the experiment given the architectural differences involved.

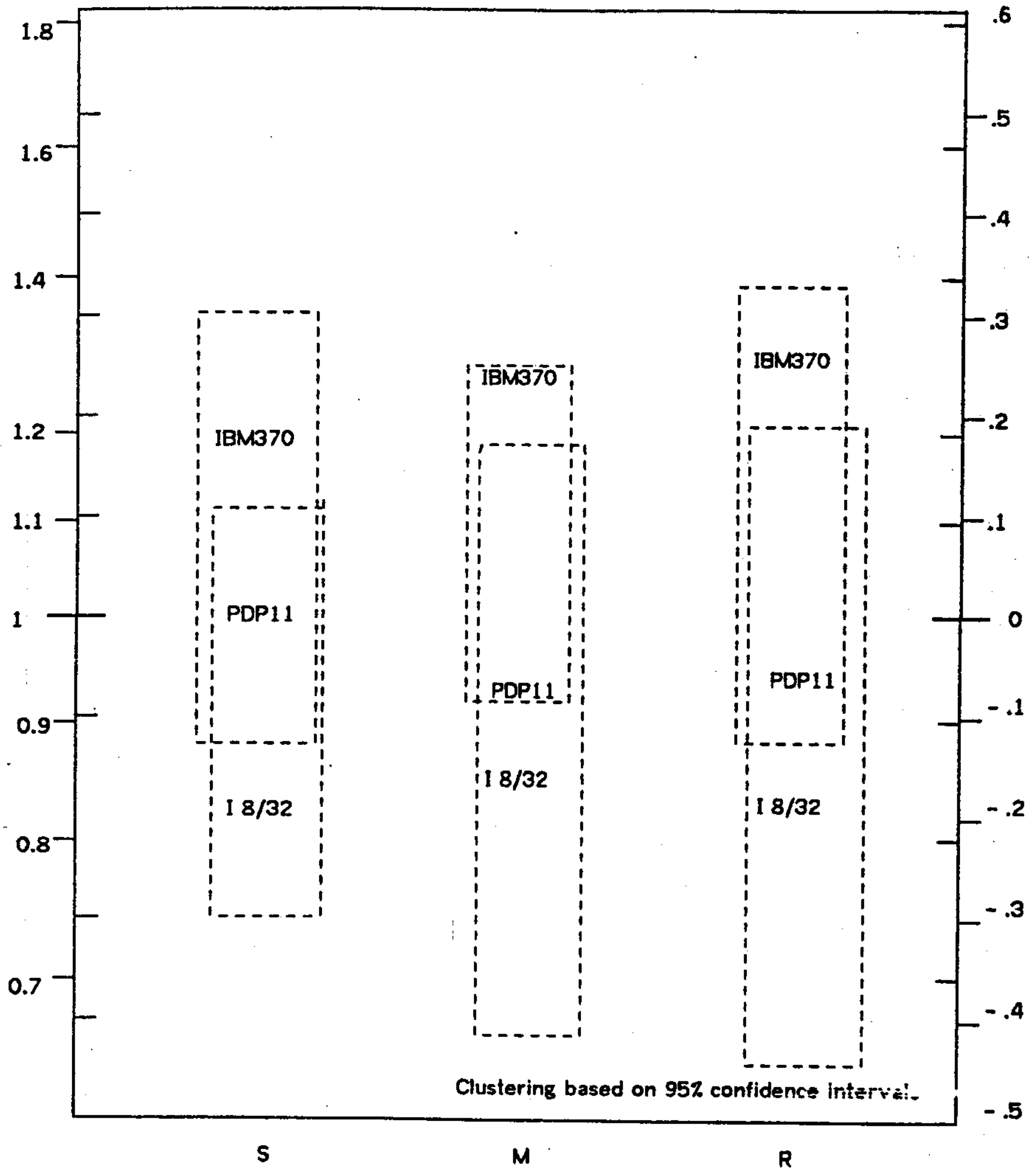

Figure 4-1: Composite Results for the First Study

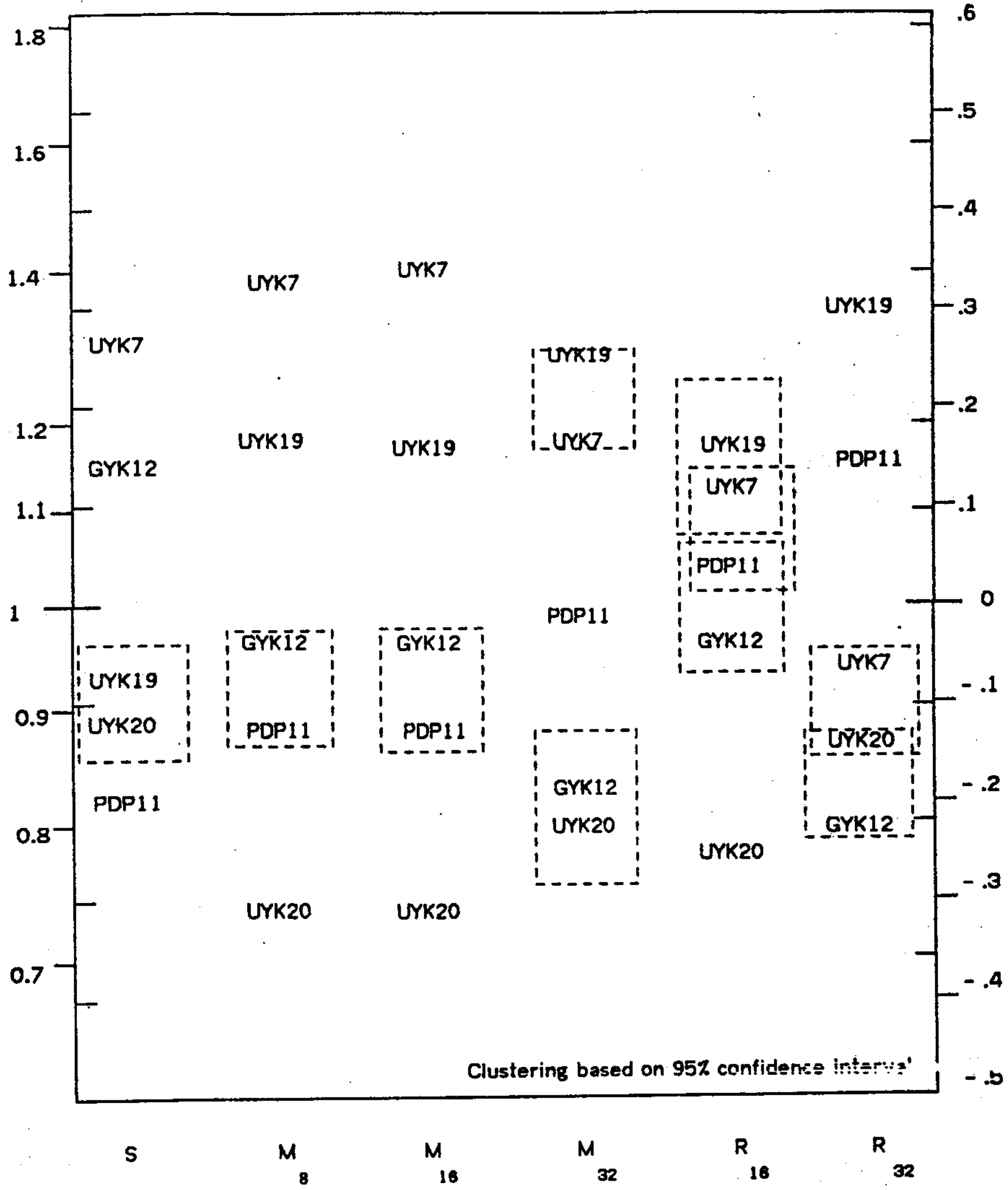

Figure 4-2: Results for the Second Study

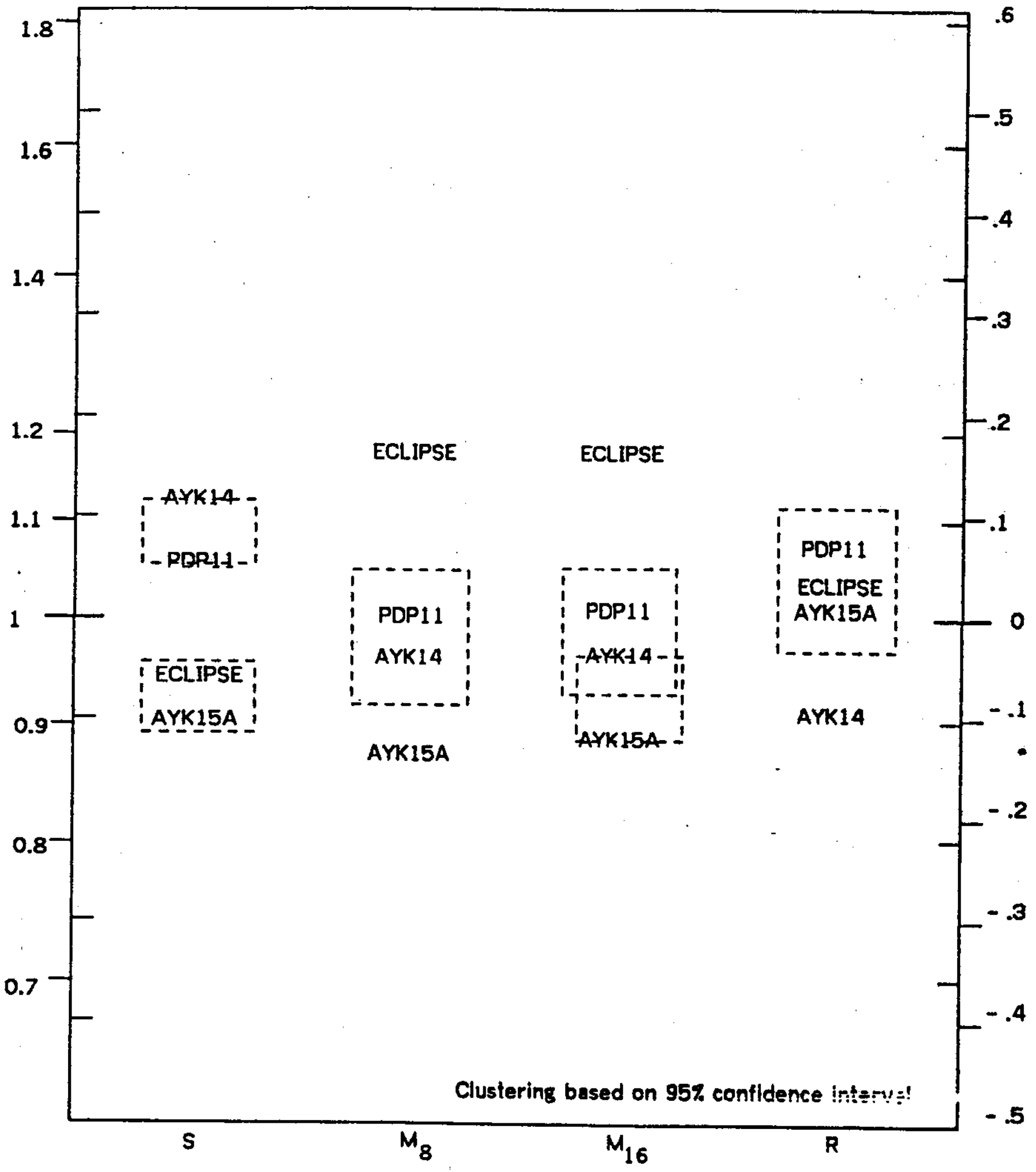

Figure 4-3: Results for the Third Study

5. Architecture validation

The implementation of an architecture (excluding build-to-print specifications which do not allow technology improvements) is very much like the coding of a program. Once the task is done the question arises "is it correct?" or "does it meet the specification?". In both cases the validation problem is in its infancy as a research question. Thus the ability to guarantee correctness is currently beyond the grasp of computer technology. Nevertheless, the grave importance of the question forces us to attempt to reach at least a reasonable certainty.

The existing software base of an established architecture provides an easy means of testing the correctness of a new implementation. Running a selected subset of programs provides a check on the ability of the new machine to perform the same tasks as the old machine. However, since running the entire software base on the new machine under all possible input conditions is impossible, the question of coverage arises. It is difficult to determine whether all architectural features have been tested. The development of programs which test the correct operation of each feature of the architecture is the most desirable course of action. Such programs provide immediate diagnosis of problems in the event of error, thereby avoiding the introduction of delays in the development cycle. The only nagging question which remains concerns the correctness and completeness of these validation programs.

It is important to note that the validation programs are complementary to the existing software test, since the latter is most likely to test the frequently used features of the architecture. The validation programs must therefore place particular emphasis on exploring the "dark corners" of the architecture, as these are least likely to be tested by the existing software.

Our approach to developing the validation programs is proceeding in parallel on two fronts: (1) automatic generation of validation programs from ISP, and (2) iterative development of hand coded validation programs.

Generating validation programs by hand is a difficult, tedious, and error-prone process. Current research at CMU indicates that it may well be possible to automatically generate validation programs directly from the formal ISP specification of the architecture [Oakley79]. By performing a symbolic execution of the ISP description, a list of all architecture features that must be present in any implementation can be compiled and used as a check list to generate the machine diagnostics that must be executed in the new implementation[1]. This approach provides better coverage than the standard machine diagnostics currently in use. The remaining task is to select sequences of instructions from those described by the ISP to exercise each of these features and differentiate between successful and unsuccessful completion of each such test.

Since the ability to automatically generate validation programs is still a subject of research, we are pursuing in parallel an iterative manual technique. In this approach, validation tests are manually coded for each architectural feature. These tests are then run under the ISP simulator. The simulator provides a check of the correct functioning of the tests and a measure of their completeness. The statistics gathering facilities of the simulator provide data as to which features have and have not been exercised by the validation program. Thus, by an iterative process of coding and testing, we are able to develop comprehensive tests which are proven to cover all features of the architecture.

[1]It should be mentioned that although Oakley's work had the generation of diagnostics as the initial goal, it was soon evident that the range of its application was much broader. For instance, we are currently interfacing the results of the symbolic executor with the input required by the Production Quality Compiler Compiler (PQCC) project [Leverett79].

6. References

[Barbacci77] M.R. Barbacci, G.E. Barnes, R.C. Cattell, and D.P. Siewiorek, "The ISPS Computer Description Language", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1977.

[Barbacci78] M.R. Barbacci and R.A. Parker, "Using Emulation to Verify Formal Machine Descriptions", COMPUTER, Vol. 11, No. 5, May 1978.

[Bell71] C.G. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Company, New York, 1971.

[Burr77] W.E. Burr, A.H. Coleman, and W.R. Smith, "Overview of the Military Computer Family Architecture Selection", AFIPS National Computer Conference, 1977.

[Dietz78] W. Dietz and L. Szewerenko, "Phase III - Comparative Evaluation of Candidate Computer Architectures", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978.

[Fuller78] S.H. Fuller, G. Mathew, and L. Szewerenko, "Phase II - Comparative Evaluation of the MCF Computer Architectures", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1978.

[Leverett79] B. Leverett, *et al.*, "An Overview of the Production Quality Compiler-Compiler Project", Technical Report, Department of Computer Science, Carnegie-Mellon University, 1979.

[Oakley79] J. Oakley, "The Symbolic Execution of Formal Machine Descriptions", PhD Thesis, Department of Computer Science, Carnegie-Mellon University, 1979.

[Wald77] B. Wald and A. Salisbury (guest editors), "Military Computer Architectures: A Look at the Alternatives", COMPUTER, Vol. 10, No. 10, October 1977.