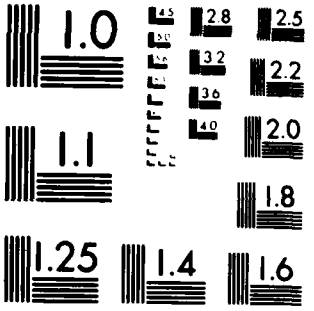
AD-A130 704 DIRECTLY EXECUTED LANGUAGES(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB M J FLYNN JUN 83 ARO-18553.4-EL DAAG29-82-K-0109 1/1

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
8-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 18553.4-EL	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Directly Executed Languages		5. TYPE OF REPORT & PERIOD COVERED: Annual Report 19 April 1982-18 April 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Michael J. Flynn		8. CONTRACT OR GRANT NUMBER(s): DAAG29-82-K-0109
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, Post Office Box 12211, Research Triangle Park, NC 27709		12. REPORT DATE: June 1983 13. NUMBER OF PAGES: 7
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE JUL 27 1983 A

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

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

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

computer architecture
directly executed languages
computer programming

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

This work compared computer architectures by measuring the execution of an identical set of high level language programs. Comparative studies are difficult and expensive as they require an environment in which all the architectures can be analyzed on a common basis. Simulation has been used, but the slow speed makes it prohibitively long to collect a significant sample. Performance measures, such as the number of instructions, reflect not only architectural differences but factors (such as compilers) not related to the architecture.

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A130764
DTIC FILE COPY

DIRECTLY EXECUTED LANGUAGES

Final Report

Prepared for

Department of the Army
U.S. Army Research Office
Research Triangle Park, NC

Contract No. DAAG29-82-K-0109

April 19, 1982 - April 18, 1983

by

Michael J. Flynn
Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

DTIC COPY INSPECTED 2

The views and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

83 07 27 063

Table of Contents

1. Architectural Analysis 1
2. Directly Executed Languages 2
3. Concurrent Execution 3
   - 3.1 Level 0 - Pipelined Execution 4
   - 3.2 Level 1 - Transparent Concurrency 4
   - 3.3 Level 2 - Machine Detectable Concurrency 4
   - 3.4 Level 3 - Algorithm Detectable Concurrency 5
4. Scientific Personnel 6
5. T.R.'s and Publications Sponsored Under Contract 7

List of Figures

Figure 3-1: Pipelined Execution 3
Figure 3-2: The Ultimate Pipeline 4
Figure 3-3: Ratios for Different Levels of Concurrency 5

1. Architectural Analysis

This work compared computer architectures by measuring the execution of an identical set of high level language programs. Comparative studies are difficult and expensive as they require an environment in which all the architectures can be analyzed on a common basis. Simulation has been used, but the slow speed makes it prohibitively long to collect a significant sample. Performance measures, such as the number of instructions, reflect not only architectural differences but factors (such as compilers) not related to the architecture.

The instruction streams of the IBM S/370, DEC PDP-11, and P-code machines were measured using a microprogrammable processor--Emmy. The measurement mechanism is embedded into the interpreter (an emulator) for the machine, and has access to all aspects of the instruction execution. The DEC VAX instruction stream was measured on a VAX 11/780 using a trace feature in the architecture. A set of FORTRAN programs was used for measurements, and reflect a scientific work load.

The analysis first studied the composition of the instruction stream. Total number of instructions executed, shows the VAX architecture to be most efficient, but measures of the activity necessary by the interpreter indicate that the S/370 representation is fastest to interpret. Memory reference behavior indicated that the 8-bit displacement used by the VAX *is very effective for local referencing, but VAX suffers in referencing* global objects. Measurements of branch behavior have shown that the PDP-11 and VAX architectures require 10-15% more branch instruction than an ideal representation of the program would indicate. This architectural defect results from the short range of conditional branch instructions.

This work analyzed the interaction between compiler optimization techniques and the instruction streams that result from optimization. Six S/370 compilers generated different representations of the test work load, and produced the data base for study of high level language behavior and architectural analysis. Optimization, while reducing the resource demands of a program, does not apply uniformly to all aspects of instruction execution. The fixed-point computation and memory reference demands are greatly reduced, but the control requirements of a program are largely unaffected. Because the absolute occurrence of control related instructions is constant, their relative frequency increases with optimization.

88 07

2. Directly Executed Languages

A natural way to make compilation as straightforward as possible is to make the execution architecture fit the high-level language. This work centers on a family of execution architectures called directly executed languages, or DELs. High-level language statements are closely represented by DEL instructions. DEL representations minimize the number of bits needed in the instruction stream for operand specification, without resorting to encodings that require knowledge of the frequency of occurrence of individual operands. A Pascal-to-DEL compiler and a DEL processor emulator are used to measure the number of instructions required to run five test programs to completion. The number is also measured for the HP 1000F, the IBM 370, Pascal P-code and the DEC Vax. The average of the ratios of the numbers for these to the number for Adept is 3.46. The number of main memory bytes read for data is 5.42 times that for Adept, and the ratio for bytes written is 14.73. These results show that it is possible to make an execution architecture suitable for a high-level language in a way that results in architectural measures that may indicate a higher speed of execution and a lower cost of implementation than some familiar architectures.

3. Concurrent Execution

The execution time of instruction can be effectively reduced by overlapping the start of the execution of one instruction with the end of the execution of another. This process of overlapping instruction execution is called *pipelining* and it is used on all "super computers". An example of an instruction stream which has been pipelined is shown in Figure 3-1.

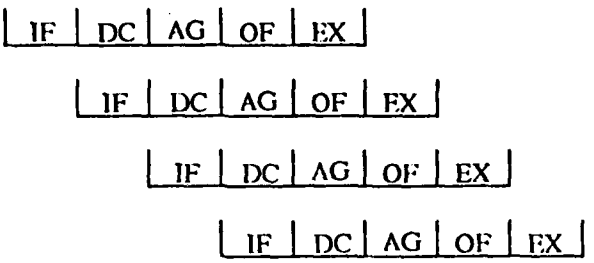


Figure 3-1: Pipelined Execution

Extending the concept of pipelining to its ultimate would generate an execution sequence where all instructions execute at the same time. But since simultaneous execution of all instructions would mean that inputs are fetched at the same time and results are produced at the same time, simultaneous execution could only be performed correctly if *no* instruction in the task required the completion of *any* other instruction to perform its function. Since this is only possible in the most trivial of cases, the correct execution of all other programs can only occur when instructions wait for other instructions to execute producing results which are needed for their correct execution. An example of such an ultimate pipeline is illustrated in Figure 3-2. Executing multiple instructions at a time will be called *concurrent* execution of instructions as opposed to parallel execution which usually refers to a single instruction stream, multiple data stream style of computation.

There are different degrees to which concurrent execution of instruction streams can take place. Using a minimal amount of hardware, a small amount of concurrency can be detected providing a modest increase in execution speed. The amount of concurrency detected in these schemes can be compared to the serial speedup of traditional machines which have instruction prefetch but little or no pipelining. Introducing more hardware increases the amount of concurrency which can be detected and subsequently the execution speed of the task. Concurrency such as this can be compared to a highly pipelined machine which allows out of order execution, multiple path exploration, and interleaved memory traffic.

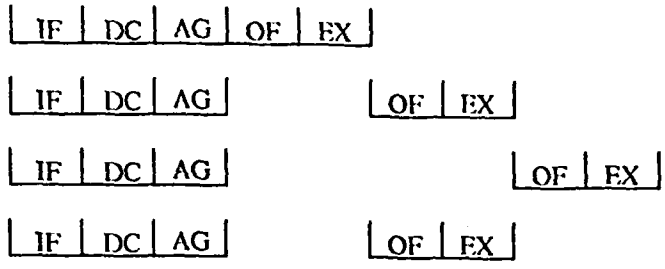


Figure 3-2: The Ultimate Pipeline

Although the degrees of concurrency detection can actually be thought of as a continuum, four distinct levels have been defined in our research. These levels are defined so that increasing level numbers increase the amount of concurrency detected with a corresponding increase in the amount of difficulty of detection and subsequently the amount of hardware needed for implementation. They include:

3.1 Level 0 - Pipelined Execution

The main feature of Level 0 concurrency detection is that there is no concurrency actually detected at all. Level 0 concurrency is characterized by the fastest program execution possible with the single restriction that an average maximum of one instruction is executed in each machine cycle.

3.2 Level 1 - Transparent Concurrency

Level 1 concurrency is characterized by a direct examination and exploitation of the concurrency which existed in the original task. In this level of concurrency, only the concurrency which was explicitly apparent in the task is detected.

3.3 Level 2 - Machine Detectable Concurrency

Level 2 concurrency is marked by taking advantage of all the concurrency which a machine can detect without resorting to algorithm recoding or code manipulation.

3.4 Level 3 - Algorithm Detectable Concurrency

Level 3 concurrency is characterized by analyzing the job to be done and restructuring it in hardware and software to produce a representation which will execute in a minimal amount of time using a minimum number of steps. More precisely, Level 3 concurrency is all concurrency which can be detected algorithmically. This detection process can occur at both the hardware and software level.

We have completed a fairly extensive study of the speedup potential at each of the above levels. One such study[1] found the concurrency available in a sample DEL program, shown in 3-3. Note: level 3a is compile time detection only while level 3b represents both compile and runtime detection.

	Level of Concurrency				
	0	1	2	3a	3b
Dynamic Number of Instructions	435	435	435	390	390
Machine Cycles to Execute Task	435	306	180	121	93
Speedup	1.00	1.42	2.42	3.22	4.19

Figure 3-3: Ratios for Different Levels of Concurrency

[1] *Performance Evaluation of the Execution Aspects of Computer Architectures*, by M. Flynn, J. Huck, S. Wakefield and R. Wedig, International Workshop on High-Level Language Computer Architecture, Ft. Lauderdale, FL, December 1932.

4. Scientific Personnel

Michael J. Flynn, Principal Investigator
Professor, Department of Electrical Engineering
Stanford University

Jerome Huck
Research Assistant
Stanford University
(Received Ph.D. degree in Electrical Enginering, March 1983)

Chad Mitchell
Research Assistant
Stanford University

Johannes Mulder
Research Assistant
Stanford University

Evan Tick
Research Assistant
Stanford University

Scott Wakefield
Research Assistant
Stanford University
(Received Ph.D. degree in Electrical Engineering, December 1982)

Robert Wedig
Research Assistant
Stanford University
(Received Ph.D. degree in Electrical Engineering, June 1982)

5. T.R.'s and Publications Sponsored Under Contract

"Detection of Concurrency in Directly Executed Language Instruction Streams", by Robert G. Wedig, Ph.D. Thesis, Stanford University, June 1982. (Also available as Technical Report No. 238, Computer Systems Laboratory, Stanford University, Stanford, CA.)

"Performance Evaluation of the Execution Aspects of Computer Architectures", by M. Flynn, J. Huck, S. Wakefield, and R. Wedig, Proceedings of the International Workshop on High Level Language Computer Architecture, December 1-3, 1982, Ft. Lauderdale, Fla.

"Studies in Execution Architectures", by Scott Wakefield, Ph.D. thesis, Stanford University, December 1982. (also available as Technical Report No. 237, Compter Systems Laboratory, Stanford University, Stanford, CA.)

"A Local Variable Storage Mechanism", by Scott Wakefield, COMPCON Proceedings March 1983, San Francisco, CA.

"Execution Architecture: The DELtran Experiment", by M. Flynn and L. Hoevel, IEEE Transactions on Computers, Vol. C-32, No.2, February 1983 (ISSN 0018-9340), pp. 156-175.

"Comparative Evaluation of Computer Architectures", by Jerome Huck, Ph.D. thesis, Stanford University, March 1982.

"Comparative Evaluation of Computer Architectures", by J. Huck and M. Flynn, IFIPS Proceedings, Paris, September 1983.

END

DATE
FILMED

8 — 83

DTIC