UNOS 1480

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title: **Relative CPU Performance on C Language Programs** Date: **September 28, 1979**

Other Keywords: TM: **79-3624-5**

Author(s)	Location	Extension	Charging Case: 49343-11
Jerome Feder	**MH 2D-230**	**2461**	Filing Case: **40125-004**

ABSTRACT

The C language has been receiving increasingly heavy use throughout the Bell System. This memorandum compares a variety of CPUs with respect to execution speed and object code size using seven benchmark programs written in C. The CPUs span the range from microprocessors to large mainframe computers and include the BTL MAC 8, BTL 3B-20, Intel 8086, Interdata 8/32, Tandem 16, Data General Eclipse M600, Honeywell 6080, Digital Equipment Corporation (DEC) VAX 11/780, Univac 1100/81, IBM 370/168, IBM 3033, Amdahl V7, and most of the DEC PDP-11 models. The comparisons necessarily include the compilers for the respective machines. The benchmarks are intended to be representative of C language usage by telephone company Operations Support Systems (OSS) and by the UNIX* operating system; they contain very little arithmetic in general, and no floating-point arithmetic. In particular cases, the effects of compiler options, alternate compilers, memory caches, memory management and clock speed are examined. Properties of programs which affect execution speed are discussed.

* Unix is a trademark of Bell Laboratories.

Pages Text: 12	Other: 13	Total: 25
No. Figures: 5	No. Tables: 18	No. Refs.: 8

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

+ANSELMO,DONALD R
ARTHURS,E
+BALDWIN,GEORGE L
+BECKETT,J T
BELL,P E
+BERRYMAN,R D
+BRANDT,RICHARD B
+CANADAY,RUDD H
+CANDELORA,RAYMOND M
CANNATA,PHILIP E
CAREY,F E,JR
+CHELLIS,ALICIA L
CHE,HEP-DAW
CHRISTENSON-BECKER,MARY
CLARK,DAVID L
COHEN,AARON S
+CROSS,AARON C,JR
+CROWLEY,T H
+DANIELSON,W E
DE GRAAF,D A
DEAN,JEFFREY S
DOLOTTA,T A
+DONOHUE,B P,3RD
DWORAK,F S
+EDWARDS,ROGER
+EHLINGER,JAMES C
EISEN,STEVEN R
+EVERETT,W W
FARRELL,JAMES W
FAULKNER,ROGER A
FEDER,J
+FELDER,D E
+FELKER,J H
FIORE,RHODA J
+FLAVIN,M A
+FLETCHER,R C
FRITZ,THOMAS E
+GIBSON,H T,JR
+GRAMPP,F T
+GROSS,ARTHUR G
+GUIDI,PIER V
HAGGERTY,JOSEPH P
HAIGHT,R C
HALEY,DEBRA T
+HALL,ANDREW D,JR
+HEIZER,ISAAC J
+HOFFNER,C W,2ND
HORNER,GEORGE G

COMPLETE MEMORANDUM TO

IMAGNA,C P
JIMENEZ,DELMA E
+JOHNSON,RANDOLPH W
+JOHNSON,STEPHEN C
KIRSLIS,PETER A
KOENIG,ANDREW R
+KORN,DAVID G
KOWALSKI,THADDEUS J
KYNDU,SUKHAMAY
+LARSON,ALLEN L
+LAYTON,H J,JR
+LEONARD,D H
+LEONARD,D J
LOIKITS,E A
LONG,P F
+LUCKY,R W
LUDERER,GOTTFRIED W R
+MAGEE,FRANCIS R,JR
+MANOCHIO,L J
MARANZANO,J F
+MATHEWS,M V
MATHIOT,MARTIN J
+MC LAUGHLIN,GEORGE E
MC PHERSON,A P
MEE,C,III
+MENDELL,HARRY B
METZLER,HELEN M
+MILTON,DONN B
MOLLENAUER,JAMES F
+MORGAN,SAMUEL P
+MORRIS,ROBERT J T
+MULLINS,JOE H
NELSON,NILS-PETER
OLSSON,S BERT
+ORCHARD,R A
PAO,T W
PEREZ,CATHERINE D
+PINSON,ELLIOT N
PUTTRESS,JOHN J
RALEIGH,T M
REICHERT,ROY S
ROSENFELD,PETER E
ROSENTHAL,VICKI H
ROSLER,LAWRENCE
RUGABER,JON S
+SHEEHAN,JOHN J
SHULMAN,SHERYL
+SIMONE,C F
+SMITH,DALE W
+SMITH,W B
SNYDER,GINI A
+SNYDER,JAMES H
+SO,HON HING
+SPENCER,A E,JR
+STAEHLER,R E
+STALLMAN,F B,JR
STUCK,B W
+STURMAN,JOEL N
TAGUE,BERKLEY A
+TAMMARO,ENN

COMPLETE MEMORANDUM TO

+THOMAS,D G
+THOMPSON,K
+TUOMENOKSA,LEE
+VAN ORNUM,J H
VOGEL,GERALD C
+VYSSOTSKY,V A
WAGNER,MARY R
+WALDHUTER,JEFFREY S
WEHR,LARRY A
+WEI,MARTIN C
WELSCH,RICHARD J
+WHITE,RALPH C,JR
+YALE,WILSON W
+YEH,YERN
122 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ABATEMARCO,TERESA M
ABATE,JOSEPH
ACKERMAN,A FRANK
ACKERMAN,J T
AHO,ALFRED V
AHRENS,RAINER B
ALBAGLI,V R
ALCALAY,D
ALEXIS,A D,JR
ALLEN,JAMES R
ALLISON,C E,JR
AMITAY,N
ANDERSON,KATHRYN J
ANDERSON,L J
ANDERSON,MILTON M
ANDERSON,NILS H
ANDERSON,R E
ANDERSON,R R
ANDREWS,W J
ANTOLICK,DAVID R
ARCHER,RUSSELL E,JR
ARNOLD,GEORGE W
ARNOLD,JAMES Q
ARNOLD,PHYLLIS A
ARNOLD,THOMAS P
ARVIDSON,W P
ASMUTH,RICHARD L
ASTHANA,ABHAYA
ATAL,BISHNU S
ATKINS,JOAN
AXELSON,A L
BABECKI,GLENN R
BAGLEY,JOHN D

COVER SHEET ONLY TO

BAILEY,CATHERINE T
BAKER,MITCHELL B
BALDINI,J J
BALENSON,CHRISTINE M
BALLANCE,ROBERT A
BALL,MARSHALL
BARNHARDT,KARL R
BAROFSKY,ALLEN
BARON,ROBERT V
BARRESE,A L
BATTAGLIA,FRANCES
BATTISTA,RALPH N
BAUER,BARBARA T
BAUER,H C
BAUGH,C R
BAXTER,LESLIE A
+BAYER,D L
BECKER,JACOB I
BECKER,RICHARD A
BECKER,SHELDON T
BECK,ROBERT J
BEIGHLEY,KEITH A
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BENNETT,RICHARD L
BENOWITZ,P
BENSING,JAMES EDWARD
+BERENBAUM,ALAN
BERGLAND,G D
BERKEY,M A
BERK,DONALD A
BERNOSKE,BEVERLY G
BERNSTEIN,L
BERNSTEIN,PAULA R
BHATIA,RAJIV
BIANCHI,M H
BICKFORD,NEIL B
BILASH,TIMOTHY D
BILLEK,F L
BILOWOS,R M
BIREN,IRMA B
BISHOP,J DANIEL
BITTNER,B B
BITTRICH,MARY E
BLACKWOOD,N T
BLAKE,GARY D
BLAZIER,S D
BLEIER,JOSEF
BLINN,J C
BLOSSER,PATRICK A
+BLUMER,THOMAS P
BLUM,MARION
BOCKUS,ROBERT J
BOCK,NANCY E
BODDIE,JAMES R
BODEN,F J
BOEHM,KIM R
BOGART,F J
BOGART,THOMAS G

COVER SHEET ONLY TO

+BOIVIE,RICHARD H
BONANNI,L E
BOND,P C
BORG,KEVIN E
BORISON,ELLEN A
BOSE,DEBASISH
BOSTON,RONALD E
BOSWELL,PAULA S
BOURNE,STEPHEN R
BOWIE,LEON A
BOWLES,PAUL W
BOWYER,L RAY
BOYCE,K J
BOYCE,W M
BOYER,PHYLLIS J
BRADFORD,EDWARD G
BRADLEY,M HELEN
BRANDAUER,C M
BRANTLEY,JOAN T
BRAUN,DAVID A
BREILAND,JOHN R
BRENSKI,EDWIN P
BRESLER,RENEE A
BRITT,WARREN D
BRONSTEIN,N
BROOKS,CATHERINE ANN
BROSS,JEFFREY D
BROYMAN,INNA
BROWN,ELLINGTON L
BROWN,LAURENCE MC FEE
BROWN,MARK S
BROWN,STUART G
BROWN,W R
BROWN,W STANLEY
BROZ,G C
BUCK,I D
BURGESS,DIANE MAHER
BURGESS,JOHN T,JR
BURG,F M
BURKE,R J
BURNETTE,W A
BURNETT,DAVID S
BURNET,ROSE M
BUROFF,STEVEN J
BUTLETT,DARRELL L
BUTTON,BEVERLY
BYORICK,ROBERT S
BYRNE,EDWARD R
CALLAHAN,R L
CAMPBELL,JERRY H
CAMPBELL,MICHAEL R
CANDREA,RONALD D
CANTWELL,RICHARD F
CAREY,J E
CARPENTER,THOMAS L,III
CARRAN,JOHN H
CARR,DAVID C
CARTER,DONALD H
CASEY,J L
CASPERS,BARBARA E
...
1120 TOTAL

+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:
362-SUP 3731-SUP 3624

COVER SHEET TO:
373-SUP

COCSPM = COMPUTING SYSTEM PERFORMANCE MEASUREMENT, SIMULATION, ACCOUNTING
MP# = MICROPROCESSORS/GENERAL DOCUMENTS
UN# = UNIX/SURVEY DOCUMENTS ONLY

HO CORRESPONDENCE FILES
HO 5C101

TM-79-3624-5
TOTAL PAGES 27

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

subject: **Relative CPU Performance on C Language Programs**
Case: **49343-11**
File: **40125-004**

date: **September 28, 1979**

from: **Jerome Feder**
MH 3624
2D-230 x2461

TM: **79-3624-5**

MEMORANDUM FOR FILE

1. INTRODUCTION

Much of the software used by the Bell System is written in the C language. A Center 362 task force* concerned with the development of processors for telephone company application has devised a set of seven benchmark programs to evaluate C language processor performance. These benchmarks are intended to imitate the style of C language use in Bell System applications and by the UNIX† operating system. Characteristic of these applications is a fair degree of emphasis on string and C *structure* manipulation and very sparse appearance of significant arithmetic computation; many applications do no floating-point computation at all.

This memorandum compares the execution speeds and program sizes obtained by running the seven benchmark programs on a variety of machines spanning the range from microprocessors to some of the largest mainframes. The following machines were measured:

- Amdahl: V7 (IH Computer Center)
- BTL: 3B-20 (with memory cache and IS25 instruction set), MAC 8
- Data General (DG): Eclipse M600
- Digital Equipment Corporation (DEC): VAX 11/780, PDP 11/70, PDP 11/45, PDP 11/34, PDP 11/23, LSI 11. (MOS memory for all models)
- Honeywell: 6080 (MH Computer Center)
- IBM: 370/168, 3033 (IH Computer Center)
- INTEL: 8086 Single Board Computer (SBC)
- Interdata: 8/32
- Tandem: Tandem 16
- UNIVAC: 1100/81

The comparisons necessarily include the compilers for the various machines which in some cases are brand new and in others are well seasoned and honed. Sections of this memorandum deal with the effects of compiler options, alternate compilers, clock speed, caches, and the addition of memory management.

The work to be described comprises a first step in measurements currently underway on operating systems in conjunction with CPU's.

1.1 The Benchmarks

The code for the seven benchmarks is given in Appendix A. In all but the first case (Ackerman's function), the code was extracted from an existing program. The original programs all have different authors.

* Task force membership: A. D. Berenbaum, C. W. Hoffner, P. Lutz, R. W. Mitze, D. A. Poplawski and E. P. Schan Jr..

† UNIX is a trademark of Bell Telephone Laboratories

b1 Ackerman's function: computes the doubly recursive function *ack(3,8)* once. Since negligible amounts of computation are involved, this benchmark tends to stress the subroutine call mechanism. *ack(3,8)* issues a complex pattern of calls to itself repeatedly growing and shrinking the stack by large amounts; the maximum depth of recursion is 2044.

b2 Word sort: accepts a list of English words and counts the number of times each word occurs by sorting the words into a binary tree. Lots of subroutine calls as well as pointer and string manipulation are involved. To avoid disk access, the input array of 1000 words is furnished as part of the program. The sorting of this array is repeated 30 times.

b3 Quicksort: lexicographically sorts a list of seven English words using the *quicksort* algorithm. The code stresses the ability to call subroutines, access the stack, and to efficiently handle strings using character pointers. The seven word sort is repeated 30,000 times.

b4 Terminal handler: examines and translates ASCII characters. The code for this benchmark was extracted from the SPS operating system of the MAC 8 PLAID system. This code is also somewhat similar to terminal handling code in UNIX. The translation of a sequence of characters is repeated 30,000 times.

b5 Symbol table insert: performs string and list manipulation to look up a symbol in a hashed symbol table. This benchmark was derived from the MAC-8 assembler. The symbol lookup operation is repeated 30,000 times.

b6 Buffer release: performs list manipulation and other operations to release an operating system buffer. The code for this benchmark was extracted from the UNIX operating system. The benchmark emphasizes the ability to manipulate C language structures such as occurs with high frequency in UNIX. The buffer release operation is repeated 100,000 times.

b7 Statistics: computes the mean and first moment of an array of ten integers using integer arithmetic (16 bits permissible). This benchmark was derived from a program to compute disk seek statistics. This is the only benchmark with even modest arithmetic activity. The first moment computation involves a multiply operation per array element as well as a divide operation at the end of the computation. The computation is repeated 60,000 times. Despite the relative leniency and simplicity of the computation, this benchmark executed notably slowly on the low-end processors.

The following rules are observed by all of the benchmarks.

- No operating system calls are made. This involved removal of all I/O activity from the programs segments from which the benchmarks were constructed.
- No C library routines are referenced (except for the subroutine call save/restore mechanism). Any library routines needed are included with the benchmarks.

Prototypes of *b3-b7* were used originally by Berenbaum, Mollinelli, Pekarich and VanOrnum in a cycle count comparison of the MAC 8 with several commercial microprocessors.[1] The prototypes were revived and modified, and *b1* and *b2* were added, when a need arose to evaluate alternate instruction sets for a processor under development in Center 362. The major benchmark changes were the inclusion of "null" subroutines to satisfy subroutine references and the addition of sufficient repetitions to yield easily measurable running times.

The benchmarks were examined during their development using a C compiler instrumented to furnish the static (each instruction in the program assumed to be executed once) frequency of occurrence of primitive operations yielded by C code. These frequencies were compared to frequencies obtained from code used in a number of UNIX Operations Support (OSS) applications; approximately the same occurrences of the most frequent primitives were found.

The author made a number of minor changes to *b1-b7* to increase portability and to enable them to run on all the machines in question. None of the changes noticeably affect execution time for the PDP 11/70 or the VAX 11/780. The use of these benchmarks represents a deliberate choice to use existing tools as opposed to developing new ones; some limitations of the benchmarks and suggested areas for improvement are given in Section 5.

Ackerman's function, because of its unusual stack growth, proved to be the most troublesome benchmark. For most of the machines, large areas for stack growth were reserved initially before running the benchmark. When running under UNIX, stack allocation was left to the operating system. UNIX allocates a modest amount of stack space initially; stack growth beyond this initial allocation causes the operating system to be invoked to furnish more stack space which then remains with the program thereafter. The effect of this operating system intervention was verified to be negligible by creating a trial program that ran *ack(3,8)* twice in succession and observing that the execution time (on the PDP 11/70 and VAX 11/780) was essentially twice that of running *ack(3,8)* once.

1.2 Compilers

In general, the results in Sections 2 and 3 are for the best available compiler and also reflect the use of code optimizers and space saving compiler options where available. The effects of alternate compilers and compiler options are explored in Section 4.

For the PDP 11 and LSI 11 machines, the standard well-seasoned PDP 11 compiler, written by D. M. Ritchie, was used. S. C. Johnson has developed a *portable* C compiler which has been used as the basis for most of the other C compilers that exist. For the MAC 8 there exist "old" and "new" compilers, based on the Ritchie and Johnson compilers, respectively. Since the new compiler is intended to replace the old one, it was chosen for the comparisons. A new VAX compiler version, incorporating several refinements, became available towards the end of this study and is the one that was used. The IBM compiler on which results are based is the one currently used with the IH TSS system; alternate IBM compilers were not examined.

The compiler and optimizer for the revised (IS25) 3B instruction set became operational just in time for 3B-20 results to be included; this compiler is at a very early evolutionary stage and is likely to be improved.

1.3 How the Benchmarks Were Run

Where possible, the benchmarks were run on "quiet" machines i.e., the benchmark was the only program running at the time. For the larger machines where this was not possible (Honeywell, IBM and Amdahl machines), the benchmarks were run during a period of light machine use. In the latter cases, each benchmark was run a number of times to ascertain that there were only slight deviations in execution time between runs and the execution times were averaged.

The operating (or other supervisory) systems used are given below†

- IBM 370/168, IBM 3033 and Amdahl V7: IBM TSS
- DEC VAX 11/780, DEC PDP 11 (all models), Interdata, 8/32 and Univac 1100: UNIX
- DEC LSI 11: stand alone operating system developed by T. J. Kowalski (Dept. 3624)
- DG Eclipse M600: AOS
- Tandem 16: Guardian
- Intel 8086 (without memory management): BASIC 16 system, The Intel In Circuit Emulator (ICE) system used to emulate the 8086*
- BTL 3B-20: none
- BTL MAC 8: PLAID system

IBM TSS CPU accounting was used to measure the IBM and Amdahl execution times. For the runs using UNIX, the execution times are the *user* times as furnished by the UNIX *time* command. (These are essentially the same as overall elapsed time in a quiet environment). A special very accurate ad-hoc timing scheme, devised by R. Fleisleber (Dept. 3622) , was used for the

† Operating system "overhead" is normally small in a quiet environment and is neglected here. Strictly, many of the measurements represent available CPU resource under the particular operating system used.

* According to Intel, the ICE system furnishes run times that are equal to 8086 run times in the configuration used here.

Intel 8086 measurement. The 3B-20 times were obtained from its real-time clock. The remaining machines were timed using a stopwatch and therefore subject to errors as much as 5%. In view of measurement error, speed differences of individual machine samples, and the general benchmark limitations discussed in Section 5, the relative speeds given in Section 2 should be taken as only approximate.

2. Execution Speed Results

Relative execution speeds for the seven benchmark programs are summarized in Figs. 1 and 2 and in Table 1. The execution speeds were obtained by normalizing each benchmark execution time to the execution time for the same benchmark on the PDP 11/70 and then inverting the result. The actual execution times are given in Appendix B. The results in both figures have been sorted in terms of decreasing average execution speed for benchmarks *b2-b6*.

CPU	*Average*	*Range*
Amdahl V7	7.6	6.0 - 8.2
IBM 3033	6.7	5.7 - 7.3
IBM 370-168(3)	2.81	2.64 - 3.13
UNIVAC 1100/81	1.55	1.44 - 2.42
BTL 3B-20 (IS25, cache)	1.16	.63 - 1.59
DEC VAX 11/780	1.13	1.03 - 1.25
DEC PDP 11/70	1.00	1.00 - 1.00
DG Eclipse M600	.93	.56 - 1.76
DEC PDP 11/45 (cache)	.73	.71 - .80
Tandem 16	.73	.63 - 1.05
Honeywell 6080	.66	.56 - .90
Interdata 8/32	.62	.56 - .79
DEC PDP 11/34 (cache)	.49	.41 - .54
Intel 8086 (8 mhz)	.49	.37 - .57
DEC PDP 11/45	.42	.40 - .49
Intel 8086 (5 mhz SBC)*	.307	.228 - .359
DEC PDP 11/34	.296	.246 - .345
DEC PDP 11/23	.281	.209 - .321
BTL MAC 8 (3 mhz)	.254	.049 - .347
BTL MAC 8 (2 mhz)	.169	.033 - .232
DEC LSI 11	.123	.087 - .128

Table 1
Normalized Speed Factors

* Results for Intel 8086 with memory management do not differ significantly with Intel SBC times

Fig. 1 shows the execution speeds for the individual benchmarks for selected machines. Each dot in Fig. 1 represents the execution speed of a benchmark; seven dots for each machine describe the execution speeds of the seven benchmarks. Insufficient memory on the available LSI 11 installation prevented the running of *b1* and *b2;* the five dots for this machine in Fig. 1 represent *b3-b7*. For the most part, the normalized execution speeds for the different benchmarks on a given machine did not differ greatly. Many machines performed relatively well on *b1;* very likely a reflection of an efficient (compared to the PDP-11) subroutine call mechanism for small numbers of arguments and registers. A relatively poor showing on *b7* was also common on the low- end machines. The most dramatic instance of this occurs for the MAC 8 which shows itself a factor of five slower on this benchmark than it does on the others. *b7* requires multiply and divide operations which are accomplished using lengthy subroutines on the MAC 8. Changing these operations to addition and subtraction brought the MAC 8 execution speed for *b7* into line with its execution speed for the other benchmarks. The other low-end machines, the Intel 8086, PDP 11/34, PDP 11/23 and LSI 11, ran 17-30% slower on *b7*. This is likely due to lack of appropriate CPU features and heavy reliance on microcode for the implementation of arithmetic multiply and divide operations. Poor performance by the Eclipse on *b7* is attributed* to lack of a sufficient number of registers.

Fig. 2 and Table 1 show the range of speeds encountered; the high and low for the seven benchmarks as well as the average of *b2-b6*. *b1* (Ackerman's function) was omitted from the average because its atypical nature. *b7* was omitted because it contains multiply and divide operations. Inclusion of *b7* would lower the Eclipse and 3B-20 averages about 8% and the Intel 8086, PDP 11/34, PDP 11/23 and LSI 11 averages 3-5%.

2.1 Wait States and Clock Speed

A *wait state* is an extra machine clock cycle added to a memory access operation to give a slow memory system time to respond.

* Personal communication by Guy Riddle of Dept. 9155 who wrote the compiler for this machine.

When comparing microprocessor chips by measuring computers based on them, it is necessary to take into account the number of wait states and the clock speed used in the configurations that are measured. The PLAID system for the MAC 8 uses zero wait states and was measured operating at 2 megahertz. The Intel SBC used for measuring the 8086 uses one wait state on memory reads, two wait states on memory writes, and operates at 5 megahertz. (Intel feels that the memory choice for the SBC is a reasonable compromise between memory cost and overall speed.) Memory is available which will permit operation of the 8086 chip using zero wait states yielding an estimated (approximate) 15% improvement in speed; caution must be exercised in comparing with the MAC 8 zero wait state result.

The Intel 8086 and MAC 8 measurements were made at the highest clock speeds for which configurations were readily accessible. Versions of the Intel 8086 are available which will run at 8 megahertz and MAC 8 versions are available which will run at 3 megahertz. Memory is available for both of these machines which will run at the higher speeds without change in the number of wait states required. Thus, speed should scale with clock rate; this implies the availability of Intel 8086 and MAC 8 configurations capable of operating 1.6 and 1.5 times faster, respectively, than the systems that were measured. Fig. 2 (and Table 1) gives both measured and projected higher clock rate speeds for these machines as well as speeds for some other machine variants to be described*

2.2 Intel 8086 with Memory Management

Addition of memory management to microprocessors can degrade execution speed due to the requirement for additional wait states. H. B. Mendell of Dept. 3356 has developed a memory management unit for the Intel 8086 which uses one wait state on memory reads and writes (slightly less than used by the Intel SBC). This unit was furnished with a 4.096 megahertz clock in the installation that was measured. Execution times for the seven benchmark programs, scaled to reflect a 5 megahertz clock frequency, averaged about 2% faster than those obtained with the Intel SBC. (The measurement was made using a stopwatch so that small percent differences are not significant).

2.3 PDP 11/45 and PDP 11/34 with Add-On Memory Caches

The PDP 11/45 and PDP 11/34 computers do not have memory caches as standard equipment. A number of manufacturers, however, market add-on caches for these machines. The measured speed improvement (run time without cache/run time with cache) for a PDP 11/45 with a Fabritek cache and for a PDP 11/34 with a Digital Equipment Corporation cache are summarized in Table 2.† The machines had MOS memory.

	Average	*Range*
PDP 11/45	1.74	1.64 - 1.86
PDP 11/34	1.65	1.56 - 1.70

Table 2
Speed Improvement with
Add-on Caches

The PDP 11/34 measurements are for the same machine with and without the cache enabled. The PDP 11/45 measurements are for two different PDP 11/45 machines, one with and one without cache; the speed ratios are thus subject to speed variation of the individual basic machines. (The PDP 11/45 is prone to this).

These benchmarks exhibit unusually high cache hit ratios; the improvements displayed are thus higher than would be observed for most code. The impact of caches is discussed further in Section 5.

3. PROGRAM SIZE

Program text and data sizes for the seven benchmark programs are summarized in Figs. 3-5 and Table 3. The sizes in bytes have been normalized to the corresponding PDP-11 sizes and are ordered in terms of increasing average text size. (Non-normalized program sizes are given in

* The 8086 8 megahertz projection is based on the 5 megahertz SBC result. Zero waits states would yield a speed about 15% higher than that shown. Memory that will operate at 8 megahertz with zero wait states is currently expensive and would be impractical for most applications.

† The actual improvements measured for *b1-b7* were
PDP 11/45: [1.64, 1.70, 1.81, 1.79, 1.76, 1.64, 1.86]
PDP 11/34: [1.56, 1.57, 1.70, 1.68, 1.70, 1.69, 1.64].

	Text		*Data*		*(Text + Data)*	
	average	*range*	*average*	*range*	*average*	*range*
DEC VAX 11/780	.84	.70-.95	1.44	.95-2.14	.96	.70-1.38
BTL 3B-20 (IS25)	.97	.84-1.04	1.44	1.00-2.00	1.07	.94-1.38
DEC PDP 11	1.00	-	1.00	-	1.00	-
DG Eclipse	1.03	.81-1.18	1.01	1.00-1.03	1.05	.82-1.16
Intel 8086	1.06	.96-1.19	.99	.92-1.00	1.05	.97-1.19
BTL MAC 8	1.11	.92-1.34	.97	.75-1.07	1.08	.90-1.31
Tandem 16	1.16	.97-1.29	1.03	.93-1.10	1.15	1.00-1.26
Interdata 8/32	1.82	1.54-2.26	1.47	.92-2.29	1.74	1.50-2.26
IBM 370	2.16	1.85-2.67	3.57*	1.39-∞	2.30	1.40-3.41
H6080					2.35	1.42-2.98
Univac 1100	3.40	2.81-3.99	3.18*	1.44-∞	3.18	1.47-4.14

Table 3
Normalized Benchmark Sizes

Appendix C). The sizes do not include library routines nor do they include stack growth (significant for *b1)* which occurs during program execution. Where code optimizers are available, the sizes shown are for the optimized code. The VAX code sizes, except for *b2,* are for code generated using the complier option which assumes address offsets to be less than 32K in magnitude. (The data array in *b2* was too large for this option to be used.)

For most of the machines, the text sizes were taken as the text size of the ".o" file produced by the compiler. Data sizes were computed from the data sizes of the "a.out" files by subtracting the data size of a null program; the latter procedure was necessary to include the size of uninitialized global data areas.

Figs. 3 and 4 show the text and data sizes, respectively, of the individual benchmarks. Fig. 5 shows the range of sizes encountered — the high, low and the average for text, data, and text plus data. The Amdahl instruction set is identical to that of the IBM machines so that the text and data sizes are the same. Only composite text plus data sizes were obtainable for the Honeywell 6080.

The text sizes for the VAX, 3B-20, PDP 11, Eclipse M600, Intel 8086, MAC 8 and Tandem machines are fairly close and significantly smaller than the text sizes for the Interdata, IBM and Univac machines. Data sizes for the MAC 8 and all 16 bit machines are nearly the same and are much smaller than the sizes for the larger word size machines which require more space to store C language *integers* and *pointers.* Some types of address information which constitute text on the other machines are treated as data on the Univac and IBM machines. As a result, the data sizes for this machine are markedly higher. *(b1,* in fact, required no data storage space at all on all but the IBM and Univac machines, yielding a normalized data size for *b1* of infinity).†

* Average without b1

† Fig 5 shows occasional instances in which averages of text plus data are either less than or greater than both text and data averages. This is *not* an error, but rather a property of the calculation. (Intuitively, for certain benchmarks one or the other of the text or data components predominates in the total size thus nullifying the effect of the other component on the text plus data average.)

4. COMPILER EFFECTS

4.1 Alternate Compilers

The results in Sections 2 and 3 were obtained using the standard (Ritchie) PDP-11 compiler and the new MAC 8 portable compilers. Results obtained with the portable PDP 11 compiler and with the older MAC 8 compiler are summarized in Table 4. The percentage numbers are the result of comparing with corresponding figures for the compilers used in Sections 2 and 3. For relative speed, positive percent values indicate faster execution. For program size, positive percent values indicate larger size. All results are for optimized code. Detailed results are given in Appendix D.

Execution times and data sizes for the portable PDP 11 compiler are very close to those obtained with the standard compiler. The text sizes averaged 11.3% larger. Execution times and text space for the new MAC 8 compiler are nearly the same as obtained with the old version, with a small improvement in average data size. (The new compiler represents a considerable improvement in terms of incorporating recent C language changes and features.)

4.2 Peephole Optimizers

The PDP 11, VAX, Interdata, MAC 8 and 3B-20 compilers incorporate what are referred to as "peephole" code optimizers. These operate on the assembly language output of the basic compiler to remove extraneous register loads and jump sequences and to perform miscellaneous local transformations, usually idiosyncratic to the particular target machine, which serve to reduce space and/or running time requirements. None of the optimizers affected the data space required by any of the programs. The time and space improvements introduced by the optimizers are given in Tables 5 and 6, respectively, expressed in percentages relative to the non-optimized code. Negative entries indicate that the optimized code actually ran slower than the non-optimized code. The VAX comparisons (except for *b2)* assume the small address compiler option. The MAC 8 and 3B-20 optimizers were not investigated.

	Relative Speed avg/(range)	*Relative Text Size avg/(range)*	*Relative Data Size avg/(range)*
Port. PDP11	-1.2% (-3.1 - 0)%	+11.3% (5.3 - 34.9)%	-1.2% (-8.1 - 0)%
Old MAC8	0% (-4.0 - 3.9)%	-.1% (-10.8 - 10.8)%	4.3% (0 - 28.6)%

Table 4
Portable PDP-11 and Old MAC 8
Compilers compared with Standard Versions

	VAX	*PDP 11*	*(Port) PDP 11*	*Inter.*
b1	6.8%	4.5%	.2%	-4.5%
b2	3.9%	4.1%	0.	.3%
b3	8.2%	6.1%	1.0%	.6%
b4	8.3%	6.5%	3.5%	1.1%
b5	6.0%	6.7%	1.7%	1.1%
b6	5.9%	3.5%	.9%	1.8%
b7	5.3%	2.8%	-.2%	
avg	6.3%	5.0%	1.4%	-.1%

Table 5
Speed Improvement By Optimization

	VAX	*PDP 11*	*(Port) PDP 11*	*Inter*
b1	-28.6%	-20.4%	-3.3%	-6.7%
b2	-10.1%	-8.8%	-4.0%	-9.1%
b3	-13.8%	-7.3%	-2.4%	-10.5%
b4	-14.8%	-8.7%	-4.7%	-15.2%
b5	-17.1%	-11.0%	-4.9%	-9.8%
b6	-23.0%	-5.8%	0	-8.4%
b7	-14.0%	-10.1%	-4.1%	-10.6%
avg	-17.4%	-10.3%	-3.3%	-10.1%

Table 6
Program Text Size Reduction
By Optimization

The PDP 11 execution speed results are for a PDP 11/70.* The effect on execution speed varies with the optimizer but, in general, is modest; the best case, a 6.3% average improvement, occurrs for the VAX 11/780. For the Interdata, in two cases the optimized code actually ran slightly slower than the standard code. Optimization yields substantially greater improvements in program size; average text size reduction exceeds 10% except for the portable PDP 11 compiler. Optimization in the latter case was found to have only a very small effect on space and time requirements. The optimized code for the portable and standard compilers are close in time and space requirements; the non-optimized code for the standard compiler is poorer.

* For other PDP 11 models the average speed improvements differed slightly as follows: PDP 11/45 — 5.7%; PDP 11/34 — 6.1%; PDP 11/23 — 5%.

4.3 VAX Small Address Option

As stated earlier, results for the VAX excepting *b2* were obtained using the *-d2* option which informs the compiler that all address offsets are smaller than 32K; this enables the compiler to use smaller bit fields for addresses yielding smaller and very slightly faster running object code modules. If during the formation of an object module the 32K limit is exceeded, the compilation aborts with an error message. At this point, a second try without the option can be made. The increase in text size for the benchmarks encountered when the option was not used is given in Table 7.

b1	13.3%
b2	-
b3	8.5%
b4	16.5%
b5	14.9%
b6	17.5%
b7	7.0%
avg.	13.0%

Table 7
VAX Text Size Increase Without
-d2 Option

The option did not affect the data size of any of the benchmarks. The improvement in average execution speed in using the option was found to be negligible (.3%).

5. DISCUSSION

During the course of running the benchmarks, a number of factors became evident which would cause the benchmark results to deviate from what might be experienced in practice by the class of programs of interest here.

5.1 Character String Manipulation

No attempt was made to tailor the benchmarks in any way so that they would perform well on any of the machines in question. Nevertheless, the selection of existing code as a basis for the benchmarks tends to favor the PDP 11, the most common target machine. Experienced coders usually know which of alternate possible C language constructs will yield the most desirable PDP 11 object code. One major aspect of this is the technique for string manipulation. PDP 11 coders commonly do this by establishing

pointers to positions within byte arrays and manipulating these pointers and their targets to achieve desired results. This works very well on the PDP 11, and acceptably well on most of the other machines examined here. Some machines, such as the Honeywell 6080, Tandem 16 and UNIVAC 1100/81, however, require inordinate numbers of instructions to manipulate strings in this fashion. For these machines, string computations execute much faster if programmed in terms of arrays and indices. It is interesting to note that the Honeywell 6080 and Tandem 16 machines have special hardware instructions for rapid byte string manipulation which could be (actually are in the case of the Honeywell) incorporated into the C library string manipulation functions.

Four of the benchmarks, *b2* - *b5*, manipulate byte strings using character pointers and are thus affected. To obtain a rough idea of the magnitude of the effect, *b2* was experimentally altered for the Honeywell and Univac machines. Substitution of the Honeywell C library string compare function for the equivalent function incorporated into *b2* (the latter was taken from the C language manual[2]) yielded a 30% improvement in execution speed. Manipulation of *b2* to better suit it to the Univac environment yielded a 37% speed improvement. Improvements of similar magnitude could also be expected for the Tandem machine.

The issue of efficient access to data by pointer versus by indexed array applies generally to all types of data. For these benchmarks, however, character data predominates.

5.2 Memory Caches

The VAX 11/780, PDP 11/70, 3B-20, and the high end IBM, Amdahl and Univac machines have memory caches as integral parts of their designs. The remaining machines, unless otherwise noted, do not. The caches on the IBM, Amdahl and Univac machines are quite large — large enough to hold many if not most programs in their entirety. For the PDP 11/70,VAX 11/780, and 3B-20 machines, entire program cache residency would be the exception rather than the rule, and cache hit ratios would tend to fall significantly short of 100%.* Cache hit ratio depends heavily on the program being executed. DEC literature[3,4] quotes overall cache hit ratios for their machines in the 80% — 95% range with a typical value of 95% for the VAX 11/780 and somewhat less for the PDP 11/70.

Behavior of the cache in the VAX 11/780, PDP 11/70 and 3B-20 machines complicates the interpretation of these benchmark measurements. While it was not feasable to measure the hit ratios attained, it is reasonable to expect atypically high values overall due to the use of short programs and data, tight program loops, and because there was no competition from other programs. In particular, *b3* - *b7* should fit entirely within the cache and achieve virtually 100% hits. *b1* and *b2* are small programs but encounter cache misses - *b1* because of its stack activity and *b2* because of its large data arrays. The hit ratios for *b1* and *b2* are difficult to predict. Benchmark hit ratios that are higher than obtained by a normal mix of programs cause the machines affected to appear faster than would be observed in normal service.

An experiment was performed to help estimate cache effects on the PDP 11/70. *b2* - *b7* consist of loops that repeatedly execute a relatively short program segment to achieve measurable running times. To determine the transient aspects of this code execution and to obtain an idea of how a benchmark without this somewhat artificial type of construction might perform, the cache was initially invalidated by repeatedly executing a large irrelevant program segment several times and then a hardware monitor was used to measure execution times for successive passes through the loops in *b2* - *b7*.

A Comten Dynamite 8016[5] hardware monitor provided four external timers, each accurate to 10 μs. Software access to the hardware monitor to trigger the timers was provided by an otherwise unused floating-point maintenance register on the PDP 11. This method was taken from Hayden[6]. It was found that by the third pass through each loop, execution time had essentially

* The PDP 11/70 cache consists of 1024 16-bit words organized into two 512 word *groups*. The VAX 11/780 cache is somewhat similar to the PDP 11/70 cache but consists of two groups of 1024 32 bit words. The 3B-20 cache consists of four groups of 512 32 bit words.

stabilized. Table 8 shows the times for the first and second loop passes, normalized to the steady-state value and corrected for timer start/stop overhead.

	Pass 1	*Pass 2*	*Steady State*
b1	-	-	-
b2	1.00	1.00	1.00
b3	1.48	1.01	1.00
b4	1.24	1.00	1.00
b5	1.21	1.05	1.00
b6	1.50	1.03	1.00
b7	1.24	1.00	1.00

Table 8
PDP 11/70 Cache Effect on Execution Time For Successive Loop Passes

A loop of *b2* required about 700 ms; line clock interrupts as well as the overall length of the time period obscure any initial cache loading transient. For the other benchmarks, the initial loop executed from 21% to 50% slower.

The cache hit ratios for this experiment are unknown. Only the outermost benchmark loop was broken permitting significant cache hits due to internal loops and repeated access to data values. *b6* which showed the largest (50%) difference is a linear code segment with no loops other than the one measured. A similar experiment using a repeated execution of the UNIX/TS *getpid* system call on the PDP 11/70 showed an 87% longer time for the first execution. Differences of 50% or more, however, would only be encountered for programs with unusually low cache hit ratios.

A second experiment, in which the benchmarks were run with the caches on the VAX 11/780, PDP 11/70 and 3B-20 completely disabled, helps set a lower bound on the speed of the three machines. Average speed for the seven benchmarks was found to degrade by a factor of 2.83 for the VAX 11/780, 2.96 for the PDP 11/780, and 1.42 for the 3B-20.* The 3B-20 result indicates far less dependence on the cache than occurs for the DEC machines. (This is expected since the 3B-20 uses relatively fast memory.)

Instruction times on the PDP 11/70 tend to be dominated by memory access. A review of memory access times for the VAX 11/780, PDP 11/70, and 3B-20 is helpful in determining the effect of cache hit ratio on execution speed. The access times for cache hits and misses are given in Table 9.[3,4]

	Cache Hit	*Cache Miss*
PDP 11/70	.3μs	1.3μs
VAX 11/780	.2μs	2.0μs
3B-20	.25μs	.80μs

Table 9
Memory Access Time For Cache Hits and Misses

For hit ratios near 100%, small changes in hit ratio can yield large percentage changes in memory access time. Effective PDP 11/70 memory access times increase 33% in going from 100% to 90% cache hit ratios. Effective VAX 11/780 memory access times increase 45% in going from 100% hit rate to the 95% value quoted by DEC as typical for this machine. 3B-20 access times increase 11% in going to 95% hit rate. The relatively large effect for the VAX is perhaps why the VAX uses a much larger cache than the PDP 11/70 plus an instruction lookahead mechanism which lessens the effect of some of the cache misses on machine speed. (The 3B-20 also usessuch a mechanism).

The 3B-20 speeds in Section 2 should require only minor downward correction to adjust to cache hit ratios obtained in normal use. For the DEC VAX 11/780 and PDP 11/70, the required corrections are more significant. Since memory access times are only a component of total instruction times, it is difficult to say how much of a correction to apply without further experiments or simulations; a 20% downward correction in speed factor for the PDP 11/70 appears reasonable, based on a review of the effect of memory access times on instruction times* and the evidence in Table 8. A reasonable VAX 11/780 correction is probably on the

* The degradation in execution speeds of *b1 - b7* obtained by disabling the VAX 11/780, PDP 11/70, and 3B-20 caches were as follows
VAX 11/780: [.40, .31, .35, .37, .36, .35, .32]
PDP 11/70: [.40, .36, .32, .33, .32, .34, .31]
3B-20: [.92, .65, .67, .69, .66, .68, .72]

* A 20% correction is that obtained in adjusting from 100% to 90% hit ratio for a PDP 11 *add* A.B. instruction using indexed address modes (mode 6).

same order, but is more difficult to estimate with the evidence available.†

5.3 Word Size

These benchmarks, unfortunately, provide no advantage to the machines with word size larger than 16 bits. Larger word size yields correspondingly higher precision results for arithmetic on integers. It also permits faster bulk data copies — something that operating systems do frequently. As an example, a small test program to simply copy an array of data from one place to another in the same address space achieved a rate of 1.6 megabytes/second on the 32 bit VAX 11/780 as compared to .92 megabytes/second on the 16 bit PDP 11/70. This is a 74% difference between the two machines as compared to an average 13% difference obtained with the benchmarks.

Probably most important aspect of a larger word size, however, is the increased ability to deal with large addresses. This permits the manipulation of very large data arrays and allows very large programs to be maintained intact rather than being written as smaller pieces that intercommunicate. This difference is to some extent fundamental in terms of the architectural freedom it allows and the types of problems that can be handled. It also enables more efficient operating system design. (It is often possible to handle large address spaces using small word size machines by appropriately manipulating memory map hardware; the cost in software complexity and performance, however, usually makes it advisable to choose a more suitable processor.)

5.4 CPU Features to Support an Operating System Environment

In general, only minimal CPU features, present in all of the machines, are required to execute the benchmarks used here. Other features, such as memory management, aids to context-switching, program interrupt handling, virtual address capability, I/O channels, DMA capability, memory protection and error recovery can dramatically affect attractiveness and performance of the machine for an operating system environment. These features are present in different forms on most of the machines and tend to be omitted or skimpy on the low-end machines. The minimal nature of the benchmark requirements has the effect of making the microprocessors look more attractive when compared to high-end machines than would be the case if the entire picture was viewed.

5.5 Other CPU Comparisons

Three other studies have compared machines on the basis of C code execution speed for non-arithmetic computation. The methodologies of these studies differed. along with the code segments used for comparisons. For the most part, the studies were of a predictive nature, performed in the absence of hardware and in some case compilers for the machines in question; in no case were benchmark programs actually executed.

W. Rash[7] compared the MAC 8 to the Intel 8086 and Zilog Z8000 processors by hand compiling three small test programs according to a stated set of rules. Calculated static cycle counts, adjusted by machine clock rate, were used as a basis for speed comparison. Optimistic assumptions were used for the hand compilations, yielding MAC 8 cycle counts which bettered by 23% those achievable with the then available MAC 8 C compiler.

A. Berenbaum, J. Molinelli, S. Pekarich and J. VanOrnum[1] compared the MAC 8, Intel 8086, Zilog Z80A and Zilog Z8000 using benchmarks which included prototypes of *b3-b7* used here. Available compilers were used for all but the Z8000. For the Z8000, C code was passed through the PDP-11 compiler and the resulting assembly instructions were hand translated into Z8000 instructions. Speed comparisons were based on calculated cycle counts, adjusted by clock rate.

Most recently, Boivie and Rutter[8] compared the Intel 8086 to the DEC LSI 11, PDP 11/23, and PDP 11/45 computers using the C source for a variety of UNIX commands. For the Intel 8086, the programs

† The VAX and 3B-20 designs also include an address translation buffer cache to hold 128 recent virtual to physical page address translations. The benchmarks here would run with a 100% hit ratio for this mechanism (as would almost all programs run in stand-alone mode). Hit rates below 100% would likely be encountered on a heavily loaded machine. Time loss due to cache misses of this type is probably best treated as extra overhead on an operating system context switch.

	MAC 8 (2 mhz)	*Intel 8086 (5 mhz)*	*LSI 11*	*PDP 11/23*	*PDP 11/45*	*Zilog Z80A (4 mhz)*	*Zilog Z8000 (4 mhz)*
Rash	1	2.97	-	-	-	-	2.71
Berenbaum	1	2.25	-	-	-	.74	2.64
Boivie	-	1.93	.728	1.19	2.03	-	-
Feder	1	2.10	.728	1.66	2.49	-	-

Table 10
Comparison of Speed Factors Obtained

were first compiled into PDP 11 assembly code and then this code was mechanically translated into Intel 8086 instructions. Static counts of the 20 most frequent instruction/address-mode combinations were used in conjunction with manufacturers stated instruction times to estimate relative program run times.

A comparison of these results with the results obtained here is given in Table 10. Since the benchmarks are all different, only ratios of machine speeds are available. To compare results, speeds have been normalized where possible to the MAC 8. Since Boivie and Rutter did not include the MAC 8, their speed factors have been adjusted so that the LSI 11 speed factor matches that reported here (last row). Zero wait states are assumed for the MAC 8, Intel 8086 and Zilog machines.

The MAC 8 result for Rash is that given for the actual as opposed to hypothetical, MAC 8 C compiler. The Intel 8086 speed factor in the last row is the value measured here adjusted upward by 15% to reflect zero wait states.

Allowing for differences in benchmarks, compilers (or compiler assumptions), sample-to-sample machine speed variations and the need to rely on manufacturers statements concerning instruction timing, there is for the most part reasonably good agreement between results. This indicates the availability of a variety of methods for predicting potential C language CPU performance in the absence of hardware or compilers.

5.6 Benchmark Improvements

The benchmarks used here were developed on a constrained time schedule with attention to only some of the factors that affect speed of program execution. Tools for measuring program properties are being developed and improved; these tools should improve understanding of the current benchmarks and yield new and better ones. Some areas for improvement are listed below:

- Use of dynamic as opposed to static occurrences in attempting to match properties of existing code.
- Control of benchmark properties that affect cache hit ratio.
- Control of pattern of occurrence of program control transfers to accurately portray pipelined processors.
- Attention to the number of registers that can be advantageously used during execution.
- Attention to distribution of the magnitudes of references to storage to accurately portray machines that efficiently encode small address offsets.
- Important operating system functions should be identified and appropriately represented in the benchmarks.
- It would be helpful if the benchmarks generated some kind of result (perhaps displayable only during trial runs) to verify correct program execution.

6. SUMMARY

A set of numbers has been presented comparing the execution time and space requirements of seven C programs run on a wide variety of machines. The approach has been to use existing benchmark tools, supplemented by additional data and cautions in known weakness areas, to provide timely information. Due to benchmark limitations, speed variations in individual machines and ongoing C compiler changes, the results only approximately portray the CPUs in question; the reader is cautioned against fine-grained viewing of these numbers.

J. Feder

MH-3624-JF-st **Jerome Feder**

Att' Refs, Ack, App, A-D. Figs 1-5

REFERENCES

1. A. D. Berenbaum, J. J. Molinelli, S. P. Pekarich, and J. H. VanOrnum "Suitability of the MAC-8 and Selected 8 and 16 bit Bit Commercial Microprocessors for Supporting the C Language," File Case 40256, June 27, 1978.

2. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice hall, Englewood Cliffs, N. J. 1978.

3. *PDP11/70 Processor Handbook*, Digital Equipment Corporation, Maynard Massachusetts.

4. *VAX 11/780 Architecture Handbook, Vol 1*, Digital Equipment Corporation, Maynard Massachusetts, 1978.

5. Dynamyte Users Manual, Comten Inc., Rockville, Md., 1974.

6. D. F. Hayden, "Some MERT Performance Measurements in Transaction Oriented Processing," MF-77-3124-12, July 28, 1977.

7. W. Rash, "MAC-8, Zilog Z-8000 and Intel 8086 Processor Comparisons for C Language Execution," TM 78-5334-1, May 19, 1978.

8. R. H. Boivie and P. E. Rutter, "8086 Throughput", File Case 40221-3, April 26, 1979.

Acknowledgments

The benchmarks used in this study were developed by A. D. Berenbaum, C. W. Hoffner, P. Lutz, R. W. Mitze, D. A. Poplawski and E. P. Schan, Jr.

The author wishes to thank the people listed below for their help in running the various benchmarks and for many suggestions provided along the way which helped shape the course of this study. The cooperation of these people involved the expenditure of significant time and effort and is ultimately what made the study possible.

Machine	*People*	*Dept.*
Amdahl V7	W. A. Felton	3761
BTL 3B-20	S. W. Ng E. P. Schan Jr.	3623 3626
BTL MAC 8	N. R. Miller	3625
D. G. Eclipse M600	B. Ruddock, S. Collins (DG)	9155
DEC VAX 11/780	L. A. Wehr	3624
DEC PDP 11/45	P. Fermin	WeCo
DEC PDP 11/34	T. J. Kowalski	3624
DEC PDP 11/23	C. B. Haley	1271
DEC LSI 11	T. J. Kowalski	3624
Honeywell 6080	L. Rosler	3731
Intel 8086 SBC	R. C. Fleisleber	3622
Intel 8086 +mm	T. P. Blumer	3621
IBM 370/168	W. A. Felton F. T. Grampp, R. C. Haight, A. R. Koenig	3761 3624
IBM 3033	W. A. Felton	3761
Tandem 16	A. M. Usas	1352
Univac 1100/81	G. Ronkin, J. Levy (Univac)	9176

The author is especially grateful to D. A. DeGraaf who helped initiate the study and who provided many suggestions during its course. A. D. Berenbaum, D. L. Bayer, T. B. London, J. M. Mollinelli, D. A. Poplawski, G. C. Vogel and members of Center 362 are also thanked for their valuable discussions and contributions.

APPENDIX A
BENCHMARK PROGRAM LISTINGS


```
/* benchmark program number 1 */

/* ackermann's function */

main() {

        ack(3,8);
        }

/* ackermann's function */
ack(m,n) int m,n; {

        if (m == 0)
                return(n+1);
            else
                if (n == 0)
                        return(ack(m-1,1));
                    else
                        return(ack(m-1,ack(m,n-1)));
        }
```

```
/* benchmark program number 2 */

/* word count */

#define NULL (struct tnode *) 0

struct tnode {
        char *word;
        int count;
        struct tnode *left;
        struct tnode *right;
        };

char *wdarray [1000] = {
"Maintenance",
"Documentation",
"for",
 .
 .
 .
"used"
        };
```

1000 WORD ARRAY

```
struct tnode tnarray [1000];
int t;

char chrs [10000];
int c;

main() { /* count occurrences of words */

        int i,n;
        struct tnode *root, *tree();

        for (i=1; i <=30; i++) {
                t = 0;
                c = 0;
                root = NULL;
                for (n=0; n<1000; n++)
                        root = tree(root, wdarray[n]);
                treeprint(root);
                }
        }

struct tnode *
tree(p,w) struct tnode *p; char *w; { /* add word to tree */

        struct tnode *talloc();
        char *strsave();
        int cond;

        if (p == NULL) {
                p = talloc();
                p->word = strsave(w);
                p->count = 1;
                p->left =
                p->right = NULL;
                }
        else if ((cond = stcmp(w,p->word)) == 0)
                p->count++;
        else if (cond < 0)
                p->left = tree(p->left,w);
        else
                p->right = tree(p->right,w);
        return(p);
        }

struct tnode *
talloc() { /* allocate a tnode */

        return (&tnarray[t++]);
        }

char *
strsave(w) char *w; { /* make a copy of string w */

        char *cp,*cps;

        cps =
        cp = &chrs[c];
        while (*cp++ = *w++)
                c++;
        c++;
        return (cps);
        }

treeprint(p) struct tnode *p; { /* print tree */

        if (p != NULL) {
                treeprint(p->left);
                treeprintt(p->right);
                }
        }

stcmp(s, u)
register char *s, *u;
{
        for( ; *s == *u; s++, u++)
                if (*s == '\0')
                        return(0);
                return(*s - *u);
}
```

```
[illegible]

start:
        [illegible]

                goto start;

                qstexc([illegible]);
                [illegible]
}

qsexc(i, j)
char *i, *j;
{
        register char *ri, *rj, c;
        register int n;

        n = qses;
        ri = i;
        rj = j;
        do {
                c = *ri;
                *ri++ = *rj;
                *rj++ = c;
        } while(--n);
}

qstexc(i, j, k)
char *i, *j, *k;
{
        register char *ri, *rj, *rk;
        register int c;
        int n;

        n = qses;
        ri = i;
        rj = j;
        rk = k;
        do {
                c = *ri;
                *ri++ = *rk;
                *rk++ = *rj;
                *rj++ = c;
        } while(--n);
}


ldiv(l,m,q)
{
}
stcmp(ss1, ss2)
char *ss1,*ss2;
{
        register char *s1, *s2;
        s1 = ss1; s2 = ss2;

        while (*s1 == *s2++)
                if (*s1++== '\0')
                        return(0);
        return(*s1 - *--s2);
}
```

```
/* benchmark program number 4 */

/* tty driver fragment */

#
struct  q_list  {
        int     q_count;
        char    *q_char;
        char    *q_free;
};
struct  packet  {
        char    *nextpkt;
        char    pdata[6];
};
#define CERASE  '#'
#define CEOT    004
#define CKILL   '@'
#define CQUIT   034
#define CINTR   0177
#define SIGINT  2
#define SIGQIT  3
#define ERROR   (-1)
#define NULL    0

int t_col;
#define CANBSIZ 132

char    *cbp1;
char    canonb[CANBSIZ];


main()
{
        extern char *cbp1;
        register int p;
        register int i;

        for( i = 0; i < 30000; i++) {
                cbp1 = &canonb[1];
                canon(p='t',canonb);
                canon(p,canonb);
                canon(p,canonb);
                canon(p,canonb);
                canon(p = CKILL,canonb);
                canon(p = 't', canonb);
                p = 0;
        }
}

ttyoutput(ac,queue) int ac; char *queue;
{
        register int c;
        int extern t_col;

        c = ac&0177;
        if (c=='\t') {
                do ttyoutput(' ',queue); while (t_col&07);
                return;
        }

        if(c == '\n') ttyoutput('\r',queue);
        while(putq(c,queue) == ERROR) wflushtty(queue);

        switch (c) {

        default:
                t_col++;
                break;

        case 010:
                if(t_col) t_col--;
                break;

        case '\n':
                t_col = 0;
                break;
        case '\r':
                ttyoutput(0177,queue);
                ttyoutput(0177,queue);
                ttyoutput(0177,queue);
                ttyoutput(0177,queue);
                t_col = 0;
        }
}

canon(ac,queue) int ac; char *queue;
{
        register char *bp;
        register int c;

        c = ac;
        bp = cbp1;
        if(bp[-1] != '\\') {
                switch(c) {
                        case CERASE:
                                if(bp > &canonb[1]) --bp;
                                break;
                        case CKILL:
                                bp = &canonb[1];
                                ttyoutput('\n',queue);
                                break;
                        default:
                                *bp++ = c;
                }
        }
        else    if(c == CERASE || c == CKILL) bp[-1] = c;
                else *bp++ = c;
        if(c == '\n' || c == CEOT || bp > &canonb[CANBSIZ]) {
                cbp1 = bp;
                bp = &canonb[1];
                while(bp < cbp1) putq(*bp++,queue);
                bp = &canonb[1];
        }
        cbp1 = bp;
}

putq(c, queue)
char c, *queue;
{
        return(0);
}

wflushtty(q)
char *q;
{}
```

```
/* benchmark program number 5 */

/* symbol table insert */

#
#define ERROR   (struct symbl *) -1
#define MAXHASH 8
#define NIDENT  8
#define OTHER   0
#define ALPHA   1
#define NUMERIC 3
#define ARITH   4
#define EFATL   0
struct symbl {
        char ident[NIDENT];
        int reloc;
        int sdind;
        int sclass;
        int stype;
        int value;
        struct symbl *next;
        struct symbl *prev;
        struct symbl *order;
};

struct symbl *fsym = 0;
int sdindex;
struct symbl *hashtb[MAXHASH], *lsym, undef;
int p2flg, slno;

main()
{
        register int i;
        struct symbl *addsym();
        for( i = 0; i < 30000; i++) {
                addsym("asymbol",2,5);
        }
        return;
}

struct symbl * addsym(name, val, rel)
char *name;
int val, rel;
{
        extern int sdindex;
        extern struct symbl *hashtb[];
        extern int p2flg, slno;
        extern struct symbl *fsym, *lsym, undef;
        register int i;
        struct symbl *alloc(), *pl;
        register struct symbl *p, *pn;
        char *c;
        if (*name == '\0') {
                error(EFATL,"syntax requires symbol",
                                        name, sdindex);
                return(&undef);
        }
        sdindex++;
        if ((p = alloc(sizeof *p)) == ERROR) {
                p2flg = 1;
                slno = 0;
                c = name;
                while (type(*c) & 1)
                        c++;
                *c = 0;
                error(EFATL, "%s - too many (%d) symbols",
                                        name, sdindex);
                exit1(0);
        }
        for (i = 0; i < NIDENT; i++) {
                p->ident[i] = *name;
                if ((type(*name++) & 1) != 1) break;
        }
        p->ident[i] = '\0';
        p->value = val;
        p->reloc = rel;
        p->sdind = 0;
        p->stype = 0;
        p->sclass = 0;
        p->next = 0;
        i = hash(p->ident);
        p->order = hashtb[i];
        hashtb[i] = p;
        if (fsym == 0) {
                fsym = p;
                lsym = p;
                p->next = 0;
                p->prev = 0;
                return(p);
        }
        lsym->next = p;
        p->prev = lsym;
        lsym = p;
        return(p);
}
hash(ap)
char *ap;
{
        register char *p;
        register int h;
        p = ap;
        h = 0;
        while (type(*p) & 1) {
                h = *p++;
                if (p >= ap + NIDENT - 1)
                        break;
        }
        return(h & (MAXHASH - 1));
}

error(i, s1, s2, j)
char *s1, *s2;
{}
struct symbl *
alloc(i)
{
        return(&undef);
}

type(c)
char c;
{
        if (c) return(1);
        return(0);
}
exit1(status)
{}
```

```
/* benchmark program number 6 */

/* release buffer (UNIX) */

struct buf
{
        int     b_flags;        /* see defines below */
        struct  buf *b_forw;    /* headed by devtab of b_dev */
        struct  buf *b_back;    /*  "  */
        struct  buf *av_forw;   /* position on free list, */
        struct  buf *av_back;   /*     if not BUSY*/
        int     b_dev;          /* major+minor device name */
        int     b_wcount;       /* transfer count (usu. words) */
        char    *b_addr;        /* low order core address */
        char    *b_xmem;        /* high order core address */
        char    *b_blkno;       /* block # on device */
        char    b_error;        /* returned after I/O */
        char    *b_resid;       /* words not transferred after error */
};

struct dummy {
        int integ;
} stuff;
struct dummy *PS = &stuff;
struct buf bfreelist;

#define B_WRITE 0
#define B_READ  01
#define B_DONE  02
#define B_ERROR 04
#define B_BUSY  010
#define B_PHYS  020
#define B_MAP   040
#define B_WANTED 0100
#define B_AGE   0200
#define B_ASYNC 0400
#define B_DELWRI 01000

main()
{
        long i;
        struct buf *rbp;
        rbp = &bfreelist;
        rbp->av_forw = &bfreelist;
        rbp->av_back = &bfreelist;
        rbp->b_forw = &bfreelist;
        rbp->b_back = &bfreelist;
        for( i = 0; i < 100000L; i++) {
                bfreelist.b_flags = (B_WANTED | B_AGE);
                brelse(&bfreelist);
        }
}

brelse(bp)
struct buf *bp;
{
        register struct buf *rbp, **backp;
        register int sps;

        rbp = bp;
        if (rbp->b_flags&B_WANTED)
                wakeup(rbp);
        if (bfreelist.b_flags&B_WANTED) {
                bfreelist.b_flags &= ~B_WANTED;
                wakeup(&bfreelist);
        }
        if (rbp->b_flags&B_ERROR) {
                rbp->b_dev |= 0200;
                rbp->b_flags &= ~(B_ERROR|B_DELWRI);
        }
        sps = PS->integ;
        spl6();
        if(rbp->b_flags & B_AGE) {
                backp = &bfreelist.av_forw;
                (*backp)->av_back = rbp;
                rbp->av_forw = *backp;
                *backp = rbp;
                rbp->av_back = &bfreelist;
        } else {
                backp = &bfreelist.av_back;
                (*backp)->av_forw = rbp;
                rbp->av_back = *backp;
                *backp = rbp;
                rbp->av_forw = &bfreelist;
        }
        rbp->b_flags &= ~(B_WANTED|B_BUSY|B_ASYNC|B_AGE);
        bfreelist.b_wcount++;
        PS->integ = sps;
}

wakeup(bp)
struct buf *bp;
{}
spl6()
{}
```

```
/* benchmark program number 7 */

/* statistics */

int array[10] = {
        500,
        1000,
        2000,
        4000,
        100,
        200,
        400,
        800,
        1600,
        3200
};

main()
{
        register unsigned i;

        for ( i = 0; i < 30000; i++
                stat(array,10);
        for ( i = 0; i < 30000; i++
                stat(array,10);
        return;
}

stat(arr,nel)
int     *arr;
int     nel;
        {
        register int    i;
        register int sum,mean;
        int getmean();

        sum = 0;
        for(i=0;i<nel;i++)
                sum += arr[i];
        mean = getmean(arr,nel,sum);
        getmedian(arr,nel,sum);
        }


int     getmean(arr,nel,sum)
int     *arr,sum;
        {
        register int    i;
        register int    mean;

        mean = 0;
        for(i=0;i<nel;i++)
                mean += arr[i] * i;
        mean /= sum;
        return(mean);
        }


getmedian(arr,nel,sum)
int     *arr,sum;
        {
        register int    i;
        register int    part_sum;

        part_sum = 0;
        for(i=0;i<nel;i++)
                {
                if((part_sum += arr[i]) > sum>>2)
                        break;
                }
        }
```

Appendix B - BENCHMARK EXECUTION TIMES

	Amdahl	3033	370/168	Univac	3B-20	VAX	1170	Eclipse	Tandem	H6080	Inter	11/45	*8086* (5MHz)	11/34	11/23	MAC 8	LSI11
b1	10.51	11.32	29.62	34.3	52.1	66.1	82.9	47.0	78.8	97.4	104.5	170.3	231	240.6	258.1	358	-
b2	2.83	3.16	7.37	14.0	18.6	17.5	21.2	19.0	26.0	32.1	34.0	48.7	68	68.4	71.8	125	-
b3	4.68	5.48	11.89	23.7	36.5	33.6	37.2	44.0	59.1	65.9	62.8	91.0	132	132.3	140.0	230	304
b4	2.05	22.40	5.93	9.2	13.5	15.4	15.9	18.0	21.5	21.9	27.1	38.4	51	54.9	57.5	100	130
b5	2.44	2.62	6.30	11.5	13.7	14.1	16.6	19.5	22.5	29.1	26.7	40.1	55	57.7	59.9	101	137
b6	1.33	1.55	3.90	7.2	8.8	9.6	10.9	11.0	14.8	12.1	16.3	25.3	33	34.6	37.1	55	85
b7	2.32	2.42	4.63	9.3	22.2	12.5	13.9	25.0	20.9	18.3	24.7	34.9	61	56.4	66.4	426	160

Table B1
Execution Time In Seconds

	Amdahl	3033	370/168	Univac	3B-20	VAX	11/70	Eclipse	Tandem	H6080	Inter	11/45	8086 (5MHz)	11/34	11/23	MAC 8 (2MHz)	LSI11
b1	7.89	7.32	2.80	2.42	1.59	1.25	1.00	1.76	1.05	.85	.79	.49	.359	.345	.321	.232	-
b2	7.49	6.71	2.88	1.51	1.14	1.21	1.00	1.12	.82	.66	.62	.44	.312	.310	.295	.170	-
b3	7.95	6.79	3.13	1.57	1.02	1.11	1.00	.85	.63	.56	.59	.41	.282	.281	.266	.162	.122
b4	7.76	6.63	2.68	1.73	1.18	1.03	1.00	.88	.74	.73	.59	.41	.312	.290	.277	.159	.122
b5	6.80	6.34	2.64	1.44	1.21	1.18	1.00	.85	.74	.57	.62	.41	.302	.288	.277	.164	.121
b6	8.20	7.03	2.80	1.51	1.24	1.14	1.00	.99	7.04	.90	.67	.43	.330	.315	.294	.198	.128
b7	5.99	5.74	3.00	1.49	.63	1.11	1.00	.56	.67	.76	.56	.40	.228	.246	.209	.033	.087

Table B2
Execution Speed - Normalized to PDP 11/70

Appendix C - BENCHMARK SIZES

	VAC	*3B-20*	*PDP11*	*Eclipse*	*8086*	*MAC8*	*Tandem*	*Inter.*	*IBM 370*	*Univac*
b1	60	84	86	70	102	94	194	104	230	320
b2	392	428	414	340	424	459	400	736	886	1328
b3	424	500	530	606	550	510	684	816	982	1620
b4	436	468	462	454	488	547	554	834	964	1300
b5	348	436	420	490	476	486	500	788	932	1520
b6	228	244	260	278	250	239	288	460	508	1036
b7	172	180	214	252	218	287	254	372	472	724

Table C1: Text Size in Bytes

	VAX	*3B-20*	*PDP11*	*Eclipse*	*8086*	*MAC8*	*Tandem*	*Inter*	*IBM 370*	*Univac*
b1	0	0	0	0	0	0	0	0	64	36
b2	35768	35764	25758	26272	25758	25757	25764	36280	35870	37148
b3	68	72	62	62	62	61	64	72	172	176
b4	140	156	148	152	136	158	138	136	288	236
b5	164	160	110	112	110	109	114	168	288	300
b6	60	56	28	28	28	21	30	64	176	136
b7	40	40	20	20	20	20	22	40	128	112

Table C2: Data Size in Bytes

	VAX	*3B-20*	*PDP11*	*Eclipse*	*8086*	*MAC8*	*Tandem*	*Inter*	*IBM 370*	*Univac*
b1	0.698	.977	1.000	0.814	1.186	1.093	1.209	2.256	2.674	3.721
b2	0.947	1.034	1.000	0.821	1.024	1.109	0.966	1.778	2.140	3.208
b3	0.800	.943	1.000	1.143	1.038	0.962	1.291	1.540	1.853	3.057
b4	0.944	1.013	1.000	0.983	1.056	1.184	1.199	1.805	2.087	2.814
b5	0.829	1.038	1.000	1.167	1.133	1.157	1.190	1.876	2.219	3.619
b6	0.877	.939	1.000	1.069	0.962	0.919	1.108	1.769	1.954	3.985
b7	0.804	.841	1.000	1.178	1.019	1.341	1.187	1.738	2.206	3.383

Table C3: Text Size - Normalized to PDP 11

	VAX	*3B-20*	*PDP11*	*Eclipse*	*8086*	*MAC8*	*Tandem*	*Inter*	*IBM 370*	*Univac*
b1	1.00	1.0000	1.000	1.000	1.000	1.000	1.000	1.000	∞	∞
b2	1.389	1.389	1.000	1.020	1.000	1.000	1.000	1.408	1.393	1.442
b3	1.097	1.161	1.000	1.000	1.000	0.984	1.032	1.161	2.774	2.839
b4	0.946	1.054	1.000	1.027	0.919	1.068	0.932	0.919	1.946	1.595
b5	1.491	1.455	1.000	1.018	1.000	0.991	1.036	1.527	2.618	2.727
b6	2.143	2.000	1.000	1.000	1.000	0.750	1.071	2.286	6.286	4.857
b7	2.000	2.000	1.000	1.000	1.000	1.000	1.100	2.000	6.400	5.600

Table C4: Data Size - Normalized to PDP 11

Appendix D

COMPARISON OF ALTERNATE COMPILERS FOR PDP 11 AND MAC 8

	*Relative Speed**	*Relative Text Size†*	*Relative Data Size†*
b1	-1.7%	+34.9%	0
b2	-0	+5.3%	0
b3	-1.3%	+5.3%	0
b4	-3.1%	+6.5%	-8.1%
b5	-2.4%	+10.5%	0
b6	-2.7%	+6.9%	0
b7	-.1%	+9.4%	0

Table D1
Comparison of Portable PDP 11 Compiler
With Standard Version

	*Relative Speed**	*Relative Text Size†*	*Relative Data Size†*
b1	0	-1.1%	0
b2	0	-1.1%	0
b3	+3.9%	+10.8%	+1.6%
b4	0	-2.2%	0
b5	-4.0%	-3.9%	0
b6	+3.6%	+8.0%	+28.6%
b7	-3.5%	-10.8%	0

Table D2
Comparison of Old MAC 8 Compiler
with New Version

---

* Positive percent values indicate faster execution

† Positive percent values indicate larger size

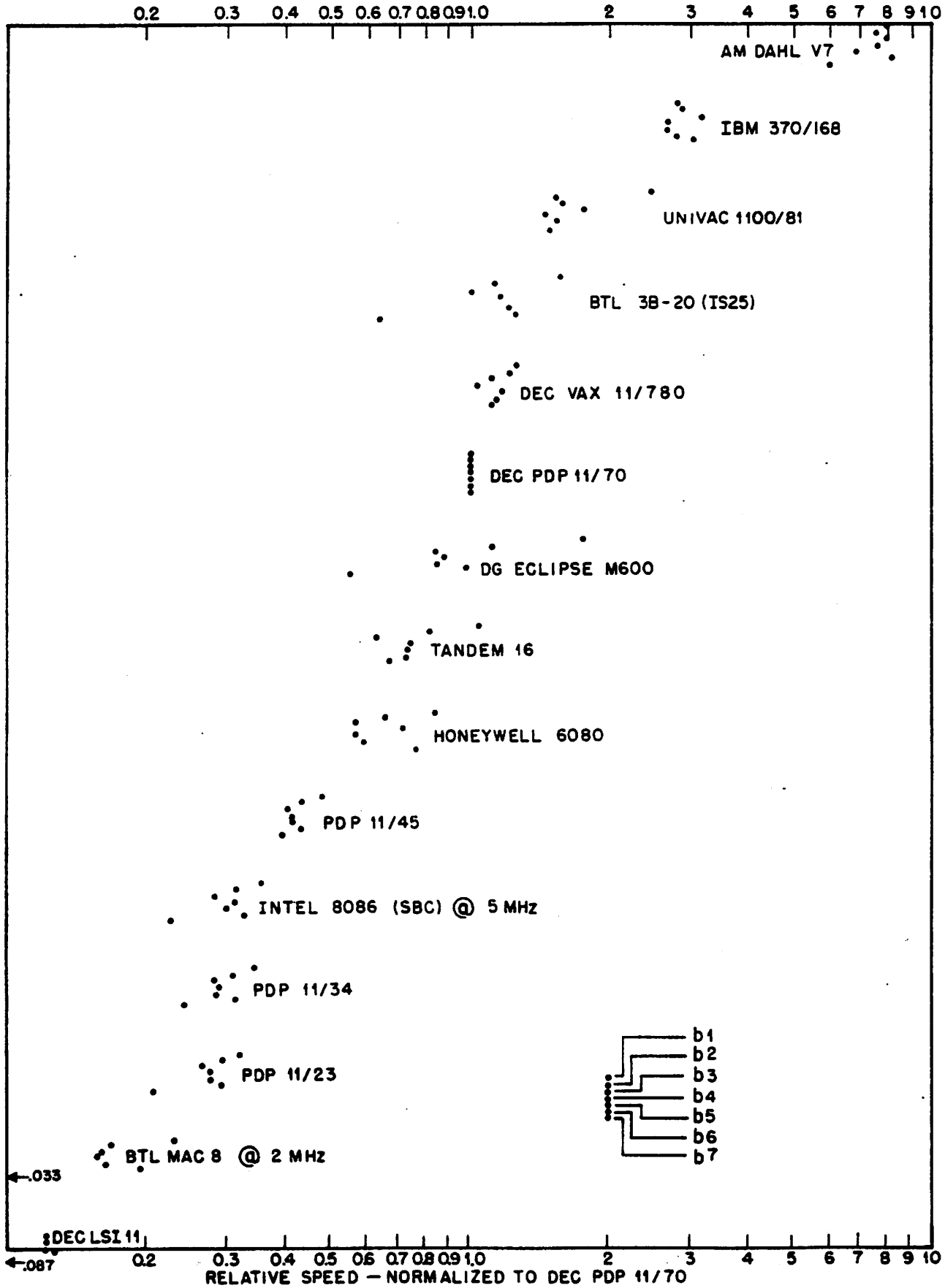

FIG.1 BENCHMARK EXECUTION SPEED

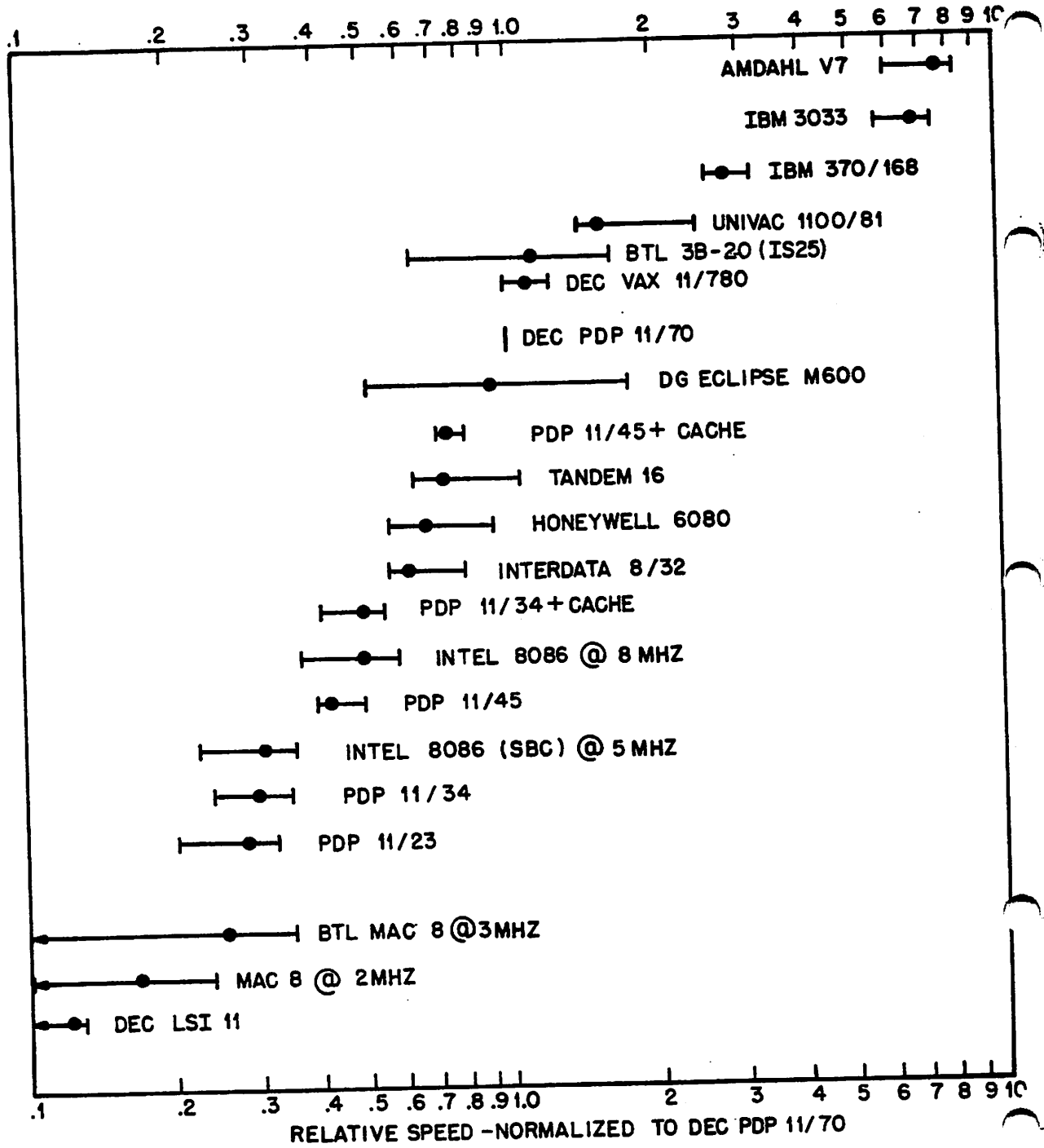

FIG 2 RANGE OF NORMALIZED EXECUTION SPEED

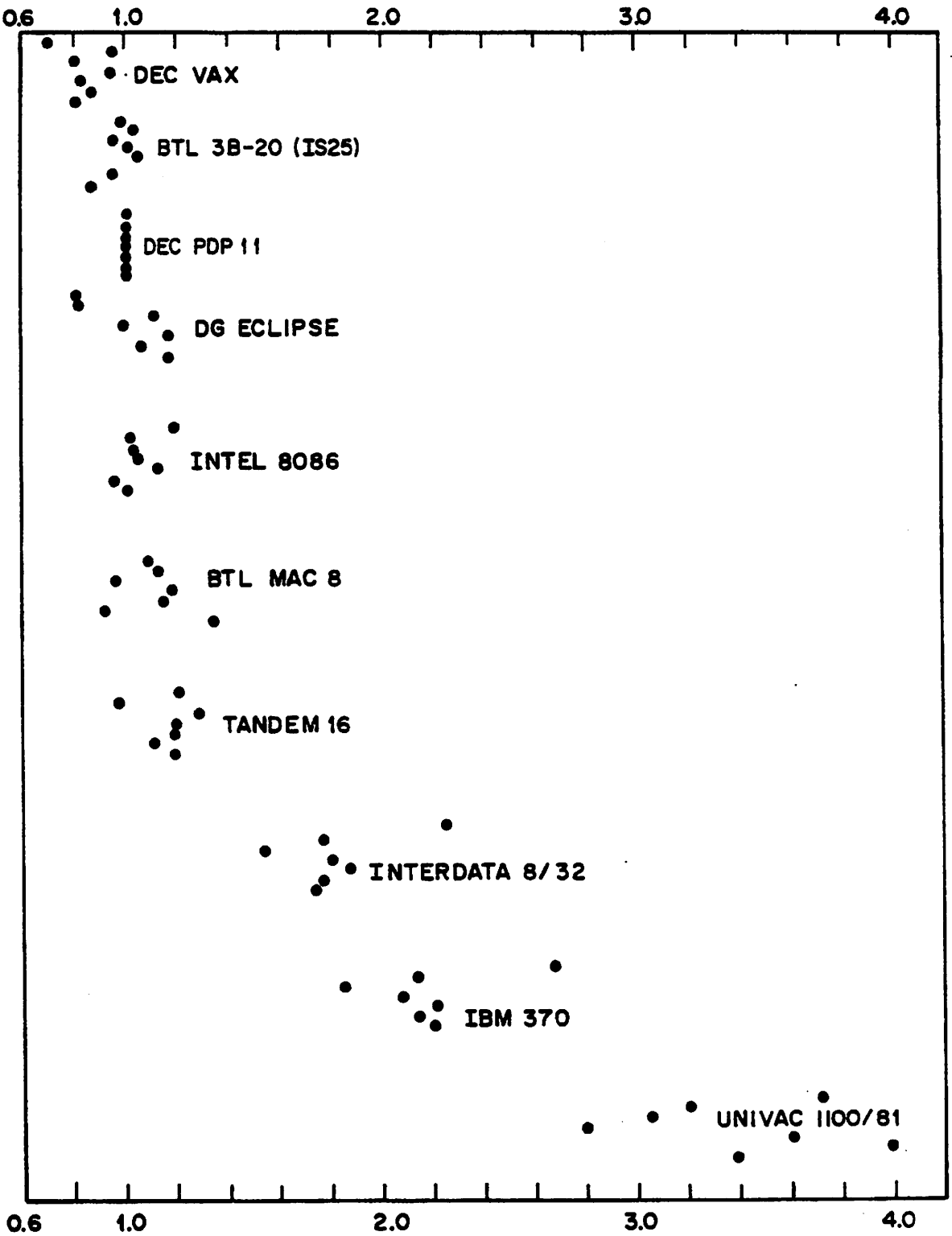

FIG. 3 BENCHMARK TEXT SIZE

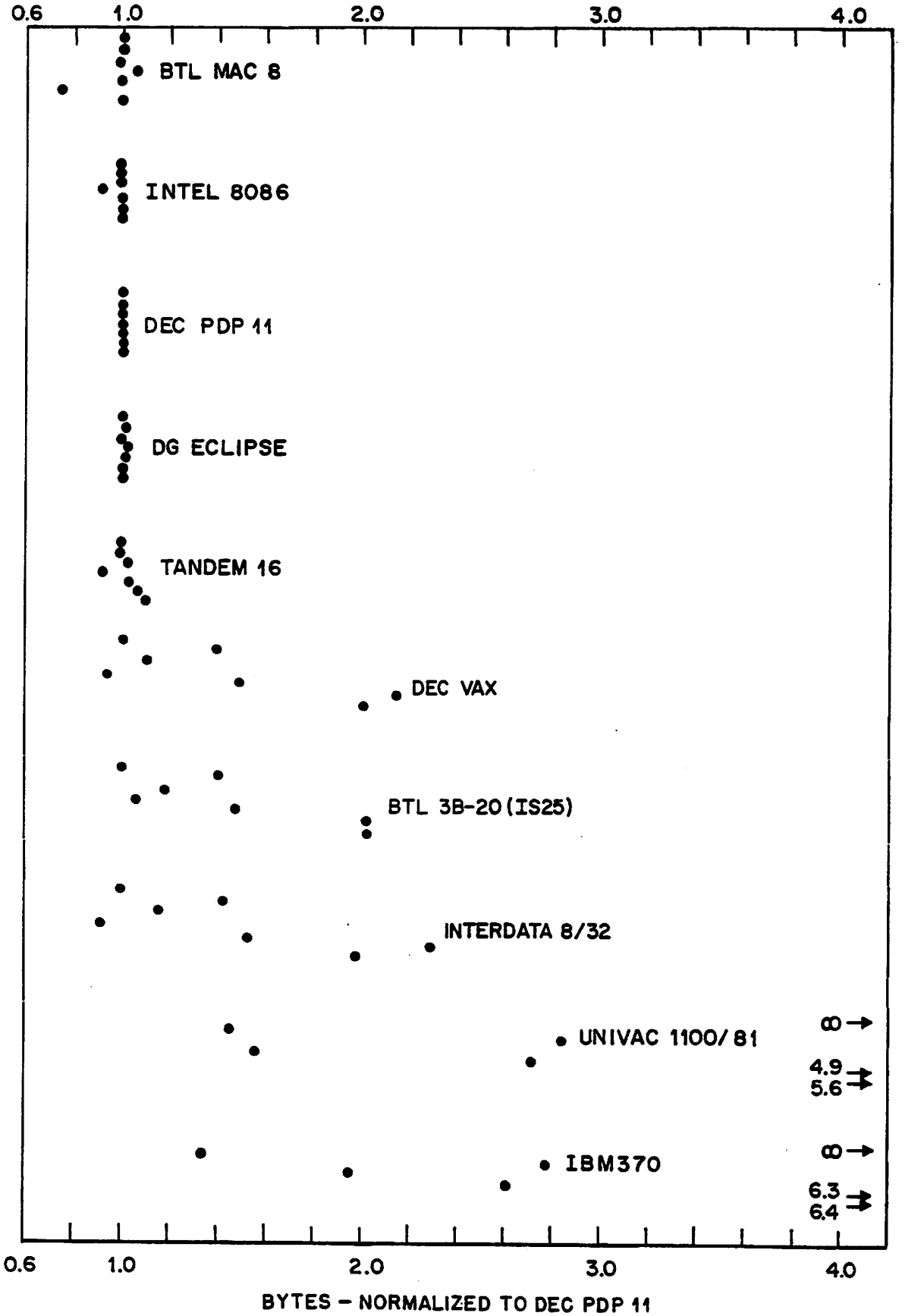

FIG. 4 BENCHMARK DATA SIZE

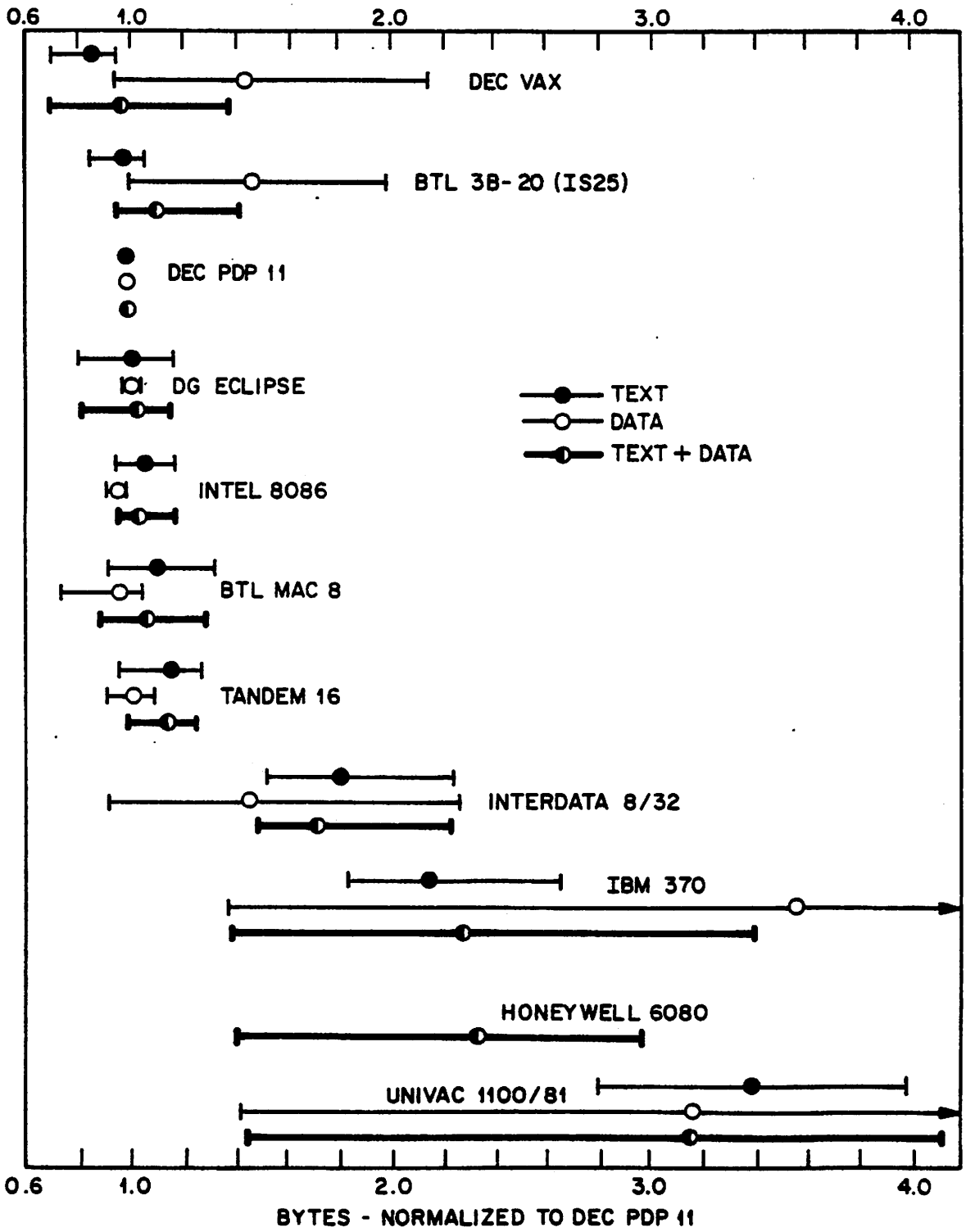

FIG. 5 BENCHMARK SIZE