Computer Systems

G. Bell, D. Siewiorek, and S.H. Fuller, Editors

Empirical Evaluation of Some Features of Instruction Set Processor Architectures

Åmund Lunde
Carnegie-Mellon University

This paper presents methods for empirical evaluation of features of Instruction Set Processors (ISPs). ISP features are evaluated in terms of the time used or saved by having or not having the feature. The methods are based on analysis of traces of program executions. The concept of a register life is introduced, and used to answer questions like: How many registers are used simultaneously? How many would be sufficient all of the time? Most of the time? What would the overhead be if the number of registers were reduced? What are registers used for during their lives? The paper also discusses the problem of detecting desirable but non-existing instructions. Other problems are briefly discussed. Experimental results are presented, obtained by analyzing 41 programs running on the DECsystem10 ISP.

Key Words and Phrases: computer architecture, program behavior, instruction sets, opcode utilization, register structures, register utilization, simultaneous register lives, instruction tracing, execution time

CR Categories: 6.20, 6.21, 6.33

This work was supported in part by the Advanced Research Projects Agency of the office of the Secretary of Defense (F44620-73-C-0074) monitored by the Air Force Office of Scientific Research, in part by The Norwegian Research Council for Science and the Humanities (Norges Almenvitenskapelige Forskningsråd).

Author's present address: EDB-sentret, Universitetet i Oslo; P.O. Box 1059, Blindern, Oslo 3, Norway.

* Notes for this article appear on p. 152.

1. Introduction

A quick survey of current computers reveals a great variation in the structure of Instruction Set Processors.[1]* This observation is true even for computers intended for the same general market. Current ISPs designed with the scientific market in mind, for example, have word lengths ranging from 24 to 64 bits; the number of different instructions varies from about 70 to over 400; register structures span the area from one accumulator plus a few index registers, through designs with 8 to 24 general or specialized registers, to designs with up to 64 registers, again relatively general. A natural conclusion from such a survey is that very little is known about the optimal structure of ISPs. Further study reveals that very little has been published about measuring techniques or other methods designed to obtain such knowledge.

This paper presents a step towards the development of such measuring techniques. It describes methods designed to study the detailed behavior of programs as they are executing on some ISP. Experimental results are presented which reflect the behavior of one particular set of programs on one particular ISP.

The need for such measures and their utility is vindicated by the results found by the designers of the Burroughs B1700 central processor [14, 15]. These results clearly show the dependence of program efficiency on a good ISP.

Previous authors have measured the frequency of execution of the individual instructions or groups of instructions [7 (The Gibson mix), 8, 4, 12, and 9]. Only a few more comprehensive studies are known to this author: Foster et al. [5, 6] have developed measures of opcode utilization and studied alternative encodings of the opcodes into fewer bits than those required by a conventional encoding. Similar results are presented by Wilner [15]. Winder [16, 17] has gathered miscellaneous statistics on ISP usage. Alexander [1] has made extensive study of how one particular programming language uses ISP features.

None of the above studies report on ISP behavior reflecting more than two or three consecutive instructions. Also, register use is barely touched upon. The methods described in this paper improve this situation. They are only to a small extent, or not at all, restricted to the study of a small fixed length sequence of instructions. On the contrary, we may follow a phenomenon for as many instructions as seems relevant, while at the same time retaining full knowledge about every instruction executed.

2. Basic Methodology

The basic idea of the methods is to analyze traces of a representative set of programs, the *subject set*, written as these are executed by an interpreter for the ISP being studied. Information is recorded for every instruction executed by the *subject program*. The major advantages of this approach are:

—ISP behavior may be studied in great detail.
—The methods are not restricted to special languages or compilers.
—Analysis programs are easily written, and programs for new analysis methods may be developed after the data have been collected.
—All analyses of the same program (trace) see exactly the same instruction stream; hence the results are not perturbed by random influences caused by external devices or by multiprogramming of jobs.

Each individual analysis, therefore, studies the behavior of a user program running on the user ISP and the suitability of this ISP to that particular program, as opposed to studying the suitability of the full ISP to a collection of multiprogrammed programs. For the latter purpose a device is needed to trace executive mode programs, probably at full speed. Statistical validity comes from studying many programs individually.

The methods are easily modifiable to apply to all register structured ISPs, and to some extent even to stack ISPs or other ISPs. The specific results obtained are, however, strongly dependent on the structure of the ISP analyzed. The extent to which they can be applied to similar ISPs depends on the degree of similarity and on the result in question. On the other hand, the results are relatively independent of technology; hence they may be used by ISP architects to compare the cost/utility ratios of different structures across different technologies.

Our methods evaluate ISP features in terms of their associated time cost, i.e. the change in execution time or instruction count caused by including or removing the feature. Of these, the instruction count is most independent of technology, but it hides the fact that certain operations take a longer time than others, regardless of technology. Hence execution time is also computed in some cases by summing the individual instruction execution times.

Other relevant costs are the space occupied in primary memory by program and data, and the cost of designing, coding, and debugging programs. Both of these are highly dependent on the ISP, and are as important to a good design as the time cost. They are not, however, measured by our methods, but should be otherwise measured or estimated by the ISP architect before he makes his decisions.

3. Experimental Environment

3.1 ISP Studied

The emphasis of our experimental work was on studying the methods, and estimating the dependence of the results on the major parameters of the subject set. In order to reduce the work, experiments were performed on one ISP only: the DECsystem10 (KA-10). The structure of this ISP is unusually general; some of its properties are:

(1) It has a large instruction repertoire of about 420 user instructions including:
—A rich set of instructions for arithmetic and bitwise comparison. These compare memory, register, immediate or implicit (0) operands, and all 6 arithmetic conditions are available.
—Programmer defined stacks.
—Three different mechanisms for subroutine calls.
—All 16 Boolean functions of two variables.
—Immediate operands and several result destinations (register, memory or both) for arithmetic and logic instructions.
—31 Monitor calls and 32 user-definable trap instructions (UUOs).

(2) The register structure is equally general. The 16 registers are part of the memory address space. All of them may be used for all standard purposes with only insignificant exceptions.

(3) Indirection may be carried to any depth, with indexing at each level.

Hence this ISP is a good starting point for detection of unnecessary generality or superfluous features. This is vindicated by our results reported here and in [10]. We did, however, also discover features which we would like to see incorporated into this ISP. Some of these have, in fact, been included in later processors of the DECsystem10 family.

3.2 Subject Set

Another restriction on our experiments was that we analyzed programs only from a scientific environment. On the other hand, we tried to choose a subject set which would show the influence of the choice of algorithm, programming language, and compiler.

Hence one part of our subject set consisted of six algorithms from *Collected Algorithms of the ACM* (CALGO). These were selected to contain as many as possible of the commonly used program structures, and to give a reasonable covering of the modified SHARE classification for algorithms. Each of these algorithms was coded in four languages: ALGOL, BASIC, BLISS, and FORTRAN. Two different FORTRAN systems were used. BLISS [18] is a high-

Table I. Distribution of Lives by Lifelength, Unweighted Sum of All Programs—Logarithmic Table Division.

Length	No. of lives	Fraction	Cum. fraction
1 - 1	174927	0.09	0.09
2 - 3	728346	0.38	0.48
4 - 7	547072	0.29	0.77
8 - 15	252508	0.13	0.90
16 - 31	116404	0.06	0.96
32 - 63	41673	0.02	0.98
64 - 127	17790	0.01	0.99
128 - up	15603	0.01	1.00
Total number of lives		1894323	

level language for systems programming. The other languages should be well known. The six algorithms were:

No. 30: Polynomial roots by Bairstow's method (Bairstow)

No. 43: Linear equations by Crout's method (Crout)

No. 113: Treesort

No. 119: PERT

No. 257: Numerical integration by Håvies method (Håvie)

No. 355: Generation of Ising configurations (Ising).

The latter could not easily be coded in BASIC, hence that version was omitted.

To investigate the influence of coding style, we included an algorithm for polynomial interpolation (Aitken) as coded in BLISS by four different programmers, plus a carefully tuned version of this algorithm. These are denoted E (efficient), B, A, L, and G. A medium-sized numeric FORTRAN program, SEC, was also analyzed. Again both FORTRAN systems were used. Finally we analyzed the five compilers used for the CALGO set: these are denoted ALGOL, BASIC, BLISS, FORFOR, and FORTEN. ALGOL, BASIC, and FORFOR are written in MACRO (the assembly language), BLISS and FORTEN are written in BLISS.

Thus our final subject set consisted of 41 programs, comprising about 5.3 million instructions or 16.8 seconds of CPU time. 38 of these were written in high-level languages. One would a priori expect that such programs do not make as good use of the ISP as do assembly language programs. On the other hand, we are already restricted to the user ISP, and certainly the majority of user programs are written in high-level languages.

4. Register Structure

Methods were developed for two problems connected with register structure:

—How many registers are used efficiently?
—What is the need for generality of registers?

Both are attacked through the concept of a *register life*. A register life consists of all activity associated with a given register during a period of time starting with a load into that register, and terminating with the last use of the register before the next load into it. A register is *loaded* when a new value is brought into it which is unrelated to its old value. Use of the old value during address calculation is not considered a relation in this context.

The start of a register life is analogous to the "open effects" situation described by Tjaden and Flynn [13]. The terms *live* and *dead* now have obvious meanings. A register is *dormant* when it is live but not used. The resolution of our time measure is one instruction. Hence two successive lives of the same register may overlap if the old value is used to load the new one. Usually there will be a dead period between two consecutive lives of a register. Finally we note that for a machine with several registers, any number of them may be live at any given time.

It seems unreasonable to use these concepts unmodified for registers which have long dormant periods. Hence the results below were obtained under the assumption that a register was dead when it had been dormant for 200 or more instructions. This is discussed further below.

4.1 Analysis Program

The analysis program detects register lives, classifies them according to the operations they contain, and finds the number of live registers at each point in time during program execution.

As the trace is read, one can not in general tell whether a register is dead or live until the next LOAD into it is encountered. This may be any length of time after the register actually died. Hence the analysis of register usage is a two-phase process. In the first phase register lives are detected and classified. Phase I also writes a file of descriptions of each life which is used by phase II. Phase II then finds how many registers were live at each point in time, and computes various results based on this.

In the analysis a relatively fine classification was used for the lives. For purposes of presentation the following seven classes were considered:

—All lives (the total class—TOT).
—Lives used for indexing (INX).
—Lives used for temporary storage only (TMP).
—The four classes defined by the "strongest" arithmetic used:
 No arithmetic (NOA).
 Fixed-point additions and subtractions (FAS).
 Fixed-point multiplications and divisions (FMD).
 Floating-point operations (FLO).

The latter four classes are disjoint and their union is the class TOT.

4.1.1 Phase I. As the trace is read, phase I keeps track of the times of the most recent load and the most recent use of each register. Hence each time a register

Table II. Average Lifelength in Instructions.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		12.3	12.3	11.2	12.9	12.9	12.3
Crout		13.6	11.3	18.2	15.1	15.9	14.8
Treesort		6.1	11.9	9.0	4.2	5.8	7.4
PERT		10.9	11.4	8.4	5.0	7.9	8.7
Håvie		16.6	11.2	13.5	14.3	20.0	15.1
Ising		16.5	-	9.7	5.5	9.2	10.2
Secant		-	-	-	8.1	9.6	8.9
Programmer:		E	B	A	G	L	Mean
Aitken		14.3	14.7	13.0	8.9	11.9	12.6
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		17.4	23.8	9.7	14.9	11.4	15.4
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	18.7	12.7	11.6	11.8	9.3	11.6	11.9

Table III. Usages per Register Life.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		4.6	3.6	4.6	4.6	4.4	4.4
Crout		3.8	3.7	6.6	3.7	3.9	4.3
Treesort		3.9	3.5	4.8	2.9	2.9	3.6
PERT		4.1	3.4	3.8	3.1	3.2	3.5
Håvie		4.4	3.7	5.8	5.4	5.2	4.9
Ising		4.0	-	4.5	3.1	3.3	3.7
Secant		-	-	-	3.8	3.8	3.8
Programmer:		E	B	A	G	L	Mean
Aitken		5.4	5.5	5.2	3.9	5.2	5.0
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		3.7	6.0	3.5	4.1	3.2	4.1
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	4.6	4.1	3.6	4.8	3.8	3.8	4.2

Table IV. Average Number of Live Registers, Computed as ⟨Sum of Lifelengths⟩/⟨Program Length⟩.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		4.4	3.6	3.8	3.8	4.0	3.9
Crout		6.0	3.7	4.7	6.4	6.0	5.4
Treesort		2.5	3.5	3.1	1.8	2.7	2.7
PERT		4.2	3.6	3.6	2.0	3.0	3.3
Håvie		6.0	3.5	3.7	3.6	4.5	4.3
Ising		6.5	-	3.6	1.9	3.2	3.8
Secant		-	-	-	3.0	3.4	3.2
Programmer:		E	B	A	G	L	Mean
Aitken		4.4	4.5	4.2	3.9	3.7	4.1
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		5.1	4.5	3.6	5.1	4.2	4.5
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	4.9	4.9	3.6	3.9	3.2	3.8	3.9

Table V. Memory References per Instruction Excluding Instruction Fetches.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.61	.52	.50	.62	.60	.57
Crout		.44	.59	.50	.55	.64	.54
Treesort		.65	.50	.51	.57	.63	.57
PERT		.51	.47	.53	.69	.63	.57
Håvie		.30	.45	.31	.44	.35	.37
Ising		.40	-	.60	.67	.60	.57
Secant		-	-	-	.60	.53	.57
Programmer:		E	B	A	G	L	Mean
Aitken		.45	.48	.52	.50	.53	.50
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.40	.32	.45	.42	.40	.40
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.38	.49	.51	.48	.59	.57	.51

Table VI. Register References per Instruction.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		1.66	1.05	1.58	1.35	1.37	1.40
Crout		1.67	1.21	1.67	1.56	1.46	1.51
Treesort		1.62	1.04	1.65	1.28	1.32	1.38
PERT		1.58	1.05	1.61	1.25	1.22	1.34
Håvie		1.57	1.14	1.61	1.36	1.16	1.37
Ising		1.58	-	1.66	1.11	1.13	1.37
Secant		-	-	-	1.39	1.33	1.36
Programmer:		E	B	A	G	L	Mean
Aitken		1.66	1.67	1.69	1.69	1.64	1.67
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		1.09	1.13	1.32	1.39	1.17	1.22
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	1.20	1.61	1.10	1.59	1.33	1.28	1.40

Table VII. Fraction of Lives with No Arithmetic.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.213	.637	.574	.494	.470	.478
Crout		.528	.716	.214	.349	.440	.449
Treesort		.315	.686	.257	.784	.565	.521
PERT		.597	.735	.547	.457	.416	.550
Håvie		.628	.680	.482	.496	.412	.540
Ising		.695	-	.620	.744	.622	670
Secant		-	-	-	.263	.266	.265
Programmer:		E	B	A	G	L	Mean
Aitken		.317	.390	.402	.475	.391	.395
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.844	.744	.921	.802	.886	.839
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.797	.496	.691	.498	.512	.456	.538

is loaded, the endpoints of its previous life are immediately available. For each register life, phase I determines its class, and also the number of references to it. Finally phase I computes the total number of register references and memory references. The data items written on the file for phase II contain most of this information, together with the register name.

Some results from phase I are given in Tables I through XII. Results are given for each individual program, as well as the averages for each algorithm, for all the compilers, and for all programs written in each language. All the programs are equally weighted in these averages.

We note that most lives (68% of the total) are between 2 and 7 instructions long. Only 4% are 32 instructions or longer. For each individual program over half the lives are less than 8 instructions long. Only 3 programs have more than 10% of their lives 32 instructions or longer. The average lifelength is 11.9 instructions, but ranges from 4 to 24 instructions for the individual programs. The average number of references to a life is 4.2, it ranges between 3 and 7 for the individual programs. The average number of simultaneously live registers ranges between 2 and 6. Operands, including indices and nominators (indirect addresses), are found in registers 2 to 4 times as often as in primary memory.

The classes FLO and FMD are significant only for those algorithms that use floating-point arithmetic, or where FMD arithmetic is used to access data. This is as one would expect. Even for highly numeric programs at most 50% of the lives are in class FLO, less than

Table VIII. Fraction of Lives with Fixed Point Add/Subtract.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.504	.106	.054	.118	.141	.185
Crout		.304	.009	.096	.186	.122	.143
Treesort		.355	.103	.710	.208	.056	.286
PERT		.380	.122	.397	.516	.552	.393
Håvie		.278	.085	.149	.123	.156	.158
Ising		.300	-	.373	.250	.370	.323
Secant		-	-	-	.359	.303	.331
Programmer:		E	B	A	G	L	Mean
Aitken		.210	.202	.302	.423	.389	.305
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.130	.234	.074	.190	.108	.147
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.185	.354	.085	.268	.251	.243	.245

Table IX. Fraction of Lives with Fixed Point Multiply/Divide.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.009	.001	.018	.042	.019	.018
Crout		.006	.064	.433	.156	.142	.160
Treesort		.317	.000	.011	.000	.370	.140
PERT		.002	.000	.004	.006	.006	.004
Håvie		.002	.001	.031	.018	.015	.013
Ising		.006	-	.007	.006	.008	.007
Secant		-	-	-	.175	.199	.187
Programmer:		E	B	A	G	L	Mean
Aitken		.000	.000	.000	.000	.085	.017
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.026	.019	.005	.009	.008	.013
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.018	.057	.013	.046	.058	.108	.054

Table X. Fraction of Lives with Floating Point Arithmetic.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.274	.256	.354	.347	.369	.320
Crout		.163	.211	.257	.306	.296	.247
Treesort		.014	.211	.022	.008	.009	.053
PERT		.021	.143	.053	.021	.026	.053
Håvie		.092	.233	.339	.363	.418	.289
Ising		.000	-	.000	.000	.000	.000
Secant		-	-	-	.203	.232	.218
Programmer:		E	B	A	G	L	Mean
Aitken		.473	.408	.296	.102	.136	.238
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.000	.003	.000	.000	.000	.001
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.001	.094	.211	.188	.178	.193	.162

40% in all but two programs. In spite of the fact that all variables in BASIC are floating point, the percentage of FLO lives in the BASIC programs is never above 25.

For the classes FAS and NOA, the dependence on language is larger than the dependence on algorithm. This is in particular true for ALGOL and BASIC, which enforce a stronger regimen on programs than do the other languages.

Between 18% and 68% of the lives, 39% on the average, are used for indexing.

4.1.2 Phase II. Phase II reads the file written by phase I in reverse order, and simulates a backwards execution of the subject program. Initially the descriptions of the last lives of each register are read. For each register the program keeps the description of one life, viz. that which is now valid, or will next be valid, during the backwards simulation. The loading and final uses of each register are entered in a list sorted by decreasing time. This list is processed in order, and a counter of live registers is suitably updated.

Each time a loading use of a register is processed, all information about that life may be discarded. The program is then ready to receive the description of the previous (at execution) life for that register. This description was written by phase I as it processed the same load instruction which is now being processed by phase II. Hence the desired data item is in the correct position to be read off the file.

We now know exactly how many registers were live at each point in time, and the fraction of the total time when exactly N registers were live can easily be computed for each N. Since the usage class was written on the intermediate file, this analysis may be done simultaneously for any suitably defined classes of lives. The results for the 7 classes previously defined are given in Tables XIII through XV.

As is seen, no program uses more than 15 registers simultaneously. 17 of the 41 programs would get by with 10 or fewer registers. This maximum is only used for short periods of time. Thus 10 registers would suffice 90% of the time for all 41 programs, 98% of the time for 36 of the 41 programs. The results for the compilers and for the BLISS programs (BLISS has a highly optimizing compiler) show that neither the size and complexity of the programs nor their efficiency imply the use of many registers. On the contrary, the BLISS results seem to indicate the opposite conclusion. Hence we would attribute the relatively high number of live registers for the other compilers to the fact that these are written in assembly language. If specialized registers were to be used, it would seem appropriate to have 2 floating point accumulators, 2 fixed-point accumulators, and 8 index registers with simple fixed-point operations.

4.2 Reducing the Register Block

The results just presented suggest that programs might run almost equally time-efficiently on an ISP with fewer registers than the one analyzed, but otherwise having the same structure. Increased execution time would ensue from having to store and reload registers whenever the number of lives in the original version was too high. We use two methods, called *interleaving* and *bedding*, to compute an upper bound on this increase in execution time.

4.2.1 Interleaving. Interleaving is applied in phase II. Assume that our reduced ISP has M registers. For each period when the program requires N registers, $N > M$, we select the $N - M$ least useful lives as described below, and assume the associated values to be stored in memory. Each time one of these values is

Table XI. Fraction of Lives Used as Temporaries Only.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.028	.067	.179	.101	.121	099
Crout		.018	.101	.049	.137	.142	.098
Treesort		.001	.107	.000	.000	.001	.022
PERT		.016	.128	.188	.069	.104	.101
Håvie		.072	.279	.062	.250	.019	.136
Ising		.059	–	.086	.147	.067	.090
Secant		–	–	–	.041	.030	.036
Programmer:		E	B	A	G	L	Mean
Aitken		.062	.078	.092	.112	.015	.072
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.096	.089	.180	151	153	134
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.112	.032	.136	.097	.106	.069	.090

Table XII. Fraction of Lives Used for Indexing.

Language:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow		.513	.407	.226	.341	.251	.347
Crout		.519	.374	.520	.195	.244	.370
Treesort		.482	.412	.683	.431	.476	.497
PERT		.592	.421	.556	.445	.497	.502
Håvie		.524	.365	.387	.278	.203	.351
Ising		.571	–	.484	.267	.249	.393
Secant		–	–	–	.376	.406	.392
Programmer:		E	B	A	G	L	Mean
Aitken		.185	.196	.232	.318	.474	.281
Compiler:		ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
		.401	.364	.341	.509	.313	.386
Language:	MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	.425	.534	.396	.378	.333	.332	.391

needed, some register has to be temporarily stored, and the required value loaded into it. Hence each reference to one of the selected lives costs at most two STORE LOAD pairs.

The following four criteria were used for usefulness of lives:

—The number of references to the life was high.
—The density of references to the life was high.
—The life was long.
—The life was short.

The fourth criterion never gave the lowest cost. The third one rarely gave a low cost, the first two gave the lowest cost almost equally often. Furthermore the criterion that gave the lowest cost often changed with M within the same analysis. The interleaving cost is computed only when needed, i.e. when $N > M$. On the other hand, neither the selection of useless lives nor the cost computation takes local properties of the lives into account; both are based on their global characteristics.

4.2.2 Bedding. The bedding method, on the other hand, is based on the local properties of lives. The idea is to store ("bed") registers in memory when they have long dormant periods. In each such period the number of live registers is reduced by one, at the cost of one STORE LOAD pair. Such periods are known during phase I, but the information is not easily carried into phase II. In phase I, however, we do not know when registers are scarce ($N > M$). Hence bedding must be applied each time a life has been dormant longer than some time K, regardless of the need for registers during that time.

Our results were obtained using a hybrid method. Registers were bedded by phase I whenever they were dormant more than 200 instructions, and interleaving was used in phase II. The results, given as relative increase in instruction count, are displayed in Table XVI. As is seen, the increase caused by a reduction to 8 registers is less than 1% for 21 of the 41 programs, less than 5% for 30 of them, but runs as high as 50% or more in a few cases. The average increase is 7.9%.

We investigated the bad cases further by using lower values for K, i.e. lives were bedded when they had been dormant for as little as 22 instructions (in one case). Interleaving was applied in phase II as before. As K is reduced the interleaving cost decreases, since there are fewer periods when $N > M$. On the other hand, the bedding cost increases since there are more dormant periods. We have at present no way of telling which K will give the best result. In fact, in a similar analysis of two programs where the cost for $K = 200$ was already low, we found that the cost was lower for $K = 200$ in one case, $K = 100$ in the other. To produce the results given in Table XVII, different values of K were tried until a minimum seemed close. As is seen, the cost has been dramatically reduced for all of the programs, although it still is high for some. These results would reduce the mean of Table XVI from 7.9% to 2.7%.

The values obtained by bedding and interleaving are upper bounds, in the sense that any satisfactory compiler or programmer, knowing the local properties of the program, will select better "useless" lives, and only store them when N is high. He will also avoid unnecessary STOREs. On the other hand, the results were obtained using complete knowledge of the path taken through the program. When the code is written, all possible paths have to be provided for. This implies a less than optimal use of registers in each particular execution. In view of the fact that most lives are short, it is reasonable to assume that the gain by the former factor far outweighs the loss by the latter.

5. Operator Utility

We also used traces to study the utility of data types, data operators, and control operators. For existing operators and types, frequency counts were used. Some desirable but nonexisting operators were detected by observing frequencies of dynamic sequences of instructions.

Frequency studies for individual instructions or groups of instructions have been reported by various authors [1, 2, 4, 8, 9, 12, 16, 17]. Our results agree well with those of Gibson [7] (the Gibson mix), which

Table XIII. Number of Registers Sufficient 100%, 98%, and 90% of the Time ($K = 200$).

Lanquage:		MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow	100%		13	10	9	13	12	11.4
	98%		11	7	6	10	9	8.6
	90%		8	6	5	9	7	7.0
Crout	100%		13	7	7	13	12	10.4
	98%		11	7	7	12	8	9.0
	90%		10	6	6	10	7	7.8
Treesort	100%		14	7	6	4	12	8.6
	98%		4	7	5	4	5	5.0
	90%		3	6	5	3	4	4.2
PERT	100%		14	10	7	11	12	10.8
	98%		10	7	6	8	8	7.8
	90%		8	6	5	3	5	5.4
Håvie	100%		14	10	9	10	13	11.2
	98%		11	6	6	6	9	7.6
	90%		9	5	5	5	5	5.8
Isinq	100%		14	-	7	11	12	11.0
	98%		11	-	5	7	9	8.0
	90%		10	-	5	3	6	6.0
Secant	100%		-	-	-	13	12	12.5
	98%		-	-	-	6	6	6.0
	90%		-	-	-	5	5	5.0
Proqrammer:			E	B	A	G	L	Mean
Aitken	100%		7	7	8	7	8	7.4
	98%		7	7	7	7	7	7.0
	90%		7	6	6	6	7	6.4
Compiler:			ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
	100%		15	11	13	13	11	12.6
	98%		10	9	6	8	8	8.2
	90%		8	7	5	7	6	6.6
Lanquage:		MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	100%	13.0	13.7	8.8	8.2	10.7	12.1	10.4
	98%	9.0	9.7	6.8	6.5	7.6	7.7	7.6
	90%	7.3	8.0	5.8	5.7	5.4	5.6	6.1

Table XIV. Number of Registers Sufficient 90% of the Time for the Arithmetic Classes FLO, FMD, and FAS (FLO = Floating, FMD = Fixed Mul/Div, FAS = Fixed Add/Sub).

Lanquage:		MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow	FLO		2	1	2	2	2	1.8
	FMD		1	0	0	1	0	0.4
	FAS		4	2	2	1	2	2.2
Crout	FLO		1	1	1	3	2	1.6
	FMD		0	1	2	4	2	1.8
	FAS		5	1	3	3	3	3.0
Treesort	FLO		0	1	0	0	0	.2
	FMD		1	0	0	0	1	.4
	FAS		1	2	3	1	2	1.8
PERT	FLO		0	1	1	0	0	.4
	FMD		0	0	0	0	0	.0
	FAS		4	2	3	2	3	2.8
Håvie	FLO		1	2	2	2	2	1.8
	FMD		0	0	1	0	0	.2
	FAS		5	2	2	2	3	2.8
Isinq	FLO		0	-	0	0	0	.0
	FMD		0	-	0	0	0	.0
	FAS		5	-	4	1	3	3.3
Secant	FLO		-	-	-	2	1	1.5
	FMD		-	-	-	1	1	1.0
	FAS		-	-	-	2	4	3.0
Proqrammer:			E	B	A	G	L	Mean
Aitken	FLO		2	2	2	2	2	2.0
	FMD		0	0	0	0	1	.2
	FAS		3	2	3	4	3	3.0
Compiler:			ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
	FLO		0	0	0	0	0	.0
	FMD		0	1	0	0	0	.2
	FAS		3	2	2	2	2	3.2
Lanquage:		MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	FLO	.0	.7	1.2	1.2	1.3	1.0	1.0
	FMD	.3	.3	.2	.3	.9	.6	.4
	FAS	2.3	4.0	1.8	2.8	1.7	2.9	2.4

should be well known. We refer the reader to [10] and [11].

274 of the over 400 instructions were used by our subject set. 75% of the instructions executed were accounted for by the 29 most executed instructions. 133 instructions accounted for 99% of the executed instructions. Over 40% of the executed instructions were moves between registers and primary memory, almost 30% were branching instructions, 12% were fixed-point adds or subtracts. The other categories of [7] each accounted for less than 5%.

We would also point out one particular result, relating to the addressing problem for tests, where the rich set of test instructions on the DECsystem10 permitted some possibly new observations. The test instructions were divided into groups according to the form of their operands, as seen in Table XVIII. Similarly, the programs were divided into three obvious groups. The programs were weighted in inverse proportion to their instruction count, and the distribution of the different groups of test instructions was observed.

Table XVIII clearly shows that comparison of two nonzero values is twice as common as comparison with zero. This is particularly true for recently computed values (contained in registers), in which case the factor is 3. Hence one is led to doubt the utility of condition codes as compared with the more general test instructions. Also noteworthy is the fact that compilers frequently test against small values known when the compiler was written (immediate operands).

5.1 Instruction Sequences

We now describe our attempt to detect data types and operators that could be included in the ISP at a benefit. Such operators manifest themselves as sequences of instructions, viz. those sequences used to interpret the desirable instructions in terms of the existing instruction set. Since such sequences may be of considerable length, a major difficulty is to limit the space and time used by the analysis program. Thus, for one of our subject programs, the number of different pairs of instructions was as high as 2000. If all these were to be extended to triples, quadruples or longer sequences, both space and time required for the analysis would be prohibitive.

We avoided this problem by using a multipass algorithm. Each pass scanned the whole trace; the first pass built the pairs, successive passes extended the existing sequences by one. After each pass the data structure was pruned; only those sequences thought to be significant were retained. The program ran until no sequences were retained, or until an arbitrary preset length of 20 was reached (after 19 passes). Before the results were printed, the counts for all those sequences which had been extended were reduced by the counts of the extensions. Hence only the unextendable fraction of each sequence was included in the final counts.

Five heuristics were used to detect candidates for deletion:

—All sequences whose counts were low compared to the most frequent sequence of the same length were deleted.

—All sequences that were not a significant extension of their leading and trailing longest subsequences were deleted. The intent was to isolate the common part of overlapping sequences as the interesting part.

—By the algorithm used, loops of length L may be represented at L different places in the data structure. When sequences of length $L + 2$ had been generated, all those for which the two last and two first instructions were the same, and which contained a jump instruction, were assumed to be loops of length L. One representation of such loops was retained, the others deleted.

—An attempt was made to detect all but one of several overlapping sequences representing the same longer sequence. Assume that the sequence A B C D E F G occurs frequently in the trace. At the end of pass 4 the sequences A B C D E, B C D E F, and C D E F G are observed to have approximately the same count. The latter two may be deleted, since the former will be extended in later passes.

—An attempt was made to detect all but the most frequent of long sequences with a large degree of overlap.

Using these pruning heuristics, about half the analyses produced one or more sequences of length 20. All analyses produced sequences of length 10 or more.

The heuristics above, as used in our experiments, were not as good as one might desire. In particular, in most analyses several of the sequences obviously overlapped. This caused the reduced counts for the overlapping parts to be much too low. Other sequences were extended too much, or they included only part of what was known from other considerations to be "the right" sequence. Hence a manual, and therefore subjective, analysis was necessary to extract significant results. This was also needed to relate the results back to program fragments with more or less intuitive meaning. During this analysis, the final results were compared with the unreduced counts printed after each pass. This manual analysis could be reduced by improving the existing heuristics and devising new ones. More accurate counts could be obtained by running a second analysis, observing only predetermined sequences or classes of sequences. This was, however, not done.

5.2 Sequence Results

Specific results are presented in [10]. Below we give a survey of those that seemed most important, and a few specific examples.

5.2.1 Subroutine calling sequences. Calling sequences for subroutines should be better supported by suitable

Table XV. Number of Registers Sufficient 90% of the Time for the Classes NOA, INX, and TOT (NOA = No Arithmetic, INX = Indexing, TOT = Total Class).

Language:			ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow	NOA		4	4	3	7	5	4.6
	INX		6	3	2	5	5	4.2
	TOT		8	6	5	9	7	7.0
Crout	NOA		6	4	2	3	5	4.0
	INX		9	3	3	2	3	4.0
	TOT		10	6	6	10	7	7.8
	NOA		2	4	2	2	2	2.4
	INX		2	3	3	2	2	2.4
	TOT		3	6	5	3	4	4.2
PERT	NOA		4	4	2	2	3	3.0
	INX		7	3	3	2	2	3.4
	TOT		8	6	5	3	5	5.4
Håvie	NOA		5	3	2	2	2	2.8
	INX		8	3	2	2	2	3.4
	TOT		9	5	5	5	5	5.8
Ising	NOA		6	–	2	2	4	3.5
	INX		9	–	2	2	4	4.3
	TOT		10	–	5	3	6	6.0
Secant	NOA		–	–	–	2	2	2.0
	INX		–	–	–	2	2	2.0
	TOT		–	–	–	5	5	5.0
Programmer:			E	B	A	G	L	Mean
Aitken	NOA		4	4	4	3	2	3.4
	INX		4	3	3	2	5	3.4
	TOT		7	6	6	6	7	6.4
Compiler:			ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
	NOA		6	5	4	6	4	5.0
	INX		4	4	2	4	2	3.2
	TOT		8	7	5	7	6	6.6
Language:		MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	NOA	5.7	4.5	3.8	2.9	2.9	3.3	3.5
	INX	4.0	6.8	3.0	2.9	2.4	2.9	3.5
	TOT	7.3	8.0	5.8	5.7	5.4	5.6	6.1

Table XVI. Sum Interleaving and Bedding Costs for $K = 200$ When the Number of Registers is Reduced to 10, 8, or 7, Given as Relative Increase in Instruction Count.

Language:			ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Bairstow	10 rg		.057	.000	.005	.017	.009	.018
	8 rg		.231	.001	.005	.136	.095	.094
	7 rg		.371	.002	.009	.254	.184	.164
Crout	10 rg		.077	.000	.004	.440	.016	.107
	8 rg		.385	.000	.004	.757	.022	.234
	7 rg		.773	.000	.004	1.046	.097	.384
Treesort	10 rg		.002	.000	.011	.000	.015	.006
	8 rg		.005	.000	.011	.000	.016	.006
	7 rg		.007	.000	.011	.000	.016	.007
PERT	10 rg		.017	.000	.000	.004	.004	.005
	8 rg		.133	.000	.000	.036	.038	.041
	7 rg		.213	.001	.000	.053	.067	.070
Håvie	10 rg		.060	.000	.002	.001	.006	.014
	8 rg		.575	.001	.003	.005	.045	.126
	7 rg		.734	.003	.008	.018	.072	.167
Ising	10 rg		.068	–	.005	.002	.005	.020
	8 rg		.438	–	.005	.010	.052	.127
	7 rg		.998	–	.005	.031	.106	.285
Secant	10 rg		–	–	–	.004	.005	.005
	8 rg		–	–	–	.012	.017	.015
	7 rg		–	–	–	.018	.023	.021
Programmer:			E	B	A	G	L	Mean
Aitken	10 rg		.003	.003	.002	.001	.002	.002
	8 rg		.003	.003	.002	.001	.002	.002
	7 rg		.003	.003	.013	.001	.005	.005
Compiler:			ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
	10 rg		.031	.004	.000	.013	.008	.011
	8 rg		.081	.040	.002	.072	.016	.042
	7 rg		.134	.085	.010	.225	.030	.097
Language:		MACRO	ALGOL	BASIC	BLISS	FORFOR	FORTEN	Mean
Mean	10 rg	.016	.047	.000	.004	.067	.009	.035
	8 rg	.064	.295	.000	.004	.136	.041	.079
	7 rg	.148	.516	.001	.007	.202	.081	.137

instructions to handle parameter transmission, return addresses, and to save and restore registers and other parts of the runtime representation.

The cost of call administration is easily detected for BLISS programs, since stack instructions are used only in this context. There is, however, no reason to believe that this cost is less for other languages usually considered to be "efficient."

The BLISS compiler, which is written in BLISS, and which contains many small subroutines for trivial bookkeeping tasks, spent approximately 25% of its time (to compile the BLISS version of Treesort) in call administration. For one of the FORTRAN compilers, which is also written in BLISS, the same number was approximately 15%.[2]

About $\frac{1}{8}$ of the instructions executed by the BLISS compiler could be saved if the subroutine call and exit instructions (PUSHJ and POPJ) were extended to manipulate the run-time registers, and to remove parameters from the stack on exit.

This would reduce 6 or 8 instructions to 2, and 10 or 12 memory cycles to 5, for each subroutine call. This improvement would fit well into the existing instruction format. In the case of FORTRAN programs it would be useful if parameter descriptors were recognized by the hardware, so that local copies of the actuals could be made by the calling instructions.

The suggested improvements would force representations on the language implementors, and hence reduce flexibility. However, such representations are rarely changed once they are decided, so this would not be a serious objection, particularly not if the instruction set were microprogrammed.

Another observation is interesting in this context: From observing the use of the stack instructions, we know that the BLISS compiler saves and restores about 16,000 registers per second (about 1.15 per routine call). This is the same number as would be saved and restored by 1,000 complete process swaps per second. We believe this to be a high frequency of process swaps for the KA-10 processor. Hence it seems that the cost of register saving caused by routine calls may be considerably larger than the corresponding cost caused by interrupts.

One remark is in order: the BLISS compiler has very many small and frequently called subroutines, and is not typical of common or garden programs. We do not, however, consider this a deficiency. Subroutines are an important ingredient in structuring programs, and should be cheap to use. The experimental results support our plea for more efficient hardware to handle registers and state information in calling mechanisms.

5.2.2 Vector descriptors and operands. A vector type should be introduced. This is motivated not only by the importance of vectors as a mathematical structure, but also by the vector structure of central memory and the effect this has on program structure in general. A vector descriptor should be provided. This should make no distinction between vectors allocated by the compilers, and those allocated at run time. Furthermore, it should permit easy description of both row and column vectors of matrices. Operations should include common mathematical operators such as inner product, and also moves, summation, searches in ordered vectors etc. By permitting vectors of different lengths, and in particular length 1, interesting specializations may be obtained, such as initialization by a constant value.

Table XVII. Best Upper Bound for Relative Increase in Instruction Count, Selected Subject Programs, Best *K* Tried.

Language:	ALGOL	FORFOR	ALGOL	FORFOR
Algorithm	Bairstow	Bairstow	Crout	Crout
Bedding cost	.049	.017	.078	.114·
Interleaving cost	.007	.011	.001	.015
Total cost	.056	.028	.079	.129
K where obtained	25	40	27	22
Same cost for K = 200	.231	.136	.385	.757

Language:	ALGOL	ALGOL	ALGOL
Algorithm	PERT	Håvie	Ising
Bedding cost	.043	.065	.102
Interleaving cost	.001	.005	.008
Total cost	.044	.070	.110
K where obtained	25	30	27
Same cost for K = 200	.133	.575	.438

Table XVIII. Use of Test Instructions, Percentages of Total Instruction Count.

Program type	Compilers	Non-numeric programs	Highly numeric programs	Total subject set
Instruction form				
Register vs. memory	3.0	4.9	4.5	4.5
Register vs. immediate	7.7	1.7	1.0	2.1
Memory vs. 0	2.3	1.7	.9	1.3
Register vs. 0	2.5	1.8	2.1	2.0

Vector types would, in the extreme, change the ISP radically, as is exemplified by the CDC STAR. We do think, however, that some vector operations would be useful even in more conventional ISPs. Examples are frequent in our programs, although none are as dramatic as the others cited in this section.

5.2.3 String handling. Introduction of a "character string" type would speed up the compilers by a significant amount. Instructions operating on this type should be controlled by a table, indexed by the set of possible characters. The options for each character should include substitution, removal, branching to a special action routine, and termination of the instruction. It should be easy to use these instructions to change encodings, move strings, remove multiple blanks, remove extraneous characters etc. Analysis of routines for I/O formatting, and of COBOL programs, would suggest further options. Typical examples which illustrate the need for such instructions come from the compilers, particularly from BASIC.[3]

5.2.4 Run-time support for languages. The routines for run-time space management, parameter transmission and similar functions in ALGOL and similar languages are exceedingly expensive. They may consume as much as 50% of the execution time of some ALGOL programs.[4]

5.2.5 Miscellaneous data operators. Other data operators which could be included are: memory to memory moves (unless subsumed under the vector type), type conversions, and packing and unpacking of partwords. Some of these are already in the DECsystem10 ISP, but are not accessible to high-level language programmers. Hence this is a language problem as much as an ISP problem.[5]

5.2.6 Loop control. There should be an instruction for loop control which increments a fullword counter in one register and tests it against a fullword upper bound in another register. This instruction is also easily accommodated within the DECsystem10 ISP structure. It would save up to 5% of the execution time of some programs, reduce program size, and increase readability.[6]

6. Conclusions

In spite of the restricted set of experiments performed, we believe some of the results produced to be valid, not only for the DECsystem10, but for all register structured ISPs. This is in particular true for the results on simultaneous use of registers, and on the cost of subroutine calls.

It seems, for instance, that eight registers would be sufficient for a general register ISP similar to the DECsystem10. The result is no longer valid when the registers are used for other tasks than in this ISP, such as base register addressing, program counter, hardware defined stacks, etc.

Similarly the results on overhead in subroutine calling are both important and portable. Results from other ISPs would often exhibit an even worse situation, since the handling of return linkages for recursive or reentrant subprograms is more cumbersome. On the other hand, the situation can easily be improved by introducing instructions tailored to the needs of the commonly used languages. An ideal solution would be to permit a restricted form of writable microprogram, defining special instructions for each language. This would also be helpful with respect to run-time support for ALGOL and other languages.

Some of the results presented here and in [10], particularly those stemming from unnecessary generality, might seem like a severe criticism of the DECsystem10. This is a consequence of the deplorable fact that our methods only measure the time cost of ISP features. The richness and generality of the DECsystem10 ISP make it a good ISP to program for, and contribute to a low programming cost and a low memory space for programs. For our other points of criticism we note that although the DECsystem10 leaves room for improvement, the problems we point out are not solved in a better way in other common ISPs.

Our work has barely scratched the surface of a large area of investigation. In particular, it would be interesting to study information used for address calculation and information used for control purposes. We would like to know more about how such information is computed, and how the two kinds interact. We hope to make this the subject of further research. The various solutions to the addressing problem for test instructions should also be investigated.

Acknowledgment. W.A. Wulf provided initial impetus to and considerable support and ideas throughout the project that led to this paper.

Received April 1975, revised January 1976

Notes

1. By an Instruction Set Processor, or ISP [3], we mean the logical processor which processes the instruction set, as divorced from its physical realization. Example: The IBM 360/370 is one ISP which has several physical realizations.

2. This is illustrated by the following sequences from the BLISS compiler:

PUSH PUSHJ JSP PUSH HRRZ	(14.3% of the execution time)
JRST POP POPJ SUB	(7.2% of the execution time)
JRST POP POP POPJ SUB	(3.5% of the execution time)

Only 3 of these 14 instructions are used in connection with parameter transmission; the rest are used for state saving, environment definition, and linkage handling.

3. The sequence:

SKIPE ILDB JRST CAIE CAIN CAIN CAIE CAIN CAIE CAIN CAIG CAIA CAIGE IDPB SKIPE SOSLE AOJA

consumed 20.7% of the compilation time. Its purpose is to move a line while removing extraneous characters like TABs, LINEFEEDS, etc. Similarly the sequence

ILDB CAIN IDPB JRST

moves a line stopping at a RETURN. It consumed 8% of the compilation time.

4. The following example is from the Ising program:

AOBJP MOVE MOVE ADDI HLLZ SETZB ROTC EXCH ROTC ROT ANDI HLRZ HRRZ ANDI LSH ANDI LSH

It consumed 19% of the time. From PERT we have:

XCT PUSHJ PUSHJ MOVE PUSH MOVEI MOVE PUSH HLRZ PUSHJ MOVE ADD MOVE POPJ POP POP TLNE POPJ MOVE POPJ

This is a complete call of a formal parameter by name (thunk), starting at the call within the procedure body (XCT) and ending at the POPJ back into it. The actual parameter is a vector element. Time consumed by this sequence was about 20% of the total.

5. An example is the sequence MOVE IDIV, used to unpack left halfwords, which consumes 45% of the time for the FORTEN version of Treesort. The HLRZ instruction used for the same purpose in the BLISS version consumes only 7.5% of the time of that version. The rest of these routines are about equally efficient.

6. The function shows up as:

ADDI AOJL or GAMGE AOJA MOVEM in FORTRAN
JRST AOS CAMLE in ALGOL,
MOVE FADR JRST CAMLE MOVEM in BASIC, and
AOJA CAMLE in BLISS.

References
1. Alexander, W.G. How a programming language is used. Rep. CSRG-10, Comptr. Res. Group, U. of Toronto, Toronto, Canada, Feb. 1972.
2. Arbuckle, R.A. Computer analysis and thruput evaluation. *Computers and Automation* (Jan. 1966), 12–15 and 19.
3. Bell, C.G., and Newell, A. *Computer Structures, Readings and Examples*. McGraw-Hill, New York, 1971.
4. Connors, W.D., Mercer, V.S., and Sorlini, T.A. S/360 instruction usage distribution. Rep. TR 00.2025, IBM Systems Development Div., Poughkeepsie, N.Y., May 8, 1970.
5. Foster, C.C., Gonter, R.H., and Riseman, E.M. Measures of opcode utilization. *IEEE Trans. Computers C-20*, 5 (May 1971), 582–584.
6. Foster, C.C., and Gonter, R.M. Conditional interpretation of operation codes. *IEEE Trans. Computers C-20*, 1 (Jan. 1971), 108–111.
7. Gibson, J.C. The Gibson mix. Rep. TR 00.2043, IBM Systems Development Div., Poughkeepsie, N. Y., 1970.
8. Gonter, R.H. Comparison of the Gibson mix with the UMASS mix. Pub. No. TN/RCC/004, Res. Comptg. Center, U. of Massachusetts, Amherst, Mass.
9. Herbst, E.H., Metropolis, N., and Wells, M.B. Analysis of problem codes on the MANIAC. *Math. Tables and Other Aids to Comput. 9* (Jan. 1955), 14–20.
10. Lunde, Å. Evaluation of instruction set processor architecture by program tracing. Ph.D. Th., Dep. Comptr. Sci., Carnegie-Mellon U., Pittsburgh, Pa., July 1974 (available as AD A004824 from Nat. Tech. Inform. Service, Springfield, Va).
11. Lunde, Å. More data on the O/W ratios. A note on a paper by Flynn. *Computer Architecture News 4*, 1 (March 1975), 9–13.
12. Raichelson, E., and Collins, G. A method for comparing the internal operating speeds of computers. *Comm. ACM 7*, 5 (May 1966), 309–310.
13. Tjaden, G.S., and Flynn, M.J. Detection and parallel execution of independent instructions. *IEEE Trans. Computers C-19*, 10 (Oct. 1970), 889–895.
14. Wilner, W.T. Design of the Burroughs B1700. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 489–497.
15. Wilner, W.T. Burroughs B1700 memory utilization. Proc. AFIPS 1972 FJCC, Vol. 41, AFIPS Press, Montvale, N.J., pp. 579–586.
16. Winder, R.O. Data base for computer performance evaluation. RCA-reprint PE-517, RCA David Sarnoff Res. Ctr., Princeton, N.J., 1971.
17. Winder, R.O. A data base for computer evaluation. *Computer 6*, 3 (March 1973), 25–29.
18. Wulf, W.A., Russell, D.B., and Habermann, A.N. BLISS: A language for systems programming. *Comm. ACM 14*, 12 (Dec. 1971), 780–790.

Computer Systems — G. Bell, D. Siewiorek, and S.H. Fuller, Editors

Memory Management and Response Time

R.M. Brown, J.C. Browne, and K.M. Chandy
The University of Texas at Austin

This paper presents a computationally tractable methodology for including accurately the effects of finite memory size and workload memory requirements in queueing network models of computer systems. Empirical analyses and analytic studies based on applying this methodology to an actual multiaccess interactive system are reported. Relations between workload variables such as memory requirement distribution and job swap time, and performance measures such as response time and memory utilization are graphically displayed. A multiphase, analytically soluble model is proposed as being broadly applicable to the analysis of interactive computer systems which use nonpaged memories.

Key Words and Phrases: memory management, system performance, queueing network models, interactive computer systems

CR Categories: 4.32

This research was partially supported by the National Science Foundation under Grants GJ-1084 and DCR 74-13302. Authors' address: Department of Computer Science, University of Texas, Austin, TX 78712.