```
+444444444444444444444444444444444444444444444444444444444444444444444444444444+

                                  $$$    $$$$
                                 $   $   $   $
                                 $   $   $   $
                                 $   $   $$$$
                                 $   $   $
                                 $   $   $
                                  $$$    $

                         USER=OP  QUEUE=LPT  DEVICE=@LPA1
                SEQ=16  QPRI=127  LPP=63  CPL=80  COPIES=1  LIMIT=41

                           CREATED:  30-MAR-77   9:48:26
                          ENQUEUED:  14-JUN-77  13:07:46
                          PRINTING:  14-JUN-77  13:07:48

                              PATH=:MEMO:MEMO$196.LS

   $   $  $$$$$  $   $   $$$     $      $      $$$     $$$            $        $$$
   $$ $$  $      $$ $$  $   $   $$$$   $$     $   $   $               $       $   $
   $ $ $  $      $ $ $  $   $  $ $    $ $     $   $  $                $       $
   $ $ $  $$$$   $ $ $  $   $   $$$     $      $$$$  $$$$             $        $$$
   $   $  $      $   $  $   $     $ $   $         $  $   $            $           $
   $   $  $      $   $  $   $  $$$$     $        $   $   $     $$     $       $   $
   $   $  $$$$$  $   $   $$$     $    $$$$$   $$      $$$      $$     $$$$$    $$$

+444444444444444444444444444444444444444444444444444444444444444444444444444444+


                                AOS XLPT REV 01.00
```

To: J. Clancy, C. Mundie, S. Schleimer, W. Slack,
R. Belgard, J. Dooda, R. Gruner, S. Redfield, S. Wallach

From: J. Ahlstrom, M. Druke, W. Wallach

MEMO 196 MARCH 12, 1977

Subject: SUMMARY MIX AND FUNCTION BY MODULES FOR
COBOL, FORTRAN, SPL, OSkernel

Note: The following mix weights are totally SWAG

MIX SUMMARY

COMMERCIAL INSTALLATIONS

The standard mix for commercial installations is guessed to be:

70%	Cobol	object programs
30%	Spl	compilers, data base, debuggers, OS

NUMERICAL INSTALLATIONS

The standard mix for numerical installations is guessed to be:

60%	Fortran	object programs
40%	Spl	compilers, OS, debuggers, editors

MIXED INSTALLATIONS

Much more variability in mix can be expected from mixed installations than from exclusively commercial or numerical ones. This mix is presented for a mythical "perfectly balanced" mixed installation:

33% Cobol
33% Fortran
34% Spl

CONTRIBUTIONS TO PERFORMANCE BY OPERATION BY LANGUAGE

	COMMERCIAL	NUMERICAL	MIXED
COBOL			
addpacked	4%		2%
cmprpacked	7%		3%
cmprdisplay	30%		15%
movechars	20%		10%
adddisplay	4%		2%
mpypacked	2%		1%
FORTRAN			
add floating		11%	6%
index add		11%	6%
cmpr&branch incl do update		10%	5%
move		7%	3%
mpy floating		5%	3%
indirection		4%	2%
go to (unconditional)		3%	2%
format edit		2%	1%
radix convert		2%	1%
SPL			
move	15%	18%	14%
goto	8%	10%	7%
call	5%	7%	4%

compare&branch	9%	12%	8%
bittest&branch	5%	6%	4%
arithmetic	3%	5%	3%

MODULE ORIENTED FUNCTIONS

We can abstract from these language-oriented operations to module-oriented functions producing the following breakdown of what JP modules must be able to do well to produce competitive machines:

PARSE

Deliver cannonical operand specifiers to FETCH at the rate of one per cycle. Note that this is not possible for SPL, COBOL and PL/I when operand lengths are specified by structured literals. This argues for longer fixed length literals for Cobol and PL/I.

Completely process unconditional jumps invisibly to other units.

Prefetch both targets of a conditional branch waiting for the condition to be resolved only to decide which to process.

The parse's relation to exception handling:

external interrupts,
s_op dependent faults,
machine checks

is yet to be specified (TBS).

FETCH

Accept cannonical operand specification, generate and pass to cache ADD and fetch length, modify remaining length and address and specify its own next nano instruction address in 1 cycle. Where fetch length is:
the minimum of JPD-bus width and (remaining) operand length.

When the amount of data to be fetched is less than one JPD-width specify justification, extension and fill characteristics.

For multiple JPD-width operand fetches, if the length is not yet exhausted and the condition, if any, is not yet detemined by the execute box, send ADD and length to cache, modify length and address and specify own next nano instruction in one cycle. Specify justificaion, extension, and fill for short lengths.

Handle compiler detected or user specified array operations, to fetch and store elements of vectors that are being opeated on as aggregates rather than single elements.

Abort multi-JPDB-width fetches when execute has already determined result of comparison.

Handle overlapping strings when compilers cannot or do not handle them.

Exception handling TBS.

INTERPRETER

Extract and insert arbitrary fields in arbitrary length operands.

Access known structures through physical addresses.

Generate memory addresses to chase linked data structures.

Generate own next nano address based on extracted fields and several staticised bits--perhaps 16 to 64 way CASES.

Exception handling TBS.

EXECUTE

Packed decimal arithmetic and comparisons including digit validity.

Display comparisons including weird sign conventions.

Packing, unpacking and editing including digit validity checks.

Overflow on 32 bit stores, and 64 bit calculations.

Binary comparisons signed and unsigned.

Conversion from binary to decimal radices.

Floating point arithmetic.

Fixed point arithmetic.

Exception handling TBS.

APPENDIX I: SPL OPERATIONS AND OPERANDS

OBJECTIVE

To characterize system programming fundamental operations the efficient execution of which is essential to SPL program performance.

OBSERVATIONS

Burroughs uses a very PL/N like language called SDL as the implementation language for the B1700/1800 systems. Though SPL is different from both these languages the problems it will be called on to solve are similar. It can be expected that SPL programs will use many more strange-length variables than SDL, and more subscripting. Additionally, the S_languages are philosophically quite different between SPL and SDL. This study of DYNAMIC execution characteristics of the B1700 MCP can only characterise the kinds of source language operations that systems programming languages specify--not the details of the actual s_ops that SPL will execute. Two dynamic traces of approximately 4,000,000 s-ops each (obtained by tracing the 1700 MCP) agree quite well in their frequencies. One study is multiprogramming a number of jobs in ample memory with memory management activity essentially limited to changes in their working sets. The second is thrashing in less memory with the MCP spending a substantial proportion (11%) of its time executing the s-op that searches through memory looking for available space.

	% of operations executed when		
Operation	not thrashing	thrashing	
move	10.6	11.3	
arithmetic	4.23	4.83	
not	2.00	1.80	
add	1.15	1.44	
sub	.84	1.23	
comparison	4.69	5.04	
eql	2.58	3.00	
neq	1.32	1.14	
gtr	.48	.53	
boolean	1.20	1.38	
program cntrl	18.22	17.22	
conditional	7.07	6.80	
ifthen	4.25	4.47	
ifelse	1.61	1.27	
leavec	1.21	1.06	conditionally leave block
unconditional	11.15	10.42	
call	4.93	3.72	
leave	2.46	3.00	uncondtionally leave
return	1.50	1.47	function value return
cycle	1.12	.82	next iteration
exit	.71	.92	procedure return
case	.43	.49	
construct	<8.00	<8.00	
parameter and local variable packets (S_language architectural overhead caused by processing environment.)			
load address	<50.00	<50.00	but not much <
or value on stack (S_langugage overhead...)			

Like SPL SDL allows the specificaion of variable length integers and bit strings as well as character strings. For data that are NOT incuded in structures (records) this facility is little used in SDL partly because it is fewer keystrokes to specify a full 24 bit integer and partly because there is no space saving in specifying stack-frame variables of less than 24 bits rather than one of 24 bits. Whether it is used in SPL I suspect will be more a matter of management than of technology, if it is as easy to specify the exact interval of a variable rather than some standard or default interval then that will be done. In 1,556,823 references to variables not in structures (implying approximately 2,500,000 references to variables in structures) in the 1700 MCP's dynamic trace, the distribution of lengths with non-zero frequencies is:

bit length	reference frequency	reference %	
1	76,385	5	
2	798	-	
3	3,766	-	
4	353	-	
5	268	-	
6	627	-	
7	22,629	1	size of i/o channel field
8	2,079	-	not including 1 char strings
12	1,588	-	
16	597	-	
20	285	-	
24unsigned	1,323,423	85	these are the two lengths that
24signed	124,025	8	can be specified without thougt

SPL will much more strongly encourage the declaration and use of strange width non-structured variables than SDL does, thus, making the numbers in this table only representative of languages that allow this facility, not at all typical of the lengths we will actually encounter in SPL. References to variables in structures will 'always' be to 'strange' lengths.

To the extent that SPL programs are similar to the B1700 MCP, they will exhibit the following characteristics:

- 28% stores
- 30% unconditional transfers of control
    - 12.5% call
- 30% conditional branching — 40% conditions true
    - 20% requiring comparison
    - 10% bit testing only
- 8% arithmetic

To the extent that SPL has explict semanticaly rich operations for functions that must be composed out of SDL s-ops, these % will be reduced--particularly the program control ones.

APPENDIX II: CORE_FORTRAN

ABSTRACT:
A study of Fortran performance is undertaken by analyzing recent publications, resulting in a dynamic mix of Fortran primitives.

This memo is an attempt to identify the "core" operations which must be executed efficiently by or machine to insure competitive Fortran performance. The numbers in this report were deciphered from various inputs including:

1) a large static and a small dynamic analysis of Fortran programs done by Knuth at Stanford,

2) two static studies which Robinson and Torsu reported in the British Computer Journal, and

3) a static and dynamic analysis of Algol performed by Wichmann.

The algorithm used to combine these inputs and derive the dynamic mix was roughly:

1. Determine the static distribution of the 8 most frequently occurring executeable statements.

2. Using the dynamic study as a basis, infer a dynamic distribution of statement occurrences.

3. Determine the types and distribution of primitive operations that each statement could compile into.

4. Combining the results of 2 and 3, produce a dynamic mix.

Each step in this procedure adds to the error already present in the inputs, resulting in an uncomfortably low confidence factor in the final conclusions. However, I believe this algorithm is the best technique available to produce these results; when new data and more informed intuition are obtained, further iterations of this algorithm should converge on a "correct" mix. In order to identify the areas where errors could be introduced, each assumption that was made is recorded; any refinements or second opinions would be very useful in producing a better iteration of this mix.

STATIC DISTRUBUTIONS

The static frequency of occurence of the 8 most commmon executeable statements occuring in the sample programs are enumerated in the following table. These numbers are normalized to reflect true percentages of executable statements, i.e. those statements which only affect compilation are removed (e .g. CONTINUE, DIMENSION, END, etc.). Each study provides data on two sample sets:

Knuth presents results of a huge sample of programs written at Lockheed Corp., as well as a much smaller set written by students at Stanford.

The British Computer Journal article (B CJ) reports on a "system" and a "student" sample.

It is interesting to note that the two "commercial" (i.e. Knuth's Lockheed and BC J's system) samples agree much better than the student samples. This is somewhat reassuring since these samples will surely be more similar to the typical Fortran programs written on our machine than the "toys" (as John Pilat calls them) written by the students.

Consequently, when computing the average percentage of each staement, the commecial saples were weighted 3:1 over the student samples. Logical IF's, i.e. {IF (.cond_expr.) "statement are treated as two statements, one IF and one "statement".

	KNUTH		BCJ		WEIGHTED
	LOCKHEED	STUDENT	SYSTEM	STUDENT	AVERAGE
Assignment	46.0	60.1	48.1	50.3	49.1
IF	16.3	10.0	16.7	11.2	15.0
GOTO	14.6	9.4	13.1	12.6	13.1
DO	4.5	5.9	5.2	7.8	5.4
CALL	9.0	4.7	4.3	4.0	6.1
RETURN	2.2	2.4	2.0	2.8	2.2
WRITE	4.5	5.9	8.5	7.9	6.6
READ	.3	1.2	1.3	2.3	1.0

DYNAMIC DISTRIBUTIONS

The only explicit dynamic information available results from tests performed by Knuth on his student "toys". Other tidbits pf information can be inferred from various data, but more reliable numbers cannot be assembled without more inputs. Knuth's dynamic data is summarized in the following table; also depicted is an attempt to determine a more accurate dynamic mix by assuming that the dynamic/static ratio is invariant, therefore allowing a normalized dynamic average to be computed from the static averages. It is important to note that these dynamic distributions are not weighted.

by estimated execution times. Such a transformation would defeat the purpose of this exercise, which is to determine which operations provide more "leverage", i.e. to determine which operations, when accelerated, contribute most to an overall increase in Fortran performance.

Statement	Knuth Static	Knuth Dynamic	dyn/ static	ave. Static	normalized computed Dyn
Assignment	60.1	64.4	1.1	49.1	56.6
IF	10.0	10.5	1.1	15.0	17.3
GOTO	9.4	8.6	.9	13.1	12.4
DO	5.9	9.6	1.6	5.4	9.0
* CALL	4.7	2.9	.7	6.1	3.2
* RETURN	2.4	2.9	1.3	2.2	-
WRITE	5.9	1.0	.2	6.6	1.3
READ	1.2	0.0	0.0	1.0	0.0

* Of course, the dynamic frequencies of CALL & RETURN must be equal, therefore, in computing the normalized computed dynamic frequency they were combined as a single dynamic statement

whose frequency is assumed to be the average of the two results of multiplying the static frequencies by their dynamic/static ratios. The other frequencies were adjusted to reflect this merger.

STATEMENT BREAKDOWNS

In this section each of the eight statements are analyzed in detail to determine a plausable mix of primitive operations that each statement could compile into. Architectural overhead loads and stores are assumed to be nonexistent since the S-ops executing these common statements will surely be semantically rich.

ASSIGNMENT

All the studies provide information about the relative occurence of operators within assignment statements, from which the following distribution of operators is derived (note: add includes sub):

add	60%
mpy	26%
div	8%
library functs	4%
user functs	2%.

The problem then reduces to determining the average number of operators per assignment statement. The answer was obtained by making two approximations:

1) 45% of all dynamic occurences of assignments are moves and

2) in the remaining 55%, there is an average of 2 operators per expression. This results in the conclusion that the average assignment statement is executed as:

move	.45
add	.66
mpy	.29
div	.09
library functions	.04
user functions	.02

The next primitive operations resulting from assignment statements are index manipulations. The first bit of information necessary is the following distribution of subscripts among variables:

0	63%
1	25%
2	10%
>2	2%

Assuming reasonble compiler optimization we can, perhaps a little optimistically, assume that all singly-subscripted variables require no index arithmetic, all doubly-subscripted variables require an index add, and all variables with more than 2 subscripts require and index multiply and an index add. This, together with an assumed average of 2.5 variables per assignment, result in the conclusion that the average assignment statement will require .30 index adds and .05 index mpy's.

Variables that are arguments to or results from a called function are referenced indirectly; this overhead should also be computed. However, since these indirect references occur whenever a CALL occurs, this analysis is postponed until the section on CALL.

IF

The two classes of IF statements, arithmetic and logical, must be analyzed seperately. Logical IF'S, which comprise approximately 70% of all IF's, are straightforward; each compile into a simple compare&branch operation. Arithmetic IF's, however, contain an arithmetic expression as well as three possible branch addresses.

The three address question was resolved by assuming that 10% of all arithmetic IF's (3% of all IF's) specify three different address and therefore require an additional compare&branch. The expressions within an arithmetic IF were assumed to be comparable to those in assignments. All this results in the following conclusions about the average IF:

1.03 compare&branch
.20 add
.08 index add
.08 mpy
.03 div
.02 index mpy
.01 library function.

DO

The DO statement is executed twice, once for loop setup and again for loop iteration. The average loop was assumed to be executed 10 times, requiring the loop setup operation frequencies to be attenuated by a factor of ten. DO loop iteration requires an index add and an index comp&branch. Although there is a difference between an index comp&branch and the IF comp&branch, (the loop count is incremented as a side effect) they are similar enough to be treated as the same primitive operation in the mix. The complexity of the DO loop setup depends on whether the loop increment is the default of one (95%) or some specified value (5%). If the increment is one, the loop count can be determined by a simple subtraction, the entire loop setup is a move and an index add(sub). If, on the other hand, the increment is not one, an additional index add and index divide is necessary to compute the count. This results in the average DO statement being executed as:

1.1 index add
1.0 compare&branch
.1 move
.005 index divide

GO TO

The GOTO is the simplest of the statements. Except for the totally non-occuring assigned GOTO(0%) and the very infrequent computed GOTO(1%), the GOTO maps directly into a branch. In fact, since 50% of all GOTO's occur in logical IF's, they compile into a conditional branch which has already been counted in the IF analysis. Therefore the average GOTO statement is executed as:

.49-goto(unconditional);
.01 compted go to.

CALL/RETURN

The CALL/RETURN pair is straightforward to analyze. It expands into a state save, a state restore, and two unconditional branches. In addition, arguments and results are passed using pointers in the stack. Therefore the overhead of indirect references are associated with CALL/RETURN. The assumption was made that there, are on the average, 5 indirect references per CALL. Therefore the average CALL/RETURN pair is executed as:

1-state save;
1-state restore;
2-unconditional go to;
5-indirections.

WRITE

Although WRITE occurs roughly 1% of the time, it has been observed that it actually consumes 25-50% of execution time. This is caused by two factors:

1) The WRITE statement could contain an "implied DO" or a list of variables to be written, therefore the average WRITE statement really involves multiple WRITE's. The assumption was made that the average WRITE executes 7 times. There is a tremendous deviation here because an instance of a WRITE could specify a single variable or a 100X100 matrix.
2) the data to be written must be converted from binary to decimal and edited according to a format specification. These "primitive" operations are quite complex and time-consuming, causing the typical WRITE dynamic execution weight to be much higher than the other statements. This is a fundamental problem with this type of analysis, the fact that some operation occurs .1% of the time is not enough information to discount it; if it takes 100 times as long to execute as another statement occuring 10% of the time it is of equal significance.>

Therefore the following mix for write is computed:

7-format edit;
7-radix convert
7-index add
7-compare&branch
1-interdomain call to write.

READ

Since READ occurs very infrequently it is not handled in detail also it is very similar to WRITE, and acceleration of formatting and radix conversion should be bidirectional.

DYNAMIC MIX

This section contains the final results of this study; the conclusions of sections 2&3 are combined to produce a SWAG Fortran mix.

STATEMENT SUMMARY

Statement	dynamic weight	primitive op	freq.	weighted freq
Assignment	.57	add	.66	.38
		move	.45	.26
		ndx add	.30	.17
		mpy	.29	.16
		div	.09	.05
		ndx mpy	.05	.03
		lib.fun.	.04	.02
		user_fun.	.02	.01
IF	.17	comp&branch	1.03	.18
		add	.20	.03
		ndx add	.08	.01
		mul	.08	.01
		div	.03	.00
		ndx mul	.02	.00
		lib.fun.	.01	.00
GOTO	.12	goto(uncond.)	.49	.06
		case	.01	.00
DO	.09	ndx add	1.1	.10
		comp&branch	1.0	.09
		move	.1	.01
CALL/RETURN	.03	state save	1.0	.03
		state restore	1.0	.03
		goto(uncond.)	2.0	.06
		indirection	5.0	.15
WRITE	.01	format edit	7.0	.07
		radix conv.	7.0	.07
		ndx add	7.0	.07
		comp&branch	7.0	.07
		I/O directive	1.0	.01

DYNAMIC MIX

primitive	weighted freq.	normalized freq.
add	.41	.20
ndx add	.35	.17
comp&branch	.34	.16
move	.27	.13
mul	.17	.08
indirection	.15	.07
goto(uncond.)	.12	.06
format edit	.07	.03
radix conv.	.07	.03
div	.05	.02
ndx mul	.03	.01
lib. fun.	.02	.01
user fun.	.01	<.01
I/O directive	.01	<.01
case	<.01	<<.01

APPENDIX III: CRUCIAL COBOL OPERATIONS

STUDY Z A STATIC AND DYNAMIC STUDY OF COBOL SOURCE ELEMENT FREQUENCIES

This study shows static and dynamic occurrence of Cobol verbs and their operands for 9,000,000 Cobol verb executions of a 15000 verb program. According to this study the dynamic distribution of verbs is:

static	dynamic	ratio d:s	verb
26	42.7	1.7	IF
33	25.6	.75	MOVE
20	12.0	.6	GO TO (conditional and unconditional)
4.8	9.5	2	ADD
.55	2.2	4	MPY
.57	2.1	4	SUB
6.5	1.5	.22	PERFORM
.26	.4	1.5	DIV

The strong disparity of static and dynamic frequencies and the interchange of the 1st and 2nd most frequent verbs confirms my prejudices against static studies.

The dynamic distribution of operands by verb (as % of all verb executions and as % of all executions of this verb) is (where bin is subscript, exd is display, pck is packed, lit is literal):

verb		% of all	% of verb	
ADD	bin, bin	.66	7.0	bin probably is a local
	exd, exd	2.9	30.6	equivalent of pck and
	exd, pck	.38	4.1	will be so considered.
	pck, pck	.04	.4	
	lit, bin	3.1	33.1	if bin is really the
	lit, exd	1.22	12.8	equivalent of index
	lit, pck	.1	1.1	rather than packed
	exd, pck, bin	.67	7.1	or sometimes one or the
	lit, pck, bin	.16	1.7	other we are misled.

if bin is assumed to be pck these percentages change to become

	% of all	% of verb
pck, pck	.7	7.4
lit, pck	3.2	34.2

Four accelerated S-ops:

add display to display	2.9
increment packed by literal	3.2
increment display by literal	1.2
add packed to packed	.7

would account for 7% of all executed Cobol instructions.

verb		% of all	% of verb	
DIV	exd, exd, exd	.05	12.9	
	exd, lit, exd	.04	10.9	
	pck, exd, pck	.21	54.4	
	lit, exd, exd	.85	21.8	
IF	x, x	3.0	7.0	x is display alphanumeric
	x, lit	11.3	26.5	
	bin, lit	6.0	14.0	
	exd, exd	.4	1.1	
	exd, lit	5.4	12.7	
	x, x, bin	1.7	4.1	
	x, bin, lit	1.2	2.8	

```
        x,   x,   x,   x 1.6     3.8
        lit, x,   x,   x 1.5     3.5
        exd, lit, x,   x  .47    1.1
        lit, x,   lit, x 2.3     5.5
```

Four accelerated compare and branch instructions:

```
    compare display to display,             6.3
    compare display to literal,            12.8
    compare packed to literal,              6.0
    compare display numeric to literal      5.4
```

would account for 30% of all dynamically executed Cobol verbs.

```
MOVE    x,   x              6.8      27.3
        exd, exd             .8       3.0
        exd, x              1.2       4.5
        lit, x              2.0       8.6
        lit, bin            1.6       6.2
        lit, exd            1.8       7.0
        x,   x,   bin       1.8       7.3
        x,   bin,            .85      3.3
        exd, rpt, bin        .72      2.8
        x,   x,   bin, bin.31         1.2
        x,   bin, x,   bin3.1        12.0   ??  ??  ??
```

Four accelerated s_ops:

```
    move display to display,        9.8
    move lit to display,            2.0
    move packed to packed           3.0
    move lit to display numeric,    1.8
```

would account for 16.6% of all dynamically executed Cobol verbs.

APPENDIX IV: COBOL ACCELERATORS

OBJECTIVE

To determine what if any components should be added to FHP hardware to improve the performance of Cobol programs.

BACKGROUND

There is a possibility of providing operation acceleration features on FHP systems that can enhance their execution of Cobol programs. To decide what operations to accelerate we would like to know the relative frequencies of Cobol verbs and data types. Unfortunately there is a dearth of reliable information on this topic and we are reduced to applying liberal doses of intuition to what studies are available.

There are 3 DYNAMIC studies which address this question:

360/85 design study
360 instruction frequency study
STUDY Z dynamic/static Cobol verb study.

The first two were done to characterize current 360 instruction execution frequencies, the last to study actual Cobol dynamics.

One static study of 360 code generated by the DOS ANSI compiler for a DGC application program provided some surprises. Two other static studies are most noteworthy for their discrepancy with dynamic data:

Guelph University study of university administrative programs
STUDY Z static Cobol verb study.

In STUDY Z the 6 dynamically most frequently occurring Cobol verbs, their static frequencies, the ratio of dynamic to static frequencies and comparison with IBM dynamic and guelph static frequencies are:

verb	%dyn		%stat		dynZ/statZ
	Z	IBM	Z	Guelph	
IF	43	12.5	26	14-30	1.65
MOVE	26	35	33	30-40	.79
GOTO	12	33	20	14-30	.60
ADD	9.5	4.5	4.9	2-3	1.9
MPY	2.2		.55		4
SUB	2.1		.57		3.7

The last column indicates that the dynamic frequency of operations is typically from twice to 1/2 their static frequency and, therefore, that the ratio of two dynamic frequencies is from 4 times to 1/4 that of the ratio of their static frequencies.

In the IBM studies the Cobol verbs have disappeared in the 360 opcodes. At first we felt that we could isolate the "architectural overhead" instructions from the "substantive" ones. Examination of the code generated by the DOS ANSI compiler shakes that belief. We had guessed 30% to 40% overhead. In fact each Cobol verb is compiled into a STATIC average of 3 360 instructions. Unless the dynamically most frequent instructions compile into substantially fewer instructions than average we are faced with perhaps 50% to 60% overhead. (Interestingly one dynamic study shows that 7 of the most obviously substantive instructions dynamically account for 40% of all instructions. I was too shy to guess that this was in fact all the substantive instructions and that 60% rather than 30% were overhead. Unfortunately trying to induce the Cobol verbs which correspond to these substantive operations produces the very different dynamic frequencies in the above table.)

OBSERVATIONS

Despite these confusions there are some underlying simularities among all these most frequent substantive Cobol verbs:

1. They address 2 streams of data being read from memory and compare them IF
2. They address 2 streams of data, 1 being read and 1 being written, perhaps after a "trivial" transformation MOVE
3. They address 2 streams of data being read from memory and "combine" them to produce a 3rd stream to be written to memory ADD SUB etc

GOAL

The goal of Cobol accelerators should be to allow these kinds of operations to proceed at memory-cache-JPDbus bandwidth.

REQUIREMENTS

To meet this goal we may need special purpose accelerators in the following areas:

1 fetch can send 1 address to cache each cycle

2 cache can send 1 JPDbus width of data each cycle

3 execute can
 compare
 pack
 unpack
 add, subtract
one JPDbus width of data each cycle. IBM checks all such operations for valid data and optionally aborts on invalid data. To be comparable to IBM in this matter and meet our performance goals we may have to add special purpose checks.

4 The Cobol standard defines several bizarre data formats that we must support. To do so in a reasonable fashion may require special decode ROMs for ASCII and EBCDIC separate and overpunch signs.

Note that accelerators for functions 1 and 2 will also accelerate the operation of Fortran and SPL programs and of kernel functions like LAT.

SUMMARY

FHP hardware believes that this goal and these requirements to meet this goal are worth investment in special purpose hardware and will add such hardware as appears to be feasible to FHP systems, either in all systems or as special optional Cobol accelerator packages. FHP hardware solicits FHP software support, comment or correction of this position.