SOFTWARE OPTIMIZATION
THROUGH RESEGMENTATION

by

MIKE CLARKSON

presented to:

HP-3000 USERS GROUP MEETING

in

MIAMI, FLORIDA

February 27, 1975

I. INTRODUCTION

During the past couple of years, we at Data Base Management Systems, Inc., have developed quite a few large, sophisticated software packages. These packages tax quite heavily the resources of the HP-3000 and tend to exercise the machine, both hardware and software, to its limits. In order to keep our program development from overwhelming the 3000, we have developed methods for efficiently maintaining our source code and for optimizing the execution of our programs. It is extremely important to have the most efficient software possible when dealing with a computer such as the 3000, which is very sophisticated, but yet has such small memory limitations. The 3000 can perform very well if the programmer will take time to optimize his software so that it complements the 3000's resource management. This paper will outline several areas for program optimization and will expound on the area of program resegmentation.

II. SOFTWARE OPTIMIZATION

The need for program optimization should be intuitively obvious, since it is very unlikely that a program will exist in its most optimal form when it is first written. Since most sophisticated programs will require optimizing before they will run at all, programmers will make a "first approximation" before the program is even run for the first time. From the initial run on, the programmer will continuously learn new short cuts as well as observe his program's running characteristics; thus, there is the continuing need for program optimization. There are many areas available that the programmer can attack in order to make his programs more efficient. Following are several examples:

Since MPE is a relatively unknown factor to most programmers, one should avoid calls to MPE as much as possible. If a call to MPE is unavoidable, one should attempt to call the lowest level intrinsic available in order to avoid unnecessary overhead, even if it means making the call in privileged mode. Privileged mode execution need not be avoided as long as one is very careful of each instruction that is executed while in privileged mode. The higher level, user mode MPE intrinsics have a lot of overhead determining if the user has the capability to perform the desired operations.

Another area for consideration is minimizing disc accesses. If small scratch files are necessary, one should weigh the advantages of a local dynamic array or an "OWN" array as opposed to a disc file (keep in mind, however, that a larger stack could cause more disc accessing by memory management). Also, we have found that it is much more efficient to have our own buffer manager which uses direct access disc files with "no buffering" option, rather than depending on the file system to optimize our disc accesses.

Another area which is not in the scope of this paper, but is well worth mention, is modular, structured programming with clean paragraphing in order to minimize program maintenance time. [1]

Finally, but not exhaustively, is the area of program resegmentation which is discussed in detail in subsequent sections.

III. RESEGMENTATION

We have found that one of the best and yet simple methods of program optimization is resegmentation. Since the memory size of the 3000 is limited to a maximum of 64K words and many SPL programs require more code than available memory, the programmer is able to group his code into segments by using the SPL command "$CONTROL SEGMENT=name". This is called segmenting the code. One should attempt to gather into a segment, procedures which call each other since there is additional run time overhead involved when one procedure calls a second procedure that is in a different segment. If a relatively small procedure is called very often by some other procedure, and the small procedure is in a different segment and cannot be moved, then it is useful to make a copy of the small procedure and make it a subroutine inside the calling procedure. This eliminates both overhead of PCAL execution and segment swapping. We have found that an optimal code segment size is around 2000(octal) words. Reasonable limits are from 1400(octal) to 3000(octal) words. Procedures which are called infrequently such as initialization, termination, and error handling procedures should be grouped into larger segments. One should use as many segments as necessary to make his program run efficiently, but at the same time, should be frugal with segments, since there is a hardware limitation of 256 segments which can be active at any one time.

Not only is it advisable to segment one's code optimally, but at times one should also segment his data, the latter being slightly more complicated. For example, say a program needs a 4000 word buffer at various times during its execution, but it is undesirable to incorporate the buffer into the normal stack, then an extra data segment can be utilized. The intrinsic GETDSEG can be called, giving it the desired buffer size and it will return the DST number of the extra data segment. Then, during further execution whenever it is necessary to access the buffer, call the intrinsic EXCHANGEDB(DST#) and the DB register will then point to the extra data segment and it can be accessed just like normal DB storage. Remember that the regular storage normally accessed via the DB register and the extra data segment cannot be accessed simultaneously. In order to switch back to the regular DB storage, call EXCHANGEDB(0) and the DB register will be returned to its original value. Many extra data segments can be maintained in one program, but it is the responsibility of the program to keep track of each extra data segment's DST number. When the program is through with its processing, it is necessary to release each obtained extra data segment back to MPE by calling the intrinsic RELDSEG(DST#). The program must be in privileged mode in order to call any of these intrinsics. It is important to release all extra data segments back to MPE, or they will be lost until the system is COLDSTARTED (or COOLSTARTED).

Once a program is segmented, it can be run along with one of the available monitoring programs such as TRACE (described in subsequent sections) which will produce statistics describing segment swapping, et cetera. The program can then be resegmented by either recompiling after moving the "$CONTROL SEGMENT" cards or by using the SEGMENTER command NEWSEG. The produced statistics will indicate which procedures call which other procedures in other segments, and using this information, the programmer can regroup procedures which call each other most often. The resegmenting, monitoring process can be performed repeatedly until the most optimal segmentation is achieved.

IV. AVAILABLE PROGRAMATIC ASSISTANCE

Presently, there are three packages available to aid the systems programmer in optimizing his software. They are (1) SAMPLER; (2) AUTOSEG; and (3) TRACE. SAMPLER is a software sampling system used for measuring the relative time spent executing various sections of code. SAMPLER requires an extra clock/TTY interface board to be installed in the system before it can run. The SAMPLER documentation in the appendix completely describes all necessary steps. Be sure to read the complete documentation before attempting to use SAMPLER. AUTOSEG is a performance enhancement tool which provides for the automatic resegmentation of programs based on data gathered under actual program operation. AUTOSEG in its present form is not very useful because it is not quite smart enough to resegment a USL file any more optimally than could casually be done by a programmer. Sketchy documentation does exist in the appendix describing the three functions of AUTOSEG. AUTOSEG does prompt the user for the necessary information. TRACE uses the hardware "trace" facility to collect data pertaining to processes at the time of intersegment transfers caused by PCAL's and EXIT's. This package is very useful in helping the programmer resegment his

programs by hand. The following section describes TRACE in detail and additional information is located in the appendix.

V. TRACE

This section will discuss in detail the software monitoring package TRACE which analyzes procedure calls external to a segment, showing caller, callee, presence of the called segment and frequency. In order to use TRACE, a new version of MPE must be generated and the produced tape must be COLDSTARTED. The version of TRACE to be discussed runs under MPE32000B.00.09. It is important to note that TRACE periodically halts the machine and should therefore be run only when a program is to be TRACEd and the regular operating system should be reloaded before any normal machine usage is resumed.

Following are the steps necessary to generate a TRACE cold load tape:

1. Restore all files from the distribution tape which belong in group/account "OUR.SYS."
2. Patch INITIAL to obtain a data segment for TRACE by changing an "IF FALSE THEN" to an "IF TRUE THEN". The octal instruction to be changed is 25001 which is found around address 2244(octal). Enter the following command sequence:

```
:RUN PATCH
FILE=? INITIAL
? M,2,2244
25001, 0600
? E
```

3. Execute the following jobstream to generate the TRACE version of MPE:

```
:JOB MANAGER.SYS,OUR
:PURGE ININ
:SPL M10M000B,,$NULL,S10S000B,NEWM10M
:SPL ININY,,,NEWM10M
:PREP $OLDPASS,ININ;CAP=PM
:SAVE ININ
:PURGE NEWM10M
:PURGE  EXIN
:SPL M11M000B,,$NULL,S11S000B,NEWM11M
:SPL  EXINX,,,NEWM11M
:SAVE $OLDPASS,EXIN
:PURGE NEWM11M
:EOJ
:JOB MANAGER.SYS
:FILE  SYSTAPE;DEV=TAPE
:TELLOP MOUNT BLANK TAPE ON TAPE DRIVE UNIT
:SYSDUMP  *SYSTAPE
 YES
 H8
                                  << BLANK CARD>>
 NO
 NO
 NO
 NO
 NO
 NO
 NO
 YES                              << SYSTEM PROGRAM CHANGES>>
 ININ,ININ.OUR
EXIN,EXIN.OUR
                                  <<BLANK CARD>>
 NO
                                  <<BLANK CARD>>
:EOD
:EOJ
```

4. Note that EXIN is a USL file and ININ is a PROG file.
5. The tape produced by SYSDUMP should be loaded with the COLDSTART option.

Once a TRACE cold start tape has been generated and a program is ready to be traced, perform the following steps to obtain printer output describing the program's segment activity:

1. COLDSTART the TRACE version of MPE.
2. Run the program to be TRACEd from a terminal, generating an LMAP on the line printer. If the CSTs are not allocated contiguously, abort the program and run it again until the CSTs are contiguous. Make sure the program does not terminate until after step # 3.
3. From the console :ALLOCATE the running program.
4. Allow running program to terminate.
5. From console :RUN TRACE.OUR.SYS - respond to prompts as follows:

```
? INIT  0
? TRACE  %n/%m
```

where n is the first CST number on LMAP and m is last CST number on LMAP

```
?RUN
```

If the system is equipped with an extra clock/TTY interface board, rather than entering 0 with INIT, enter the

decimal DRT number of the clock board (eg. ?INIT 12).

6. Hang scratch tape on tape unit #7 with write ring in.
7. From the terminal run the program to be TRACEd with valid input, output, etc.
8. When the program terminates on the terminal:

   From console continue responding to TRACE prompts as follows:

   ```
   ?EOF
   ?STOP
   ?CLEAR  %n/%m          [see above]
   ?EXIT
   ```

9. From console :RUN TRACERED.OUR.SYS
10. Respond to TRACERED prompt with:

    ?SEGMENT CALLER EXIT

11. Statistics will be printed on line printer.

(Note: If the machine halts during execution of the program being TRACEd, merely hit the RUN/HALT switch and the machine will continue executing. The halts are caused by such things as no write ring in tape or printer being used by another process. Also make sure that the program to be TRACEd is the only program running on the system [ie. besides TRACE.OUR.SYS]).

The next page is an example of one page of the output generated by TRACERED.

PAGE 1 HP3000 SEGMENT TRACE DATA REDUCTION PROGRAM

SEG	ENTRY CNT.	ABSENCES	%ABS.	STT	ENTRY CNT.	%SEGE	TIMED ENT.	%STTE	TIME/C	SDEVT/C	PIN SEG DELTP	CALLS	%STTE
121	564	73	12.9	001	564	100.0	432	76.6	.28	.13	026 144 00234	564	100.0
122	2	1	50.0	001	1	50.0	0	.0	.00	.00	026 136 00402	1	100.0
				002	1	50.0	0	.0	.00	.00	026 136 00375	1	100.0
126	72	13	18.1	001	24	33.3	24	100.0	.16	.01	026 130 02041	24	100.0
				002	24	33.3	17	70.8	.24	.00	026 130 02060	24	100.0
				003	6	8.3	4	66.7	.30	.28	026 133 00364	6	100.0
				004	11	15.3	11	100.0	.27	.04	026 133 00356	6	54.5
											026 136 03337	5	45.5
				005	5	6.9	5	100.0	.12	.00	026 133 00346	5	100.0
				006	1	1.4	1	100.0	.19	.00	026 133 00331	1	100.0
				007	1	1.4	0	.0	.00	.00	026 133 00342	1	100.0
127	45	40	88.9	001	45	100.0	4	8.9	1.39	.32	026 133 00227	45	100.0
130	512	61	11.9	001	256	50.0	135	52.7	1.21	.21	026 133 00173	256	100.0
				002	256	50.0	150	58.6	1.11	.15	026 133 00151	256	100.0
131	2	2	100.0	006	1	50.0	0	.0	.00	.00	026 133 00250	1	100.0
				010	1	50.0	0	.0	.00	.00	026 133 00252	1	100.0
132	1	1	100.0	001	1	100.0	0	.0	.00	.00	026 131 00642	1	100.0
133	342	17	5.0	001	342	100.0	20	5.8	.47	.04	026 122 00767	8	2.3
											026 134 00773	4	1.2
											026 135 01111	266	77.8
											026 136 03421	64	18.7
134	1	1	100.0	001	1	100.0	0	.0	.00	.00	026 135 00467	1	100.0
135	165	30	18.2	001	119	72.1	115	96.6	.09	.00	026 133 00621	44	37.0
											026 134 00735	4	3.4
											026 136 03136	2	1.7
											026 136 03215	69	58.0
				002	2	1.2	2	100.0	.69	.11	026 133 00272	2	100.0
				003	40	24.2	0	.0	.00	.00	026 133 00216	40	100.0
				010	4	2.4	3	75.0	.24	.01	026 134 00774	4	100.0
136	15	5	33.3	006	1	6.7	0	.0	.00	.00	026 122 00222	1	100.0
				010	5	33.3	5	100.0	.09	.00	026 153 00066	1	20.0
											026 157 00006	1	20.0
											026 160 00044	1	20.0
											026 161 00106	1	20.0
											026 163 00124	1	20.0
				012	9	60.0	7	77.8	.09	.00	026 122 00731	9	100.0
137	1	1	100.0	001	1	100.0	0	.0	.00	.00	026 135 01334	1	100.0
140	35	31	88.6	001	35	100.0	0	.0	.00	.00	026 133 00421	35	100.0
141	20	12	60.0	001	1	5.0	1	100.0	.38	.00	026 147 00043	1	100.0
				002	19	95.0	9	47.4	1.42	.44	026 147 00100	7	36.8
											026 147 00206	12	63.2

There are sixteen columns of values, four of which are non-zero because there was an extra clock/TTY interface board in the system when the program was TRACEd.

Following is a description of each of the sixteen columns of information on the TRACE reduction printout (the first eleven columns comprise the called segment statistics and the last five columns comprise the caller statistics):

1. three digit octal CST (code segment table) number of the segment
2. total PCAL entries to the segment
3. number of PCAL absences only (EXIT absences cannot be traced)
4. absences as a percentage of total entries
5. three digit octal STT (segment transfer table) entry number
6. total calls to that STT entry
7. STT entries as a percentage of the total entries
8. number of timed STT entries (total number of STT entries on which it was possible to gather timing information)
9. number of timed entries as a percentage of the total STT entry count
10. average time per STT call in milliseconds with all TRACE overhead removed (this time is the time spent in this segment less the time spent in other segments which were being traced and were called by this segment)

11. standard deviation of the time per call in milliseconds
12. three digit octal process identification number of caller
13. three digit octal CST number of caller
14. five digit octal delta P (ie. offset into calling segment to PCAL made to called segment)
15. total calls made to this STT by the caller from this delta P
16. number of calls as a percentage of the STT entries

An example of how to recognize potential optimizations is as follows: Notice that the segment at CST #140 was called 35 times which caused 31 disc accesses or 88.6% of the total. This is a very high percentage to have causing disc accesses. The only entry point in the segment which was called was STT #001 and it was called from only one place, which was 421(octal) words into the segment at CST #133. Now one would look at the LMAP describing these segments and determine which relative segments within the program were loaded at CST entries 140 and 133. Then looking at the PMAP of the traced program, one would analyze the present size of the segment corresponding to CST #133 to see if it has room to add the code of the RBM located at STT #001 of the segment corresponding to CST #140. If there is room in the segment, the RBM can be moved by using the SEGMENTER

command NEWSEG or by changing the appropriate "$CONTROL SEGMENT" card in the SPL source and re-compiling the procedure. This one optimization would save 31 disc accesses. It is important to also determine if there are any other procedures in the CST #140 segment which make internal segment calls to the STT #001 procedure; since they could make more than 35 calls, the expected optimization could cause a degradation.

The timing information found on the TRACE reduction printout is also useful. Through close observation of the timing statistics, one can learn where in the code a major portion of execution time is being spent. For example, two entry points (STT #001 and STT #002) in the segment loaded at CST #130 were each called 256 times. Note from column ten of the TRACE reduction printout that each call to these entry points averaged 1.21 milliseconds and 1.11 milliseconds respectively in duration. This time factor is somewhat high relative to most of the other TRACEd STT entries. Given this fact, one should analyze the code of the procedures which correspond to these STT entries and determine if any code optimization could be performed in order to help these procedures execute more quickly.

Following are some helpful hints for using TRACE that should be utilized. In order to get good timings for every RBM of the program,

the USL file should first be run through AUTOSEG's NICRSP processor to generate worst case segmentation so no internal segment PCALs are executed. This is done by (1) make a copy of the USL file; (2) :RUN NICRSP.OUR.SYS and give it the name of the USL copy; (3) then prepare the resegmented USL copy and run it through TRACE. There are two things to keep in mind when using TRACE: (1) TRACE.OUR.SYS runs in privileged mode so the user needs privileged and account capabilities; (2) TRACE.OUR.SYS uses its own tape driver for I/O so MPE must not be allowed to use any other tape drive on the same controller that TRACE is using. An extra clock/TTY interface board may be easily added to the system by assigning the extra board an unused DRT number and plugging the board into the system. MPE need not be reconfigured to recognize the new DRT.

VI. CONCLUSIONS

We hope that the reader has gained an appreciation for program optimization, especially through resegmentation. We at DBMS have enjoyed performance improvements of as much as 400% to 500%, using resegmentation alone. Resegmentation is an easy first pass to make before getting into the optimizations that require such things as changing existing code. The software monitoring packages described in this paper will be released on the contributions tape and are stored in the group/account "OUR.SYS". If any system anomalies occur during the use of TRACE, it should be assumed that they are attributable to that usage and, therefore, no problem reports should be submitted to Hewlett-Packard. The appendix contains a description of the contents of OUR.SYS which will be on the tape that is distributed to the users. If there are any questions concerning the software monitoring packages, please direct them to me at the following address:

Mike Clarkson, Vice President
Data Base Management Systems, Inc.
12100 N.E. 16th Avenue
North Miami, Florida 33161

REFERENCES

1. HP-3000 Users Group Proceedings, May 10, 1974, p. 69,

PROGRAM PERFORMANCE, by Stephen Sontz

APPENDIX

CONTENTS OF GROUP OUR.SYS

1.	SAMPLE	
2.	SAMPLES	«SOURCE»
3.	SAMPLING	
4.	SAMPLINS	«SOURCE»
5.	DRS	
6.	DRSS	«SOURCE»
7.	SYSGEN	«OBSOLETE»
8.	M10M000B	«PATCH DECK»
9.	M11M000B	«PATCH DECK»
10.	S10S000B	«SOURCE»
11.	S11S000B	«SOURCE»
12.	EXINX	«PATCH DECK»
13.	ININX	«PATCH DECK»
14.	ININY	«PATCH DECK»
15.	TRACE	
16.	TRACES	«SOURCE»
17.	TRACERED	
18.	TRREDS	«SOURCE»
19.	NICRSP	
20.	NICRSS	«SOURCE»
21.	ORSP	
22.	ORSS	«SOURCE»
23.	ORTDRP	
24.	ORTDRS	«SOURCE»

SOFTWARE SAMPLING SYSTEM

[redacted]

March 7, 1973

Contents

		Page
A.	Introduction	1
B.	Hardware Requirements for Sampler	2
C.	Sampler Operation and Format	3
D.	Setting up the Sampler	4
E.	The Data Reduction Program	6
F.	A Case Study	8
G.	Miscellaneous Comments	11

A. Introduction

The Software Sampling System is a useful tool for measuring the relative time spent executing various sections of code. The system consists of three parts:

I. The Sampler interrupts the CPU at a preselected frequency and records on tape the following information for any number of selected code segments:
   1. Number of code segment interrupted.
   2. Relative P in that code segment prior to the interrupt.
   3. Approximate size of the stack.

   All user interaction with the sampler is done via the Setup program.

II. There is a setup program with provision for initializing the sampler, selecting segments to be sampled and setting the sampling interval.

III. A Data Reduction Program is available to process the data tape. This program will provide the following histograms:
   1. Relative time spent in each segment sampled.
   2. Relative activity within selected segments.
   3. Size of stack from DS to S for each sample.

A. Introduction

The Software Sampling System is a useful tool for measuring the relative time spent executing various sections of code. The system consists of three parts:

I. The Sampler interrupts the CPU at a preselected frequency and records on tape the following information for any number of selected code segments:
   1. Number of code segment interrupted.
   2. Relative P in that code segment prior to the interrupt.
   3. Approximate size of the stack.

   All user interaction with the sampler is done via the Setup program.

II. There is a setup program with provision for initializing the sampler, selecting segments to be sampled and setting the sampling interval.

III. A Data Reduction Program is available to process the data tape. This program will provide the following histograms:
   1. Relative time spent in each segment sampled.
   2. Relative activity within selected segments.
   3. Size of stack from DB to S for each sample.

B. Hardware Requirements for Sampler

To run the Sampler in its present form requires the following hardware:

1. A 3000 System which will run MPE and for which the maximum DRT number has been set to 70 greater than required by any device.
2. An extra clock/TTY board with a known distinct device number.
3. A dedicated tape drive and controller.
4. An extra terminal from which to run the setup program.

C. Sampler Operation and Format

The sampler is an interrupt handler which resides in the upper DRT table together with its buffer. This interrupt handler is activated by timer interrupts from the extra clock/TTY interface board.

The sampler then traces stack markers backwards to determine which code segment was executing prior to the interrupt. This code segment number is used to index into an internal segment bit-table which determines whether this segment's data should be recorded. If the segment has been selected the contents of its status register is entered in the sampler's buffer together with the relative location (within that segment) of the next instruction to be executed. The relative value of the top of the stack is also entered and a user stack/interrupt control stack bit is set.

If the buffer is full interrupts are disabled and the buffer is written directly onto tape. If constant timing interval has been specified at setup time the count register of the timer is cleared just prior to exit. However, if the randomized jitter has been specified on the timing interval, a pseudo-random number between 0 and 255 is loaded into the count register. The switch register is used as one of the parameters of the random number generator to allow the operator to influence the sequence generated.

Tape records produced by the sampler are 128 words in length consisting of 16 logical records of 8 words each. The contents of each logical record is as follows:

Word 0. Status word of interrupted procedure.
Word 1. Relative location of next instruction to be executed within that procedure.
Word 2. S-DB if on interrupt control stack and (DB-S) if on user stack. Note that bit 0 indicates ICS/User Stack.
Word 3. Unused at present.
Word 4. Record type. Bit 0 is 1 to indicate that the record was produced by the sampler.
Words 5, 6, 7. Unused at present.

D. Setting up the Sampler

Before starting-up MPE ensure that the extra clock/TTY interface board is inserted in the highest priority polled I/O slot. Coolstart MPE and reply "Y" to the question "ANY CHANGES?" When the question "HIGHEST DRT NUMBER = XX.?" is printed reply with a number which is 70 greater than the highest DRT used by the system. No other changes are required. Mount a tape fitted with a write ring and select unit 0. When the system is up log on and :RUN SETUP1. The machine replies "TRACER/SAMPLER" and prompts for commands with "?". The first command to be entered must be "$LOAD" (as described below) to initialize the sampler after which sampler commands may be entered in any order.

Commands specifically affecting the sampler are strings of length less than 7 having "$" as the first character. The second character determines the command, and following characters are optional. All parameters are in standard 3000 form ("%" prefix for octal numbers).

$LOAD <timer device number>, <starting drt number>

This loads the sampler into the drt table starting at <starting drt number> (which must be at least one greater than the highest drt entry used by the system). The <timer device number> is the device number of the extra clock/TTY interface.

$TIME <count> <quantum>

This command sets the time between completion of one sample and start of the next.

<quantum> is "U" = microseconds
"M" = milliseconds
or "S" = seconds

<count> is an integer less than 64K. The timer is set-up so that the precision of sampling interval is better than 0.1%. This command also halts the sampler if it was running.

$RAND This command introduces a random jitter to the sampling interval. The purpose of this jitter is to avoid the possibility of the sampling intervals becoming synchronized with the sampled code. The switch

register is used as one of the parameters for the random number generator. Statistical tests have shown that %176523 is a good number to set in the switch register. Zero in the switch register resets the $RAND command.

$SET <segment range list>

This command specifies the segments to be sampled. <segment range list> is a list of ranges of segment numbers separated by commas, where a range is a segment number or first segment number/last segment number.*

$CLEAR <segment range list>

This command inhibits sampling for all segments in the ranges of the <segment range list>*.

$GO Starts the sampler. (At least a "$LOAD" and a "$TIME" command should have been issued previously.)

$HALT Halts the sampler.

The following general commands are also useful.

EOF Write an end of file mark on the tape.

DEBUG Enter debug.

EXIT Terminate the setup program.

There are various self-explanatory error messages. However, at present there is no error checking on the parameter values of the "$LOAD" command. Since the program runs in privileged mode care must be exercised when entering the $LOAD command to avoid crashing MPE.

---

*<segment range list>:: = <segment range> |<segment range list>, <segment range>
<segment range>:: =<segment number> | <segment number>/<segment number>
<segment number>:: = any integer between 0 and 255

E. The Data Reduction Program

A program is availabel to process the data tape produced by the sampler. Output is in the form of histograms of segment usage and relative activity within each segment.

The data reduction program is activated by the following sequence of commands:

```
:FILE FTN09; DEV = LP ; CCTL
:FILE FTN07; DEV = TAPE; REC = 128, 1, F, BINARY; NOLABEL
:RUN -TAPE11 TAPE5
```

The program makes a first pass through the tape, processing all records, if any, produced by the old (now obsolete) trace routine and obtaining normalizing information about sampler records. At the end of the first pass the total number of records on the tape is printed on the line printer. Assuming all records were produced by the sampler "ENTERING DATA REDUCTION FOR SAMPLER" is printed on $STDOUT, followed by a request for segments for which an internal histogram is desired. Enter these in the free format <segnum>, <p interval> followed by carriage return where <segnum> is the number of the segment and <p interval> is the "width" of the bars of the histogram. If <p interval> is zero the program chooses a bar width such that the histogram for that segment will fit on a single page. Of course, if <p interval> is 1 the histogram will show the number of samples which hit each individual instruction.

Prefix octal numbers with "%". If <segnum> is a valid number for a segment but no samples interrupted that segment the machine replies "BAD SEGMENT NUMBER". To terminate the list of segment number enter a <segnum> greater than 255.

A response of "Y" to the question "COMPLETE PLOT OF STACKSIZE?" will product a histogram of the value of S-DB for every sample. This histogram of stacksize consumes much time and paper and is usually of little interest.

A second pass through the tape is then made, after which the histograms requested earlier are printed on the line printer. The number of the segment

is shown at the top of each histogram. Each bar in an internal histogram represents the number of samples which hit the range of code locations shown at the left of the bar. The actual number of samples is shown at the right. Bar lengths are normalized separately for each segment to emphasize the high-usage sections of code within a segment.

The machine then prompts "SEGMENT HISTOGRAM?" on $STDOUT. A reply "Y" will cause a histogram of segment usage to be printed on the line printer. This segment histogram shows the number of samples which hit each segment. For each segment which was sampled at least once a bar appears in the histogram showing at the left, the segment number and, at the right, the number of samples which hit that segment. Once again the bar lengths are normalized so that the longest bar spans the page.

The machine then prompts "EXIT?" on $STDOUT. A reply "Y" terminates the program, whereas "N" rewinds the tape in preparation for repeating the second pass.

F. The Sampling System in Action: A Case Study

This section describes an example of the use of the software sampling system to improve the performance of a specific program, namely the data reduction program described in section E (henceforth referred to as DRS.)

DRS was written in Fortran and run originally on the HP 2100 and DOS-M. When transferred to HP 3000 Fortran and run under MPE execution time became distressingly large. In an attempt to improve this the sampler was run while DRS was executing. The following is a description of the procedure used.

The system was started up with the extra DRT space (maximum DRT set to 101) and with the extra clock/TTY interface board (device number %13 = 11) inserted in a polled I/O slot. Two terminals were required for the measurement, a running terminal and a sampling terminal. Since the only tape controller available would be required by the sampler it was not possible to run DRS with input from tape. The utility program FILECOPY was used to copy a data tape (from a previous sampling session) to a disc file called DATAS, and DRS was run using DATAS as the input file.

Before running DRS from the running terminal the following sequence of commands was entered from the measuring terminal (system output underlined):

```
:RUN SETUP1

TRACER/SAMPLER

? $LOAD %13, 31
? $SET  0/255
? $TIME 4 M
? $RAND
        (%176523 set in the switch register.)
? $CLEAR %107
        (%107 was segment number of the dispatcher.)
```

At this point the sampler is set up to interrupt at 4 millisecond intervals with random jitter (i.e. time between samples is randomly chosen in the range

3.75 to 4 milliseconds). All segments are set to be sampled except the dispatcher (thus no tape will be written while the system is paused waiting for input).

At this time DRS was started on the running terminal. As soon as DRS started executing the command

```
? $GO
```

was entered on the measuring terminal to start the sampler. When DRS terminated, the sequence of commands

```
? $HALT
? EOF
? EXIT
```

was entered on the measuring terminal to stop the sampler, write an end of file mark on the tape and then terminate the setup program.

The data tape was then manually rewound and DRS was run with the data tape as input to produce the histogram of segment activity shown in Figure 1. This histogram immediately indicates that the Formatter consumes more than twice the CPU time that the actual DRS code does. To determine exactly where in the formatter this time was spent histograms of activity within segment 132 were produced (during subsequent repetitions of the second pass). One such histogram is shown in Figure 2. Checking the high usage sections of code suggests that formatter time could be reduced considerably by changing some data definitions, and by modifying the statement which reads from tape (from a do-implied list to an array read), thereby reducing the number of calls to the formatter.

These changes were made to DRS and the above measurement procedure was repeated. The resulting histogram of segment usage is shown in Figure 3. (Note that in this measurement the dispatcher was sampled also.) This shown that the relative formatter time has been reduced to less than one quarter of its previous value. In fact total execution time of DRS was reduced 58%.

3.75 to 4 milliseconds). All segments are set to be sampled except the dispatcher (thus no tape will be written while the system is paused waiting for input).

At this time DRS was started on the running terminal. As soon as DRS started executing the command

? $GO

was entered on the measuring terminal to start the sampler. When DRS terminated, the sequence of commands

? $HALT
? EOF
? EXIT

was entered on the measuring terminal to stop the sampler, write an end of file mark on the tape and then terminate the setup program.

The data tape was then manually rewound and DRS was run with the data tape as input to produce the histogram of segment activity shown in Figure 1. This histogram immediately indicates that the Formatter consumes more than twice the CPU time that the actual DRS code does. To determine exactly where in the formatter this time was spent histograms of activity within segment 132 were produced (during subsequent repetitions of the second pass). One such histogram is shown in Figure 2. Checking the high usage sections of code suggests that formatter time could be reduced considerably by changing some data definitions, and by modifying the statement which reads from tape (from a do-implied list to an array read), thereby reducing the number of calls to the formatter.

These changes were made to DRS and the above measurement procedure was repeated. The resulting histogram of segment usage is shown in Figure 3. (Note that in this measurement the dispatcher was sampled also.) This shown that the relative formatter time has been reduced to less than one quarter of its previous value. In fact total execution time of DRS was reduced 58%.

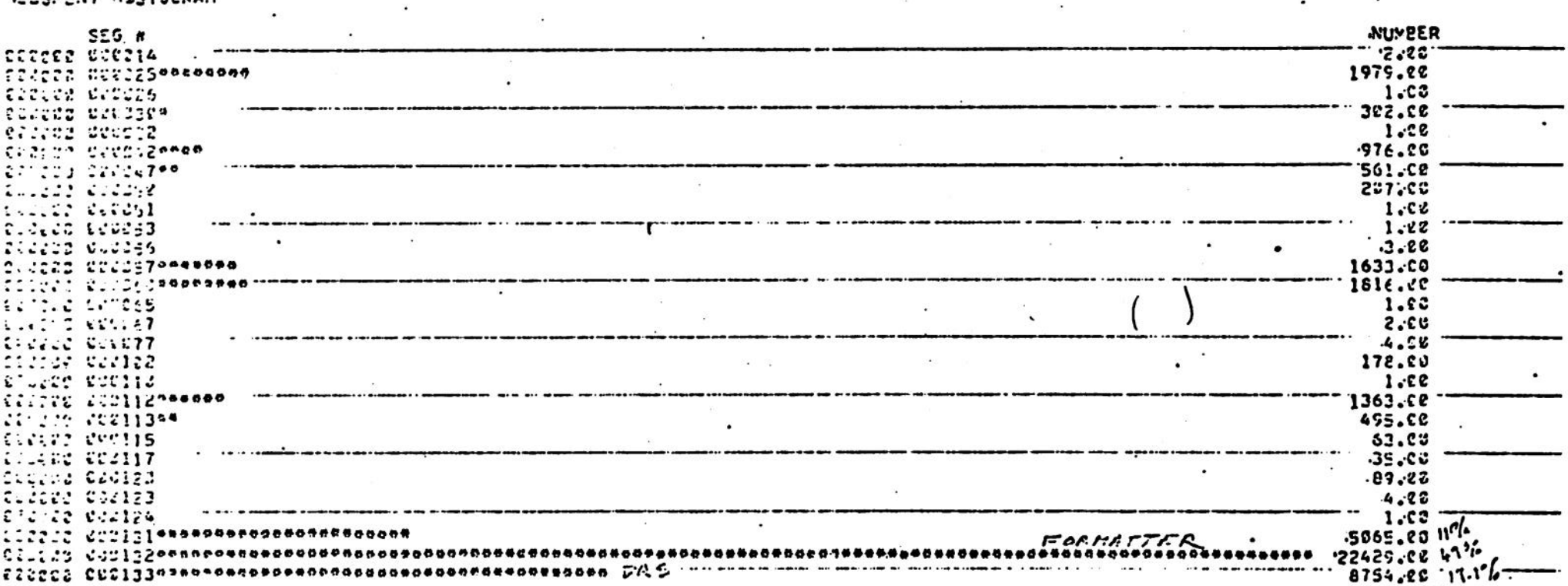


Figure 1. Histogram of segment usage (excluding the dispatcher) for original version of DRS.

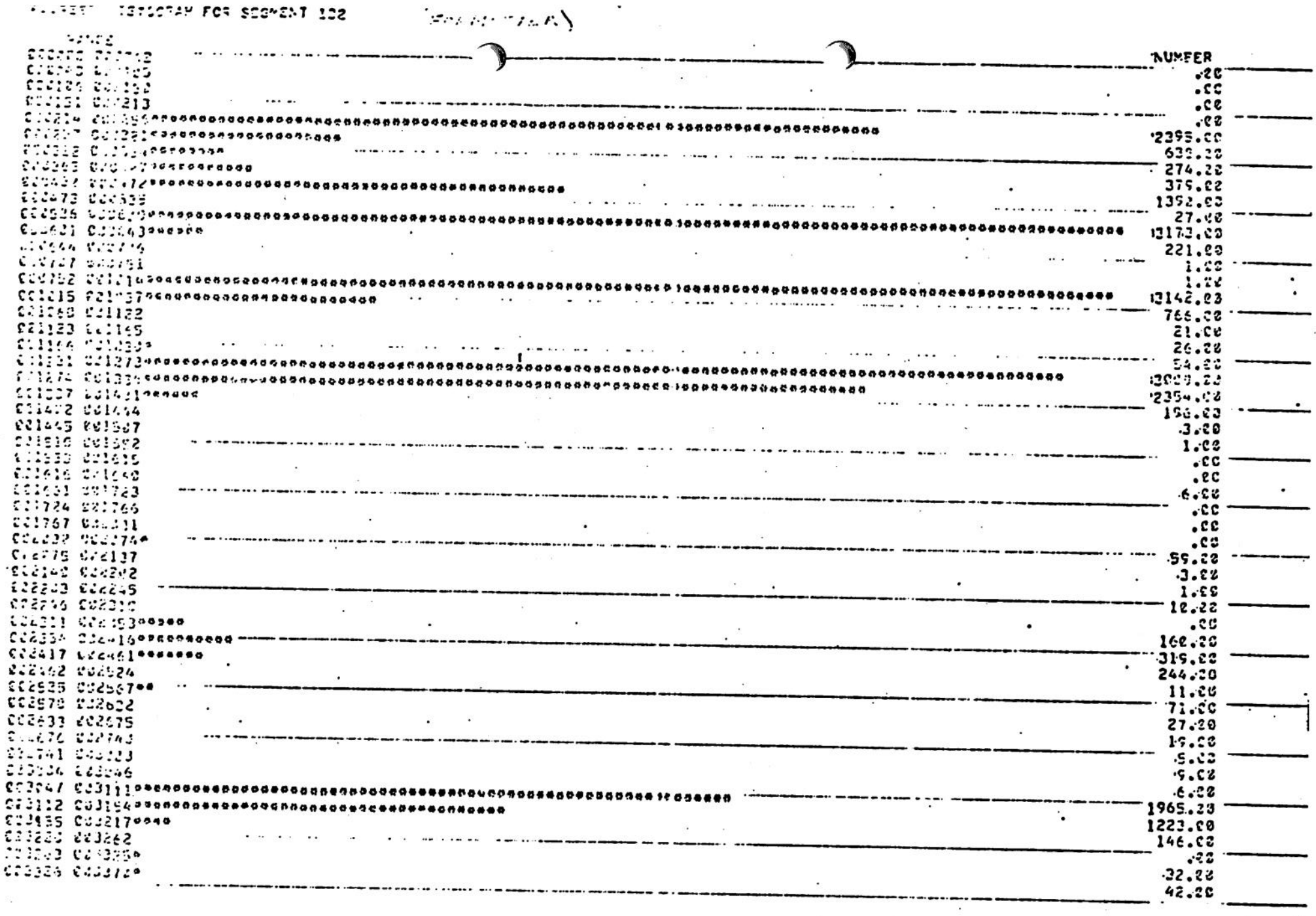


Figure 2. Histogram of activity within Formatter (original DRS).

1 SEGMENT HISTOGRAM

SEG #		NUMBER	
000000	000023	1093.00	
000000	000031	1339.00	
000000	000042	3.00	
000000	000045	5819.00	
000000	000057	9120.00	
000000	000071	4646.00	
000000	000101	2280.00	
000000	000105	3.00	
000000	000106	11741.00	DISPATCHER 14%
000000	000110	2.00	
000000	000111	3095.00	
000000	000113	1438.00	
000000	000115	3.00	
000000	000116	4.00	
000000	000130	2.00	
000000	000131	9393.00	FORMATTER 14.4%
		15086.00	DRS 23%

Figure 3. Histogram of segment usage for DRS with improved formatter calls.

Although the formatter time was reduced it was still significant. To determine whether this could be reduced any further histograms were produced showing activity within the formatter. Figure 4 shows the default histogram with bar width of %43 (obtained by entering "%130, 0" when prompted for segment number of DRS). Clearly most of the formatter time is spent in the section of code between %430 and %472. To examine this in more detail a histogram was produced with unit bar width (by entering "%130, 1" when prompted for segment number) showing the relative usage of each instruction. Figure 5 shows a portion of this detailed histogram. It was interesting to note that the instructions in locations 467, 470 and 471 account for 75% of the formatter time and 10% of the total execution time! Examining the formatter code shows that these instructions comprise the inner loop of a FOR statement used to fill a buffer with blanks.

Rather than attempting to modify the formatter, a tape reading routine was written in SPL using file system instinsics and linked into DRS thereby bypassing the formatter. This version of DRS was more than three times faster than the original version. Its segment usage histogram is shown in Figure 6.

The sampling system could doubtlessly be used to improve the performance of DRS still further by concentrating now on the program itself and possibly the file system.

ADDRESS HISTOGRAM FOR SEGMENT 130 (FORMATTER)

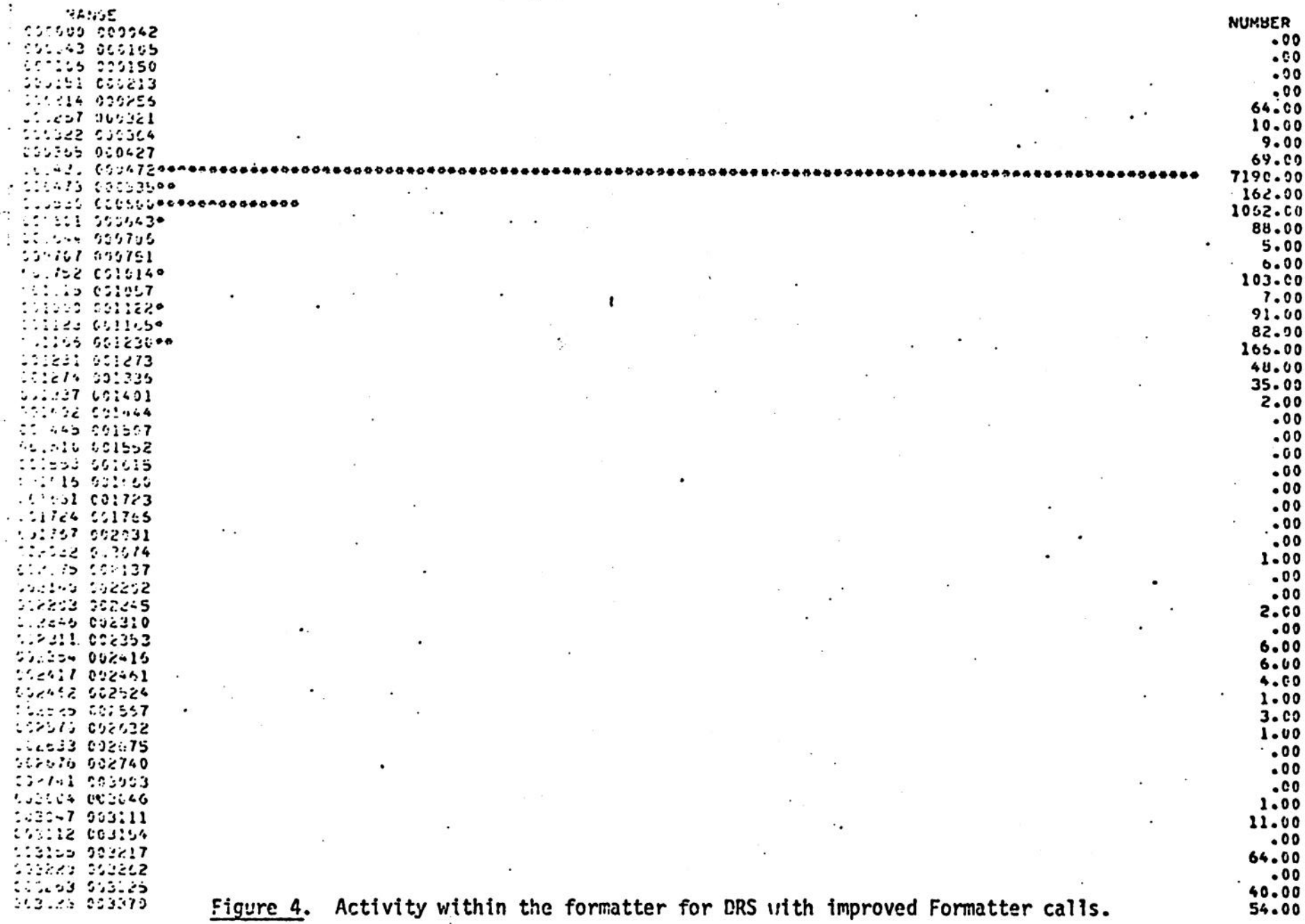


Figure 4. Activity within the formatter for DRS with improved Formatter calls.

Figure 5. Detail of Formatter Activity (compare Figure 4).

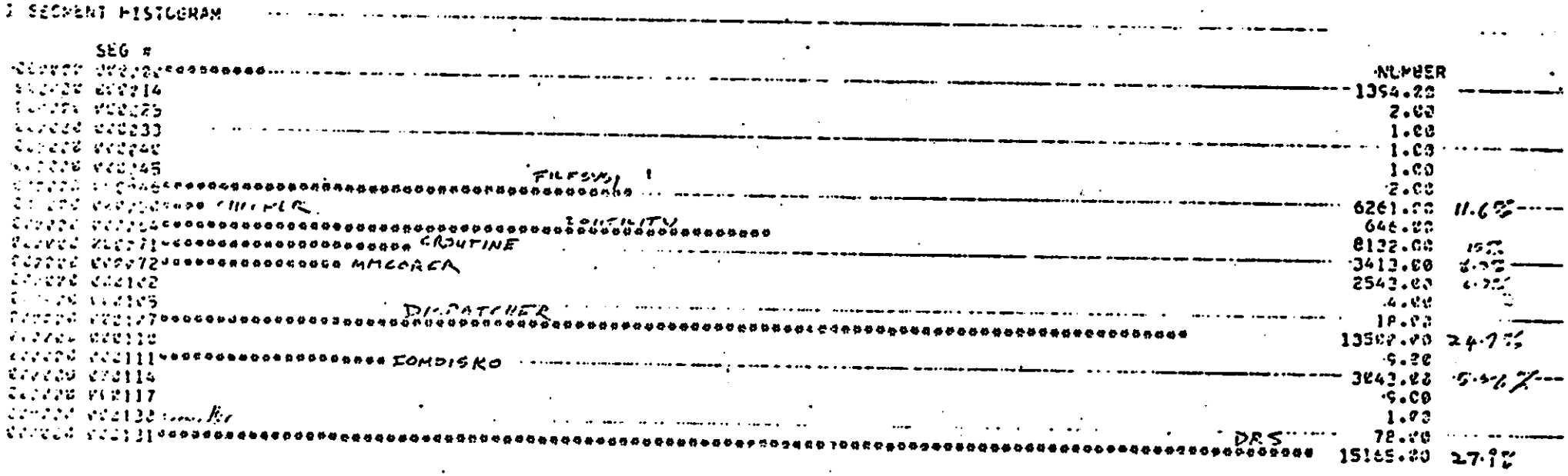


Figure 6. Segment usage of DRS using direct file system intrinsics.

G. Miscellaneous Comments

It must be appreciated that the sampler does disturb the system. At least the timer interrupt will cause an extra dispatch if the CPU is operating on the user stack. If the sampling interval is made too small the sampler might be measuring the effects of its own perturbations. A sampling interval of greater than 3 milliseconds has been found to yield accurate measurements. However, when measuring events of short duration shorter sampling intervals may be necessary. In such cases it is important that the sampler does not interrupt the dispatch which it causes. To prevent this the sampling interval should never be less than 350 microseconds. If the sampler does interrupt its own dispatch the CPU will be locked up in an infinite loop. To terminate this loop the sampler can be manually disabled by setting bit 0 to 1 in the word at absolute location

4* <starting drt number> +3.

The sampling interval selected by means of the $TIME command is the time between exit from the sampler and the subsequent interrupt returning to the sampler. Thus the time spent in the sampler itself is excluded and the only perturbation is the possible extra dispatch (approximately 300 microseconds). However, relative timing of other I/O operations is disturbed. When measuring programs with high I/O activity longer sampling intervals are recommended.

Overhead when sampling all segments at 3 millisecond intervals causes approximately 20% degradation in throughput.

In order to measure activity in segment zero (external interrupts) it is necessary to ensure that the extra clock/TTY interface is polled for highest priority. This will also guarantee the precision of the sampling interval. However, unless precise sampling intervals are specifically desired it is strongly recommended that the randomizing option ($RAND) be used. When using constant sampling intervals the resultant histograms have been seen to exhibit spurious spikes due to synchronization of the sampler and the sampled program. The randomizing option has produced reliable data in all cases.

The sampling system is not particularly elegant and many improvements could be made in its operation. However it is a useful tool for determining which sections of code consume the most processing time. These results are sometimes surprising even to the programmer who wrote the code. Optimization of software can certainly be accelerated if it is possible to identify the "10% of code which consumes 90% of CPU time".

[redacted]

DATE May 1, 1973

TO Measurement Distribution

SUBJECT Modifications to The Software Sampling System

The original sampler used the switch register to provide a parameter to the random number generator for the randomized timing interval option. To free the switch register this paramter is now fixed and resides in the sampler's data area. A "$FIX" command has been added to reset the "$RAND" command. The default is a constant sampling interval; "$RAND" introduces randomized jitter and "$FIX" resets the sampling interval to be constant.

The magnitude of the randomized jitter has been doubled by using a random number between 0 and 511 to load the count register of the extra clock/TTY board. This requires that randomization should not be used for sampling intervals less than 600 microseconds. However, the greater randomness of the samples improves the validity of results. For most measurements a sampling interval of 11 milliseconds with randomized jitter will produce reliable results. In this case the sampling interval will vary randomly from 6 to 11 milliseconds ensuring that dispatches caused by the sampler are not sampled.

To conserve tape and reduce the overhead caused by the sampler the output record format has been modified. Two word records are now produced with a blocking factor of 64. The first word of each record contains the status register of the interrupted segment and the second word contains its P register. The data reduction program has been modified to handle the new format. Of course, stack size statistics are no longer available but could be resurrected if necessary.

The new sampler can be set up using the program SETUP9 and the associated data reduction program is TAPE9.

Overhead caused by sampling all segments with a randomized sampling interval of 11 milliseconds is 7%. The Central Limit Theorem predicts that 25,000 samples produce results accurate to 1% of total number of samples with 99.9% confidence. Indeed, experiments have shown that segment histograms produced by the sampler agree to this extent with measurements made using the SUM hardware monitor. These figures do not account for artifacts caused by disabling interrupts.

It should be noted that if interrupts are enabled just prior to an exit the sampler can only interrupt after the exit instruction has executed. This can produce spikes in the histograms of segments calling procedures which disable interrupts. The aberration will be avoided if there is at least one instruction between the enable and the exit.

In Sampler EOF now rewinds tape.

In Debug Use, :FILE FTN07; DEV=TAPE; REC=128,1,F; NOLABEL

DATE May 14, 1974

SUBJECT Program Sampling

The 3000 System Section has a program, SAMPLER, which determines relative time spent in the various portions of a program. SAMPLER is very useful in optimizing program code by determining what sections of a program is most heavily used. Unfortunately SAMPLER requires hardware and software modifications to a 3000 system to run. A new version of SAMPLER called SAMPLE is now available which requires no hardware changes and only a minimal software change. Specifically about seven lines of SPL code must be added to procedure TIPC of EXIN. Some capabilities have been lost in SAMPLE that was available in SAMPLER. However the lost capabilities only affect users who are sampling system code segments. For user code segments SAMPLE is fully as capable as SAMPLER. SAMPLE is available for internal use from the 3000 System Section. A write up on SAMPLE has been enclosed for evaluation.

DL/kg

DL

INTRODUCTION

SAMPLER is a package of three programs for use in measuring the relative time spent within various portions of a program. The three programs are:

SAMPLE which the user runs to initiate the sampling of the program code,

SAMPLING which performs the actual sampling and

DRS which reduces the data generated by SAMPLING to a user readable form.

OPERATING INSTRUCTIONS

1. Load the program to be sampled and obtain the CST numbers of the segments to be sampled.
2. Run SAMPLE.
3. Run DRS to reduce the data tape created by SAMPLE.

SAMPLE writes the sample data on a file called SAMTAPE which is assumed to be a magnetic tape but can be a disk file. DRS reads the sample data from a file called DRSTAPE, also assumed to be a magnetic tape and lists the output on a file called DRSLIST, assumed to be a line printer.

MESSAGES

MACHINE ID? From SAMPLING. Type in name of 3000 that test is being run on.

SAMPLE ID? From SAMPLING. Type in identification for sample run.

SAMPLING INTERVAL = 50 MS? From SAMPLING. Type "Y" if a sampling interval of 50 milliseconds is to be used, else type the number of milliseconds between samples. The minimum interval is 10 milliseconds and the maximum interval is 1000 milliseconds. The shorter the interval the greater the number of samples that will be obtained in a given time span. The larger the number of samples, the more valid the results will be. However a short sampling interval will result in considerable degradation of system performance. At 10 milliseconds the system can be expected to run 70 percent slower, at 50 milliseconds degradation will be about 15 percent. A minimum of 7 percent degradation is to be expected.

CST #'S? From SAMPLING. Type in a list of CST numbers of segments to be sampled. The numbers are assumed to be octal and are typed in the following format

```
<CST list>::=<number range>|
            <number range>,<CST list>
<number range>::=<CST number>|
<CST number 1>/<CST number 2>
```

<CST number 1> should be less than <CST number 2> and specifies that all segments with CST numbers between <CST number 1> and <CST number 2> inclusive is to be sampled. <CST list> can be continued onto a second line by typing an &.

Example:

```
231, 240/243, &
277
```

Indicates that segments 231, 240, 241, 242, 243 and 277 are to be sampled.

PRINT CSTAB(*)? From SAMPLING. Type "Y" to get a listing of CST numbers that will be sampled else hit carriage return.

TYPE 'STOP' TO STOP: From SAMPLE. Type "STOP" at any time to stop sampling.

UNABLE TO CREATE SAMPLING PROGRAM From SAMPLE.
Possible causes are

1. SAMPLE do not have PH capability.
2. User or account does not have AS priority.
3. SAMPLING exists in another group or account.

START = <CST number 1> > STOP = <CST number 2>.
From SAMPLING. Self-explanatory. If this error occurs, check CSTAB(*) afterwards.

*** BAD NUMBER *** From SAMPLING. A bad CST number was inputed. This message will be preceded by <CST list> up to and including the errant number but not beyond. Most likely cause is typing in decimal CST numbers or extraneous % preceding CST numbers. If this error occurs, check CSTAB(*) afterwards.

UNABLE TO OPEN SAMTAPE. From SAMPLING. Will be followed by file error information.

UNABLE TO WRITE HEADER From SAMPLING. Will be followed by file error information.

QUIT P=11 This is a MPE message. SAMPLING QUITs with P=11 when an error occurs on SAMTAPE.

DRS OUTPUT

DRS prints:

1. A header page which is self-explanatory.
2. A summary histogram showing relative time spent in each segment that was sampled.
3. Detail histograms of each segment that was sampled.

The summary histogram consists of:

1. The CST number of the segment.
2. The histogram.
3. The number of samples taken from the segment, percentage of total samples the numbers represents and cumulative percentages.

The user will be prompted for how detail he wants the detail histograms to be. The prompt is

SEG # <CST number> (<number of samples>SAMPLES)?

Hit carriage return to omit the histogram of this segment. Type zero to get a one page histogram. DRS will scale the histogram to fit in one page. Type an integer n to get a histogram where the interval are n words wide. The detail histograms consists of:

1. Segment CST number.
2. Two PB relative addresses in octal specifying the beginning and end of each interval.
3. The histogram.
4. The number of samples in the interval.
5. The percentage the number is of the total number of samples for this segment.
6. Cumulate percentage.

DRS contains a restart facility and will ask the user if he wishes to restart. Type "Y" or "N" as desired.

NOTES:

SAMPLE and SAMPLING require PH and PN capability.

SAMPLING requires AS priority.

AUTOMATIC PROGRAM RESEGMENTATION

DATE: February 1, 1974

SUBJECT: Automatic Program Resegmentation on the HP 3000

A new performance enhancement tool has been developed. This tool provides for the resegmentation of programs based on data gathered under actual program operation. The resegmentation system will be discussed first, followed by a case study.

The resegmentation system consists of three programs, and utilizes the segment trace facility to gather the data necessary for resegmentation. See Figure 1 for an overall view of the operation. Only two items are needed to start the process. These are a USL file of the program requiring resegmentation, and an adequate test case to exercise the program when segment trace is performed.

The first program (NICRSP) takes the USL file and resegments it such that no procedure in a segment calls any other procedure in that segment. This elimination of internal segment calls is necessary for segment trace since internal procedure calls cannot be traced. The resultant USL file is then prepared into a program file. The program file is run and the segment trace data gathered. Then this USL file and the trace tape are input to program two (ORTDRP). This program reduces the trace data into a usable form for the third program and puts the data in a disk file. Finally, the resegmentation program is run (ORSP) using the file built by program two and the USL file. The output is the resegmented USL file.

It is a good idea to use a copy of the USL for resegmentation. Then the original can be kept in case another resegmentation is desired with a different

segment size. The maximum segment size is kept in DB+Ø of both program one and three. This location may be modified by calling debug when the program is started. This may be done by starting the program at its secondary entry point DBG. Note that program's one and three require privileged mode and process handling capability. Another requirement is that when the program is traced, the segment numbers are assigned in a linear order. This may be verified by running the program with the LMAP option. This assignment order can be insured by starting immediately after cold load.

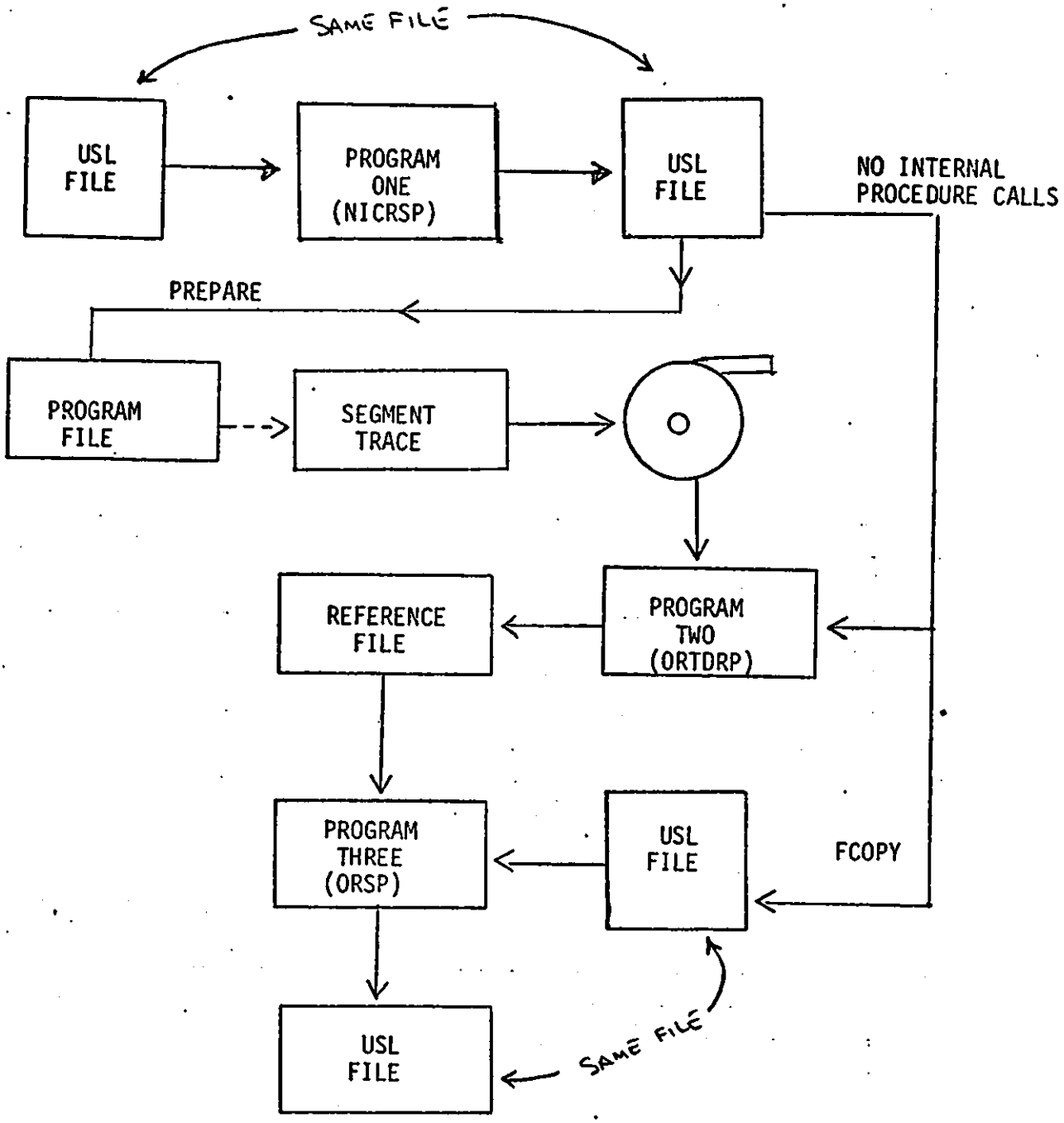


Figure 1

Resegmentation Procedure

A Case Study - COBOL

The COBOL compiler was chosen as a test case because it is one of the largest subsystems and its authors had already carefully segmented the code by hand. Thus, it would test the program's ability to resegment a large system, and the results could be directly compared to the original compiler. Four different tests were conducted. The first test consisted of resegmenting the compiler based on the results of tracing one compile. This same compile was then tested against the original compiler. The following three tests were conducted on a version of the compilation of nine different source files. Tests were made on three over the compile of nine different source files. Tests were made on three modes of operation; stand-alone, multiprogramming against itself (code sharing) and multiprogramming against another large subsystem. The results of each test are shown in tables one through four.

Each test was conducted on four different maximum segment sizes; one thousand through four thousand word segments in multiples of one thousand words.

PROGRAM	Number of Segments	Elapsed Time	Code Segment Faults
COBOL	28	147	587
4K Reseg.	20	206	1423
3K Reseg.	28	195	1376
2K Reseg.	39	162	949
1K Reseg.	56	132	451

Table 1 - One program traced & run alone

PROGRAM	Number of Segments	Elapsed Time	Code Segment Faults
COBOL	28	336	1069
4K Reseg.	23	484	2874
3K Reseg.	28	342	992
2K Reseg.	34	397	2078
1K Reseg.	52	325	859

Table 2 - Cobol run alone (4 Compiles)

PROGRAM	Elapsed Time	CPU Run Time	Code Segment Faults
COBOL	197	83	1915
4K Reseg.	178	77	1414
3K Reseg.	167	73	1450
2K Reseg.	165	71	1680
1K Reseg	153	71	1631

Table 3 - Cobol multiprogrammed against itself (one compile)

PROGRAM	Elapsed Time	CPU Run Time	Code Segment Faults
COBOL	206	85	1807
4K Reseg.	225	82	2023
3K Reseg.	189	80	1589
2K Reseg.	201	83	1857
1K Reseg.	225	94	2321

Table 4 - Cobol multiprogrammed against another large subsystem (one compile)

An examination of the tables shows that programmatic resegmentation can do about as well as a human. Different segment sizes yield much different results, so some experimentation must be done to pick a good segment size. For this experiment, the 3K segment size appears to be about the best of the four sizes used.

SEGMENT TRACE SYSTEM

Segment Trace System

The Segment Trace System (STS) uses the hardware "trace" facility to collect data pertaining to processes at the time of intersegment transfers caused by PCAL's and EXIT's. Sufficient information is available to gather statistics on both code and data of processes, or to examine an individual segment. Data are collected on unit #0 of the system magnetic tape unit (DRT 6) and reduced offline on the 3000 by a data reduction program. Because of the amount of code executed for each transfer traced and the non-overlapped tape I/O, tracing large numbers of segments can cause severe system performance degradation. The trace segment has been made an integral part of the operating system to reduce the problem of incorporating it after each MPE update and to make it available on every development system.

CONSTITUENTS OF STS

STS consists of the following three software components:

1. An MPE system containing a special version of the ININ code segment and a 300 word data segment.
2. A program to control the tracing process (TCP).
3. A data reduction program.

STS is inert when no trace bits are set and should cause no noticeable loss of system performance.

USING STS

1. COLD load a copy of MPE containing the trace segment.
2. Ready a mag tape on unit 0.
3. Run the TCP program from a session. Use the command sequence (explained in detail on page 7).

INIT	clock dirt
SET	set trace bits
RUN	tracing begins
STOP	tracing ceases
CLEAR	Clear trace bits
EOF	writes EOF ON TAPE
EXIT	terminates TCP

4. Rewind and dismount tape.
5. Process tape using reduction program.

STS - TRACE

Segment trace data are collected by a segment added to segment ININ in MPE. This segment is called from the absence trap segment (%14) and the trace trap segment (%16) when these segments have determined that tracing is to be done. The trace trap segment processes the Break and Control - Y features of a session, and these will pre-empt a segment trace. To aid understanding of the capabilities and limitations of trace, a description of the hardware trace feature follows.

A procedure call to a segment will cause a trap to the trace segment if bit 2 of the first word of that segment's CST entry is set to one and bit 0 is not set to one (absence).

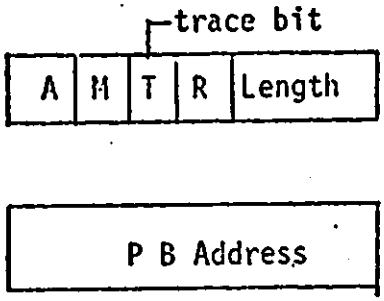


Figure 1 CST Entry

Trace and absence traps due to PCAL's are identical in their effect on the stack. The result is two stack markers and an external label.

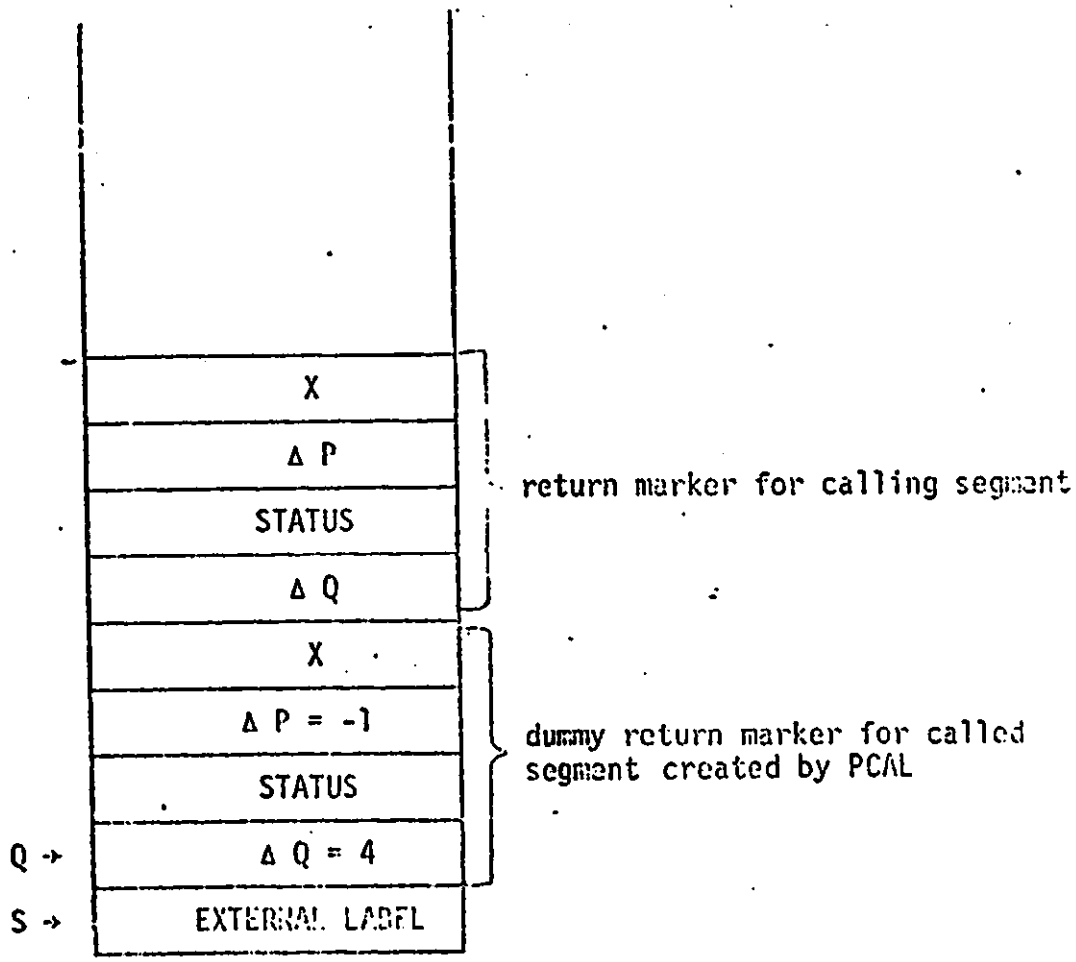


Figure 2 - Stack After PCAL Trace or Absense Trap

The external label is a copy of the one referenced by the PCAL in the calling segment. With this label, the trace routine can calculate the correct delta P in the dummy marker to enter the called segment.

An EXIT trace occurs if bit 0 of delta P in the return marker is a one and an EXIT instruction is executed using that marker. The marker is left on the stack and control is passed to the trace trap segment. The value N from the

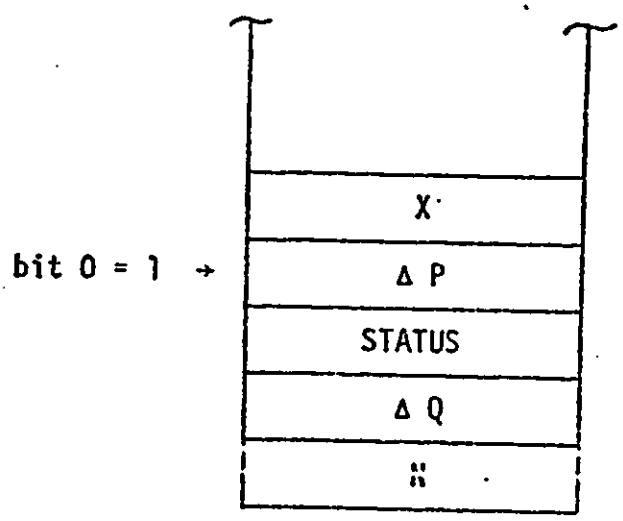
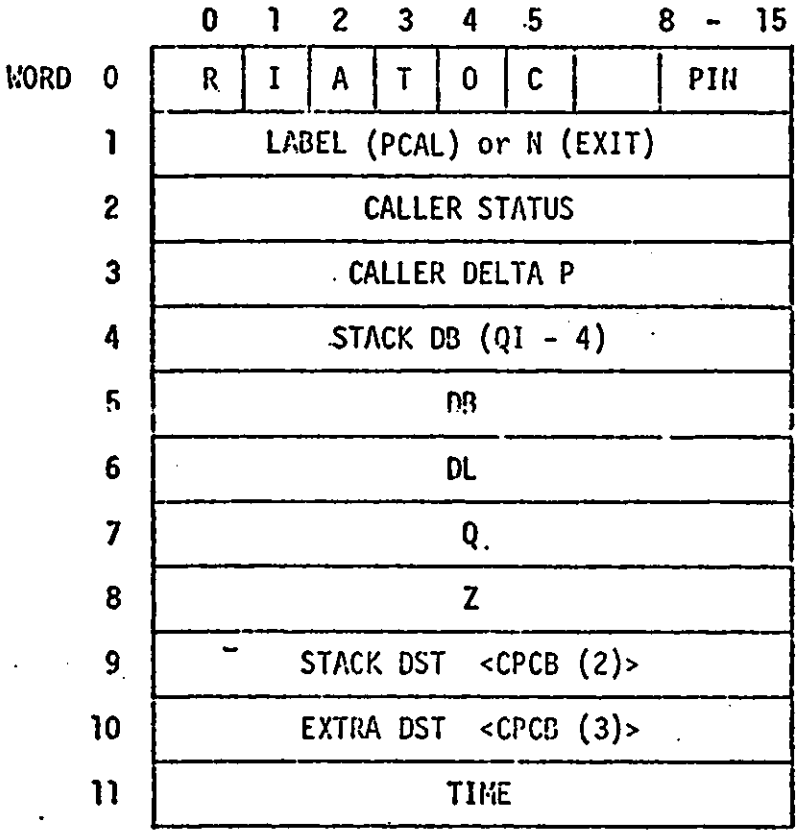


Figure 3 - Stack After EXIT Trace Trap

EXIT N of the called segment is pushed on the stack so the trace trap processor can XEQ the correct EXIT N off the stack at the completion of the trace process. The trace routine must reset delta P to the correct value before the EXIT is executed. Currently, bit 1 of delta P determines if it is a trace trap (bit 1=1) or a break trap and the Trace Segment (%16) takes the appropriate action.

The hardware trace facility has some limitations. PCAL's and EXIT's within a segment cannot be traced, nor can interrupts be traced. Another important exception is a PCAL of a segment to itself via an external label. The PCAL can be traced, but the corresponding EXIT cannot. The trace segment will supress tracing such EXIT's.

The trace data collection segment works as follows. When the trace segment is called, the trace options word in the system global area (SYSPG + 245)(figure 4) is checked. If bit 1=Ø then the trace segment will exit to the caller with condition code set to less than. This means that no tracing is being done. If bit 2 = Ø, the trace request will exit without collecting any data. This bit is used to start and stop the physical collection of data. If bit 3 = 1 and the trace was from a PCAL, delta P of the return segment will be set so the corresponding EXIT can be traced if a segment is not calling itself via an external label. If the Clock DRT is non-zero, timing measurements will be made.



where R = Record Type Ø = Trace Record
1 = INFORMATION RECORD

I = Interrupt Bit - on interrupt occurred between this record and the previous record.

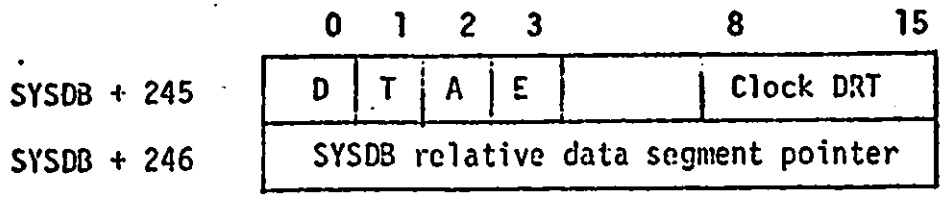
A = Absence - the call for this record came from the absence trap.

T = extra clock being used.

O = timer overflow bit (18th bit).

C = timer carry bit (17th bit).

Figure 4 - Trace Record Format



where D = dispatch bit (set to one each time the dispatcher is executed)

T = trace bit (set to one while tracing is active)

A = arm bit (if this bit is a one, trace trap will be processed, else they will be ignored)

E = exit bit (if this bit is a one, the trace processor will set the EXIT marker to invoke a trace trap each time a PCAL trace is processed)

Figure 5 - Trace Control Format in System Global Area

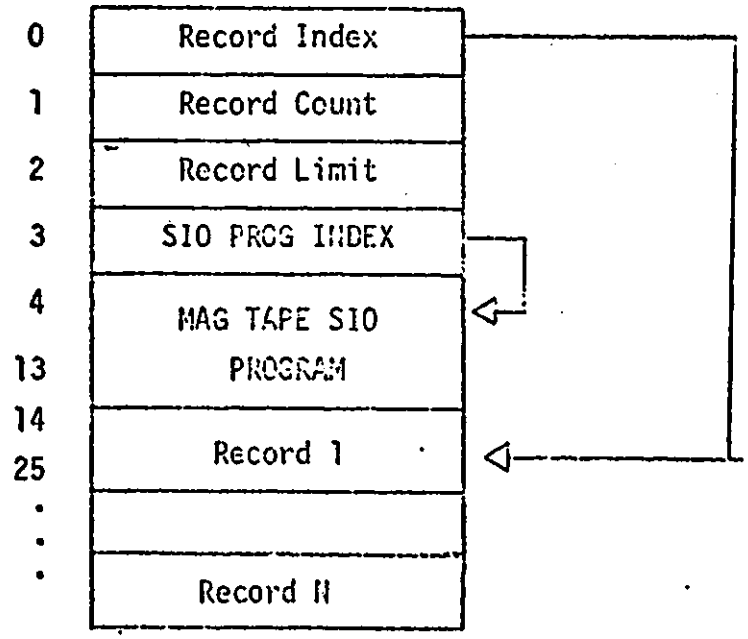


Figure 6 - Trace Data Segment Format

After the options check, the trace tape record is built. Word Ø contains 6 bits of status information and the process identification number as shown in Figure 4. Word 1 contains the label if a PCAL is traced or N if an EXIT N is traced. Words 2 and 3 contain the status and delta P as they appear in the calling segment stack marker. Word 4 contains the current process stack DB located in Q1 - 4. Words 5 through 8 contain the values of the process stack registers. Words 9 and 10 contain the stack DST and extra data segment DST located in the current process control block. If timing measurements are being made, word 11 will contain a current time stamp.

An extra clock board provides the time base for timing measurements. The clock is read at the beginning of each trace, and this value is loaded back into the clock at the completion of the trace, so that the trace overhead is removed from the measurement. The LR = CR and LR = CR overflow bits of the clock are used to extend the clock period to 18 bits instead of the 16 bits available in the clock counting register. At a ten microsecond counting interval, 1.9 seconds of time can elapse before measurement overflow occurs. This cannot happen since time of day updates occur once each second in MPE and timing analysis will not be done through interrupts.

Trace Control Program

The Trace Control Program (TCP) provides the user with a convenient means of setting and clearing trace bits, and initializing and controlling the trace operation. TCP must run in priviledged mode so that user account capability is needed.

The commands to TCP may be input from a terminal (TCP prompts with a "?") or card reader, depending on the mode (session or job). Commands are entered one per line (card). Terminal input is terminated with a carriage return. Some examples of command syntax are shown in Figure 7.

A blank or comma may be used as a data item separator, but blanks are otherwise ignored. Numbers may be either octal or decimal, with a "%" character preceding an octal number. A "/" between two numbers indicates a range.

```
INIT  0        [redacted]0
TRACE          45, %27/41  %102
RUN
STOP
CLEAR          %20/130
EXIT
```

Figure 7 - TBC Command Examples

TBC features a comprehensive set of diagnostics. All but the following two abort the current command:

BAD CST NUMBER X (X <%20 or too large)

UNASSIGNED CST X (This CST not used)

If ~~there~~ This occurs within a <range>, execution of the <range> continues.

The TBC program itself is divided into three parts, which are the scanner, the interpreter, and the main program. The scanner picks "tokens" from the current line, leaving the ASCII representation in the byte array TOKEN, a type indicator in T, and the value, if numeric, in V. The main procedure of the scanner is NEXT. A token is defined to be a comma (",") a slash("/"), a carriage return, or a string of alphamerics not including blanks, commas, slashes, or carriage returns.

The main program fetches new lines, requests the first token (by calling NEXT) and attempts to interpret it as a command. If this is successful, control passes to a procedure in the interpreter which executes the command.

One can add new commands simply by adding an equate for the "type" of the command, enlarging the case statement in the main program, and inserting an IF clause in procedure COMMAND to detect it. An actual procedure must be put in the interpreter section to perform the actual execution.

```
<command>:: = TRACE <t list> |
              CLEAR <t list> |
              SHOW <addr>    |
              STORE <addr>, <value>  |
              INIT <drt> |
              RUN  |
              STOP |
              EOF  |
              EXIT

<T list>:: = <t element>, <t list>
<t element>:: = <num>|<range>
<range>:: = <lower bound>|<upper bound>
<lower bound>:: = <num>
<upper bound>:: = <num>
       <addr>:: = <num>
      <value>:: = <num>
        <drt>:: = <num>
        <num>:: = Octal number ≤ 177777
                  (leading 0's ignored)
```

Figure 8 - TBC Command Syntax

TRACE/CLEAR <t list>

Trace bits are set/cleared in CST entries described by <t list>. If a <range> is specified, the trace bits in <lower bound> through <upper bound> inclusive are affected. <lower bound> must be ≤ <upper bound>.

INIT <drt>

All necessary initialization is performed, <drt> is the DRT index of the extra clock board. This command must be issued before any RUN's.

RUN

Tracing beings.

STOP

Tracing ceases; it can be restarted with a RUN.

EOF

An end-of-file mark is written on the trace tape.

EXIT

TBS terminates (cannot be tracing when this command is executed.

Figure 9 - TBC Command Semantics

Data Reduction Program

The data reduction program will be capable of presenting the data either by process or by code segments. The segment statistics should be useful for analyzing library routines and other segments where only information about those particular segments are of interest. Process statistics can be used to examine both the code and data behavior of software sub-systems.

Segment statistics include both measurements of time and statistics concerning information about the segments which call the traced segment. Timing data includes the total time spent in the segment, the average time and standard deviation of all the calls to that segment. The calls will be broken down by STT numbers, calling segment numbers, and process numbers.

Process statistics will examine the minimum and maximum stack size; the number of times the stack is added to; the size of the user's own area, and how many times that size is changed. Extra data segment use can also be determined. The flow of program control can be shown by showing the segments used, who called them, and how many times they were called.

May 7, 1973
STS - TRACE Data Reduction Program

Segment Trace Data Reduction Program

A program is available to process the tape produced by the STS-TRACE program. (For a description of STS-TRACE, consult prior memo dated March 20, 1973.) Output is in the form of entry, caller, and timing statistics for each segment traced.

The data reduction program file name is TRACERED and is activated using the MPE Run command. The program may be run from batch or interactive access. If interactive access is used, the program prompts with a "?". Currently the program will recognize three commands, as shown:

SEGMENT - causes the program to process a trace tape and produce segment statistics. This command may be executed once only per line.

CALL - in addition to the segment statistics above, this option causes the processing of a trace tape to include the caller statistics, and the caller statistics to be included in the output. Note that this command must be

issued each time the segment command is issued if caller statistics are desired.

EXIT - causes the reduction program to terminate after other commands included in this line have been executed.

The commands may be input in any order. Any non-alpha character may be used for delimiting commands. Any alpha input other than the above will cause an error message to be emitted, but will otherwise be ignored. After input, the commands will be executed and, if no EXIT command was included, will request the next command.

All statistical output from the program is directed to the line printer. The following is an explanation of each column of data. Please refer to Fig. 1.

The first eleven columns comprise the called segment statistics. The first column is a three digit octal segment number. The second column is the total PCAL entries to that segment.

The third column is the occurrance of PCAL absences only. EXIT absences cannot be traced, and therefore cannot be displayed. The fourth column is absences as a percentage of the total entries. Column 5 is a three digit octal segment transfer table (STT) entry number. Column 6 is the total calls to that STT. Column 7 is STT entries as a percentage of the total entries. Column 8 is the number of timed STT entries. This is the total number of STT entries on which it was possible to gather timing information. Column 9 is the number of timed entries as a percentage of the total STT entry count. Column 10 is the average time per STT call in milliseconds with all trace overhead removed. Note that this time is the time spent in this segment less the time spent in other segments which were being traced and were called by this segment. Column 11 is the standard deviation of the time per call in milliseconds.

Columns twelve through sixteen comprise the caller statistics. Columns twelve through fourteen contain the caller ID, that is the process identification number, the segment number and delta P value of the calling segment. All three are represented in octal. Column fifteen is the total calls made to this segment by the caller, and Column sixteen is the number of calls as a percentage of the STT entries.

Segment Trace In Action - A Case Study

This section describes an example of the use of the software trace system. A calibration program has been written in order to calibrate the constants in the trace data reduction program which removes the tracing overhead from segment timings. All system output is underlined.

First, the trace control program is brought up and initialized.

:RUN TRACE

HP3000 TRACE CONTROL PROGRAM

? INIT 15

Next, the calibration program is started with the LMAP option to find out which segments the program would use. The segments used were %123 through %137. The calibration program requests the number of passes to make, and at this point, the following trace commands are issued.

? TRACE %123/%137
? RUN

Now trace is ready to operate, so 100 is issued to the calibration program. Tracing proceeds until the calibration program has run to completion. Next, the following commands are given to the trace control program.

? STOP
? EOF
? EXIT

This terminates the program execution. Note that the CLEAR command was not necessary since the CST entries were deallocated when the calibration program terminated.

After the trace operation is complete, the data is reduced. Two reductions were made with and without caller statistics are shown in Figures 2 and 3.

: RUN TRACERED

HP3000 TRACE TAPE REDUCTION PROGRAM

? SEGMENT
? SEGMENT CALL EXIT

Note that the measured time for an EXIT only is .02 milliseconds are shown in segments %131, %132, and %137; whereas a PCAL, EXIT pair is .05 milliseconds as shown in segments %124, %125, %133, and %134. These are approximate since an exit takes 21.7 microseconds, and both a PCAL and EXIT take 50.4 microseconds. They are not in exact agreement since the trace system only receives one clock count each 10 microseconds. Therefore the error could be as large as 10 microseconds for each segment traced.

SEG.	ENTRY CNT.	ABSENCES	%ABS.	STT	ENTRY CNT.	%SEGE	TIMED ENT.	%STTE	TIME/C.	SDEVT/C.	PIN SEG DELTP	CALLS	%STTE
124	99	1	1.0	001	99	100.0	81	81.8	.05	.00	027 123 00041	99	100.0
125	99	1	1.0	001	99	100.0	85	85.9	.05	.00	027 124 00001	99	100.0
126	99	1	1.0	001	99	100.0	92	92.9	.02	.00	027 125 00001	99	100.0
127	99	1	1.0	001	99	100.0	95	96.0	.06	.00	027 126 00001	99	100.0
130	1	1	100.0	001	1	100.0	0	.0	.00	.00	027 123 00020	1	100.0
131	1	1	100.0	001	1	100.0	1	100.0	.02	.00	027 130 00001	1	100.0
132	1	1	100.0	001	1	100.0	1	100.0	.02	.00	027 123 00017	1	100.0
133	100	1	1.0	001	100	100.0	59	59.0	.05	.00	027 123 00030	100	100.0
134	100	1	1.0	001	100	100.0	61	61.0	.05	.00	027 133 00001	100	100.0
135	100	1	1.0	001	100	100.0	61	61.0	.08	.00	027 134 00001	100	100.0
136	200	1	.5	001	200	100.0	163	81.5	.08	.00	027 135 00001	100	50.0
											027 135 00002	100	50.0
137	400	1	.3	001	400	100.0	389	97.2	.02	.00	027 136 00001	200	50.0
											027 136 00002	200	50.0

Figure 1

PAGE 1 HP3000 SEGMENT TRACE DATA REDUCTION PROGRAM

SEG.	ENTRY CNT.	ABSENCES	%ABS.	STT	ENTRY CNT.	%SEGE	TIMED ENT.	%STTE	TIME/C	SDEVT/C	PIN	SEG DELTP	CALLS	%STTE
124	99	1	1.0	001	99	100.0	81	81.8	.05	.00				
125	99	1	1.0	001	99	100.0	85	85.9	.05	.00				
126	99	1	1.0	001	99	100.0	92	92.9	.02	.00				
127	99	1	1.0	001	99	100.0	95	96.0	.06	.00				
130	1	1	100.0	001	1	100.0	0	.0	.00	.00				
131	1	1	100.0	001	1	100.0	1	100.0	.02	.00				
132	1	1	100.0	001	1	100.0	1	100.0	.02	.00				
133	100	1	1.0	001	100	100.0	59	59.0	.05	.00				
134	100	1	1.0	001	100	100.0	61	61.0	.05	.00				
135	100	1	1.0	001	100	100.0	61	61.0	.08	.00				
136	200	1	.5	001	200	100.0	163	81.5	.08	.00				
137	400	1	.3	001	400	100.0	389	97.2	.02	.00				

Figure 2.

PAGE 1 HP3000 SEGMENT TRACE DATA REDUCTION PROGRAM

SEG	ENTRY CNT.	ABSENCES	%ABS.	STT	ENTRY CNT.	%SEGE	TIMED ENT.	%STTE	TIME/C	SDEVT/C	PIN SEG DELTP	CALLS	%STTE
124	99	1	1.0	001	99	100.0	81	81.8	.05	.00	027 123 00041	99	100.0
125	99	1	1.0	001	99	100.0	85	85.9	.05	.00	027 124 00001	99	100.0
126	99	1	1.0	001	99	100.0	92	92.9	.02	.00	027 125 00001	99	100.0
127	99	1	1.0	001	99	100.0	95	96.0	.06	.00	027 126 00001	99	100.0
130	1	1	100.0	001	1	100.0	0	.0	.00	.00	027 123 00020	1	100.0
131	1	1	100.0	001	1	100.0	1	100.0	.02	.00	027 130 00001	1	100.0
132	1	1	100.0	001	1	100.0	1	100.0	.02	.00	027 123 00017	1	100.0
133	100	1	1.0	001	100	100.0	59	59.0	.05	.00	027 123 00030	100	100.0
134	100	1	1.0	001	100	100.0	61	61.0	.05	.00	027 133 00001	100	100.0
135	100	1	1.0	001	100	100.0	61	61.0	.08	.00	027 134 00001	100	100.0
136	200	1	.5	001	200	100.0	163	81.5	.08	.00	027 135 00001	100	50.0
											027 135 00002	100	50.0
137	400	1	.3	001	400	100.0	389	97.2	.02	.00	027 136 00001	200	50.0
											027 136 00002	200	50.0

Figure 3

DATE May 15, 1973

SUBJECT Changes to the Segment Trace System

cc: Measurement Distribution

The following two changes have been made to the STS trace segment. The first involves the trace record format, and the record involves the operation of the trace code segment.

The following changes have been made to the output record format of the trace segment. Refer to Figure 1. The positions of DB and DL within the record have been reversed. Bite six and seven of the first word are now being used as indicators. Bite six indicates an end-of-data condition, that is, that this record and all subsequent records are invalid. Bit seven indicates that this record was not generated as the result of a trace or absence interrupt, but was generated as the result of a direct procedure all to the EXTRACE segment.

Word	Use								
	0	1	2	3	4	5	6	7	8 ... 15
	R	I	A	T	O	C	L	E	
	LABEL (PCAL) or N (EXIT)								
	CALLER STATUS								
	CALLER DELTAP								
	STACK DB <QI-4>								
	DL								
	DB								
	Q								
	Z								
	STACK DST <CPCB (2)>								
	EXTRA DST <CPCB (3)>								
	TIME								

where R = Record Type
I = Interrupt
A = Absence
T = Segment timing
O = Timer Overflow
C = Timer Carry
L = Last Record
E = External Call

The time segment now has the capability of recovering from a tape write error. If the error occurs, the program will stop with a HALT 1. The tape controller status will be displayed in RA so that the source of trouble can be determined. Press run, and the program will issue a backspace and gap to the tape drive. If this is successful, the operation will continue in a normal manner. If unsuccessful, the program will stop with a HALT 2. Pressing run will allow tracing to continue, but the tape will contain an error which will probably inhibit the data reduction program from reading the tape past that point. This means that it is best to terminate the tracing operation and restart with a different tape mounted.