SLAC-PUB-715
March 1970
(MISC)

GENERALIZED IBM SYSTEM 360 SOFTWARE MEASUREMENT*

R. H. Johnson and T. Y. Johnston
Stanford Linear Accelerator Center
Stanford University, Stanford, California 94305

(Submitted to Datamation.)

*Work supported by the U. S. Atomic Energy Commission.

The areas of computer hardware and software measurement are currently receiving a great deal of attention. This is because computer hardware and software have become increasingly complex, and some performance evaluation techniques are necessary to improve or even measure either hardware or software efficiency. This paper is an attempt to show how a generalized software monitor might be utilized to do this.

For some time now the staff of Computation Center at the Stanford Linear Accelerator Center (SLAC) has seen the need for a way of measuring the performance of programs which run on our computer. We are presently using an IBM System 360 Model 91, primarily for the analysis of particle event data produced by the operations of the Linear Accelerator in high energy physics research. Our computing load is increasing rapidly.

Our computing load can be divided into two dominant categories.

1. Production runs of relatively few programs, of complex structure, which use large amounts of CPU time.
2. Many short runs using language translators, data set utilities, linkage editors, and other externally supplied software.

Even slight improvements in the efficiency of either our production programs or externally supplied software can produce great savings in computer time for us.

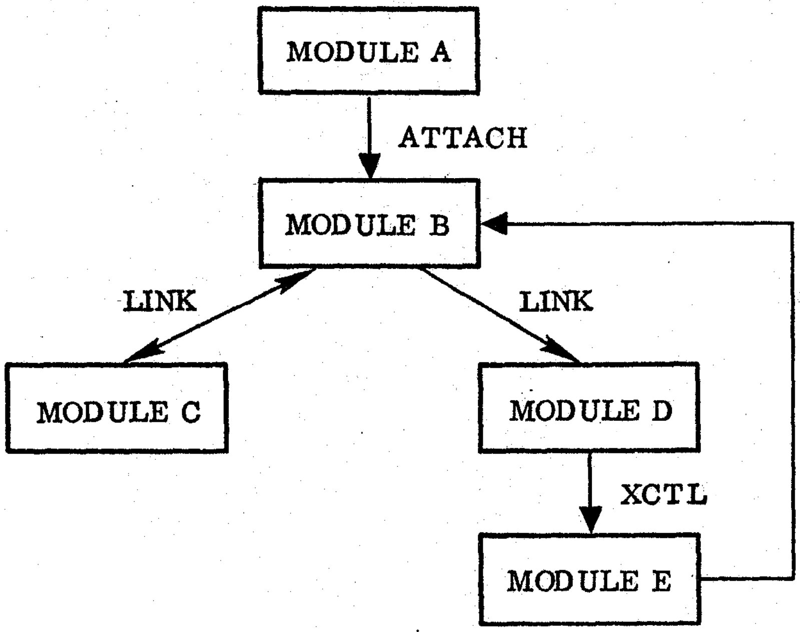
Early this year we set about to create a software measuring program. Commercial package programs were already available to perform this function, but their use was ruled out. We run the MVT option of the 360 Operating System on our computer (multiprogramming with variable number of tasks). This enabled us to design a very simple, but effective, monitoring system of our own. The

MVT option allows us to attach virtually any program structure as a subtask under our monitor.

Program structure is determined by how programs link to one another. Programs running under OS MVT can link to other programs in at least five ways.

1. LINK — Control is passed to another program module (which may have to be fetched from external storage). The parent module receives control back after the LINKed to module has finished executing.
2. ATTACH — The parent module creates a subtask. Both modules can vie for access to the CPU. The subtask merely informs the parent task when it has finished, but otherwise they operate independently of each other. For the previous case (LINK), the parent module remains inactive until control is returned to it.
3. XCTL — Similar to LINK, except that when the called module finishes executing it returns to the module which invoked the parent module and not the parent module.
4. To conserve core storage, some programs are constructed so that various of their subprograms can overlay each other in core storage as control passes from one subprogram to another. When this occurs, a part of the Operating System called the overlay supervisor is invoked which performs the function of fetching the desired subprogram, loading it into the proper part of core storage and then passing control to the loaded subprogram.
5. LOAD — A module may be brought into core storage from external storage and have control passed to it by a branch instruction later on. This differs from the use of LINK, since when a module is LINKed to, it receives control immediately.

A possible program structure might be as follows:



Our monitor program is capable of recognizing passage of control by any of the above five methods of linkage. The monitor keeps records, while the measured subtask runs, of which modules are in control and what their status is.

The subtask is relatively uneffected by the monitor, which is the main task. Periodically, at fixed time intervals, the monitor interrupts execution of the subtask and records certain statistics about the state of the subtask when it was interrupted. These include:

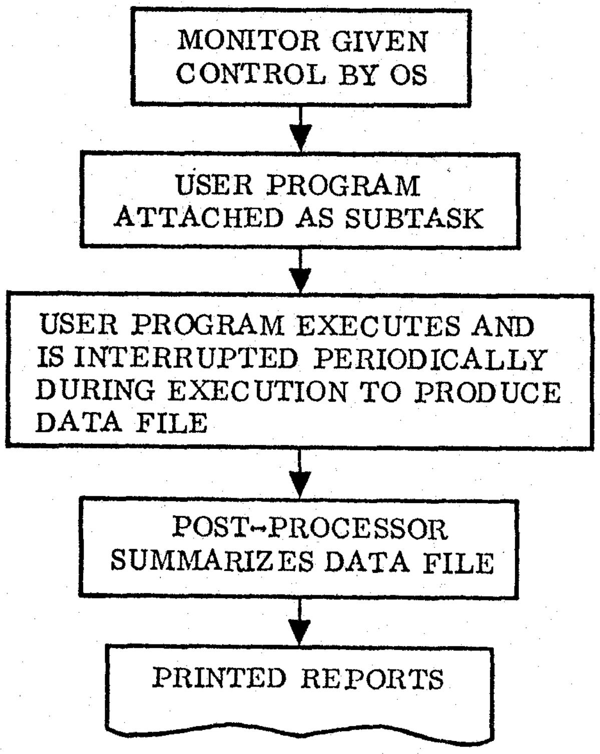
1. Identifier for task(s) currently active
2. Name of the module active in the task
3. Number of the highest overlay segment in core for the active module
4. Load address of the active module
5. Code telling whether the module was running or waiting
6. Core address when module was sampled.

After the program structure to be measured has finished execution, a post-processor is evoked which summarizes these data produced by the monitor. Outputs produced by the post-processor include:

1. A timeline showing flow of control from module to module as the subtask executed.
2. A summary of modules encountered and totals of the number of times each was found waiting or running. A typical example is seen in Table 1.
3. A wait and run histogram which graphs frequency of PSW address at interrupt versus core location range. We generally set this range (or "bucket" size) at 100_{16}. An example of a run histogram is given in Fig. 1.

Notice that bucket addresses are given both in absolute and relative to the beginning of the module load address. This greatly facilitates reference back to linkage editor maps and assembly listings.

The overall flow of control of the monitor is as follows:


MONITOR GIVEN CONTROL BY OS
USER PROGRAM ATTACHED AS SUBTASK
USER PROGRAM EXECUTES AND IS INTERRUPTED PERIODICALLY DURING EXECUTION TO PRODUCE DATA FILE
POST-PROCESSOR SUMMARIZES DATA FILE
PRINTED REPORTS

By scrutinizing this final output one can determine the percent of total time each module was active and the percent of time it was running or waiting. The histograms tell roughly what proportion of the time a module executed its own code, and the proportion of time the module spent in executing code in Operating System I/O Access Routines. Frequently executed loops show up as peaks on the histogram.

By referring to the histograms, and to assembly listings of the code being measured, it is possible to correlate peaks on the histogram with actual identifiable sections of code. In some cases, by recoding certain sections of a program, decreases in running time can be achieved.

We have already received dramatic savings in CPU time by monitoring two of our most heavily used production programs. In both cases we examined the run histograms and noticed some very large peaks which accounted for a large proportion of the total CPU time used by the programs. In one case we recoded a critical routine from FORTRAN into Assembler language and reduced total CPU time per run by 35 - 40%. The other program was one that accounts for one third of all our CPU time. In it we found an unnecessary error checking procedure. This was removed and cut total CPU time per run by 15%.

As indicated previously, the idea of monitoring the execution of a program is not new, but our monitor has one characteristic which is unique. That is, it can be used to measure the performance of system routines as well as user routines. The subtask running under our monitor can be a compiler, an assembler, a data set utility, or a linkage editor, etc., as well as a user program. Most existing software measuring programs must be somehow incorporated into the program to be measured (usually by a subroutine call or link editing) through some special processing. The combined programs are then run to make the measurements. To do this with something like the IBM Fortran H compiler would be a prodigious task. Our monitor requires no modification in the program to be measured, which means we can very easily measure the executing characteristics of a compiler.

We have already made some measurements on a few of the IBM supplied processors and would like to briefly describe them here.

Fortran H

At our installation this is the most frequently used language processor. We keep statistics on number of calls to each program on our system and Fortran H amounts to almost half of our total calls. We therefore chose this processor as the first one to investigate. The test compilation was approximately 800 cards in length and consisted of thirteen subroutines. The compiler itself is an overlay module with 13 overlay segments. The segments are called into core in a fixed order as each subroutine is compiled. The statistics we gathered are given in Table 1.

Part of the OS MVT system is a program called FETCH. FETCH is responsible for locating program modules on external storage (drum or disk) and bringing them into core memory so that they may be executed.

That part of our post-processor output which summarizes total wait and run time showed that of the 10.2 sec of CPU time used by the compiler, 30 percent of that time was spent in program FETCH bringing in overlay segments, and that 96 percent of all the wait time was also in FETCH. This would suggest that it would be futile to attempt to streamline the code of the compiler in order to decrease running time, since program FETCH is the real bottleneck! This also shows that the time it takes to compile a given program will depend heavily on how many subroutines the program has, since each additional subroutine implies a fixed number of calls to program FETCH.

PL/I F

So far we have made only a few simple tests of this compiler. We have discovered that it spends 20% of all its CPU time in program FETCH. We are in the process of making further measurements.

Assembler F versus Assembler G

In the future world of "unbundled" software the buyer of software is apt to be confronted by multiple packages which have similar external characteristics (i.e., input and output formats),but which operate quite differently internally. It is our contention that a monitor such as ours can be used to compare competing software packages. As an example of what might be done we have made a comparison of Assemblers F and G.

Assembler G is a modification of IBM's F assembler. Assembler G was produced at the University of Waterloo in Waterloo, Ontario. The main modification effected was to improve buffering of the assembler's input and output. To illustrate the effect of this change see Table 2. Our test case was an assembler language program some 6000 cards in length.

Detailed examination of the histograms showed that ASMF modules spent considerable more time in OS Access Method code (doing Input/Output) than did ASMG. This is reflected in the fact that our job accounting data shows that ASMF, has, on the average, a higher elapsed time/CPU ratio than ASMG.

While the results of this test might be of some interest, a second test we made was considerably more enlightening. It is known by those who use the assemblers that they use up vast amounts of computer time when called upon to do expansion of Macros. A Macro is a code skeleton, usually of many instructions, with dummy operands. The user writes the Macro with its actual operands just as he writes machine instructions with their operands. The assembler then "expands" the Macro by filling in the dummy operands with actual operands and generating the actual machine instructions. In a sense a Macro enables the assembly language programmer to use abbreviations for frequently written sequences of code. We decided to try and find out how ASMG and ASMF compared at Macro expansion and also to locate that section of code which uses all the CPU cycles. As a test example we used an 800 card input to the assemblers that consisted almost entirely of deeply nested Macro expansions (i.e., Macros within Macros). The results for both assemblers are summarized in Tables 3 and 4.

Even the relatively unsophisticated person can tell by glancing at the tables that the module ASMGF3 (or IEUF3) is the one using all the CPU time. It can also be seen that ASMG has at least eliminated the necessity for large amounts of input/output to do Macro expansion. For those interested, Fig. 1 is a histogram which shows how CPU cycles are distributed relative to the beginning of the ASMGF3 module. Perhaps some systems programmer might devise a way to modify the frequently executed code to increase running efficiency! This example points out, by the way, that a monitor such as ours can make it possible for a person who is not familiar with all the code in a large system to improve its running efficiency. He can do this by looking for the "critical bottleneck" (if there is one) and then optimizing its code.

Improvements

We hope soon, as part of our analysis program output, to produce a matrix showing total number of calls from module to module. This will be of help especially in programs with complex overlay structures. From the matrix a user would have a much better idea of where to place modules in his overlay segments so as to minimize the number of calls to the overlay supervisor.

Conclusion

We think we have demonstrated that it is possible to measure the external running characteristics of user and systems software and thereby gain some understanding of their faults and advantages without having to delve into the actual code of the program involved. We feel also that our monitor can be used to compare competing software packages as well as to improve existing ones.

We believe that it may even be possible, by using the monitor, to relate program structure to demands made on the system. We would like to measure programs compiled in different higher level languages that perform exactly the same function. Just as ASMG and ASMF have the same inputs and outputs, so might a user-written FORTRAN, PL/1 or ALGOL program; but yet because of techniques used by the compilers, they may make vastly different demands on system resources.

TABLE 1

FORTRAN H COMPILER IEKAAOO

Number of Overlay Segment	Number of times found waiting	Number of times found running	Percent of time in control
1	17	1	1.5
2	177	69	20.5
5	158	58	18.0
6	52	30	6.8
7	22	4	2.2
8	35	29	5.3
9	82	68	12.5
10	109	106	17.9
13	139	42	15.1

TABLE 2

	Number of times found waiting	Number of times found running	Total CPU time (seconds)
ASMF	4409	774	18.2
ASMG	1561	664	18.0

TABLE 3

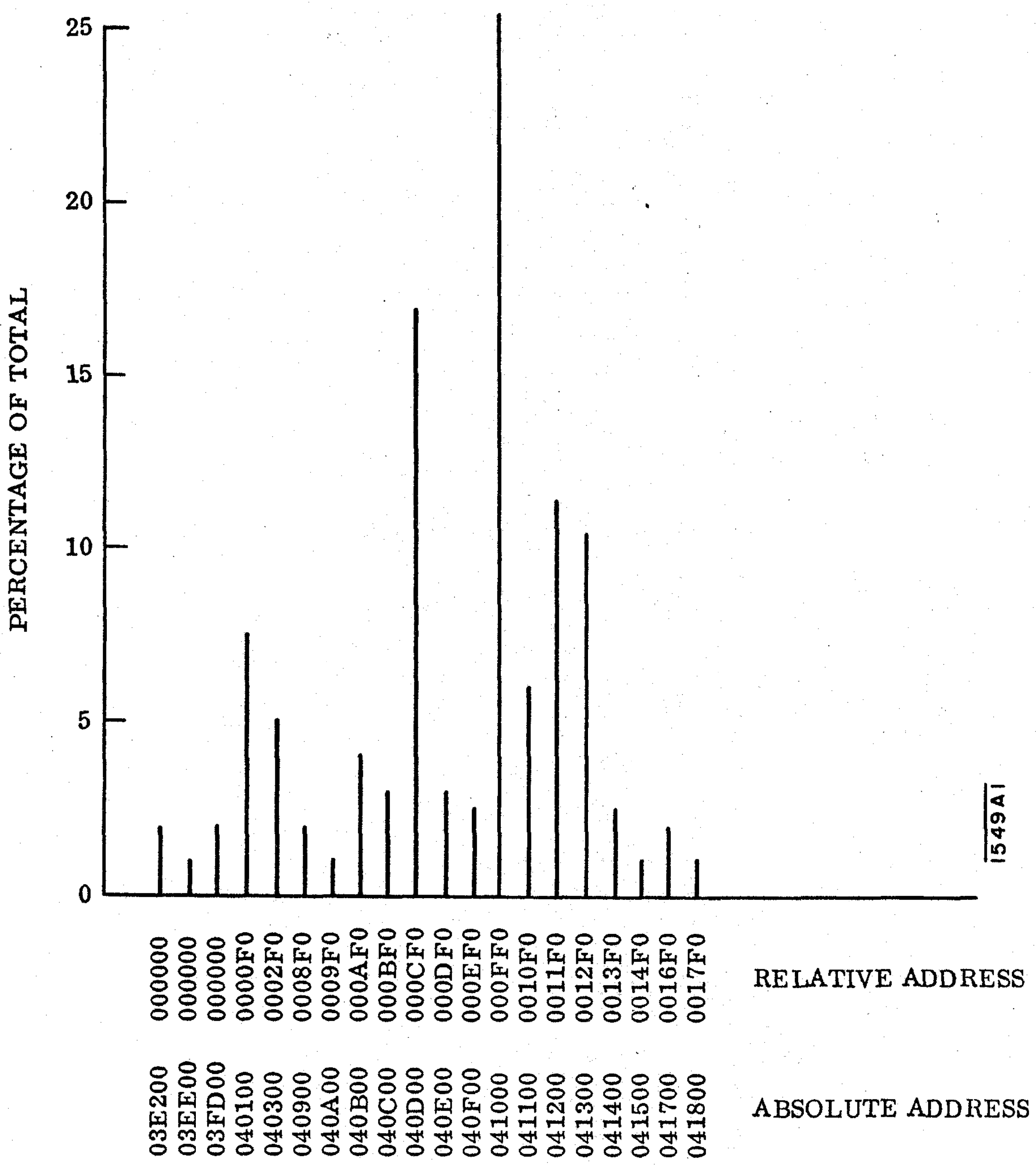
ASSEMBLER G

Module Name	Number of times found waiting	Number of times found running	Percent of time in control
ASMG	25	4	1.6
ASMGF2	47	30	4.2
ASMGIS01	2	0	0.1
ASMGF3	9	1614	89.1
ASMGRTA	2	1	0.2
ASMGF7	17	16	1.8
ASMGF8	30	17	2.6
ASMGFPP	7	1	0.4

TABLE 4

ASSEMBLER F

Module Name	Number of times found waiting	Number of times found running	Percent of time in control
ASMF	3	0	0.0
IEUMAC	16	0	0.1
IEUF1	29	5	0.2
IEUF2	209	86	1.4
IEUF3	17308	3115	96.5
IEURTA	15	0	0.1
IEUF7	91	24	0.5
IEUF1	17	2	0.1
IEUF8	148	15	0.8
IEUFPP	73	5	0.4



PERCENTAGE OF TOTAL
25
20
15
10
5
0
1549A1
RELATIVE ADDRESS
000000
000000
000000
0000F0
0002F0
0008F0
0009F0
000AF0
000BF0
000CF0
000DF0
000EF0
000FF0
0010F0
0011F0
0012F0
0013F0
0014F0
0016F0
0017F0
ABSOLUTE ADDRESS
03E200
03EE00
03FD00
040100
040300
040900
040A00
040B00
040C00
040D00
040E00
040F00
041000
041100
041200
041300
041400
041500
041700
041800

Fig. 1