Experience gained in the development and use of TSS

by RICHARD E. SCHWEMM
IBM Corporation
White Plains, New York

INTRODUCTION
Six and a half years have elapsed since W. T. Comfort described TSS/360 to the 1965 Fall Joint Computer Conference. Since that time, much has been learned by IBM and its customers about time-sharing, about TSS, and about large-scale, interactive systems in general. Scores of people have worked with the system; dozens of articles have been published; it would clearly be impossible to put in one paper a comprehensive answer to the question—what has been learned developing TSS? Yet, with the availability of Release 8.1, the major development work on the system has been completed, and this is an appropriate time to take stock of where we have been, where we are, and where we might go from here.

One summary paragraph of that 1965 paper says the following: “This report attempts to give an overall picture of the System/360 Model 67 Time-Sharing System, its system design, and major hardware and control program characteristics. The unique combination of hardware and software objectives makes a very complex problem, for which a simple and efficient solution is desired—a difficult task at best.” Indeed, bringing TSS/360 to the marketplace and making it productive in customer installations has been a very difficult task, both technically and financially.

No attempt has been made in this paper to give detailed technical descriptions of programming systems problems and solutions. Instead problems have been described in a general way, their impact on the system discussed, and the strategy for solution outlined along with the results obtained. Four major areas will be discussed:

- Lessons Learned about System Structure
- Lessons Learned about System Performance Analysis
- Lessons Learned about Software Development Tools
- Lessons Learned about Management of Software Development

SYSTEM STRUCTURE
Version 1 of TSS/360 was released in October, 1967. Use for experimental, developmental, and instructional purposes was advised. By the fall of 1968, A. S. Lett and W. L. Konigsford were able to report on a version which was more stable, performed better, and contained more function. Based on this progress made during the first year in the field, plus progress anticipated from items designed and then in development, a very positive view of the system was presented.

Several important lessons had emerged from the TSS experience by this time. It seems clear that the proper way to construct a system of this type is to build a small, hard-core base first, and add function later. As Lett and Konigsford put it—“The initial emphasis was on building a stable system, followed by extensive measurement—and—analysis efforts to identify potential system modifications.”

The same paper reported that the design of the TSS Resident Supervisor had proved sound and remained essentially as described in 1965, a statement as valid today as it was in 1968. On the other hand, major portions of the original design had to be abandoned for the simple reason that they were unacceptable to the Users. One example of this was the replacement of the original rigid set of commands with Command System II. Another was the replacement of the system’s hard-coded scheduling algorithm with an extremely flexible Table Driven Scheduler. The important thing here is that the original design was unacceptable, but that a communication channel was open from the Users to TSS Development, and that the latter was responsive to the former’s requirements.*

* In both cases cited as examples, Users made strong technical presentations. T. A. Dolotta of Princeton University and A. Irvine of System Development Corporation made a “Proposal for Time-Sharing Command Structure” at SHARE in August, 1966. The basic thinking behind table driven scheduling came from F. G. Livermore of General Motors Research.
The table below displays the functional buildup of TSS/360 by release:

<table>
<thead>
<tr>
<th>Date</th>
<th>Release</th>
<th>Major Functions Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/1/68</td>
<td>1.1</td>
<td>(Incremental reliability improvements only)</td>
</tr>
<tr>
<td>4/15/68</td>
<td>1.2</td>
<td>(Incremental performance improvements only)</td>
</tr>
<tr>
<td>7/1/68</td>
<td>2.0</td>
<td>• Support for Extended Address Translation (32 bit)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On Model 67 equipped with the special feature, users can address up to 4096 virtual memory segments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Multiple Sequential Access Method (MSAM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is a special access method for card equipment and printers which allows a single task (BULKIO) to process all spooling operations.</td>
</tr>
<tr>
<td>10/21/68</td>
<td>3.0</td>
<td>• Command System II</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This system provides a consistent command syntax, an extensive default and profile facility, a general editing capability, and allows users to define their own commands and call programs directly.</td>
</tr>
<tr>
<td>1/20/69</td>
<td>4.0</td>
<td>• Table Driven Scheduler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This technique provides installation control and dynamic regulation of scheduling parameters, for complete flexibility in obtaining optimum scheduling balance within and among tasks.</td>
</tr>
<tr>
<td>7/8/69</td>
<td>5.0</td>
<td>• Support For Duplex Configurations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This was the first system release which was tested on and suitable for a multiprocessing configuration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Virtual Access Method II</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This rewrite of VAM provides more efficient handling of physical devices, ability to create duplicate data sets, and an overall improvement to data set integrity.</td>
</tr>
<tr>
<td>10/15/69</td>
<td>5.1</td>
<td>(Incremental human factors improvements only)</td>
</tr>
<tr>
<td>3/31/70</td>
<td>6.0</td>
<td>• Resident Terminal Access Method (RTAM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removing TAM from shared virtual memory reduces paging which significantly improves conversational task performance.</td>
</tr>
<tr>
<td>12/17/70</td>
<td>8.0</td>
<td>• Multiple Terminals per Task (MTT)</td>
</tr>
<tr>
<td>10/1/71</td>
<td>8.1</td>
<td>• PL/I Compiler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A version of the OS/360 F-level compiler is modified to operate within TSS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Remote Job Entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-conversational job streams can be introduced into TSS from a remotely located work station; spooling to/from the work station is also provided.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Dynamic Catalog</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The user portion of the master catalog is made a working catalog for the duration of the user's task, eliminating contention for the master catalog.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Page Table Paging</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Demand paging of page tables allows effective use of very large virtual memories.</td>
</tr>
</tbody>
</table>

This functional buildup was accompanied by a steady improvement in performance and reliability. The performance picture is discussed in detail in a subsequent section. To date, a total of 2442 program errors detected by Users have been corrected.

Figure 1 displays the system size as a function of release. It is worthy of note that only the addition of the PL/I Compiler caused a significant increase in system size.

**SYSTEM PERFORMANCE ANALYSIS**

The economic viability of any system is determined by a complex equation which combines price, performance, and the value of function. Since a given set of hardware has a fixed price, and a given set of software functions appear to a User to have a constant value, performance is the critical factor in determining success or failure. A simple example will illustrate this:

*Let us assume that an interactive system operates on a configuration that costs $1,000,000 per year. Let us further assume that users feel that each professional employee using this system will double his productive output and that an average employee of this type costs $30,000. Then the value of the system is a simple function of how many users it can support—if it supports 33 or fewer, it will be a loser, 34 or more a winner.*
During the development of TSS/360, a comprehensive scheme for treatment of system performance evolved. The elements of this scheme are:

- Establishment of Performance Objectives
- Creation of External Performance Measurement Tools
- Creation of Internal Recording Tools and Appropriate Data Reduction Facilities

**Performance objectives**

How well will the system perform the functions it provides? This simple question receives a complex answer. TSS/360 is designed for conversational use, batch use, and a mixture of the two. A performance objective was established for each type of use. The conversational objective is the most difficult to describe and of greatest interest here, so our discussion will be restricted to it.

Basically, conversational performance is defined as the maximum number of tasks which the system will support with acceptable response time. This very general definition is made more specific by creating a benchmark terminal session and dividing the interactions created by the session into three classes—trivial response, non-trivial response, and data-dependent response. Acceptable response to trivial commands (such as text entry) is defined as four seconds. Acceptable response to non-trivial commands (such as data set creation) is defined as 7.5 seconds. Data-dependent commands (such as compilation) have no specific acceptability criteria. With this definition of conversational performance, we have a constant, calibrated yardstick that answers the performance question.

Here we should point out that no User of TSS accepts our benchmark terminal session as typical of his conversational work load. Most have specified their own benchmarks. However, there is general agreement that the definition of performance is adequate. We have a yardstick; the users have metersticks; we are in fact measuring the same thing.

The initial conversational performance objective established for TSS/360 was to support 40 tasks running the benchmark terminal session on a configuration composed of 1 processing unit, 512K bytes of memory, 1 paging drum, and disk modules on 1 channel. Subsequently, objectives for larger configurations were established.

**External performance measurement tools**

TSS operates in an extremely dynamic environment, and the load imposed upon it by live users is characterized by peaks and valleys of activity and demands for services. Yet, for performance objectives to be useful, they must be measurable, and the measurements must be repeatable. To achieve this, a measurement driver was created to simulate the actual environment under controlled and reproducible conditions.

![Figure 1—Size of TSS/360 as a function of release](from the collection of the Computer History Museum (www.computerhistory.org))
Conversational benchmarks are the & Measurement Driver

- It is capable of recovery from at least the "ordinary" kind of unexpected response, such as might result from transmission errors.
- It accepts as a parameter, a "human keying rate" for each "terminal" to adjust for the difference between machine-driven transmission rates and real keying rates.
- It records and time-stamps all transmissions in both directions on all lines, together with the control parameters and other such data, to allow data reduction and analysis at a later time.
- It is capable of terminating a run, based upon cut-off criteria specified for that run, to avoid wasteful use of machine time.
- It is data directed in its operation, so that not only the transaction to be transmitted but also the control of the individual delays between each interaction can be specified in the conversational benchmarks, in accordance with the measurement rules.

Conversational performance is measured by executing a series of driver runs varying the number of tasks for each run. A curve is then constructed which represents response time as a function of the number of tasks. Such a curve for TSS Release 6.0 is shown in Figure 3.

The measurement driver developed by IBM runs on a System/360 Model 40. Several Users of TSS have used this system to evaluate system performance on their own benchmarks. A second measurement driver has been developed by Carnegie-Mellon University. This driver has the advantage of running on the System/360 Model 67 with the version of TSS under study. Although compensation must be made for the system resources devoted to the driver function, the Carnegie-Mellon Simulator (SLIN) is compatible in script and timing characteristics with IBM's, and the output produced is comparable.

**Internal recording tools**

Answering the question "how well is the system performing?" from an external viewpoint provides little insight into the question "how can performance be improved?" For this task, internal recording tools must be used to obtain knowledge of the internal operation of the programming system and of how that system utilizes its hardware facilities. The basic measurement and recording tools used by TSS Development are Systems Performance Activity Recorder (SPAR), Systems Internal Performance Evaluation (SIPE), and Instruction Trace Monitor (ITM).

The System Performance Activity Recorder (SPAR) is a one-of-a-kind hardware recording system. It is hardwired to the system being monitored and does not cause any degradation in performance. SPAR can

* Similar capability is provided by IBM's System Measurement Instrument Unit (SMI).
provide accurate measurements of system facility utilization. The facilities monitored for TSS/360 include the CPU, processor storage, I/O channels, and direct access devices. SPAR is used to provide the following types of information:

- Time utilization of the system hardware facilities
- Counts and time relationship of identifiable events such as:
  - Time-slice ends
  - Pages transferred between devices
  - Entries to a module

System Internal Performance Evaluation (SIPE) is a software recording system that produces a degradation of approximately five percent in system performance. Hooks in the resident supervisor cause information to be collected and written on tape. The internal actions of the supervisor are then available for later data reduction. SIPE is used to provide the following types of information:

- Time utilization of system hardware facilities
- Space utilization of processor storage, drum, disk storage
- Counts of system events
- Time relationship of important internal events in the supervisor

The Instruction Trace Monitor (ITM) is a hardware and software combination that causes every instruction executed to be written in sequence on tape. The instruction sequence in resident supervisor or virtual memory is thus available for later data reduction. It should be noted that ITM causes a relatively large degradation in system performance by increasing system running time. ITM is used to provide the following types of information:

- Time sequence of virtual memory modules executed and referenced
- Time spent in each module
- SVC’s issued from each module

With these tools, and with an appropriate set of reduction programs, virtually any question dealing with the internal system performance can be answered. Access to information of this type, allows an intelligent, aggressive program of performance improvement to be carried out. The results of such a program are shown in Figure 4.

One final point should be made about internal recording tools—they have been as valuable to Users as they were to the development organization. As mentioned earlier, Users have their unique workloads, and TSS’s completely flexible scheduling technique is most effectively used with detailed knowledge of how

Figure 3—TSS/360 release 6.0 response to trivial commands as a function of number of tasks
the system is operating on that individual workload.

SIPE is in use at most TSS installations, and several Users have developed their own measurement tools. Of particular interest in this category are DEMON—a real time software measurement monitor developed at Bell Telephone Laboratories, Naperville, Illinois and XDEMON—an expanded version developed jointly by Carnegie-Mellon University and IBM's Watson Research Center.

XDEMON is a time-driven measurement tool providing statistics on the usage of resources in TSS/360. XDEMON works as a shared facility under the system maintaining a set of public tables, whose information is updated automatically and made available to the users, who can display selected information concerning TSS usage accumulated since STARTUP time and for the period since the last update. There is a privileged user, called the 'MONITOR' who has the capability of issuing some selected commands, including the updating of the shared tables, the recording of statistics for later reduction, and the monitoring of tasks that are imposing an excessive load upon the system, with the possibility of changing priorities and scheduling parameters for such tasks.

SOFTWARE DEVELOPMENT TOOLS

"Give us the tools, and we will finish the job."

Looking back over six years much can be said about the tools necessary to develop a system such as TSS, and that is the subject matter of this section.

The types of tools utilized can be broken into three general categories:

- Machinery
- Language Translator with appropriate utilities
- Debugging Aids

Little that is unique in the first two categories was done by TSS Development. As in the case of many total systems projects (hardware and software combinations) software development began well in advance of the availability of the System/360 Model 67 hardware. To overcome this situation, a Model 67 Simulator was created to operate on the standard System/360. The TSS system itself is written in Macro Assembler. Initial assemblies were accomplished on a version of the Basic Operating System/360 (BOS/360) Assembler modified to expand TSS macros. Additionally, a utility function was created which produced TSS object module format from the output of the BOS/360 Linkage Editor.

Debugging aids

The initial set of TSS debugging aids was provided by a system called Support for Test, Release and Time-Shared Operations (STRATO). STRATO was a major modification of BOS/360. It provided three major functions:

- An Interactive Console Command Language
- A Model 67 Simulator
- A Test Case Driver

The STRATO Command Language provided facilities to system programmers to load and unload programs, to start, stop, and dynamically interrupt programs, to display, alter and dump memory. The Test Case Driver would supply inputs to a version of TSS running under the Simulator and record outputs.

Early versions of the TSS Resident Supervisor were hand-built as BOS/360 Jobs and then run and tested...
under STRATO. Once the Supervisor was cycling, critical Virtual Memory Functions were integrated. Much of this work was completed on Model 40 systems before the first Model 67 was installed.

The second phase in the development of debugging aids consisted of removing the dependency upon STRATO. With the availability of the Model 67, the Simulator function was removed, and STRATO was moved to the Model 67. Integration and Test of the TSS Assembler and Dynamic Loader was the next milestone; the dependency upon the BOS Assembler was removed. STRATO itself was released with TSS, since it remained the only system programmer tool for manipulating the resident supervisor and shared virtual memory.

The Time-Sharing Support System (TSS) which was delivered with Release 3.0 removed the final dependency upon STRATO. The Resident Support Subsystem (RSS) and Virtual Support Subsystem (VSS) are more general and easier to use than STRATO command language, but they have one significant weakness. RSS itself is dependent upon some resident supervisor facilities, e.g., interrupt stacker and supervisor core allocation. The effect is that RSS is not a valuable debugging aid for those parts of the resident supervisor.

One of the most valuable aids in TSS was a fortuitous invention—dynamic modification of the system through the use of Delta Data Sets. The system initialization program, STARTUP, builds both the resident supervisor and Initial Virtual Memory (IVM) from modules stored on the Initial Program Load (IPL) Control Volume. (This is in contrast to OS/360 in which the entire supervisor is link edited together during the System Generation procedure.) If dynamic modifica-
tion of the resident supervisor or IVM is desired, the changes are prepared on a private disk and the operator informs STARTUP to search this disk for "deltas." The process is shown schematically in Figure 5. The modifications included in this manner stay in effect only until shutdown. The next time the system is started, they can be removed or altered at the user's option.

To understand the next phase in the development of debugging aids, let us summarize where we stood with Release 3.0. Figure 6 gives a schematic view of the system operation. At his individual terminal, each user had at his command a comprehensive set of debugging aids to work within his virtual memory. With the Program Control Subsystem (PCS) of Command System II, he could dynamically start, stop, or interrupt programs, and he could display or patch within them. The TSS Dynamic Loader allowed him to substitute an altered copy for any module within his virtual memory. This power and flexibility was replicated for each active user, i.e., debugging in non-shared virtual memory was time-shared.

The same was not true either for shared virtual memory or the resident supervisor. Exploratory debugging could be done with RSS and VSS but modules could only be replaced by going through STARTUP. What was desired beyond these aids were techniques to allow time-shared debugging work on both shared virtual memory and the resident supervisor.

Such a facility for modules within shared virtual memory is provided by HOOK. With HOOK a system programmer can cause a private (modified) copy of a shared virtual memory module to be used for his task only. Linkage to the private copy is established dynamically in response to a HK command. If testing shows an error in the module, the programmer can UNHOOK it, assemble further modifications, and HOOK that version for further testing. This activity goes on in parallel with other use of the system, including other users of HOOK. What was said previously about the value of measurement tools to users is probably more true of debugging aids. Recognizing this and realizing that users desire to modify shared as well as non-shared virtual memory, HOOK was made available to all TSS users late in 1971.

Achieving time-shared debugging for the resident supervisor is a more difficult challenge. To meet it, TSS Development looked outside its own domain to Virtual Machines. Such technology underlies the design of CP-67/CMS. CP-67 creates a complete Virtual Machine for each user, so several users can work with virtual Model 67 machines and debug versions of CP. It is clearly feasible to provide Virtual Machines under TSS, and the addition of such a facility would complete a most powerful set of program debugging aids. With Virtual Machines, there would be little, if any, need for single-thread, restart-prone debugging. Figure 7 shows the debugging aids as they could be with Virtual Machine added.

MANAGING SOFTWARE DEVELOPMENT

Bringing any large system from concept to fruition is a difficult task. TSS Development had many problems, some not unlike those described recently by F. P. Brooks, Jr. Probably the most valuable lesson learned in the process was how to manage a large software development project in the "steady state."

The TSS development organization was composed of three major functions: Design, Development, and Build and Test. Ideally, these functions should be accomplished serially, for in this case, the same people could do the work, and no need would exist for communications mechanisms between functions. For practical purposes, the functions do overlap. The design cannot be complete or perfect. The development job must continue to correct program errors and implement new functions. Build and Test is required until the total system is stabilized. Further, the size of TSS both in concept and design precluded implementation by a small group.

Here I will insert an opinion on the merits of large versus small development groups. A small group has two distinct advantages. It is easy to manage (one
person can direct the efforts of 7-12 people); and a uniformly high standard of quality can be maintained in the selection of the group’s members. Large groups suffer from the opposite problems. There is, however, nothing inherently bad about a large programming group. The critical requirement is for effective management.

The systems management technique employed in TSS was known as the New Scope Review Board (NSRB). Control was exercised in the following way. The design of TSS was considered complete at the Release 1.0 level so that the only source of new code would be fixes to identified programming errors (APAR’s). Any other change that was identified was designed and presented to the NSRB for approval. The board was composed of representatives of all functional areas with the System Manager serving as chairman. Approval of a given item meant inclusion of its design into the system and hence implementation, test, and integration in a System Release. With this mechanism in place, sources of new code were restricted to error correction and approved NSRB items.

Each proposal to change TSS was described in an NSRB Technical Summary, composed of eight (8) parts.

1. Title of the item and sequence number
2. Four line abstract of the item
3. Reference to the design document
4. List of all external publications affected
5. A description of the character of the change, e.g., does the item affect performance, reliability, human factors, or function—positively or negatively—and how much?
6. List of other items upon which this item depends, i.e., other NSRB items, APAR’s or hardware Engineering Changes
7. List of all parts of the system affected, e.g., what modules, DSECT’s, Macros, or Data Sets are revised, added or deleted—and, if a revision, is the change small, medium, or large?
8. Signature of the designer and his manager

With all proposals for new programming before the Board and in a standard format, the system became manageable. For instance, a concentrated program to improve performance was instituted by selecting NSRB items whose character indicated large, performance plus while rejecting others. A control on resources was also possible since each NSRB item carried with it an estimate of the work required to complete it. The System Manager could either restrict the quantity of new scope to match his resource or he could add or subtract resource to match the new scope desired.

Figure 8 illustrates the interrelationships of the various functions and the flow of information between them.

The major sources for requirements input are the System Users and a Business Planning Group. Once a requirement is known, a design is prepared and the item is submitted to the NSRB. Those items which are approved form one of two sources of work for the development group.

Here a word should be said about the unique place occupied by the design function. We have mentioned that TSS had a sound basic design, and certainly this was essential to the success of its implementation. We also subscribe to the philosophy that designers must have control over the manner in which the system is developed. Yet, design must be responsive to a multitude of outside influences, including Systems Management. Several of the best features of TSS have resulted from corrections to short-sited design.
A philosophy used throughout TSS has been to deliver the best possible code. Therefore, all program errors detected by the build and test group are submitted to the same APAR Control group as User-detected errors. APAR control verifies the error, eliminates duplicates, and tracks the status of all errors. Errors to be fixed form the second source of work for Development. Here we have had success with the idea of module ownership. The development programmer responsible for a module does all the work necessary to implement NSRB changes and APAR corrections. (Having a separate maintenance group tends to remove an incentive for quality development work, i.e., ones bugs come back to haunt the maintenance programmer not the developer.) The resultant changed modules are the source of input to Build and Test.

The cycle is completed when Build and Test ships a system release.

The table below shows the activity of TSS Development from the three sources described.

TABLE II—System Content by Release

<table>
<thead>
<tr>
<th>Release</th>
<th>Number of NSRB Items Included</th>
<th>Number of APARS Corrected</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>9</td>
<td>91</td>
<td>—</td>
</tr>
<tr>
<td>1.2</td>
<td>32</td>
<td>136</td>
<td>7</td>
</tr>
<tr>
<td>2.0</td>
<td>49</td>
<td>181</td>
<td>72</td>
</tr>
<tr>
<td>3.0</td>
<td>56</td>
<td>190</td>
<td>255</td>
</tr>
<tr>
<td>4.0</td>
<td>42</td>
<td>111</td>
<td>201</td>
</tr>
<tr>
<td>5.0</td>
<td>80</td>
<td>321</td>
<td>534</td>
</tr>
<tr>
<td>5.1</td>
<td>32</td>
<td>192</td>
<td>471</td>
</tr>
<tr>
<td>6.0</td>
<td>84</td>
<td>226</td>
<td>538</td>
</tr>
<tr>
<td>6.1</td>
<td>—</td>
<td>40</td>
<td>85</td>
</tr>
<tr>
<td>7.0</td>
<td>64</td>
<td>174</td>
<td>466</td>
</tr>
<tr>
<td>8.0</td>
<td>50</td>
<td>206</td>
<td>442</td>
</tr>
<tr>
<td>8.1</td>
<td>70</td>
<td>574</td>
<td>1016</td>
</tr>
<tr>
<td>Subsequent</td>
<td>—</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>TOTAL</td>
<td>568</td>
<td>2442</td>
<td>4087</td>
</tr>
</tbody>
</table>

CONCLUSION

Where does TSS go from here? What is the prognosis for large-scale, interactive systems in general? The first question is easier to answer.

Work remains to be done to bring TSS up to a high degree of reliability. This task is receiving the highest priority. In addition, we are dedicated to improving the system’s error recovery facilities. There is a high potential plateau for dependability and availability; we are working to reach it.

Impressive as the TSS performance story has been, we accept our users’ contention that performance is not as good as it needs to be. The proper course now is to specify new benchmarks that more closely parallel the real customers workloads, and to use the powerful tools previously described to drive performance upward.

Last, but not least, is the fact that TSS users have been and are continuing to improve the system. Both the usability and the utility of TSS continue to grow.

The prognosis? I cannot provide that, but I can put forward two thoughts—both deeply rooted in the TSS/360 experience.

I have noted that where TSS/360 has succeeded people efficiency is held more important than machine efficiency, and that it is becoming broadly accepted that the availability of skilled people, not the availability of better hardware, is the limiting factor in the growth of our industry. From this I conclude that, if our industry is to thrive, systems which go to the user, which make people more efficient, in short, interactive systems like TSS must play a dominant role in the decade ahead.

Finally, I will express my belief that the single most important requirement before us is to modernize the program development process itself. Years ago, the application of computing to hardware design and manufacture removed bottlenecks which threatened to retard industry growth. Today, the bottlenecks are in software design and manufacture.

ACKNOWLEDGMENTS

The outlook of this paper is historical not journalistic. I hope that readers who have been more deeply involved in TSS development than I will agree with the sense of the paper even though they disagree with this or that detail.

I have cited some references, and I have made use of some unpublished information. For the latter, I would like to thank M. R. Babacei, D. R. Cease, K. M. Hilmar, A. Kamerman, O. R. LaMaire, and C. E. Seabold.

Finally, it must be said that TSS/360 would have been a very uninteresting subject were it not the creation of fine people. Each man and woman who worked on the system deserves a fair share of credit. My special thanks go to Bill Florac, Scott Locken, Nick Martellotto, and Ida Scott. These people more than any others gave me insight into the nature of TSS.
REFERENCES

1 W T COMFORT  
A computing system design for user service  
1965 FJCC Spartan Books

2 A S LETT W L KONIGSFORD  
TSS/360: A time-shared operating system  
1968 FJCC Thompson Book Company

3 IBM System/360 time sharing system—Command system users guide  
IBM Corporation Form GC28-2001

4 W J DOHERTY  
Scheduling TSS/360 for responsiveness  
1970 FJCC AFIPS Press

5 D N STREETER  
Cost/benefits of computing services in a scientific environment  
IBM Research Report RC 3453

6 F D SCHULMAN  
Hardware measurement device for IBM System/360 time sharing evaluation  
22nd National ACM Conference Proceedings

7 W R DENISTON  
SIPE: A TSS/360 software measurement technique  
24th National ACM Conference Proceedings

8 CP-67/CMS system description manual  
IBM Form GH20-0802

9 F P BROOKS JR  
Why is the software late?  
Data Management August 1971

10 N A ZIMMERMAN  
System integration as a programming function  
24th National ACM Conference Proceedings