A CLASSIFICATION AND SURVEY OF COMPUTER SYSTEM PERFORMANCE EVALUATION TECHNIQUES

P. R. BLEVINS *
C. V. RAMAMOORTHY **
ELECTRONICS RESEARCH CENTER
AND
DEPARTMENT OF ELECTRICAL ENGINEERING
UNIVERSITY OF TEXAS AT AUSTIN
78712

NGR-44-012-144

* Staff Engineer with IBM Office Products Division, currently on educational leave

** Professor of Electrical Engineering and Computer Science, The University of Texas at Austin

MAILING ADDRESS

DR. C. V. Ramamoorthy
Electronics Research Center
University of Texas
Austin, Texas 78712

TELEPHONE

(512)-471-7265

ABSTRACT

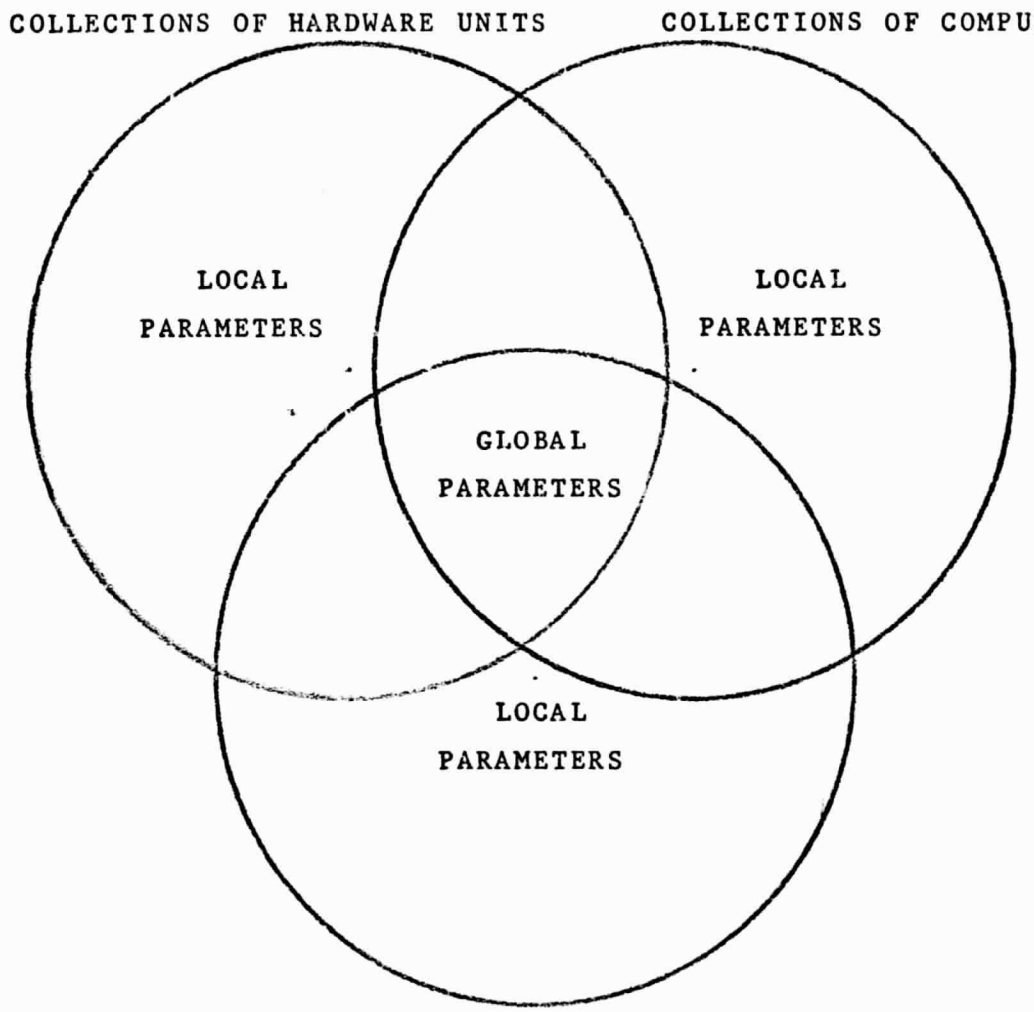
The widely distributed software capabilities coupled with increased compatabilities among hardware units is propagating an ever increasing multitude of computer systems. Such systems aim at a common goal - total system optimization. But achievement of this ultimate goal is being delayed by the absence of a theoretical basis capable of providing a reference for standardized evaluations and comparisons.

The current computer system performance evaluation technology cannot be described in terms of standardized techniques involving established parameters. However, the technology can be partitioned with respect to system component (computing job, hardware unit, and operating system). Within these classifications this paper examines the methodology (analysis, simulation, synthesis) and parameters of typical evaluation techniques presently available. Their advantages and disadvantages are delineated. A comprehensive bibliography is provided both for specific references and for genernal information.

INTRODUCTION

While awaiting fourth generation computer systems, it is timely to examine the available computer system performance evaluation technology - its methodology, its techniques, and its parameters. This technology cannot be described in terms of standardized techniques involving standardized parameters. No theoretical methods or sound conventions exist. But the assortment of techniques (the technology) can be partitioned with respect to methodology and system component. This paper examines the typical techniques available to evaluate computing jobs, hardware units, and operating systems.

The absence of standardized performance evaluation techniques has not deterred the propagation of an ever increasing multitude of computer systems. Such experimenting has been encouraged by industry compatibilities which have allowed individual computer systems to represent many hardware manufacturers, the operating systems to be either the CPU manufacturer's or any one of several software houses and the workload to range from scientific to commerical in a weird mixture of program languages. As a result a computer system must be defined as the cross product of collections of hardware units, collections of operating system programs, and collections of jobs. Considering the number of terms in the cross product the scope of the evaluation job is indeed great.



THE COMPUTER SYSTEM

FIGURE 1

A general performance evaluation approach must focus upon the isolation and characterization of significant performance parameters - those which characterize the rate and amount of work performed and the availability to perform work. From this viewpoint Computer System Performance Evaluation can be defined as the process of analyzing, simulating, and/or synthesizing the behavior of computer system performance parameters. The absence of adequate performance evaluation parameters has deterred the growth of theoretical methods of computer system organization and design and delayed the ultimate goal - total system optimization. These techniques must provide both static and dynamic measures of components in both a dedicated and a multitask system environment,

But today's system implementation technologies have far outraced performance evaluation technologies. The seriousness of the precipitating problem can be focused by noting that 1969's 21.5 billion dollar investment (53,500 installations) in computer systems is forecast to exceed 50 billion dollars (128,000) by 1975! With such investments at stake computer system optimization promises great rewards for three interest groups, the equipment manufacturers, the computation centers, and the users. Each group is motivated by different primary goals.

Users are interested in performance parameters such as response time(turnaround time), and execute time. In brief they want maximum convenience consistent with economy.

To meet the demands of their many users the computation centers seek to maximinize system resource utilization. They evaluate present systems in order to improve average throughput, average response time (turnaround time), and average operation cost while allowing non-average users to be characterized by parameter variances.

Manufacturers view performance evaluation as the key to meeting present and future demands from the varied and rapidly expanding market. Interests focus upon developing the capability to accurately evaluate present and proposed systems, and to predict performance parameter behavior for systems still in early stages of development. Their individual goals are to maintain a competitive edge by offering computer systems (hardware and software) possessing optimum cost/performance ratios.

METHODOLOGY

Surveying the past decade's literature exposes a wide range and direction of computer system performance evaluation techniques. The decade has produced computer systems ranging from simple batch processing, unit record, uni-processor systems to complex multi-programmed, telecommunication, multiprocessor systems. The corresponding performance evaluation methods have ranged from weighted instruction mix execution time comparisons (ANALYSIS) to time-sharing system simulation models (SIMULATION) to memory hierarchy modeling (SYNTHESIS).

The direction of evaluation methodology has slowly turned from analysis toward synthesis. Analysis includes the separating and breaking up of the computer system into parts so that their functions, relationships, and properties may be understood. Data can be gathered either from real computer systems or from simulation models. By analysis the cost per million instruction executions could be determined based upon the manufacturer's system costs and specifications.

Synthesis includes the combining of known components (hardware and software) into a total computer system or subsystem so that the whole system's (or subsystem's) functions, relationships, and properties may be understood and predicted. Data can be gathered either by measuring real individual parts or simulation models of individual parts. Synthesis represents the highest level of evaluation since it allows performance to be predicted rather than just to be measured. An example of synthesis would be

predicting the average access time of a memory hierarchy based upon the specifications of its components.

Simulation plays a vital role in both analysis and synthesis. This tool allows complex systems to be studied under known conditions and controls. Many real systems are too complex for theoretical analysis and the use of simulation may prove to be the only workable approach. However, the time and cost required to generate accurate simulation models can easily exceed that of direct measurement approaches. But for evaluating proposed systems which yet do not exist in hardware, simulation offers an alternative to building prototypes.

PARAMETERS

Any evaluation technique must consider a set of parameters referenced to a specific environment. Three parameters, throughput, response time, and cost per operation, commonly denote global system capacity. Common local parameters include: average access time, average data transfer rate, average instruction execution time, hardware (processor, memory, tape, drum, channel, etc.) utilization, average executive processing percentage, average job processing percentage, and effective busy time of the processor. The design of specific computer systems reflects the weights assigned to these parameters.

Availability represents a different type of global performance parameter. It is usually expressed as a percentage or as an amount

of time. Manufacturers control availability through component reliability and redundancy, error detection and correction, and preventative maintenance. Computation centers influence availability through planned graceful degradation involving backup equipment (I/O, memory, etc.) and job rescheduling.

Throughput can be considered as the steady state work capacity of a computer system . It is usually expressed as a number of specific tasks performed per time interval. For example, a system's throughput might be described by the number of typical PL/1 statements compiled per hour or by the number of information retrieval requests satisfied per hour. Many applications demand specific throughput levels. As a result the system must be designed to achieve such levels.

Response time can be considered as a transient work capacity of the computer system. It denotes the delay between submitting the job and receiving the job output. [32] Clearly it is a function not only of the system throughput but also of the system environment including the number of jobs and their priorities. For many batch processing systems the response time (turnaround time) may be relatively constant and measured in hours. But for certain time-sharing systems the response time may be only seconds. For example the response time for a stock broker's quotation request or an airline reservation inquiry may be less than 5 seconds.

The third global parameter measures performance in units of money per operation unit time. It is usually given as dollars per hour. Applications exist where more time than money is allowed.

However, definite economies of scale exist for computer systems. Solomon [35] has showed that the IBM System 360 models possess ratios of 1.6 to 2.1; i.e. by increasing the system investment by a factor K the internal operating speed increases by K raised to a power ranging from 1.6 to 2.1 depending upon the model. For example the average monthy rental for models 75 and 30 are approximately $80,000 and $8,000 respectively. With K = 10 an improvement of 100 in the internal operating speed would be expected. For a scientific instruction mix [35] the Model 75 is approximately 40 times faster. Such comparsions are usually referenced to the famous Grosch's Law which equates ''added economy only as the square root of the increase in speed - that is, to do a calculation ten times as cheaply you must do it one hundred times as fast.''[18]

The uninitiated might attempt to express system performance as a single entity involving a relationship between the global parameters, throughput, response time, operation cost, and availability. However, such a figure of merit has not evolved, because system performance is not a single entity!

USER JOB TRANSFORMATIONS

From its definition to execution a problem undergoes many transformations. The total transformation efficiency of this process can be computed as the product of the individual transformation efficiencies. As a result the user who investigates why his program possesses poor performance must carefully consider several major transformation efficiencies including the following:

1. Development of the problem statement and solution algorithm (user)
2. Generation of a high level source language program (programmer)
3. Generation of an assembly language program (compiler)
4. Generation of the appropriate machine language code (assembler or loader)
5. Ceneration of the problem's solution by managing the job's execution on the system hardware (operating system)

The first transformation is performed by the scientist or engineer who reduces the problem to a statement and developes a solution algorithm. This first transformation is not a part of computer system performance evaluation, but is noted in order to stress the importance of the communication link between the scientist and the programmer. In order to maximinize this transformation efficiency the problem must be precisely stated and an optimum solution algorithm chosen. It is usually helpful to develop several solution algorithms so that the chosen 'optimum' algorithm represents more than just one way to solve the problem.

The second transformation and probably the most important transformation is performed by the programmer who reduces the problem statement and solution algorithm into a high level source language program such as FORTRAN, ALGOL or COBOL. To maximinize this transformation efficiency the programmer must consider many parameters including: source language selection, I/O treatment, data treatment, and system resources and properties. Major errors such as wrong source language selection are usually avoided, but more subtle errors may strongly degrade the transformation effeciency. For example consider a high usage DO LOOP containing numerous initialization statements. Such statements might easily be moved outside the DO LOOP resulting in significant execution time improvements.

Familiarity with available system subroutines must not be underrated. Such subroutines usually represent sophisiticated and highly optimized algorithms in either assembly or machine language. As a result their proper use eliminates the degradation introduced by the compiler and minimizes the inefficiencies introduced by the programmer. For example consider the drastic effect (reduction) upon a program's execution time should a programmer approach the Fibonacci difference equation solution using iterative rather than recursive techniques.

Note that it is not unusual for the programmer to eliminate the step involving the high level source language transformation by directly generating the program in the appropriate assembly language. Directly generating assembly language requires additional

programming skill and time and produces a machine oriented program. However, a skilled programmer can usually generate more efficient assembly language code than the two step transformation including generating the high level source language followed by compilation. Also, for high usage programs elimination of the compiling time becomes an important consideration.

Usually the assembly language is generated by the system's appropriate source language compiler. Such compilers represent one of the most highly optimized system programs. However, each compiler represents a compromise between speed, menory requirements, and output code quality. This compromise should represent the optimum cost/performance ratio for the user, but a common underlying assumption is that only low usage programs are involved. It is this assumption that allows significant inprovements to be gained for high usage progams by the programmer working directly in assembly language.

Machine language generation probably represents the most highly efficient of all the transformations. Converting assembly language into machine code is mainly a one to one mapping procedure which needs to be performed automatically due to its tedious nature. The assembler or loader can do little to overcome program inefficiencies embedded one, two or three transformation levels. Note that it has been assumed that each transformation step possesses a maximum efficiency of one. Efficiencies greater than one would imply that the transformation process possesses 'error' correcting capabilities.

Under control of the operating system the object program is executed by the system hardware. Here the efficiency of the transformation (execution) depends upon the operating system and the system resources available. Response time and throughput depend not only on the internal speeds and capablilities of the hardware but also upon the efficiency with which the Operating System manages the execution of the user program. Clearly an inefficient Operating System globally affects system performance.

Tracing the transformations of the user problem from definition through execution introduces the entire computer system - the collections of jobs, the collections of hardware units, and the collections of operating system programs. The effect of each collection upon the system performance parameters such as throughput, response time and cost per operation can to a degree be isolated in order to establish performance bounds. Since it is difficult to determine effects of the collections in a complete, interactive system environment it is both expedient and meaningful to develop evaluation techniques of individual system components.

EVALUATION TECHNIQUES FOR COMPUTING JOBS

Evaluation techniques for the collections of jobs can produce significant benefits. Developed techniques allow dynamic monitoring of program execution in order to expose the program's traffic patterns and execution time density patterns. Other techniques provide modeling tools for studying program structures . Common purpose is to contribute input data for program improvement processes.

DYNAMIC MONITORING OF SINGLE JOBS

Three general approaches have been taken to achieve dynamic program execution monitoring. They include self monitoring by introduced artifact and external monitoring by hardware/software devices (snuping). Also, various techniques allow programs to be executed during so-called interpretative modes which are either controlled by special programs or by special hardware features.

The basic purpose of all such techniques is to gather dynamic statistics such as subroutine execution times, branching probabilities, subroutine usage and resource usage. Such statistics provide important input to program optimization processes.

ARTIFACT - Let the program be represented by a directed graph with statement(s) corresponding to nodes and statement transitions corresponding to arcs. The activity of the nodes and arcs can be monitored by inserting artifact at the desired monitor points. Such artifact can take the form of subroutine calls. In a FORTRAN

program the following subroutine calls might be used;

CALL COUNTER(I)

and CALL TSTAMP(I)

To record the usage frequency the subroutine, COUNTER, simply counts the number of times that it is called. To record execution times the called subroutine, TSTAMP, fetches the system clock time using the CALL SECOND subroutine and then labels and stores the clock time. Values of the index I correspond to unique monitor points.

The artifact technique is simple to implement, but increases overall execution time of the monitored program. Also, no theoretical methods exist to locate the strategic monitor points required to minimize the execution time cost with respect to the information collected.

The technique allows program traffic patterns and execution time density patterns to be measured as a function of the input data. Since the technique locates the high traffic program segments, it helps to locate those segments which should be considered for optimization. The technique fails to ideally record execution times since the artifact execution times are also measured by the system clock. Compensation for the artifact execution times is tedious, but often it can be ignored if the measured segments are relatively large. Similarly the execution time density patterns pinpoint those program segements which should be optimized in order to improve the program execution time.

EXTERNAL MONITORS - The hardware monitor has been used effectively to gather dynamic statistics of programs. Examples include the IBM Program Monitor [1], the IBM Time Sharing System Performance Activity Recorder (TS/SPAR) [33], and the UCLA Snuper Computer [15]. The last two examples are also highly applicable toward evaluating Operating Systems. Basically these hardware monitors provide means of detecting the contents and/or activity of logic lines and means to store such gathered data. For example the early IBM Program Monitor recorded the contents of the IBM 7090 computer's instruction counter. The monitor was able to keep pace with the 7090 by recording only certain selected types of instructions, buffering and packing the gathered data, and halting the 7090 whenever the buffer capacity was exceeded. Data was recorded on magnetic tape and later processed by several special programs. Final results were presented either as printout or on film. Principal benefits of such a technique includes the generated descriptions of the program segment execution time and frequency. Also, no artifact is introduced to affect the execution timing.

The UCLA Snuper Computer possessed several significant differences. A second computer (Sigma 7) was used to control the monitoring of the object computer (IBM 7090). Means were provided to monitor numerous internal signals and to present unique messages at the interface to the Snuper Computer. In Phase 1, artifacts were introduced into the object computer's programs at significant event points. However, the artifact consisted of so-called emitter calls which caused corresponding unique messages to be presented

at the interface to the Snuper Computer. The messages were interpreted as unique memory addresses by the Snuper Computer and the corresponding memory locations were indexed by one to provide event counters. In summary the Phase 1 Snuper Computer allowed program monitoring with reduced artifact, but required a second computer complete with monitoring hardware and programs.

For Phase 2 the Snuper Computer completely eliminated the artifact in the object computer programs by gathering sufficient data during the program compilation and loading phases to generate a Significant Event Filter (SEF). Interface messages were used to address the SEF and if the corresponding bit were a one the event was defined to be significant. Event counting proceeded as in Phase 1. In both phases gathered data was later processed in order to extract important statistics.

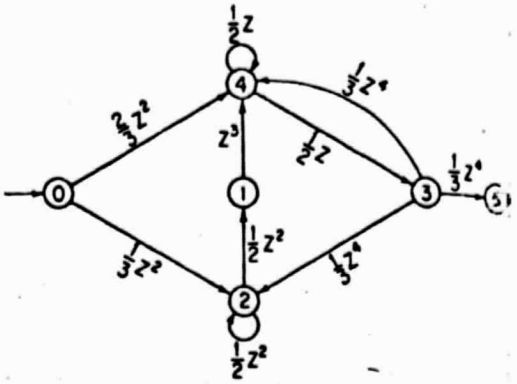
Hardware monitors such as those just described are best suited for laboratory use. They lack portability, introduce object system downtime for connections, are expensive to operate and develop, and are usually designed to monitor only a very small class of computers (probably only one model). However such techniques have been accepted by manufacturers as useful tools in program optimization processes. Even if the use of such tools cannot become wide spread, their benefits may become widespread through the distribution of optimized system programs (a user benefit).

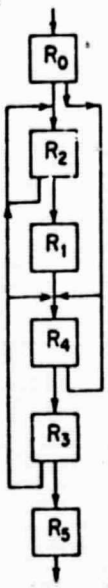
GRAPH ANALYSIS

Since the programmer's flow chart is a directed linear graph, many techniques of graph theory are applicable to the analysis and modeling of programs. A general model has the nodes corresponding to computational subsequences (statements or subroutine) and the arcs corresponding to execution flow. The model can be expanded by assigning execution times to the nodes and branching probabilities and resource requirements to the arcs. [23,28] By changing the node and arc assignments (statement, subroutine, job, etc.) the directed graph model affords a general approach toward modeling a wide range of problems associated with program execution.

There are a number of techniques [14] to transform cyclic graphs into acyclic graphs. Such schemes allow the cyclic program structures such as loops to be replaced by deterministic acyclic structures involving expected execution frequencies. Such a scheme simplifies calculating a program's expected execution time. However development of the expected branching probabilities for non-deterministic loops such as those iterated under the control of a while clause still depend upon either dynamic program monitoring statistics or a knowledgeable estimate. In either case the model is input data sensitive and a proper range of input data must be considered.

However even if all the necessary assumptions were valid, the graphical approach toward program analysis often results in a complex process. Consider the example [25] shown in Figure 2 where it is assumed that the branching probabilities are statistically independent and data insensitive. A discrete Markov analysis of the expected execution time for the six node graph ivolves a complicated process.





MATRIX OF EXECUTION TIMES

R	t	TIME UNITS
R_0	t_0	2
R_1	t_1	3
R_2	t_2	2
R_3	t_3	4
R_4	t_4	1
R_5	t_5	0

MATRIX OF BRANCHING PROBABILITIES

p_{ij}	0	1	2	3	4	5
0	0	0	1/3	0	2/3	0
1	0	0	0	0	1	0
2	0	1/2	1/2	0	0	0
3	0	0	1/3	0	1/3	1/3
4	0	0	0	1/2	1/2	0
5	0	0	0	0	0	0

$$G_{05}(z) = \frac{\frac{1}{9}z^7 - \frac{1}{18}z^9 + \frac{1}{36}z^{12}}{1 - \frac{1}{2}z - \frac{1}{2}z^2 + \frac{1}{4}z^3 - \frac{1}{6}z^5 + \frac{1}{12}z^7 - \frac{1}{12}z^{10}} \quad (1)$$

$$\bar{t} = \left.\frac{d\,G_{05}(z)}{d\,z}\right|_{z=1} = 29\ 1/3 \text{ units of time.} \quad (2)$$

FIGURE 2

EVALUATION TECHNIQUES for HARDWARE UNITS

The earliest computer system evaluation techniques focussed on evaluating the hardware units since systems at that time were greatly limited by the hardware technology. During the era of the batch processing systems performance was very strongly characterized by the performance of major hardware components such as the CPU and the memory. Evaluation techniques mostly compared the execution times of single machine instructions such as ADD, frequency weighted instruction mixes, and program kernels.

With the introduction of the multiprogrammed systems attention shifted toward developing application benchmarks. But with the introduction of the time sharing systems and the present generation of super computers (CDC 7600, IBM 369/91,195) attention shifted toward developing simulation technques.

INSTRUCTION EXECUTION TIME COMPARISONS

Probably the first technique used to compare different CPU's merely used the execution times for single instructions such as ADD or MULTIPLY. This technique is definitely application sensistive and therein is its great weakness. The performance of only a very limited class of programs can by characterized accurately by considering only the ADD and MULTIPY execution times. For example a typical matrix multiplication program consists of only about 25% ADD and MULTIPLY type instructions [35]. Monitoring scientific installations has shown that the ADD/SUBTRACT and MULTIPLY/DIVIDE instruction types compose less than 20% of the instructions executed. [3,25,35]

Other weaknesses of the technique includes the failure to compensate for the CPU's having different operand sizes, different organizations (one address, multi-address, auxiliary registers, multiple execution units, etc.), and different instruction implementation algorithms (pure binary, decimal, microprogrammed, floating point).

STORAGE CYCLE TIME COMPARISONS

Another early evaluation technique compared the storage cycle times . The usefulness of this scheme has been outdated by interleaving memory banks, high speed 'cache' memories, and large storage access widths. For example the IBM 360/85 possesses a storage access width of 16 bytes (128 bits) with either 2 or 4 way interleaving plus a 32k byte 'cache' memory with a 80 nanosecond cycle. [17] Such features would be overlooked if only the .96 microsecond storage cycle time were considered. Considering that the IBM 360/25 has a .90 microsec. storage cycle time, a very false performance ratio might be deduced. The role of the 'cache' memory drastically affects the effective storage cycle of computers. It has been reported that the IBM 360/195 'cache' memory satisfies an average of 99% of all storage requests and that varying the backing bulk memory access time from a fraction to 2 microseconds affects system performance by only 10 to 15%. Clearly the storage cycle time parameter must either be redefined or marked as insignificant when evaluating such modern computers.

The Maximum Storage Bus Rate (MSBR) has evolued as a modern figure of merit.

MSBR = (access width/storage cycle time)x interleave factor However, MSBR fails to compensate for the 'cache' memory, but would show that the IBM 360/85 has a MSBR= 533.33 megabits while the model 25 has only a MSBR= 17.77 megabits.

INSTRUCTION MIX EXECUTION TIME COMPARISONS

Recognizing some of the weaknesses of the Instruction Execution Time Comparisons, the Weighted Instruction Mix approach was developed. [3,25,35] From actual operating systems, statistics were gathered to weigh individual instructions according to their frequency of occurance in a particular set of applications. A weighted instruction mix execution time can then be calculated. Arbuckle [3] has presentedd the following scientific application mix:

INSTRUCTION TYPE	FREQUENCY PERCENTAGE
Floating Point Add/Subtract	9.5%
Floating Point Multiply	5.6%
Floating Point Division	2.0%
Load/Store	28.5%
Indexing	22.5%
Conditional Branch	13.2%
Other	18.7%

Again this approach suffers from failure to correct for the CPU's operand size, organization and algorithm implementation. Properties of the instruction stream such as sequence and I/O operations are ignored. Instruction sets which may differ greatly

in power are evaluated on the basis of certain universal instructions while leaving 20 to 30% as miscellaneous instructions. Since the statistics are collected from a particular system they reflect that system's properties such as its organization, compiler, assembler, operating system efficiency, and workload. A system's workload depends on both the job types and their number. Excessive amounts of waiting time would certainly bias such statistics abnormally. However, the technique provides very useful information to the hardware designer since high frequency instructions are pinpointed. By optimizing such high frequency instructions the average number of jobs executed per wait time can be increased.

PROGRAM KERNELS

To overcome many of the weaknesses of the Instruction Mix approach, the program kernel was introduced. [3,8,35] The kernel represents the CPU coding required to perform a significant task such as evaluating a polnominal, solving a set of simultaneous linear equations or multiplying two matrices. Obviously the kernel allows a skilled programmer to use all the features of a CPU. A significant property as a yardstick is that it possesses poor correlation with weighted instruction mix results. Just how representative of a system workload can a matrix multiplication kernel be? It is no surprise that system performance based upon such a kernel representing a very specific job possesses poor correlation with the Weighted Instruction Mix technique which is much more representative of a typical system workload. Certainly both techniques offer convenient means of gathering data. The capability of the program kernel to provide data referenced to a specific job should not be ignored.

However, kernels have proved useful in measuring the instruction execution rate of high performance computers such as the IBM 360/195. No single machine cycle can be identified with a single instruction execution due to the overlap, 'pipelining' and lookahead in instruction execution. Iterations of kernels as small as 15-20 instructions have been employed to establish effective instruction execution rates measured in millions of instructions per second. Much attention must be given to the content of such kernels since the effective instruction rate of such computers is strongly affected by the instruction stream and storage requests.

A significant weakness of evaluating system performance based upon individual kernel execution times is the failure to measure the efficiency of the Operating System. The arrival of multiprogramming and multiprocessing initiated the Application Benchmark evaluation technique.

APPLICATION BENCHMARK

Presently the internal performance of computer systems is frequently validated by application benchmarking. [7,19] This technique involves timing the execution of a typical collection of user jobs. One manufacturer suggests that the user benchmark contain 15 frequently used application jobs. Such a technique provides the user with much specific information since his own jobs are actually executed and timed. Such a collection of jobs tends to produce average performance parameter values since more events occur in the sample space of jobs. Also, it provides a test for the multiprogramming and/or multi-processing operating system. By

running the Application Benchmark on competitive systems, performance ratios may be rather accurately established by the user. Contents of the benchmark should be carefully chosen to insure that the user's tasks are fairly represented. Such results are representative of total system performance (hardware and software).

SIMULATION MODELING

Often simulation modeling provides the best if not only means of studying today's sophisticated hardware units. Designing hardware units such as CPU's and memory hierarchies [2] involves optimizing parameters beyond the scope of simply 'tuning' a hardware prototype. For example a hardware prototype would have provided an inadequate tool for evaluating a concept such as the IBM 'cache' memory. [17] Optimizing overall performance as a function of design parameters such as 'cache' memory capacity, transfer block size, backing memory access time, and program structure characteristics could definitely be satisfied best using a simulation model. As a result the development of a prototype is no longer the starting point for design optimization. It frequently represents the hardware implementation of an optimized simulation model whose primary purpose is to prove the feasibility and reliability of unknown components and whose secondary purpose is to prove the accuracy of the 'optimum' simulation model.

However, a close examination of a simulation model exposes the many assumptions and approximations used to construct the model. It is the accuracy of these numerous and often difficult assumptions that establishes the validity of the simulation evaluation results. Probably the major consideration involves the compromise which must be made between simulation detail and cost. Simulation is not the answer to every evaluation problem, but it frequently provides a technique to evalute new concepts without filling warehouses with ''golden prototypes.''

EVALUATION TECHNIQUES for COLLECTIONS OF OPERATING SYSTEMS

To a degree the performance influences of both the collections of jobs and collections of hardware units can be isolated and characterized. Such limited information can in some cases even establish bounds on total system performance. But total system performance is accurately measured only by global performance parameters such as throughput, response time, and cost per operation. These parameters are functions of all three system collections.

SIMULATION

Simulation models have been widely used to evaluate today's operating system designs. References include articles concerning a wide range of operating system types including real-time [37], time-sharing [16,24], multiprocessing [5,9,34], and multiprogrammed [20]. Simulation has become a vital part of the developement of every operating system. Often manufacturers must include the results of simulation testing as an integral part of large system proposals. The UNIVAC 1108 multiprocessor Evaluation of System Performance (ESP) model [9] exemplifies a typical simulation project. It has been employed to evaluate the Jet Propulsion Laboratory multiprocessor system. Statistics reflecting channel utilization, processor utilization, peripheral equipment utilization, system overhead, and effective processor busy time can be gathered by simulation. The model construction reflects the existing 1108 executive system (EXEC 8) including the Executive Control, Input/Output, and Scheduling functions.

System hardware resources are characterized by average access and transfer times. The user provides input job descriptions. From this information the number of interrupts and I/O transfers are calculated and their execution simulated using detailed descriptions of the executive system (EXEC 8) functions and the hardware resources. System performance can then be predicted. Up to eight reports reflecting system performance statistics are generated. They include: number of I/O channel loads including utilization percentage, executive system routine usage for the individual jobs plus the total workload, executive and workload instruction totals, interrupt summary, software utilization distribution percentages, and hardware utilization plus a general summary reflecting throughput and busy time .

SELF MEASUREMENT

A convientent approach to monitoring an operating system's performance is to introduce artifact into the operating system program. Such artifact allows statistics to be gathered dynamically. A current example is the IBM System Internal Performance Evaluation (SIPE) program which is designed for the System/360 Time Sharing System [12]. This software measurment technique introduces SIPE 'hooks' into the resident supervisor program at strategic locations. Logically these 'hooks' function as subroutine calls to introduce SIPE into the normal program stream. Each 'hook' possesses an unique identifier code which drives the SIPE program to collect specific information about the system status. This snapshot of the system status includes contents of internal registers, specific locations of main memory, and a time stamp. The collected data is buffered and later transferred to tape. From these tapes data reduction programs extract various statistics and format them for display.

SIPE was designed for customer installation use as well as laboratory use. The current IBM 360/TSS resident supervisor programs contain the SIPE 'hooks.' Unless activated the effect of the 'hooks' upon system performance is insignificant. When fully activated system performance is degraded only about 8%. As a result SIPE can be justified for frequent user use such as installation debug.

Software approaches to evaluating operating system performance face one serious compromise since executing the introduced artifact slows down the system and may percipitate abnormal system conditions. Event resolution and system degradation must be compromised.

Significant features of software approaches such as SIPE include portability, flexibility, and relatively low cost. The SIPE 'hooks' (8 bytes each) are actually part of the resident supervisor. SIPE itself must be link-edited into the system during startup. Obviously no special hardware exists in such a totally software technique. Great flexibility exists since only the resident supervisor contains the 'hooks.' All the TSS/360 utility functions are unaltered. Evaluation of new TSS configurations involves only adding or deleting 'hooks' in the supervisor. Once developed the technique offers low operating cost and possesses no holding (storage) costs such as some hardware monitors like the Snuper Computer.

EXTERNAL MONITORS

External hardware monitors offer another approach to evaluating operating systems. Devices can range from simple usage meters and channel activity counters to complex units such as the UCLA Snuper Computer [15], the UNIVAC 1108 Hardware Monitor [29], and the IBM Time Sharing System Performance Activity Recorder (TS/SPAR) [33]. Since such devices are interference free their prime advantage is the capability to gather dynamic statistics in a non-degraded system environment. But sophisticated devices such as the

last three possess properties which limit their general use. They lack portability, flexibility, and economical operation of software approaches such as SIPE. For example their large physical size limits portability and their installation can impose excessive downtime for the monitored system. Since they must interface with hardware, they are designed to monitor a particular system's internal architecture and circuit family. But most of the large hardware monitors were intended only for laboratory use. They can contribute to installation optimization directly through the development of better operating system programs.

CONCLUSION

This paper has classified and examined the presently available assortment of computer system evaluation techniques. The techniques have been partitioned with respect to computing jobs, hardware units, and operating systems. Within each classification, techniques representing analysis, simulation, and/or synthesis have been examined in order to expose the advantages and disadvantages inherent to the specific techniques.

Characterizing the present technology provides both an outline of what is available and an outline of what needs to be developed. Fundamental among the needs of the computer evaluation technology is a theoretical basis providing a reference for standardized evaluations and comparisons. First, standardized parameters should be established. Such parameters must characterize global system performance; they must provide common denominators allowing local parameters to be reduced. Then appropriate techniques may be generated to measure these parameters under known system environments and workloads. Certain typical classes of installations might be represented by standard workloads consisting of standardized kernels or application benchmarks (7). Special purpose installations must continue to develop customized application benchmarks for comparative testing in order to evaluate competitive proposals.

Besides evaluation for selection purposes there remains the important problem of monitoring the installed system. Such performance monitoring could be greatly benefited by designing snuping features into the hardware units and the operating systems (36) in order to collect utilization and usage frequency statistics. Ideally such features would operate in a parallel and interference-free fashion. Those techniques such as software artifact which do cause interference should provide means for their selective use.. Built-in snuping features would allow computation centers to continually monitor their systems in order to provide early detection of system environment changes and precipatating problems. Also, such field usage statistics would provide the manufacturer much important input for future system designs.

REFERENCES

1. C. T. Apple, "The Program Monitor-A Device for Program Performance Measurements," Proceedings 20th ACM National Conference (Aug. 1965), pp. 66-75.

2. W. Anacker and C. P. Wang, "Performance Evaluation of Computing Systems with Memory Hierarchies," IEEE Trans. Comp., Vol. EC-16, No. 6 (Dec. 1967), pp. 764-773.

3. R. A. Arbuckle, "Computer Analysis and Thruput Evaluation," Computers and Automation, Vol. 15, No. 1 (Jan. 1966), pp. 12-15,19.

4. A. J. Bonner, "Using System Monitor Output to Improve Performance," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 290-298.

5. F. R. Baldwin, W. B. Gibson and C. B. Poland, "A Multiprocessing Approach to a Large Computer System," IBM Systems J., Vol. 1, No. 1 (Sept. 1962), pp. 64-76.

6. W. Buchholz, "A Selected Bibliography on Computer System Performance Evaluation," Computer Group News (March 1969), pp. 21-22.

7. W. Buchholz, "A Synthetic Job for Measuring System Performance," IBM System J., Vol. 8, No. 4 (1969), pp. 309-318.

8. P. Calingaert, "System Performance Evaluation: Survey and Appraisal," Comm. ACM, Vol. 10, No. 1 (Jan. 1967), pp. 12-18.

9. J. A. Caron, L. R. Jorze and K. D. Tschetter, "Evaluation of System Performance Model," UNIVAC Advanced Systems and Programming Report P.X.-5426 (Aug. 11, 1969), 67 pages.

10. P. S. Cheng, "Trace-Driven System Modeling," IBM Systems J., Vol. 8, No. 4 (1969), pp. 280-299.

11. E. G. Coffman, Jr., "Analysis of a Drum Input/Output Queue Under Scheduled Operation in a Paged Computer System," ACM J., Vol. 16, No. 1 (Jan. 1969), pp. 73-90.

12. W. R. Deniston, "ATSS/360 Software Measurement Technique," Proceedings 24th ACM National Conference (Aug. 1969), pp. 229-245.

13. G. Estrin and D. Martin, "Experiments on Models of Computations and Systems," IEEE Trans. Comp., Vol. EC-16, No. 1 (Feb. 1967), pp. 59-69.

14. G. Estrin and D. Martin, "Models of Computational Systems-Cyclic to Acyclic Graph Transformations," IEEE Trans. Comp., Vol. EC-16, No. 1 (Feb. 1967), pp. 70-79.

15. G. Estrin, D. Hopkins, B. Coggan and S. D. Crocker, "Snuper Computer: A Computer in Instrumentation Automation," AFIPS Conf. Proc., Vol. 30 (1967 SJCC), pp. 645-656.

16. G. Estrin and L. Kleinrock, "Measures, Models and Measurements for Time-Shared Computer Utilities," 1967 ACM National Meeting Proceedings, pp. 85-96.

17. Donald H. Gibson and W. Lee Shevel, "'Cache' Turns Up A Treasure," Electronics, Vol. 42, No. 21 (Oct. 13, 1969), pp. 105-107.

18. H. R. J. Grosch, "High Speed Arithmetric: The Digital Computer as a Research Tool," Optical Society of America Journal, Vol. 43, No. 4 (April 1953), pp. 306-310.

19. E. O. Joslin, "Application Benchmarks: The Key to Meaningful Computer Evaluation," Proceedings 20th ACM National Conference (Aug. 1965), pp. 27-37.

20. J. H. Katz, "An Experimental Model of System/360," Comm. ACM, Vol. 10, No. 11 (Nov. 1967), pp. 694-702.

21. D. D. Keefe, "Hierarchical Control Programs for Systems Evaluation," IBM Systems J., Vol. 7, No. 2 (1968), pp. 123-133.

22. H. G. Kolsky, "Some Computer Aspects of Meteorology," IBM Journal, Vol. 11, No. 6 (Nov. 1967), pp. 584-600.

23. T. G. Lowe, "Analysis of Boolean Program Models for Time-Shared, Paged Environments," Comm. ACM, Vol. 12, No. 4 (April 1969), pp. 199-205.

24. N. R. Nielsen, "The Simulation of Time Sharing Systems," Comm. ACM, Vol. 10, No. 7 (July 1967), pp. 397-412.

25. E. Raichelson and G. Collins, "A Method for Comparing the Internal Operating Speeds of Computers," Comm. ACM, Vol. 7, No. 5 (May 1964), pp. 309-310.

26. C. V. Ramamoorthy and M. J. Gonzales, "Recognition and Representation of Parallel Processable Streams in Computer Program-II (Task/Process Parallelism)," Proceedings 24th ACM National Conference (Aug. 1969), pp. 387-397.

27. C. V. Ramamoorthy, "Discrete System Representation and Analysis by Generating Functions of Abstract Graphs," IEEE Int. Conv. Rec. 1965, pp. 68-80.

28. C. V. Ramamoorthy, "Discrete Markov Analysis of Computer Programs," Proceedings ACM National Conference 1965, pp. 386-392.

29. D. J. Roek and W. C. Emerson, "A Hardware Instrumentation Approach to Evaluation of a Large Scale System," Proceedings ACM National Conference 1969, pp. 351-367.

30. M. Schatzoff, R. Tsao and R. Wing, "An Experimental Comparison of Time Sharing and Batch Processing," Comm. ACM, Vol. 10, No. 5 (May 1967), pp. 261-265.

31. P. H. Seaman and R. C. Soucy, "Simulating Operating Systems," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 264-279.

32. Allan L. Scherr, "An Analysis of Time-Shared Computer Systems," Massachusetts Institute of Technology, Project MAC Technical Report MAC-TR-18, June 1965.

33. F. D. Schulman, "Hardware Measurement Device for IBM System/360 Time Sharing Evaluation," Proceedings 22nd ACM National Conference (Aug. 1967), pp. 103-109.

34. E. C. Smith, "A Directly Coupled Multiprocessing System," IBM Systems J., Vol. 2, No. 3 (Sept.-Dec. 1963), pp. 218-299.

35. M. B. Solomon, Jr., "Economics of Scale and IBM System/360," Comm. ACM, Vol. 9, No. 6 (June 1966), pp. 435-440.

36. W. I. Stanley, "Measurement of System Operational Statistics," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 299-308.

37. W. E. Stanley and H. F. Hertel, "Statistics Gathering and Simulation for the Apollo Real-Time Operating System," IBM Systems J., Vol. 7, No. 2 (1968), pp. 85-102.

38. E. S. Walter and V. L. Wallace, "Further Analysis of a Computing System Environment," Comm. ACM, Vol. 10, No. 5 (May 1967), pp. 266-272.

39. Peter White, "Relative Effects of Central Processor and Input-Output Speeds Upon Throughput on the Large Computer," Comm. ACM, Vol. 7, No. 12 (Dec. 1964), pp. 711-714.