MODELS OF SYNTHETIC PROGRAMS PROPOSED AS PERFORMANCE EVALUATION TOOLS IN A COMPUTER NETWORK

by

DAVID CULP

B.A., California State University, Pomona, 1972

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1976

Approved by:

Major Professor

Table of Contents

List of Figures

Introduction

With the rising cost and complexity of modern day computing systems, accurate and economical performance evaluation techniques are becoming critically important. There are three basic reasons for conducting a performance evaluation study. They are:

1. Sizing and selecting a new computer system.
2. Predicting the effects of changes in the current system or job stream.
3. Tuning present system performance.

All too often in the past a performance evaluation study was undertaken only for the initial purchase of a computer system. Performance monitering of an existing system is of at least equal importance and should not be overlooked. Monitoring provides information on the actual performance of the existing system. This information can be used to forecast the impact of system changes such as a reconfiguration of the hardware, or a change in an operating system routine. Monitoring is also useful in isolating bottlenecks in the flow of jobs through the system.

Of prime importance to the efficiency of any complex computer system or network is the performance of the operating system. Often the typical job stream of a

computer system will change or evolve over a period of time. The operating system must be periodically tuned to perform efficiently in this new environment.

Synthetic programs are rapidly becoming a valuable tool in all of these performance evaluation studies. A synthetic program is defined by Kernighan (10) as 'a highly parameterized program which uses precisely specified amounts of computing resources, but which does no useful work'.

This paper looks at synthetic programs as they have been applied in drive workload construction. Synthetic program models that have been used in performance studies on traditional computer systems are described in detail. Then new synthetic program models are developed for use in workload construction on both a back-end data base management network and on a fully distributed computer network. These models are then extended so that by using synthetic programs we can simulate various alternative I/O devices, local operating systems and machines in a particular network node. This will enable us to easily simulate one computer configuration with another similar configuration.

Additional work still must be done on the actual measurement necessary to calibrate these synthetic program models and to test the various model assumptions. After this is completed actual experimentation must be conducted to determine the adequacy of these models on predicting performance in computer networks. This report only deals

with the development of these synthetic models. The measurement and experimentation need to be conducted in future projects.

Drive Workload Construction Methods

In this chapter we will precisely define system workloads and discuss the various methods that have been used to construct drive workloads in past performance studies. This paper defines the system workload as any collection of individual jobs and data that can be processed during a specified period of time. Often it is of interest to measure the performance of an operating system under a variety of workloads, many of which are far different from the typical system workload. A basic problem in this type of analysis is in assembling the desired workloads with which to exercise the system. Once assembled, these job streams are called drive workloads. Drive workloads are needed for every evaluation technique be it an analytical model, a simulation model, or a hardware/software monitor measurement experiment. Much of the remainder of this paper deals with the construction of drive workloads.

2.1 Sampling Procedures

Several methods for constructing drive workloads appear in the literature. Perhaps the simplest method is that of using the actual job stream over some time interval as the

drive workload. This method can give the analyst a good idea of how the current or proposed system performs under the typical workload. However, it can provide no information for predicting system performance under non-typical workloads, such as extreme CPU or I/O bound situations. There is also the possibility that the time intervals selected are not truly representative of the typical system workload. To help alleviate this problem it has been recommended that instead of collecting all jobs over a continuous interval as a drive workload, a sampling procedure be employed to randomly select individual jobs over a much longer time period such as a month. This method was employed at the Iowa State University Computing Center by Shoppe (12) to obtain a representative drive workload. It appears to be a good procedure to obtain a drive workload representative of the typical real system workload, provided the analyst is able to sample from every job on the system.

2.2 Benchmarking

One of the more popular methods of constructing a drive workload is through the use of application benchmarks. The idea of application benchmarks is really just applying a stratified sampling technique to Shoppe's procedure (9). The jobs in the users real workload are divided into classes or categories. The classes or categories are chosen to reflect different types of processing or different

Division of Jobs Into Application Classes

Class 1. - FORTRAN Coded Engineering Problems

Program	Topic
1.	Volume of Spheres of Changing Radius.
2.	Satellite Tracking.

Class 2. - FORTRAN Coded Mathematical Problems

Program	Topic
1.	Multiplication of Arrays.
2.	Probability Calculations.
3.	Matrix Inversions.
4.	Statistical Routines.

Class 3. - COBOL Coded Business Problems

Program	Topic
1.	File Update.
2.	Sales Forecasting.
3.	Inventory Control.

Figure 2.1

applications. A typical division into application classes is shown in Figure 2.1. There may be one or many programs in a class. Joslin (9) recommends that each class should contain at least ten percent of the total workload time and no class should contain more than fifty percent of the total workload time. If the drive workload is to be representative of the real workload, typical jobs can be chosen in each class in proportion to the total number of jobs in that class. The chosen jobs are called benchmark programs. The drive workload constructed from these benchmarks is called a real drive workload.

Now any workload of interest can be described as a probability distribution over some predefined set of classes. By varying the number of programs selected in each class it is theoretically possible to construct a drive workload representative of any workload of interest. Under ideal conditions benchmarking can be an inexpensive and accurate way to construct drive workloads. Unfortunately these ideal conditions often do not exist. Since workload conditions of interest are often described over predefined categories, there is always the possibility that some of these categories may be empty after the sample is collected. This means any hypothetical workload giving nonzero probability to these empty classes will be impossible to construct. Also the sampling procedure requires that every job in each class be equally likely to be selected. Usually this is not the real situation. Users of most service

facilities are reluctant to supply programs, data bases and operating instructions. Security considerations may prevent many jobs from being considered. These problems can bias the drive workload. Benchmarking also presents other problems. New changes in the operating system being evaluated make it difficult to keep complex jobs viable. Since actual programs are being used, the data bases referenced by these jobs must be kept intact. Similarly, routines to handle disk and tape operations must also be included. This can be very expensive and impractical. As previously stated the major problem with benchmarks is flexibility. Many workloads of interest are just impossible to construct since the characteristics of each available job are fixed.

2.3 Synthetic Programs

To overcome many of the difficulties encountered with benchmarks, synthetic programs have recently been utilized in performance evaluation studies. To understand synthetic programs one must think of a computer system as a collection of services or resources upon which the workload places a demand. The demands on these resources or services can be considered the characteristics of the corresponding jobs in the workload. Unlike a benchmark a synthetic program is not a member of the actual job stream. However, a synthetic program can have the characteristics of a program from the

job stream. Using the same stratified sampling procedure which was described for benchmarks, a synthetic drive workload can be constructed that matches the characteristics of the real drive workload. Each benchmark program in the real drive workload need only be replaced by a dummy synthetic program matching the characteristics of the corresponding real benchmark. The synthetic workload will be a valid representation of the real system workload only in terms of the characteristics to which the synthetic model is sensitive. Consequently, care must be taken in defining and choosing proper workload characteristics. The flexibility problem of obtaining real programs to match an nontypical workload is solved through the use of synthetic programs, since only the job characteristics are needed instead of actual jcbs themselves. Similarly, since a synthetic program does not need real data, the problem of maintaining massive data bases is also eliminated. Overall, synthetic programs are probably the easiest and most economical performance evaluation technique for workload construction. The next chapter describes actual synthetic program construction models.

Synthetic Program Construction Models

In the last chapter it was stated that a synthetic program was a dummy program that matched the characteristics of some corresponding real program in a real drive workload. In this chapter we will precisely define what is meant by the term characteristics.

3.1 Basic Model Elements

As previously stated, a computer system can be thought of as a collection of resources upon which each program in the workload places a demand. Any job in the workload is characterized by the magnitude of its demands on the system resources. Examples of resources common to most computer systems and their corresponding demands are illustrated in Figure 3.1.

We can consider the demands on these system resources as the values of the characteristic variables of a program in a real drive workload. So having a synthetic program match the characteristics of a program in the real drive workload simply means that the synthetic program has the same values for its characteristic variables as the corresponding real program. When we define a computer

System Resources and Corresponding Demands

	Resource	Demand
1.	Central processor.	CPU seconds.
2.	I/O channels.	Number of I/O activities initiated.
3.	Core memory.	Amount of core allocated.
4.	Unit record devices.	Number of cards punched, read, and the number of lines printed.
5.	Auxiliary storage.	Number of data transfers to and from each type of auxiliary storage device, and the amount of data transfered.
6.	Memory bound instruction repertoire.	The number of each class of memory bound commands executed. Classified by accesses.
7.	Processor bound instruction repertoire.	The number of each class of processor bound instructions executed. Classified by operand type.

Figure 3.1

system as a collection of resources, then the classification of a program in terms of what it does is meaningless. It is theoretically possible to model, in terms of these characteristic variables, any type of processing with any other type. For example, a matrix inversion program and a sort-merge can look identical to the system in terms of these characteristic variables.

3.2 The Service Demand Approach

Given any real program in a system workload, values can easily be obtained for the first four characteristic variables in the list in Figure 3.1. These values are usually obtained through the system accounting data, or through the use of hardware/software monitors (3,10,16). However, for programs written in higher level languages measurements for the last three characteristic variables may be very difficult to obtain. To solve this problem Campbell (3) recommends using as characteristic variables the computing and I/O services demanded by a job at the language level. To illustrate this concept consider the following FORTRAN statement:

```
A = B(K)/C + Z(I,J) + D.
```

This statement can be considered to demand the following services.

1 Index calculation of a two dimensional array.

1 Index calculation of a one dimensional array.

Service to Resource Mapping

Figure 3.2

1 Real division.

1 Real assignment.

2 Real adds.

These services are then mapped by the compiler into corresponding resource demands. This mapping is depicted in Figure 3.2. Similarly, a list of typical services that can be measured at the language level are given in Figure 3.3. For any given real program, once we have measured these services, a synthetic program can be constructed that will demand the same service profile. An example of such a synthetic program construction is given in Appendix A.

Notice that the final synthetic program in this example (Appendix A) is simply a reordering of the original real example program. In order to make this type of reordering valid for all synthetic program construction methods, we will make the following assumption. It will be assumed that only the total resource or service demands of a given program are important and that the order of these resource or service demands on the various resources or services is irrelevant (3). This service demand approach is a slight improvement over the general resource demand approach for simple high level programs in a typical system workload. However, for complex programs this method can be very difficult and time consuming. In constructing a drive workload representative of the normal system workload, this method requires actual access to the real workload programs. This means the same problems encountered with benchmarking,

Typical Service at the Language Level

Type of Services Provided	Important Characteristics
1. Arithmetic and logical operations.	Type (i.e. integer, real or complex), operation (i.e. =,+,-,*,/).
2. Procedure calls (function or subroutines).	Parameters passed.
3. Array indexing.	Number of indexes.
4. Calls to run time routines.	Explicit (i.e. sqrt,rand, float), implicit (data conversion).
5. I/O	Device type, physical characteristics of the file. (i.e. record and block size, access method and organization).

Figure 3.3

such as security also apply to this construction method. Atypical workloads however, can be described as probability distributions over the resource or service space of a computer installation. With the help of the reordering assumption, this makes the construction of non-typical system workloads possible. Overall this service demand model has too much detail to be of practical value. Several less detailed models have been developed in the literature.

3.3 The Simple FORTRAN Loop Model

One of the simplest synthetic program construction models was developed by Kernighan and Hamilton (10) of Bell Laboratories. This model uses only three simple characteristic variables (denoted by X1,X2,X3). They are:

X1. Total job CPU time.

X2. Total job I/O time.

X3. Total job memory requirement.

On any sizable computer system, values for these three characteristic variables are usually available for every job by utilizing the system accounting data. If there is no system accounting data, or if it is inaccessible, the values for these characteristic variables for any job in the system can be obtained through simple hardware/software monitors while the job is running. In order to simplify the synthetic job construction, Kernighan and Hamilton made the following assumptions:

1. A job uses fixed core requirements.
2. I/O goes to only one unspecified device in unspecified amounts.
3. CPU and I/O are distributed throughout the job in an unspecified way, and are overlapped or not as the generated code and operating system dictate.

A simple synthetic program can be designed to meet these specifications. The program consists only of two nested loops inside a larger control loop. Figure 3.4 helps illustrate the structure of this synthetic program. In this model CPU time is consumed in the first inner loop (CPU loop) by any null computation like summing numbers. Similarly, I/O time is consumed in the second inner loop (I/O loop) by reading and writing a scratch file. There are two basic ways to match the total CPU and I/O times of a real program to the corresponding CPU and I/O times of this synthetic program. The first method requires that the operating system be cooperative about giving running programs access to the resource demands the program has used so far. With this type of an operating system, this synthetic program can be made self-timing. This is done by checking with the system to match the prespecified CPU and I/O time ratios for the inner loops and similarly checking for a total time requirement in the outer loop. When using this self-timing synthetic program construction method, it must be assumed that the system calls, for determining resource utilization, take a negligible amount of time in

The FORTRAN Loop Synthetic Program

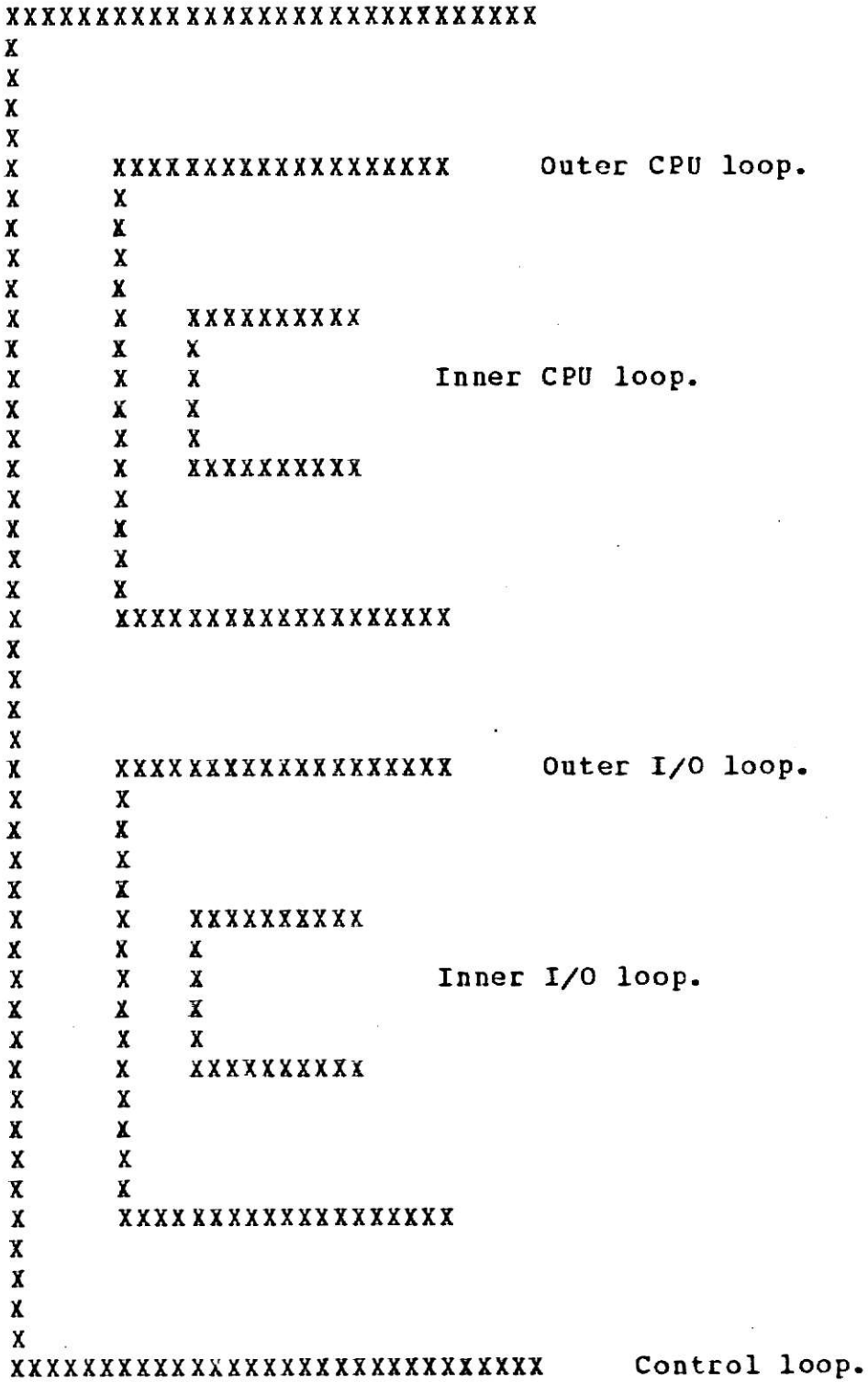


Figure 3.4

relation to the total job time. If this assumption is justified, very accurate models can be obtained from this self-timing construction method.

Another approach is to calibrate both inner loops (I/O and CPU) so that they consume the same amount of time per replication. The specified CPU and I/O time requirements can then be met by setting the loop iteration counts at construction time. This nonself-timing construction method has the advantage that there is no unaccounted for time due to system timing calls. In an environment in which the system does not give us access to the resource utilization of running jobs, this nonself-timing approach may be the only alternative for this type of synthetic program. In a non-multiprogramming system this method can also give very exact results. However, since I/O time can vary over a considerable range from run to run in a multiprogramming system, this nonself-timing construction method can lead to erroneous results. Also calibrating the two inner loops so they run for the same amount of time per replication can require a great deal of work.

In the summer of 1975 the author wrote such a nonself-timing synthetic program in FORTRAN. This program is given in Appendix B. The structure was similar to that shown in Figure 3.4. The inner CPU loop consisted of adding and multiplying the elements of an arbitrary integer array. Similarly, the inner I/O loop consisted of reading and writing N words to a scratch file. It was hoped to

calibrate the inner I/O and CPU loops so that the outer loops executed one iteration in one tenth of a second on an IBM model 370-158. A trial and error search was employed to calibrate both loops. On the CPU loop the inner loop iteration count, as well as the number of operations per iteration, was varied over a wide range. Similarly, on the I/O loop the number of read/writes per inner loop iteration, as well as the number of words transfered per read/write (N) were varied. The cost of analyzing sufficiently replicated combinations of these variables became unduly prohibitive. Consequently, this search was terminated before acceptable values for these variables were obtained. From this experience the author strongly recommends that in constructing synthetic programs of the nonself-timing variety, the programs be written in assembly language. By using assembly language, the loops can be calibrated from the manufacturer's published timing data rather than from an experiment requiring repeated runs.

One simple refinement can be added to this simple synthetic model. A separate I/O loop can be included for each type of I/O device. Using this refined model Kernighan and Hamilton constructed such a self-timing synthetic program in the FORTRAN programming language. They used this one program to construct a synthetic drive workload. Each copy of this synthetic program in the synthetic drive workload was calibrated to match a corresponding program in a real drive workload. The synthetic drive workload matched

the real drive workload within ten percent for all measurements taken. This is an extremely good match for so simple a model.

3.4 The File Update Model

A discussion of synthetic program construction methods would not be complete without a section on the original Buchholtz synthetic program. Over half of the published work the author referenced for synthetic program construction made use of some version of this program. The original version written by W. Buchholtz (2) first appeared in 1969. The program was written to be used as a tool in comparing the performance of existing computer systems. The measure of performance used was the reciprocal of the run time of this highly parameterized synthetic program. The program itself is a simplified file maintance procedure. Basically the program works as follows: The input is a file of detail records. A file of master records is searched for every detail record. Upon finding the corresponding master record a compute kernel is executed a prespecified number of times. Then both the updated master and detail records are written out. This program is cyclic in that its running time is directly proportional to the number of detail records processed. Since a compute kernel of variable length and replications is included, it is possible to simulate both input/output and processor bound situations,

or any combination thereof. The compute kernel in this program is interesting in that it is self-checking. This can be an important factor when using this program in the evaluation of a new operating system. It helps in validating the programming of both the program and the system. Basically the self-checking compute kernel works as follows: The programmer has flexibility in choosing the length and replications for the kernel by the choice of two positive integers N and A. A table is initialized to contain the first N^3 integers starting with the integer A. the kernel can be described as follows:

Denote the table entries by T_I, I=1, ... N

where: $A = T_1$ and $T_{I+1} = T_I + 1$.

The kernel consists of computing S, where:

$$S = \sum_{K=1}^{N} T_{J_K}$$

and $J_K = J_{K-1} + 6U_{K-1} + 1$

with $U_K = U_{K-1} + K$ and $J_0 = U_0 = 0$.

Now it can be shown by induction that:

$J_K = K^3$ and similarly that:

$$\sum_{K=1}^{N} K^3 = \left(\sum_{K=1}^{N} K\right)^2 = (N(N+1)/2)^2 .$$

So we have:

$$S = \sum_{K=1}^{N} T_{J_K} = \sum_{K=1}^{N} (A-1+K^3) = N(A-1) + \sum_{K=1}^{N} K^3 =$$

$$N(A-1) + (N(N+1)/2)^2 .$$

This means the arithemetic can be verified by computing B where:

$$B = (1/N)(S-(N(N+1)/2)^2) + 1$$

and checking for A = B.

A slightly modified version of the Buchholtz synthetic program written by the author appears in Appendix C. The original program is highly parameterized in that the analyst can vary the values of the following program parameters to obtain the proper I/O and processor time configurations:

P1. The number of master records.

P2. The number of detail records.

P3. The number of times the compute kernel is executed per match. (P3 is denoted by N in the above discussion).

P4. The number of times the file update process is repeated.

P5. The blocksize of the I/O buffers.

P6. The size of the records.

P7. The starting integer value A.

The major problem when using this synthetic program is in mapping specified resource demands into values for these seven program parameters. This mapping is given by the P' map in Figure 3.5. Now the F map in this Figure can easily be determined for any point in the synthetic program parameter space. This is done by running the synthetic program under the parameter values of the chosen point. The problem stated above is that of finding the inverse map F'.

Wood and Forman (16) used a slight variation of this synthetic program to construct a synthetic workload at the Mitre Corporation. They seemed to have used a trial and error search to find the synthetic program parameter

Parameter Space to Resource Space Map

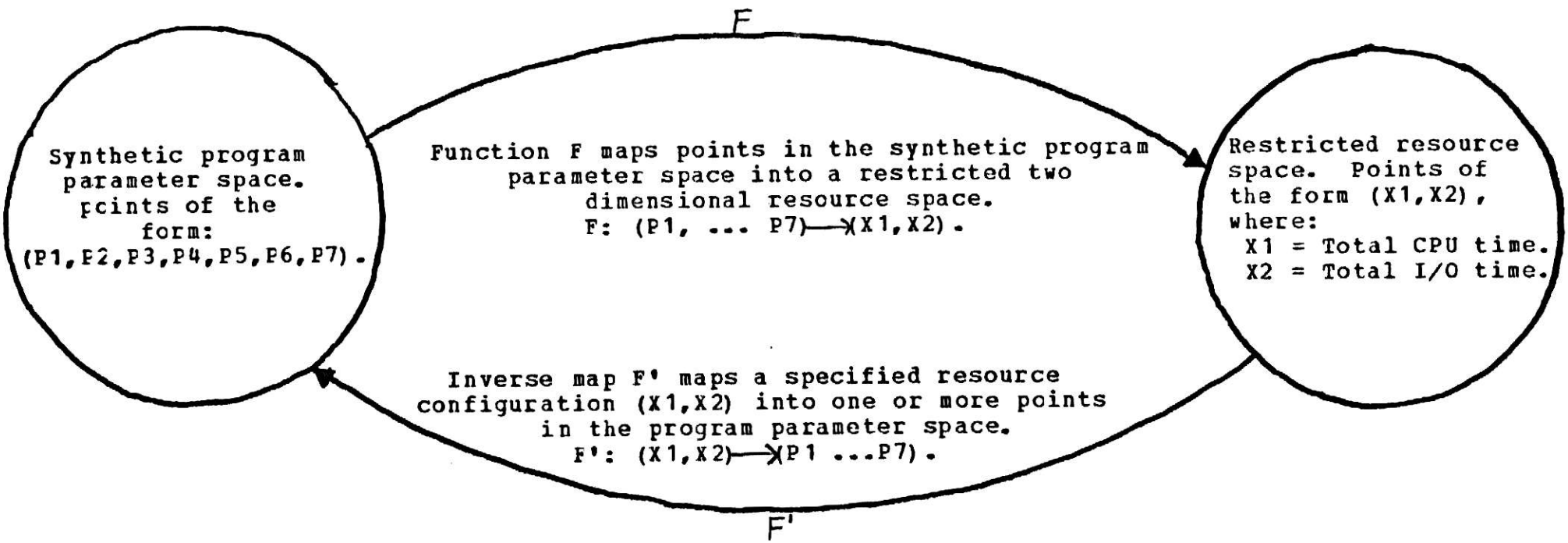


Figure 3.5

settings to match the corresponding real jobs. Synthetic workload construction using this program proved to be a fairly good tool in their throughput study. However, for general use the trail and error search seems to be a massive drawback.

3.5 An Extension to the File Update Model

Recent work by Sreenivasan and Kleinman (13) also of the Mitre Corporation, has resulted in a statistical procedure which can be used to determine these parameters. First the definition of the characteristic variables must be revised. In this study it was assumed that all jobs execute in a given partition size. This means there is no characteristic variable for total job space as before. The new characteristic variables are given by:

X1. Total job time.

X2. Number of execute channel programs (EXCP) in job.

Also we assume that P7 = A = 1. Then the functional dependency of X1 and X2 on P1,P2,P3,P4,P5 and P6 can be expressed as:

$$(12)\quad X1 = K1 + (K2)(P4) + K3(P1+P2) + (K4)(P4)(P1+P2) + (K5)(P2)(P3)(P4).$$

$$(13)\quad X2 = 2P4 + (2P4+1)((P1)(P6)/P5 + (P2)(P6)/P5).$$

The first equation is obtained by noting that X1 can be considered proportional to the total number of instructions executed. Therefore X1 seems to be dependent on the

following four variables.

1. The number of times the initial code is executed. This is denoted by P4.
2. The number of records initially created. This is denoted by P1+P2.
3. The total number of reads and writes. This is denoted by P4(P1+P2).
4. The total number of kernel executions. This is denoted by (P2)(P3)(P4).

The simple linear regression model of equation 12 readily lends itself to this structure. Now to see equation 13 we must define.

L1. Number of blocks written per update. This is given by P6(P1+P2)/P5.

L2. Number of blocks read per update. This is given by P6(P1+P2)/P5.

L3. Number of EXCP for closing files. This is given by 2.

L4. Number of blocks written for initial creation. This is given by P6(P1+P2)/P5.

Recalling that P4 is the number of times the update process is repeated, it is easy to see that.

$$X2 = 2P4 + P4(L1+L2) + L4.$$

Since L1 = L2 = L4 = P6(P1+P2)/P5 we have:

$$X2 = 2P4 + L1(2P4+1) = 2P4 + (2P4+1)(P6(P1+P2)/P5).$$

This equation is equivalent to equation 13. Using a least squares regression fit on experimentally derived data, the

following estimates were obtained for equation 12:

$$K1 = .28 \qquad K4 = .00138$$

$$K2 = .5 \qquad K5 = .00024$$

$$K3 = .00092$$

So for any given X1,X2 in the resource space we have:

$$X1 = .28 + .5P4 + .00092(P1+P2) + .00138P4(P1+P2) + .00024(P2)(P3)(P4).$$

$$X2 = 2P4 + (2P4+1)(P6(P1+P2)/P5).$$

This system of equations can be solved for integer values of P1,P2,P3,P4,P5 and P6 through the use of the following iterative technique: First integral values are chosen for P1,P2,P5 and P6. Then equation (13) is solved for P4. If the estimate of P4 (denoted by $\hat{P4}$) is positive, it is rounded to the nearest integer. If $\hat{P4}$ is negative, new initial values are chosen for P1,P2,P5 and P6. Now using the values P1,P2,$\hat{P4}$,P5 and P6, equation 13 is solved for X2. If this estimated value for X2 is not equal to the given value for X2, then P1 and P2 are altered to make these values equal. Then all these parameter values are used in equation 12 to solve for P3. Again if $\hat{P3}$ is negative, new initial values are chosen. If $\hat{P3}$ is positive it is rounded to the nearest integer. Then X1 is estimated using these values. If this estimated value of X1 is not equal to the given value, P1,P2 are varied over a small range to make them equal. Note that P1,P2 can vary over a small range without altering the rest of equation 13. These values of P1,P2,P3,P4,P5 and P6 are then used as new starting values

and the process is repeated until these values converge. The final values of these program parameters that are derived using this technique are the values that correctly parameterize the synthetic program for the given values of X1 and X2. It should be noted that there may not be a unique solution and that several settings of the program parameters may satisfy the equations. This appears, theoretically, to be a very good method for calibrating this synthetic program to meet specified resource requirements. The main disadvantage is that although solutions for P1 through P6 always exist, there may not be a solution for which all the parameters are integers. In this case we have to be content with an approximate solution.

3.6 Conclusions

All of these synthetic models have been used for workload construction in different performance studies. Clearly the detail in the service demand model makes this model unacceptable for general use. The file update model has been the synthetic model used most often for synthetic workload construction. Several performance studies have been conducted utilizing this synthetic model for workload construction (2,10,16). The results of these studies have helped prove the validity of this model.

By far the easiest synthetic model to understand and use is the simple FORTRAN loop synthetic construction model.

Unfortunately this model has only been used in one reported performance study. Although the results from this study showed this model to be very promising, more such studies must be conducted before this FORTRAN loop model can be considered generally acceptable. The ease of synthetic program construction via this model certainly makes such validition desirable. If the results of these future studies on this model do indeed validate the model, I would strongly recommend the use of this FORTRAN loop model in synthetic workload construction. In the next chapter we look at possible synthetic program applications to computer networks.

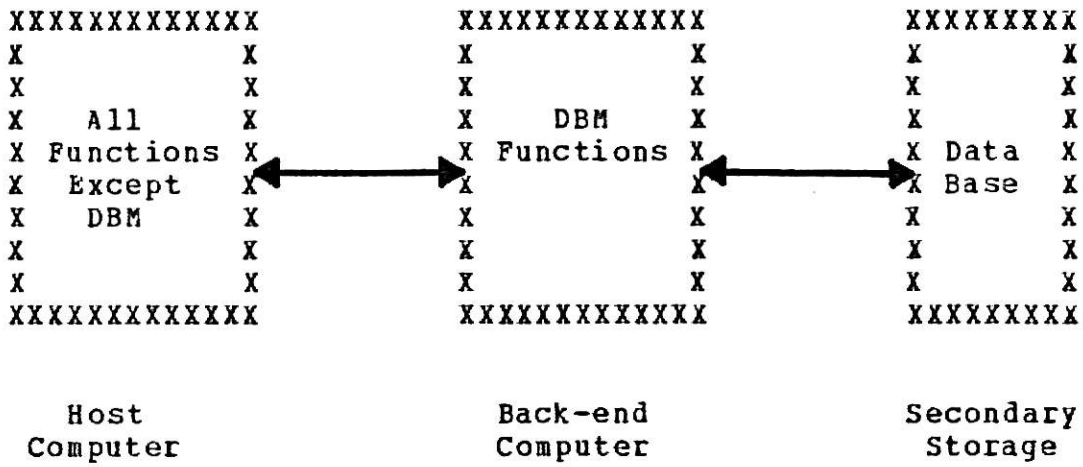
Synthetic Program Applications to Computer Networks

In this section we will describe synthetic program applications to performance studies on various computer networks. A computer network is defined as an interconnected group of two or more computers. Several significant advantages are achieved through the use of a computer network. These include increased processing power, economy through specialization and easy sharing of resources and data. Various network topologies have been proposed in the literature (14). In the remainder of this paper, we discuss synthetic program applications to network topologies currently receiving intensive study at Kansas State University.

4.1 The Back-end Data Base Management System

One of the current topics of interest in computer science, is the concept of a separate back-end computer for data base management. In a traditional system all the major software components, such as the operating system, application programs, as well as the data base management system will execute on a single large central computer.

Typical Back-end Configuration



Back-end System

Figure 4.1

This computer usually has direct access to the data base residing on secondary storage. The back-end concept replaces this traditional system with a two computer network.

In a back-end network a separate computer (usually a minicomputer) is used entirely for the data base management function and has exclusive access to the data base. This back-end computer serves as an interface between the host computer, executing application programs, and the data residing on secondary storage. A typical back-end network structure is illustrated in Figure 4.1. One of the major advantages with such a structure, is the increased efficiency that can be obtained through the specialization of both computers in the network. The front-end (host machine) is freed from the time and space consuming data base management functions. Meanwhile the back-end computer has a small set of very specialized tasks which it performs. A list of typical back-end data base management (DBM) commands is given in Figure 4.2.

Now consider the workload construction problem on such a back-end topology. A typical real drive workload of application programs to be executed on the front-end machine could be assembled via the sampling and benchmarking techniques described in Chapter 2. Exercising the network under such a real drive workload would require the inclusion of the front-end operating system as well as the back-end operating systems and data base management software. It

Back-end Commands and Data Structures

Typical Back-end Commands	Typical Data Structures
1. Find.	1. Sequential list.
2. Get.	2. Inverted list.
3. Obtain.	3. Tree.
4. Store.	4. Single linked list.
5. Modify.	5. Double linked list.
6. Insert.	
7. Remove.	

Figure 4.2

would also require the inclusion of all data bases referenced by the programs in this real drive workload. As was the case in a centralized computer system the major problem with such a procedure is a lack of flexibility. Nonstandard workloads are still next to impossible to represent by this technique. Previously the concept of synthetic programs was introduced to help overcome these difficulties. Constructing a synthetic drive workload for such a back-end configuration requires that synthetic jobs be created that match jobs in the real or hypothesized drive workload, in terms of their demands on both front-end and back-end resources.

4.2 A Synthetic Drive Workload for a Back-end Network - Model 1

The first step in creating a synthetic drive workload for a back-end network is defining appropriate characteristic variables. Suppose there are N different DBM command types supported by the back-end software. Then the following set of characteristic variables would be an appropriate starting place for this model:

X1. Total local front-end CPU time for the front-end application program. This does not include the time spent waiting for the DBM commands to complete.

X2. Total local front-end I/O time or EXCPS for

the front-end application program. This is I/O done directly on the host machine, such as card reading, printing, tape drives etc.

X3. Space requirement for the front-end application program.

Y1. The total number of executions of DBM command type 1 in this front-end application program.

Y2. The total number of executions of DBM command type 2 in this front-end application program.

.

.

YN. The total number of executions of DBM command type N in this front-end application program.

Notice that the first three characteristic variables are essentially the same resource variables that were introduced in the last section. The remaining N variables are service variables that measure the amount of intermachine communication for a given front-end application program in this back-end network. Like all service variables they must be mapped into the system resource space as shown in Figure 3.2.

In this chapter it still must be assumed that only the total resource demands are important and that the order and

relative time of the various front-end and back-end resource demands (CPU, I/O, etc.) is irrelevant. To develop a synthetic program model for this back-end network, this reordering assumption must be extended to include these DBM commands. This means it must also be assumed that only the total number of each type of DBM command is important and that the order and relative time at which a program executing on the front-end issues these DBM commands to the back-end is irrelevant. These assumptions enable us to extend the synthetic program construction models developed for a traditional system to the back-end system or any other network topology.

With the above assumptions it is relatively easy to construct a synthetic program that reflects any prespecified distribution over front-end resources and DBM command types. The synthetic program most readily extended to this is the simple FORTRAN loop program. In the discussion of this program it was recommended that a separate loop be included for each type of I/O unit on the system. Similarly, it would be a simple task to include such a loop for each type of DBM command supported. I would recommend using a self-timing version of this program, with all back-end DBM commands being executed first. This would take into account the extra CPU time inherent in the various loop overheads, and insure that all back-end DBM commands would be completed before program termination. This program structure is illustrated in Figure 4.3.

The FORTRAN Loop Model for the Back-end

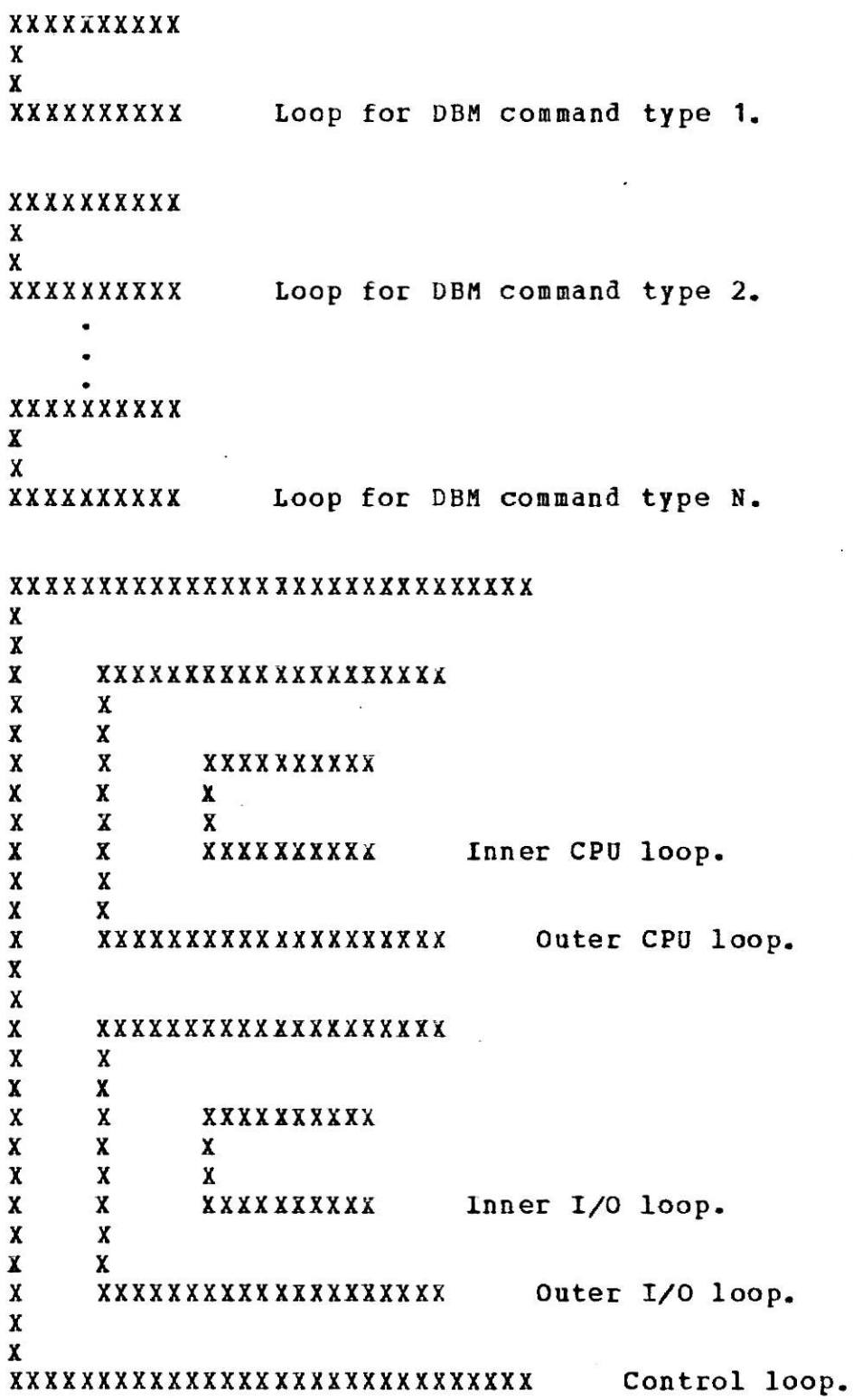


Figure 4.3

The file update synthetic program could also be used for this model. However, since the front-end machine usually does not have direct access to disk files, the temporary data sets created and accessed by this program create a problem. We could rewrite this program to insure that the update process actually does the required number of each type of DBM command. Then we could let the back-end perform the actual data base function of this update procedure. This would require a major rewriting of this synthetic program each time it is used. It would also require new calibrating equations similar to those introduced in the last section. A simpler solution can be obtained by not utilizing the back-end for the actual update procedure. This could be done by using tape files for the data sets being processed. Then the update procedure can be calibrated to run entirely on the front-end machine. The back-end DBM commands would be dummy commands that are not relevant to the actual update process. These commands could be executed in the update cycle, with an appropriate number of each command type executed per cycle.

In most previous work with synthetic programs the workloads to be modeled were defined in terms of resource demands. However, this model contains N characteristic variables at the service demand level. The appearance of these service variables in this model means that we are assuming the back-end resource utilization of any real program, can be met in a synthetic program simply by

executing the same number of every DBM command type. This is equivalent to assuming that each execution of a given back-end DBM command type uses fixed back-end resource requirements, in terms of CPU time, I/O time, etc. If we subdivide these back-end commands by the type of data structure referenced, this assumption can probably be made valid in the general case. This is because each command type could probably be subdivided into smaller and smaller classes, until the variance of resource utilization within each class becomes approximately zero.

In terms of the execution of a synthetic program this assumption is certainly acceptable. This is because the synthetic program will model N executions of a DBM command type by executing one dummy command of that type N times. So these synthetic DBM commands will be dummy commands that access garbage data files. Also each execution of the same command type should be identical in a synthetic program. Therefore, each of these DBM command types executed in a synthetic program should use fixed resource utilization requirements.

4.3 The Use and Validation of Model 1

To use this model to exercise such a back-end network, suppose we have a typical real or hypothesized drive workload that can be converted to a synthetic drive workload. First we need to obtain the values of the N+3

characteristic variables for each application program in this real drive workload. These values can be used to construct corresponding synthetic programs. These synthetic programs can then be assembled into a matching synthetic drive workload. The computer network can then be exercised by running this synthetic drive workload under the control of the front-end operating system, just as would be done with any real drive workload. As with a real drive workload each DBM command in these synthetic programs would activate the back-end to begin processing that request.

Since a synthetic workload has never been used in a performance study on a computer network, a validation experiment should be conducted on this model. Such an experiment would consist of the following steps: First it is necessary to assemble various real drive workloads that represent a wide range of front-end and back-end resource utilization. Then each of these real drive workloads should be run a prespecified number of times. While these real drive workloads are executing, measurements should be taken to determine the values of the characteristic variables (X1,X2,X3,Y1,, ... YN) for each real application program in every workload. Similarly, the front-end and back-end performance variables of interest should be measured while these real drive workloads are executing. These variables include such things as front-end/back-end computer efficiency, channel utilization, throughput and queue lengths.

The values of the measured characteristic variables are used to construct a matching synthetic program for each real application program. Once constructed these synthetic programs are assembled into corresponding synthetic drive workloads. The network is again exercised, this time using these synthetic drive workloads. Each synthetic drive workload is executed the same number of times as each real drive workload. While the synthetic drive workloads are executing, measurements are taken on the same performance variables that were monitored with the real drive workloads. Then a statistical analysis is performed to see if these performance variables behave the same under both the synthetic and real drive workloads. Since there are generally several correlated performance variables of interest, a multivariate statistical procedure should be used in this analysis. Help is readily available for the design and analysis of this statistical experiment at the Statistical Laboratory at Kansas State.

The validity of the assumption of fixed resource requirements for each DBM command type can best be checked by analysing the results from the previously described experiment. In this regard care should be taken when conducting this above experiment to insure that amoung the back-end performance variables measured are the resource utilization requirements of every DBM command execution. These measurements can then be broken down by command type. This data can then be analyzed to determine if each DBM

command type does indeed have fixed resource utilization requirements. If this assumption of fixed resource utilization does not hold for each command type, the data can then be reanalyzed to determine possible subdivisions over which this assumption is valid. It is possible that this assumption will be totally impractical no matter how we subdivide these command types. If this happens a new model must be developed, which is free of this assumption. The next few paragraphs describe such an alternative model.

4.4 A Synthetic Drive Workload for a Back-end Network - Model 2

Another approach to this workload construction problem is to define our characteristic variables entirely in terms of resource demands. Such set of characteristic variables is given by:

X1. Total local front-end CPU time for this front-end application program. Again this does not include time waiting for DBM commands to complete.

X2. Total local front-end I/O time or EXCPS for this front-end application program.

X3. Space requirement for this front-end application program.

X4. Total local back-end CPU time generated by all of the DBM requests in this front-end

application program.

X5. Total back-end I/O time or EXCPS generated by all of the DBM requests in this front-end application program.

It will probably be noticed that there is not a characteristic variable for the back-end space requirement. Now any front-end application program generates a back-end space requirement every time it executes a DBM command. However, this back-end space requirement will probably change for each DBM command type executed in a given application program. For example, the space requirement in the back-end machine for an execution of an obtain command may be far different from the back-end space requirement for an execution of a delete command. Yet both of these commands can easily be included in various real and synthetic application programs. This makes it impossible to obtain one value for a characteristic variable representing the back-end space requirement of a given front-end application program. Also such a characteristic variable is not necessary, since the back-end operating system will automatically allocate an appropriate amount of space for each type of DBM command, as it is encountered during the execution of the workload.

Suppose there are N different back-end DBM command types. As previously stated, each execution of a given DBM command type in a synthetic program uses fixed resource requirements. This fixed resource utilization can be

denoted by:

C1. Back-end local CPU time for DBM command type 1.

C2. Back-end local CPU time for DBM command type 2.

.

.

CN. Back-end local CPU time for DBM command type N.

T1. Back-end I/O time or EXCPS for DBM command type 1.

T2. Back-end I/O time or EXCPS for DBM command type 2.

.

.

TN. Back-end I/O time or EXCPS for DBM command type N.

Next we try to find a set of nonnegative integers B1,B2, ... BN such that for a given set of values for the characteristic variables (X1,X2,X3,X4,X5) representing a given application program, we have:

$$X4 = (B1)(C1) + (B2)(C2) + \ldots (BN)(CN).$$

$$X5 = (B1)(T1) + (B2)(T2) + \ldots (BN)(TN).$$

The solution for B1,B2 ... BN in the above equations can be obtained by the same iterative procedure described in Chapter 3. As in that chapter there may not be such an integer solution, and we may have to be satisfied with an approximate solution. The total back-end resource requirements are met in a front-end synthetic program by executing DBM command type I BI times, where in the synthetic program, command type I has fixed resource requirements CI,TI.

In this solution to the workload construction problem,

there is another subtle problem for which we need a different simplifying assumption. Notice that the real application program and the synthetic program representing it, will most probably execute each DBM command type a different number of times. Now there is some operating system overhead (both on the front-end and back-end) associated with each execution of a DBM command. When the synthetic program executed the same number of each command type as the real program it modeled, it was very reasonable to assume that this difference in operating system overhead between the real and synthetic programs was negligible. However, now this assumption becomes more questionable. This model requires that either we still accept this assumption of negligible operating system overhead, or that we assume that the difference in this operating system overhead will average out over the entire workload.

4.5 The Use and Validation of Model 2

This model can be used to exercise the computer network in the same manner as the previous model. As in that case a statistical experiment must be conducted to confirm the model. In this experiment we should be able to utilize some of the data available from the previous experiment. Again we need to assemble several real drive workloads, to be run on the network. While these real drive workloads are executing, measurements should be taken to determine the

values of these new characteristic variables for every application program in each workload. As before measurements must also be taken on any front-end/back-end performance variables of interest. Included in these measurements should be some measure of the operating system overhead associated with the various DBM commands. If all necessary measurements on these real drive workloads were obtained previously, in the experiment on the other model, this data can be used directly, thus eliminating the need to rerun the real drive workloads.

Next we should execute various synthetic drive workloads that execute a significant number of each type of DBM command. The purpose of such runs is to obtain information on the assumed fixed resource requirements (C1,C2, ... CN,T1,T2, ... TN), and to test the validity of this assumption. If it is available, this data could also be obtained from the measurements taken on the previous experiment. Once this is done, synthetic programs can be constructed to match the resource utilization of these real application programs in terms of these new characteristic variables. Synthetic drive workloads are then assembled and executed a prespecifed number of times on this computer network. While these synthetic drive workloads are running, measurements are taken on the performance variables of interest. These measurements should be identical to those taken while the real drive workloads were running. Then, as before, a statistical analysis could be performed to

determine if these performance variables behave the same under both the real and synthetic drive workloads. Data should also be available to test the validity of the various model assumptions.

4.6 Final Remarks on Models 1 and 2

Both of these models for this workload construction problem required the introduction of simplifying assumptions. None of these assumptions are likely to satisfy a purist. However, I am convinced that without these or similar assumptions this problem is not solvable with synthetic programs. The first major assumption of fixed resource utilization for each DBM command type is easily acceptable, especially if we consider subdividing these command types as previously described. The validity of the second major assumption on negligible supervisor overhead is going to be very dependent on the total job resource utilization as well as the total number of DBM commands issued in a given job. This assumption will most probably hold for jobs with large resource requirements and a small total number of DBM commands. However, there probably exists a break-even point where the validity of this assumption becomes questionable. It is possible this assumption will remain valid in all practical situations. It is also possible that it is hardly ever valid. The above statistical experiments can help clarify the validity of all

these assumptions.

4.7 Simulating the Back-end Machine With Synthetic Programs

Often in a back-end system it is of interest to compare the network performance under several different back-end machines, or software implementations on the same machine. For example, in a given back-end network we may currently be using a Nova minicomputer as a back-end machine and a Interdata minicomputer as the corresponding front-end machine. In this situation it might be interesting to analyze the change in network performance if both machines were Interdata minicomputers. A model will be developed in this section for using synthetic programs to make drive workloads executing on the front-end computer appear to be communicating with a different back-end. So for our above example we will use synthetic programs in this Nova back-end to make this machine appear like an Interdata back-end.

Now let us examine the software on a typical back-end data base management computer. First there is the network operating system common to both of the network machines. We will assume that the difference in the resource utilization of this network operating system between real and corresponding synthetic drive workloads is negligible. Next we have the local computer operating system and below this, we have the various executing DBM command routines. This

The Structure of a Typical Network Computer

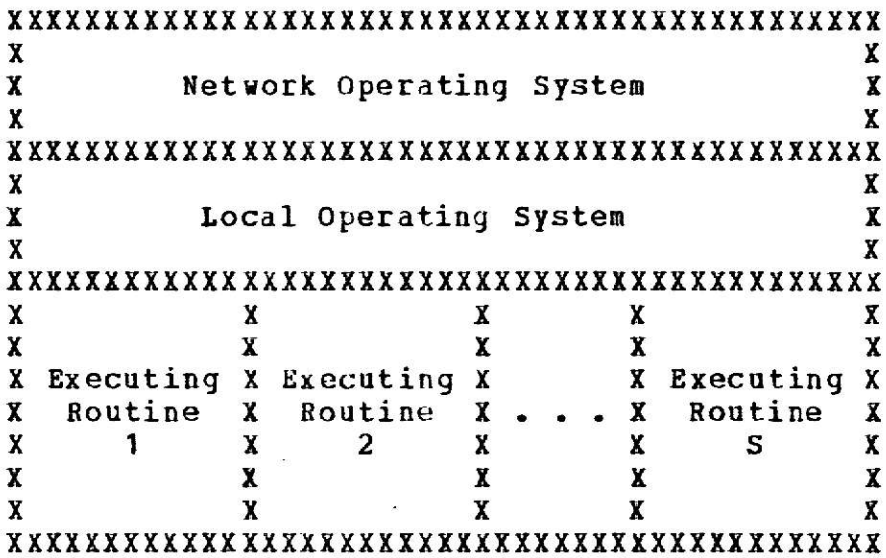


Figure 4.4

structure is illustrated in Figure 4.4.

As in Section 4.2 we will again assume that all of these DBM command routines have fixed resource utilization requirements. We will also assume that the local operating system resource utilization associated with each DBM command type is similarly fixed. These assumptions will enable us to construct synthetic programs that model the fixed resource requirements of these DBM command routines and their corresponding operating system overhead.

For simplicity, assume that all intermachine communication between the front-end and back-end machines is achieved through the familiar send/wait message routines described by Brinch Hansen (1). Further assume that any time a front-end application program issues a DBM command, the local front-end operating system suspends this process in a wait message state until an answer is received. Then the only major diffenence detected by the front-end application workload when the configuration of the back-end machine is altered, is a change in the time these front-end jobs must wait for DBM commands to complete. So we only need to determine a way to get the current back-end configuration to model the wait times of the proposed back-end configuration.

Suppose that in a given back-end system we are currently using a configuration for the back-end node that we will denote by (A). Also suppose that we are considering an alternative configuration for this back-end node that

will be denoted by (D). Now (D) could be the same machine with a different local operating system and/or DBM command software. (D) could also be a different machine with its own local operating system and DBM command software. We will assume that both the front-end computer and the (D) configuration can multiprogram a maximum of S processes and that there is sufficient space in this (D) configuration so that any combination of S of these DBM command routines will fit simultaneously.

Now suppose that we remove the back-end local operating system and DBM command software routines. We will replace this local back-end operating system with a minimal operating system consisting basically of a simple multiplexor (task switcher) and a primitive interrupt handler. This interrupt handler will only need to process interrupts generated by the receiving and sending of messages. The multiplexor will switch between at most S parameterizable synthetic routines. Each of these synthetic routines will be parameterized upon request to represent the fixed resource utilization of one of the (D) configuration's DBM command routines and corresponding local operating system overhead. The assumed fixed resource utilization of this minimal operating system must also be taken into account when parameterizing these synthetic routines. This is done by obtaining the following values for each DBM command routine in the (D) configuration software.

D1. Total CPU time for this DBM command routine

and the CPU time used in the local operating system associated with this DBM command execution.

D2. Total I/O time or EXCPS for this DBM command routine.

M1. The CPU time used in this minimal operating system in executing this synthetic DBM command routine.

M2. The I/O time or EXCPS used in the minimal operating system in executing this synthetic DBM command routine.

Then for each DBM command routine in the (D) configuration software we will calculate synthetic program parameters from the corresponding characteristic variables X1,X2 where:

X1 = D1-M1.

X2 = D2-M2.

The nonself-timing FORTRAN loop synthetic program is the easiest construction model to consider for these parameterizable synthetic routines. In this synthetic model the program parameters are the appropriate loop indexes. We will slightly alter these models so that the program parameters are stored in a common data area accesible to both the synthetic routine and the minimal operating system. Also upon completion each of these synthetic routines will send a message to the requesting front-end process and then await a new request.

4.8 Trace of a DBM Command in a Simulated Back-end Configuration

Initially all these back-end synthetic routines are suspended awaiting DBM request from the front-end application workload. Since there are S synthetic routines we can designate each of these synthetic routines to correspond to a specific front-end process number (1,2, ... S). Now suppose front-end process I issues a DBM command. The front-end operating system will send a message to the back-end configuration and suspend process I until an answer is received. The text of this message need only contain the sending process number (I) and the DBM command type. When the message is detected by the network operating system, an interrupt occurs. The interrupt handler then starts execution of a simple routine that will copy the program parameters for this type of DBM command from a preset protected data area into the common program parameter data area for synthetic routine I. This synthetic routine is then marked ready and control passes to the multiplexor. The multiplexor then controls the execution of the ready synthetic routines. When synthetic routine I completes execution it sends a return message to front-end process I. This send messsage command will cause another interrupt. The network operating system will transmit this return message. Then synthetic routine I is marked inactive and control returns to the multiplexor. Since we have a

separate synthetic routine for each front-end process, we need not worry about one DBM request resetting the parameters of a ready synthetic routine.

4.9 I/O in a Synthetic Routine

In the last two sections we have purposely ignored the consideration of I/O in our back-end synthetic routines. Since we have removed our local back-end operating system, these synthetic routines can not issue the usual operating system controlled I/O commands (read, write etc.). Also the I/O devices on our simulated (D) configuration may be far different from the actual devices on our existing (A) configuration.

Recall that these synthetic routines will have a separate I/O loop for every device supported on the (D) configuration. Instead of issuing actual I/O commands in these loops our synthetic programs will call an appropriate I/O device routine. This device routine will calculate the amount of time an incoming I/O request should take on the device being modeled. The variable amount of time required for head movements on direct access devices can be calculated in these device routines by either a random number generator or some prespecified distribution of head movements.

After calculating this required time, this device routine will send a message to one of S micro processors.

Each of these micro processors will contain a simple loop synthetic program that can be parameterized to execute for a specified amount of time. The text of this message need only contain the synthetic program parameters (probably a single loop index) necessary to correctly parameterize this micro synthetic program. When this micro synthetic program has completed, a return message is sent to the corresponding synthetic routine. Both of these messages will cause interrupts that basically will change the appropriate synthetic routine from a ready to a wait state or from a wait to a ready state respectively. It will probably be possible to use the network operating system for the transmission of these messages.

Notice that with S separate micro processors we can have a separate micro for each synthetic routine. This will solve any problems that could occur by two device routines trying to use the same micro simultaneously. Similarly, if these device routines are not reentrant we could have a separate set of device routines for each synthetic routine.

Now the actual time a synthetic routine is delayed for an I/O request can be partitioned into the following three parts:

1. Running time of the device routine.
2. Running time of the micro processor.
3. Minimal operating system overhead associated with completing this request.

We wish these three parts to sum to the correct I/O delay.

We will assume that the running time for a given device routine and the corresponding minimal operating system overhead is fixed. Then the running time of the micro processor is calculated by subtracting this fixed time (1+3) from the I/O request time. This means we can not simulate any I/O request that has a delay time less than the sum of the fixed device routine time and the corresponding minimal operating system overhead. In most practical situations this restriction should not be a problem.

4.10 The Use and Validation of the Synthetic Back-end Model

The use of this model in a performance study on a back-end network is slightly more involved than the use of the previous models. This is because we must alter the actual back-end configuration. The first step in this alteration will be to develop the multiplexor and interrupt handler for the minimal operating system. Then the actual synthetic routines must be constructed from the construction models discussed previously. Once developed this basic structure will remain the same for the simulation of any back-end configuration on the existing machine. The synthetic program parameters corresponding to the DBM command routines and the device routines must be changed for each back-end configuration we wish to simulate.

Once we have our back-end correctly structured, we are

ready to construct a front-end synthetic application workload. As with Model 1, the first step will be to define the jobs in our real or hypothesized workload in terms of the characteristic variables of Section 4.2. We then construct synthetic programs from the values of these characteristic variables. These synthetic programs can be assembled into a synthetic drive workload. The computer network is then exercised by running the synthetic drive workload under the control of the front-end operating system.

When the programs in our synthetic workloads issue DBM commands, the back-end configuration will process these requests as described in Section 4.8. Notice that the back-end will not return meaningful results for these DBM requests. This will not affect our front-end synthetic programs since they do not depend on the results from these DBM requests. However, if a real workload were driving this synthetic back-end, the effect of returning nonmeaningful results for the DBM requests would not be predictable. Consequently, a synthetic drive workload must always be utilized on the front-end when using this model.

Before accepting this model a validation experiment should be conducted and analyzed. This experiment should consist of the same basic steps described in detail in Section 4.3. Again several real drive workloads are run on a given back-end network. Then matching synthetic drive workloads are created. A different back-end machine is then

connected to the same front-end machine that was utilized with the real drive workloads. Then the local operating system and DBM command routines on the original back-end configuration are simulated on this new back-end machine as described in Sections 4.7 - 4.9. The synthetic drive workloads are then used to exercise this simulated back-end network. Each synthetic drive workload is executed the same number of times as each real drive workload. Again the performance variables of interest are monitored with both the synthetic and real drive workloads. A statistical analysis is then employed to determine if these performance variables behave the same in both of these situations.

4.11 Fully Distributed Network

The most general network topology appearing in the literature is the fully distributed network. This type of network topology can consist of any number of computers. A simple three computer fully distributed network is illustrated in Figure 4.5. In the most general case every computer in such a network executes a general purpose application workload. Also each network machine can transfer some of its processing (programs or special functions) to any other network machine. This helps keep the total network processing balanced among the various network computers. In addition all data bases are shared among the network machines. It is also possible to have

Distributed Network Structure

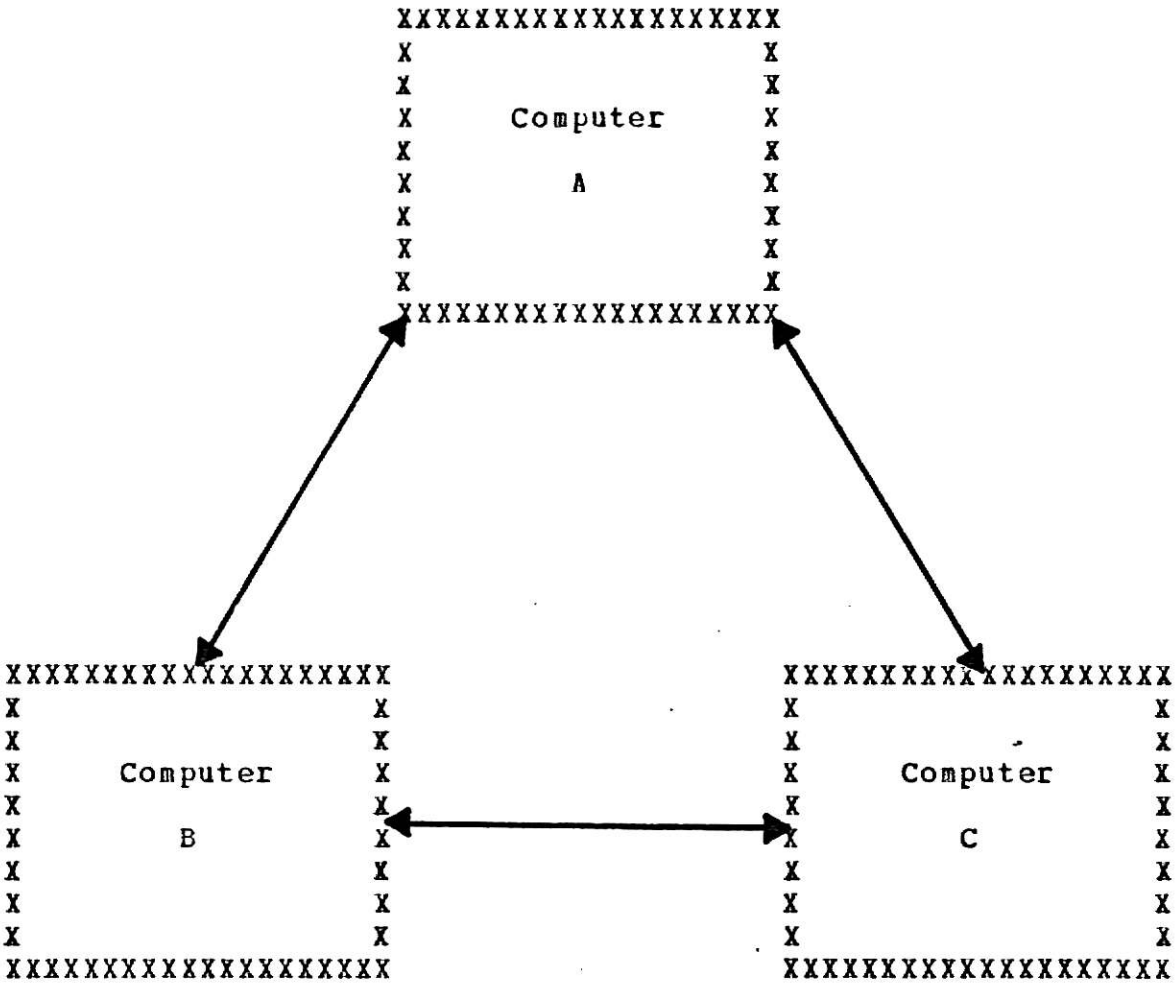


Figure 4.5

some of the network computers specialized to perform certain functions for the remaining network machines. For example, one of the machines could be a back-end data base management computer for the entire network. Overall, this fully distributed network structure allows a great deal of flexibility and has tremendous potential advantages in increased processing power. We will now extend some of the results derived for the back-end network to this more general fully distributed network. For simplicity, all results derived in the following discussion will be based on a three computer fully distributed topology as illustrated in Figure 4.5.

Now consider the workload construction problem on a fully distributed network. A typical real drive workload of application programs to be executed on the total network can be assembled via the sampling and benchmarking techniques described in Chapter 2. Since in the most general case there are three machines executing application workloads, the total system workload can be partitioned as follows:

1. Application programs originating on machine A.
2. Application programs originating on machine B.
3. Application programs originating on machine C.

Then in any performance study these three workloads are simultaneous input to the appropriate machines. Exercising the network under such real drive workloads would require the inclusion of all network and local operating systems and the inclusion of any special service routines that certain

network machines may perform for the remaining network machines. Also required would be the data bases referenced by the application programs in these real drive workloads. As with the previous cases the major problem with this procedure is a lack of flexibility. Nonstandard workloads are very complicated to construct via this technique. Previously, the concept of synthetic programs was used to help overcome this difficulity. Constructing a synthetic drive workload for this fully distributed network requires that synthetic jobs be created that match jobs in the real or hypothesized drive workload in terms of their demands on the resources of each network computer.

4.12 A Synthetic Drive Workload for a Fully Distributed Network

We will now extend the back-end model of Section 4.2 to a fully distributed network. Suppose that in our three computer fully distributed network, computer A can perform N specialized tasks or special service functions for the total network. Similarly, assume computers B and C can perform M and K such tasks or special service functions respectively. The DBM commands supported in a back-end machine are examples of special service functions. Then for a given application program, the following set of characteristic variables seem appropriate.

X1. Total local CPU time for the application

program. This is the CPU time this application program would take if it executed to completion on the computer upon which it originated. This does not include time spent waiting on the internal and external special service functions to complete.

X2. Total local I/O time or EXCPS for the application program. This is the I/O or EXCPS that this application program would take if it executed to completion on the computer upon which it originated. This does not include I/O related to the internal and external special service functions.

X3. Space requirement for this application program on the computer upon which it originated.

Y1. The total number of executions of special service function 1 on computer B that are in this application program.

Y2. The total number of executions of special service function 2 on computer B that are in this application program.

.

.

YM. The total number of executions of special

service function M on computer B that are in this application program.

Z1. The total number of executions of special service function 1 on computer C that are in this application program.

Z2. The total number of executions of special service function 2 on computer C that are in this application program.

.

.

ZK. The total number of executions of special service function K on computer C that are in this application program.

W1. The total number of executions of special service function 1 on computer A that are in this application program.

W2. The total number of executions of special service function 2 on computer A that are in this application program.

.

.

WN. The total number of executions of sprcial service function N on computer A that are in this application program.

Notice that this model has only three resource variables (X1,X2,X3). The remaining N + M + K variables are service variables that measure some of the intermachine

communication. Like all service variables they must be mapped into the system resource space as illustrated in Figure 3.2.

This model does not represent the intermachine communication generated by the swapping of entire application programs between network computers. Now this program swapping is controlled by the various operating systems resident in the network machines. So for this model we are assuming that program swapping generated by the real and synthetic drive workloads is statistically equivalent. In addition to this assumption we still need all the reordering assumptions that were introduced in the corresponding back-end case. Recall that these assumptions basically stated that the order and relative time of both the resource demands and special service functions is irrelevant. Since we are again using service variables we are assuming that the special service function types use fixed resource utilization requirements. This assumption assures that by modeling the total number of each special service function we generate the desired resource utilization due to these special service functions. To make this assumption more acceptable we could subdivide these special service function types as was recommended for the DBM command types.

Using this assumption it is relatively easy to construct a synthetic program that reflects any prespecified distribution over the host computer's resource space and the

total network special service functions. These synthetic program construction models are identical to those described for the back-end case in Section 4.2

Once an appropriate synthetic workload is assembled this fully distributed network is exercised under this synthetic workload simply by running the three synthetic application workloads on their respective machines under the control of the various local and network operating systems. As with all previous results this model and the assumptions therin should be validated through a statistical experiment. This experiment should follow the same basic steps described in Section 4.3.

As with the back-end network the solution presented here for the fully distributed network required the introduction of simplifying assumptions. The assumption of statistical equivalence between the operating system generated program swapping of real and synthetic programs seems fairly resonable. If this assumption does not hold we could have a real problem. This is because there is no way to control this program swapping without changing part of the actual operating system in each affected machines. The question of how these operating systems should be altered cannot be answered until we know how this program swapping differs between the real and synthetic workloads. The assumption of fixed resource utilization requirements for the special service functions is the most questionable. I firmly believe that this assumption can be justified over

some subdivision of these commands. If this does happen the above model will probably be adequate. However, if this assumption is invalid, a new model must be developed free of this assumption of fixed resource utilization requirements. One possibility for a new model would be a simple extension to Model 2 of Section 4.4. This extension is very straightforward and if it is needed, it should follow immediately from the material already contained in this report.

4.13 Using Synthetic Programs to Simulate a Node in a Fully Distributed Network

In a performance study on a fully distributed network it is of interest to compare network performance under a variety of interchangable computers and/or local operating systems for a particular network node. A model will be developed in this section for using synthetic programs in one of the network machines to make the configuration of this machine appear different to the drive workloads on the remaining network machines.

Now let us examine the software on a typical network computer. First there is the network operating system common to all the network machines. We will again assume that the difference in the resource utilization of this network operating system between real and matching synthetic workloads is negligible. Next we have the local computer

operating system and below this, the various executing application programs and special service routines.

As in Section 4.7 we will assume that all these special service functions have fixed resource utilization requirements and fixed local operating system overhead. We will also assume that the resource utilization of the local operating system associated with a given application program is similarly fixed. These assumptions will enable us to construct synthetic programs that model the resource utilization requirements of tasks executing on a given network machine.

Suppose that in a given three - computer fully distributed network we are currently using a configuration for the 'A' node that we will denote by (A). Also suppose that we are considering an alternative configuration for this 'A' node that we will denote by (D). In the discussion that follows we will assume that the (D) configuration can multiprogram a maximum of S tasks.

As before we will remove the local (A) operating system and resident special service routines. We will again replace this local operating system with a minimal operating system consisting of a simple multiplexor and a primitive interrupt handler. This interrupt handler will only need to process interrupts generated by the receiving and sending of messages. The multiplexor will switch between at most S parameterizable synthetic routines. Each of these synthetic routines will be parameterized upon request to represent the

resource utilization requirements of one of the following tasks:

1. A job from the (D) configuration drive workload.
2. A special service function provided by this (D) configuration.

For simplicity, we will ignore the program swapping in this model and assume that each application program executes to completion on the computer upon which it originates. Now any real or hypothesized application workload (for this (D) configuration) can be described in terms of the characteristic variables of the last section. We will of course wish to slightly alter the definitions of X1,X2 and X3 so that they reflect both the local (D) operating system and the minimal operating system overhead associated with given application programs. This alteration is identical to that described in Section 4.7. Then from these altered characteristic variables, appropriate synthetic program parameters can be determined for each job in our real or hypothesized (D) application workload. These workload program parameter values can be stored as a simple list in some common data area in our 'A' node.

We need also to obtain the synthetic program parameters for each of the (D) configuration's special service functions. These program parameters are obtained exactly as the DBM program parameters are obtained in Section 4.7. As in that case we will again store these values in some protected data area.

Our construction model for these S synthetic routines will again be the nonself-timing FORTRAN loop model. We will include a separate loop for each I/O device and each network special service function. All I/O will be done exactly as in Section 4.9. Also upon completion of a task these synthetic routines will send an appropriate message to the requestor of this task.

4.14 A Trace of Interrupts in This Simulated Node

In this section we will examine the behavior of our minimal operating system in controlling these synthetic routines. We basically need only address the question as to when interrupts occur and what happens when they occur. Initially we will assume that these S synthetic routines are executing the first S programs in the (D) application workload.

Suppose that an existing process on the B or C computer issues a request for one of the (D) configuration's special service routines. The local operating system on the B or C computer will send an appropriate message to the 'A' node. Then this requesting process will be suspended by the B or C local operating system until an answer is received. This message need only contain the special service type requested and the requesting process ID. When this message is received by the network operating system, an interrupt occurs. The interrupt handler will then start a routine

that will interpret this message. Once the special service function type is determined, the appropriate synthetic program parameters are copied from their protected data area into a list of waiting special service function requests. This list is similar to the list of the workload program parameters. Control then returns to the multiplexor.

If this special service request comes from one of the internal synthetic routines, the above steps are followed; except before returning control to the multiplexor, the internal requesting synthetic routine is placed in a wait state.

Next suppose synthetic routine I has just completed executing a task that represents an application program in the (D) drive workload. This synthetic routine will generate an interrupt. The interrupt handler will then start a routine that will reparameterize synthetic routine I. Some type of simple priority scheme must be developed so that we can determine whether to reparameterize synthetic routine I to represent a program in the (D) drive workload or a special service request. Once we have determined what synthetic routine I should do as its next task, the reparameterization is started. This reparameterization consists of copying the synthetic program parameters from the proper list into the program parameter area for synthetic routine I. Then this synthetic routine is marked ready and control returns to the multiplexor.

Now suppose the task that synthetic routine I completes

represents a special service function rather then an application program. If this special service request came from one of the internal synthetic routines, then this requesting synthetic routine must be marked ready before control returns to the multiplexor. Similarly, if this special service request came from an external process, then a return message must be sent to that process. All other steps in handling these interrupts are the same as in the above case.

The only other interrupt to be considered takes place when one of these S synthetic routines requests the execution of one of the B or C computer special service routines. This interrupt will cause an appropriate message to be sent to the B or C computer. Then this requesting synthetic routine is placed in a wait state until an answer is received. As usual control then passes to the multiplexor. When a return message is detected, another interrupt is generated which results in this synthetic routine being marked ready.

This discussion takes into consideration all interrupts that can be generated in this simulated (D) configuration. The use and validation of this model is basically identical to the discussion presented in Section 4.10.

4.15 Conclusions

In this chapter several models were developed for extending synthetic workload construction methods to various computer networks. First a model was developed for the construction of a synthetic drive workload on a back-end data base management system. Then an example was given on how to change certain model assumptions to obtain a different model for the same purpose. This will enable us to replace bad modeling assumptions with more acceptable alternative assumptions. Then we looked at the possibility of using synthetic programs to enable us to simulate one back-end configuration on another back-end configuration. A detailed synthetic model was developed as a solution to this problem.

After these models were developed for a back-end data base management network they were extented to encompass the more general fully distributed computer network. No actual measurement, calibration or experimentation was done on the proposed models. Such work would be impossible to conduct at this time because the current computer network facilities at Kansas State are inadaquate. However, the steps necessary for the measurement and validation experiments are explained in detail in this report.

It is hoped that these models will provide a valuable starting point for future work on both obtaining the measurements necessary to implement these synthetic models

and on the actual experimental evaluation of these models. Such additional work is certainly warranted by the simplicity of running this type of computer simulation utilizing synthetic programs rather than by the usual simulation techniques.

Acknowledgements

First I would like to give a special thanks to my wife Mary for her patience and encourgements while I was working on this report. Also I would like to thank my committe, Dr. Wallentine, Dr. Kemp and Dr. Hankley. I would particularly like to thank Dr. Wallentine for suggesting the topic and proof-reading the report.

References

1. Brinch Hansen, P., Operating_System_Principles, Prentice Hall,1973.
2. Buchholtz, W., 'A Synthetic Job for Measuring System Performance', IBM Systems Journal, J.8,4, 1969.
3. Campbell, J., A., Morgan, D., E., 'An Answer to a Users Plea', 1st Annual SIGME Symposium on Measurement and Evaluation, 1973.
4. Canaday, R., D., Harrison, E., L., 'A Back-end Computer for Data Base Management', Communications of the ACM, Vol. 17, No. 10, October 1974.
5. Dodd, G., G., 'Elements of Data Base Management Systems', Computing Surveys, Vol. 1, No. 2, June 1969.
6. Draper, N., Smith, H., Applied_Regression_Analysis, John Wiley, 1966.
7. Ferrari, D., 'Workload Characterization and Selection In Computer Performance Measurement', Computer, July/August, 1972.
8. Gotlieb, C., C., Performance_Measurement, Department of Computer Science, University Of Toronto, Canada.
9. Joslin, E., O., 'Application Benchmarks: The key To Meaningful Computer Evaluations', Association for Computing Machinery, (Proceedings 20th National Conference), 1965.
10. Kernighan, B., W., Hamilton, P., A., 'Synthetically

Generated Performance Test Loads for Operating Systems', 1st Annual SIGME Symposium on Measurement and Evaluation', 1973.

11. Lucas, H., C., 'Performance Evaluation and Monitoring', Computing Surveys, 3,3 Sept., 1971.
12. Shoppe, W., L., Kashmark, W., F., 'System Performance Study', Proceedings SHARE XXXIV, Vol. 1, March 1970.
13. Sreenivasian, K., Kleinman, A., J., 'On The Contruction of a Representative Synthetic Workload', Communications of the ACM, Vol. 17, No. 3, March 1974.
14. Stelmach, E., V., Introduction to Mini-Computer Networks, Digital Equipment Corporation, 1974.
15. Waldbaum, G., 'Evaluating Computing System Changes By Means of Regression Models', 1st Annual SIGME symposium on Measurement and Evaluation, 1971.
16. Wood, D., C., Forman, E., H., 'Throughput Measurement Using a Synthetic Job Stream', Proceedings AFIPS, Vol. 39, 1971.

Appendix A

An Illustration of the Service Demand Construction Method

Example FORTRAN Program in System Workload

```
      DIMENSION A(1000)
      TOT=0
      DO 10 I=1,1000
      READ(5,500) X
  500 FORMAT(F6.2)
   10 TOT=TOT+SQRT(A(I))/A(I)
      AVG=A(1)/2.0
      DO 20 I=2,1999
   20 AVG=AVG+(A(I/2+1)+A(I+1)/2))/2.0
      WRITE(6,600) TOT,AVG
  600 FORMAT(7HANSWER=,2F10.2)
      STOP
      END
```

Appendix A

Measurement of Services Provided By the Example Program

Type	Service	Measurement
1	Real assignments.	3000
1	DO loop iterations.	2998
5	Read cards.	1000
4	Implicit run-time routine to convert character to real.	1000
1	Real add/sub.	4996
4	Explicit run-time routine SQRT.	1000
3	Array index calculation.	5997
1	Real divide.	2999
1	Integer add/sub.	3996
1	Integer divide.	3996
5	Write printer.	1
4	Implicit run-time routine to convert real to character.	2

* Type refers to one of the five types of services described in Figure 3.3.

Appendix A

Synthetic Program Created From the Above Measurements

```
      DIMENSION A(1000)
      A(1)=10.
      DO 10 K=1,10
      DO 10 I=1,100
C
C           1000 DO LOOP ITERATIONS
C
      READ(5,500) X
C
C           1000 READ CARDS
C           1000 CONVERT CHARACTER TO REAL
C
  500 FORMAT(F6.2)
      Y=(SQRT(2.0)+75.0)/63.5
C
C           1000 SQRT, REAL ADD, DIVIDE, ASSIGN
C
      DO 20 J=1,2
C
C           2000 DO LOOP ITERATIONS
C
```

Appendix A

```
   20 A(J/2+2)=(Y+A(J))/(Y-A((J+1)/2))
C
C          4000 Integer ADD, DIVIDE
C          4000 REAL ADD/SUB
C          6000 ARRAY INDEX CALCULATION
C          2000 REAL DIVIDE, ASSIGNMENT
C
   10 CONTINUE
      WRITE(6,600) Y,A(1)
  600 FORMAT(2F10.2)
C
C          1 WRITE PRINTER
C          2 REAL TO CHARACTER CONVERSIONS
C
      STOP
      END
```

Appendix A

Measurement of Services Provided By This Synthetic Program

Type	Service	Measurement
1	Real assignments.	3001
1	DO loop iterations.	3000
5	Read cards.	1000
4	Implicit run-time routine to convert character to real.	1000
1	Real add/sub.	5000
4	Explicit run-time routine SQRT.	1000
3	Array index calculation.	6000
1	Real divide.	3000
1	Integer add/sub.	4000
1	Integer divide.	4000
5	Write printer.	1
4	Implicit run-time routine to convert real to character.	2

* Type refers to one of the five types of services described in Figure 3.3.

Appendix B

Simple FORTRAN Loop Synthetic Program

```
C
C
C          DECLARATIONS AND FORMATS
C
C
      INTEGER IA(1500)/1500*0/,NC1/2/,NIO1/2/,TCPU,TIO,
     1DISK1/10/,M1/1/
  100 FORMAT(4I5)
C
C
C          READ IN THE FOLLOWING SYNTHETIC JOB PARAMETERS.
C
C          1.  TCPU     THIS IS THE TOTAL CPU TIME FOR THIS
C                       JOB IN TENTHS OF SECONDS.
C          2.  TIO      THIS IS THE TOTAL I/O TIME FOR THIS
C                       JOB IN TENTHS OF SECONDS.
C
C
    1 READ(5,100,END=35) TCPU,TIO
C
C          CALCULATION OF LOOP INDEXES
```

Appendix B

```
C
C          NC:   OUTER CPU LOOP INDEX.
C
C          NIO:  OUTER I/O LOOP INDEX.
C
C          N:    CONTROL LOOP INDEX.
C
C
      N=1
      K=MIN0(TCPU,TIO)
      L=MAX0(TCPU,TIO)
      DO 11 I=1,K
      J=MOD(K,I)
      IF(J.NE.0) GO TO 11
      J=MOD(L,I)
      IF(J.NE.0) GO TO 11
      NC=TCPU/I
      NIO=TIO/I
      N=I
   11 CONTINUE
C
C
C          START OF CONTROL LOOP
C
```

Appendix B

```
C
C     DO 30 L=1,N
C
C          START OF CPU LOOP
C
      DO 17 I=1,NC
      DO 17 J=1,NC1
      K=IA(J)+IA(J+1)
   17 M=IA(J)*IA(J+1)
C
C          START OF I/O LOOP
C
      DO 18 I=1,NIO
      DO 18 J=1,NIO1
      WRITE(DISK1) (IA(N1),N1=1,M1)
      REWIND DISK1
   18 READ(DISK1) (IA(N1),N1=1,M1)
   30 CONTINUE
      GO TO 1
   35 RETURN
      END
```

Appendix C

Modified File Update Synthetic Program

```
C
C          DECLARATIONS AND FORMATS
C
      INTEGER MASDAT(3),MAST(2)/'MAST','ER  '/,U,SUM,COUNT,
     1CHECK,ETIME,STIME,MASTER(6)
     2TABLE(1000),N/10/,START/100/,DISK1/10/,DISK2/11/
      EQUIVALENCE (MASTER(1),MASKEY),(MASTER(2),MASSUM),
     1(MASTER(3),MASCHK),(MASTER(4),MASDAT(1))
  101 FORMAT(3I10)
  201 FORMAT(10X,'********** THE NUMBER OF RECORDS TO BE',
     1' PROCESSED EXCEEDS THE TOTAL NUMBER OF RECORDS ****',
     2'******')
  202 FORMAT(10X,'********** COMPUTE ERROR HALTED JOB',
     1'**********')
  205 FORMAT(10X,'********** CHECK SUM ERROR HALTED JOB ',
     1'**********')
```

Appendix C

```
C
C          CREATE TABLE ENTRIES
C
      M=N**3
      DO 1 J=1,M
    1 TABLE(J)=START+J-1
C
C
C
C          READ AND CHECK SYNTHETIC JOB PARAMETERS
C
C
C          1.  NMAS      THIS IS THE NUMBER OF MASTER RECORDS
C                        TO BE CREATED.  IF THE NUMBER READ IS
C                        LESS THEN OR EQUAL TO ZERO NO MASTER
C                        RECORDS ARE CREATED.
C
C
C          2.  NPRC      THIS IS THE NUMBER OF MASTER RECORDS
C                        TO BE PROCESSED.  IF THIS NUMBER IS
C                        LESS THEN OR EQUAL TO ZERO NO MASTER
C                        RECORDS ARE PROCESSED AND ONLY THE
C                        COMPUTE KERNAL IS EXECUTED.
C
```

Appendix C

```
C
C          3. NREPS     THIS IS THE NUMBER OF TIMES THE
C                       COMPUTE KERNAL IS TO BE EXECUTED PER
C                       RECORD.  THE COMPUTE KERNAL IS
C                       EXECUTED AT LEAST ONCE PER RECORD.
C
C
    2 READ(5,101,END=51) NMAS,NPRC,NREPS
      COUNT=0
      CHECK=0
      IF(NMAS) 3,3,5
    3 IF(NPRC.LE.NMASTR) GO TO 10
      WRITE(6,201)
      STOP
C
C          CREATE NMASTR MASTER RECORDS
C
    5 NMASTR=NMAS
      REWIND DISK1
      DO 7 J=1,NMASTR
      CHECK=CHECK+J
      MASSUM=0
      MASKEY=J
      MASCHK=CHECK
```

Appendix C

```
      MASDAT(1)=MAST(1)
      MASDAT(2)=MAST(2)
      MASDAT(3)=J
    7 WRITE(DISK1) MASTER
      GO TO 3
C
C          CHECK FOR COMPUTE ONLY
C
   10 IF(NPRC.EQ.0) GO TO 41
      ASSIGN 17 TO KRETRN
C
C          READ FIRST RECORD
C
      REWIND DISK1
      REWIND DISK2
      READ(DISK1) MASTER
C
C          COMPUTE KERNAL
C
   11 CALL INTIME(STIME)
   12 DO 16 I=1,NREPS
      SUM=0
      U=0
      J=0
```

Appendix C

```
      DO 14 K=1,N
      J=J+(6*U+1)
      SUM=SUM+TABLE(J)
   14 U=U+K
      LSUM=(N*(N+1))/2
      LSUM=(SUM-LSUM*LSUM)/N+1
      IF(START.EQ.LSUM) GO TO 16
      WRITE(6,202)
      STOP
   16 CONTINUE
      GO TO KRETRN,(17,45)
C
C          WRITE OUT UPDATED RECORD AND READ NEXT RECORD
C
   17 MASSUM=SUM
      COUNT=COUNT+1
      WRITE(DISK2) MASTER
      IF(COUNT.GE.NPRC) GO TO 31
      READ(DISK1) MASTER
      GO TO 12
   31 CALL INTIME(ETIME)
      LSUM=(COUNT*(COUNT+1))/2
      IF(MASCHK.EQ.LSUM) GO TO 35
      WRITE(6,205)
```

Appendix C

```
      STOP
   35 TIME=(ETIME-STIME)/100.
C
C          PRINT OUT RESULTS
C
      WRITE(6,209) NMAS,NPRC,NREPS,TIME
      GO TO 2
   41 ASSIGN 45 TO KRETRN
      GO TO 11
   45 CALL INTIME(ETIME)
      GO TO 35
   51 RETURN
      END
```

MODELS OF SYNTHETIC PROGRAMS PROPOSED AS PERFORMANCE EVALUATION TOOLS IN A COMPUTER NETWORK

by

DAVID CULP

B.A., California State University, Pomona, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1976

ABSTRACT

With the rising cost and complexity of modern day computing systems, accurate and economical performance evaluation techniques are becoming critically important. One of the most promising new techniques is the concept of synthetic programs. A synthetic program is defined as 'a highly parameterized program which uses precisely specified amounts of computing resources, but which does no useful work'. This type of program is valuable for fixing the resource utilization requirements of the drive workloads, under which a system is tested and analyzed.

This paper discusses the major advantages achieved by using synthetic programs in the construction of such drive workloads. Several actual synthetic program construction models are presented and contrasted. Then these the synthetic program models are extended to performance studies on minicomputer networks. This extension includes not only the workload construction applications, but also the possibility of simulating operating systems and I/O devices through the use of synthetic programs.