Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

SIMULATION OF COMPUTER SYSTEMS AND APPLICATIONS

William S. Keezer

Enterprise Storage Engineering Department
Systems Engineering
LEXIS-NEXIS, a Member of Reed Elsevier plc
P.O. Box 933
Dayton, Ohio, 45401, U.S.A.

ABSTRACT

The modeling of computer systems, particularly distributed systems, is presented with an emphasis on the system characteristics that are important to the model and some general methods to represent them. CPU, memory, DASD, network I/O, load generation, and the parameters for applications and their dynamics are discussed. This is followed by a plan for building the model, focusing on a modular approach that the inclusion of more detail as more knowledge is obtained.

1 INTRODUCTION

With the increase in the number of distributed processing systems and their attendant complexities, there is a definite need to be able to model systems during their design phases, and, in the future, to model potential changes to already operating systems. It has been the author's experience that this activity can detect problems long before they are implemented and guide corrections.

Starting from a simple perspective, we will look at distributed system as one or more units composed of a CPU, memory, DASD and/or tape storage, network connections, a suite of applications, and a workload. The underlying approach is to treat the system as an online transaction processing facility, though there is no reason not to create workload generators that are internal to the system under study, e.g. scheduled batch processing. The goal here is to provide an output that correctly predicts the behavior of the system as seen by an outside observer, not to faithfully reproduce the internal details of the system. Depending on the model, there may be a need for some internal details, but not for the entire system. Vendors have modeled individual components, such as DASD devices or memory, to a high degree of precision, and, to the degree that they have published their work or are willing to share it, great value may be obtained in understanding the internals of the system.

There is no point in modeling more precisely than the system reproducibility, and it is the author's experience that an answer within 10-20% of the true answer is often adequate for the overall system. However, some components of the system may need to be modeled more precisely. Few computer systems have such a constant or systematically varying workload as to be more reproducible than ten percent either way. In the early stages of the design work, the room for variance is even greater. Often the answers at this stage are found with quite simple models, or even with some basic systems engineering work that is used as sanity checks on the designs and the model. Thus, the process we will develop after discussing the system components and how to represent them, is one of putting together simple modules that can be replaced with more detailed versions later in the process as more knowledge of the system is obtained.

This tutorial will not discuss particular packages or any details of actual models. Once the principles of the systems are understood, their representation in a modeling system is fairly straightforward. The focus will be on the system components, how they are structured and operate with some general suggestions for representing them in a model. We will also cover application dynamics, and how to represent application work in a system. Finally there will be some discussion on how to go about building a model of a system under design and development.

2 COMPUTER SYSTEMS COMPONENTS

2.1 CPU

Distributed system CPU can often be modeled as a single resource with a queue and a time delay. However, that is not really the manner in which it occurs, and there are times when detailed knowledge of the operating system workings is important. For example, the cache manager

may be modeled separately from the rest of the operating system workload. Detailed presentations of the UNIX operating system can be found in two sources, Bach (1986) and Leffler, et al (1990). Processing in a UNIX system occurs in one of two logical divisions, user space or system space. User space contains all the application specific data, buffers, and compiled code, and system space contains all the I/O buffers, system memory, and operating system code. To maintain integrity, any data to be processed by the operating system is rewritten to separate memory locations in system space from the user space. The processing in user space is only for the application-level calculations. All work scheduling and I/O is handled by the operating system.

Multiprocessor systems have the potential for parallel processing, but only for user spaces. The operating system has only one lock on the kernel, and therefore any system work is single-threaded through that lock. System work takes precedence over user space work, and a request for a system function will pre-empt any user work in the processor. Though system work functions have priorities among themselves, no system call can pre-empt any other system call; it must wait until the current system function is complete. Once a system function is complete the next function will be the one with the highest priority, but it in turn will be in the processor for its full time-slice or until it requests an I/O. This of itself does not lead to complications, but the fact that there is only one kernel lock for the operating system means all system calls are single-threaded through the equivalent of one processor. Actually, slightly less, because of scheduling overhead and the handling of the lock between processors.

The implications of this arrangement are that a computationally intensive application or set of applications will have a high degree of parallelism on a multiprocessor system, but an I/O intensive system will be throttled to slightly more than the capacity of one processor, and all I/O handling will be serialized. It is possible to have multiple I/Os outstanding, but the actual interrupt handling is only done under the one kernel lock.

The simulation of such an arrangement is actually much more simple than might be first supposed. One models a multiple-server queue, and increases the CPU time to account for the single-threading through the kernel lock. This does not lead to inflated CPU utilization times, because when a user process is waiting for the kernel lock, it goes into a spin-loop checking for the availability of the lock. Thus the CPU is being utilized, though not directly for productive work. In I/O intensive functions, cache management may need to be modeled separately from the rest of the CPU usage(Nelson, Keezer, and Schuppe, 1996). One can treat it as a separate resource with a capacity equal to that of one processor on a multiprocessor system.

Though the additional detail of modeling the various work-spaces and the queueing for the kernel lock might provide a slightly more accurate value for CPU processing, in the author's experience, the above technique was adequate for design and development phase work. However, the exact nature of the workload sharing must be kept in mind. In cases where their is a master processor and the remainder are slaves, the master processor must be separately modeled, because its workload is different. By the time truly accurate answers are required, the system under development is usually up and running and can be measured and modified faster than a model. The exception would be a model to do what-if planning for an existing system.

As a side note, MVS systems can be readily modeled as multiple-server systems, but without the concerns over the kernel locks. MVS has matured to the point that there are thousands of kernel locks, and with fourteen levels of interrupts, system functions can be pre-empted for more critical system functions, and all processors are equal in capability.

2.2 Memory

UNIX was developed in a compute-intensive environment, and most systems come with sufficient memory as not to be memory constrained. In the case of applications with extremely large memory requirements, however, memory will have to be tracked. It is possible to allocate more memory for a process than is available as physical memory. This is called virtual memory. In principle, it is not necessary to be concerned about virtual memory; it is a programming constraint. What is important is the actual use of physical memory.

Memory is managed by keeping the most active portions in physical memory and putting the inactive portions out on specially allocated DASD, to be brought into physical memory as necessary (paging out and paging in, respectively). Physical memory is also used as the cache for I/O, those portions being handled separately by versions of UNIX that use separate caches. Some versions of UNIX, e.g. Solaris™ and SunOS™ from Sun Microsystems, do not partition off a separate cache allocation, but simply use as much memory as necessary for the function.

Memory is modeled as a pool from which allocations are withdrawn as necessary. When the pool is depleted or reaches some set level of empty pages, the least-recently used pages are paged to disc and freed for reuse. It is not necessary to track locations, simply the amount left. One can allocate precisely or with a distribution, and when the page out to reuse space occurs, add some

distributed increment back into the pool. When allocating memory, allocate the entire amount necessary, then let the paging process replenish the pool; this is what occurs in the actual system. The actual details will vary with the system under study, and the modeler must maintain communications with the development staff to help determine what the necessary parameters are. One area that requires care is the termination of an application. The amount of physical memory recovered is not equal to the total allocation, but only that part in physical memory.

2.3 DASD

It has been the author's experience that storage I/O, and in particular DASD I/O, is a major potential bottleneck in UNIX systems. DASD I/O is inherently slower than processing, with one I/O typically taking ten milliseconds, while processing speeds are measured in nanoseconds per instruction.

There has been a lot of progress in the last several years, since the development of the SCSI controller, in improving DASD performance. At one time a DASD I/O required the complete attention of the central processor. Data was moved to the system buffer for a write or the buffer allocated for a read; the data was transferred to or from the disc a block at a time, and then the results transferred back to the calling program. With the advent of SCSI controllers, the data is transferred to a system buffer and the I/O commands issued to the controller. The controller then executes the I/O and puts the results in a system buffer, reading and writing multiple blocks with DMA (direct memory access). At the completion, the SCSI controller then issues an interrupt for the system to process the result. This is quite similar to the manner in which MVS DASD functions, with DASD controllers and channel processors functioning similarly to SCSI controllers.

The discs themselves have also been improved, with on-board caches and processors, some of which could run DOS. Discs can perform prefetch for sequential data and store the results in the cache to speed up the transfer of data and reduce the potential for hardware delays. Details of typical disc operations are given in Ruemmler and Wilkes (1992,1993).

When modeling I/O for applications, the two most important operations are read and write. Open and closes generally are not of major importance, even though they can have high overhead, unless there are many of them relative to the other activities. The major impact of an open command is to increase the response time on a first return from a transaction, if it opens a file. The major impact of a close command is to increase the response time on the last return, if it closes a file.

Read operations are all handled similarly, but there are three main types of writes, synchronous, asynchronous, and DASD fast write. A synchronous write holds the program until the I/O results return to the program. This increases response time by the time of the entire write operation. Asynchronous writes do not wait for a return, and, once the write has been set up, the program continues. DASD fast write occurs on discs which have non-volatile storage (storage that maintains its contents in a power outage or power down). In this case as soon as the data reaches the on-board cache of the disc, a result is returned, eliminating the need to wait for the completion of the physical I/O.

UNIX stores data in memory in a cache as mentioned above. Before a physical I/O is performed, a check is made to see if the data is in cache. If so, then the I/O consists of a simple transfer from cache to the user space. Generally, cache hits are approached on a percentage basis, which leads to a probability for modeling purposes. If the I/O is intensive, and the number of files are small relative to the amount of cache allocated, in some versions of UNIX, the cache manager must be explicitly modeled as a significant contributor to response time.

Details on programming UNIX I/O can be found in Nelson, B. L., Keezer, W. S., and Schuppe, T. F. (1996). As a first step in the earliest models, one can replace the details of the I/O with a branch to either a cache hit or a physical I/O with a distribution of possible response times. Generally a skewed distribution with a mode close to the minimum response time is adequate. Separate provision may be made for the CPU involvement. The next complexity would be to break down the I/O into disc fixed overhead, seek time, rotational delay, and data transfer time. Finally, one can model I/Os with a high degree of precision using the methods in the referenced paper.

As RAID (redundant arrays of inexpensive discs) devices become more common for servers, the models will simplify. RAID architecture separates the physical I/O from the requested logical I/O with large caches, and performance times for cache misses tend to be more uniform. With the very large caches seen on some RAID controllers, cache hit percentages can routinely run in the 90-99%. RAID controllers can also reduce CPU overhead values since mirroring and other management functions can be handled in hardware in some products.

2.4 Network

Networks have been the subject of intense simulation activity for years. For the purposes of this tutorial, networks are simply a combination of CPU and memory overhead and a time delay, possibly distributed. The

details of network protocols are usually not important in modeling the interaction of applications in systems. Results are mostly dependent on how long a transmission takes, regardless of protocol.

3 WORKLOAD COMPONENTS

3.1 Load Generation

Creating a realistic workload for the system is a critical part of the model. Because of the complexity of the transaction interactions at a resource level, small changes can sometimes have major effects. Arrival rates, transaction mixes, and transaction sources are important. The sources of transactions are other systems, internal schedulers, and external users.

The easiest to handle are the internal schedulers. One can simply generate transactions similarly to the schedule in the system under study. The other two sources are more difficult. Transactions from other systems can be handled in two different ways. If the other systems are part of the model, then their requests will naturally arise from the execution of the model. If the request-generating systems are external to the model, then they need to be handled similarly to external users.

The three main parameters to consider with external users are how many are there at one time, what transactions are they submitting, and how long do they think between transactions. The number of users can be handled as an arrival rate problem, based on the average session time and the arrival rate. The simulation of transaction choices can depend on how many transactions there are from which to chose. If there are a number of repetitive transactions, such as those from data entry clerks, then each of these could be simply modeled as separate generators with correct arrival rates.

If, as in the author's experience, there are a large number of transactions, and the choices of the next transaction are varied, a transition matrix has proven to be a good method for generating the arrivals. Each user is generated as an entity and then chooses a transaction starting at a given place in the matrix. The choices are based strictly on probability of a transition from the last transaction to any given transaction. A corresponding pair of matrices with the user think times as a mean and standard deviation is used to generate the delay between transaction submissions. The details of the method may be found in Keezer, Fenic, and Nelson (1992).

3.2 Applications

There are two ways to model applications, create a sub-model for each different transaction of the application(s) or parameterize the transactions and use a generic sub-model. If the transactions are complex the impact on the system may vary greatly with small changes. In such a case the individual sub-model approach is necessary. In the author's experience this creates a high maintenance effort and large amounts of code, since in the design and development phases, changes are constantly being made in response to model results or to programming problems.

The author has used parameterization to simulate relational database transactions successfully (Keezer, in preparation). For standard applications, there are five basic parameters, system id (if the model consists of more than one system), CPU used, memory used, the number of DASD I/Os, and the number of network I/Os. To parameterize transactions, create tables with one column for each system in which the transaction occurs, e.g.,

system ID
CPU
memory
DASD I/O
network I/O
others as needed.

One may then index into the table to obtain the necessary parameters. The distribution of the work in the generic sub-model would be via a set of indexed loops. The indices would be for the number of DASD I/O's, the number of network I/Os, and their total plus one or two for distributing the CPU. The CPU data should only include the requirements for processing, not for the system calls. Those are provided by the system call simulation. The parameterization of relational database procedures is not as simple, though the method is much the same. In relational databases there are both the various tables and the operations on them by the transactions to parameterize. The details are beyond the scope of this tutorial.

Unless there is a known large skew in the way resources are consumed, The most straightforward method of modeling their consumption is to divide the total CPU requirement by the total DASD and network I/Os plus one, and the DASD I/Os by the network I/Os. One then models a cycle of CPU consumption followed by a DASD I/O, and occasionally one of CPU consumption followed by a network I/O. The final pass is the remaining CPU consumption. The memory is allocated at the beginning of the sub-model, similarly to the way most transactions work, unless it is known that additional allocations of memory are done inside the transaction. In which case the additional memory allocations can be done intermittently as were the network I/Os.

It is not desirable to group CPU consumption and I/Os together unless it is known to occur in the real transaction. This is because it will create artificially long blocks of resource utilization, penalizing short transactions compared to reality and making long transactions look better than reality.

4 BUILDING THE MODEL

4.1 Input Data

Because of the high susceptibility of simulation models to GIGO (garbage in, garbage out), considerable attention needs to be paid to the input data for the models. The data can be thought of as occurring in four types, runtime input, application parameters, configuration data, and system data. Runtime inputs are such values as the number of processes, the active transactions, the number of users, the arrival rates of transactions, and the batch scheduling. Application parameters include the amount of CPU required, the number of I/Os, the amount of memory, and the number of network I/Os. Configuration data are items such as the number of drives on a system, the number of systems in the model, the network delays, the amount of memory and cache, and the number of files stored on DASD. System data include the CPU speeds, the various parameters for the physical I/O to DASD, backplane speeds, and controller speeds.

System data require the greatest attention during their creation. System data appears in every calculation, and small errors can be multiplied many times. There are a number of sources for system data, among them benchmarks, manufacturers' literature, e.g., Sun Microsystems' white paper (1992), published literature, e.g., Alexander, et al, (1994), and experience. Any values based on the modeler's experience should always be subject to review in light of further data. Manufacturers' data and published literature sometimes require careful analysis to obtain the numbers actually required. Frequently the desired values have to be derived from reported values, and often the conditions of their generation are not necessarily similar to those being modeled.

Runtime inputs are under the control of the modeler as are the configuration parameters. Application parameters are somewhat more difficult. During the design phases, frequently only guesses can be made for those values. This is where the experience of the modeler and the developers becomes important. Usually reasonable estimates may be made of the parameters, and, with the cooperation of the developers, the actual values may be obtained from test runs of the programs.

4.1.1 Benchmarks

The most critical system values should be obtained from benchmarks. Generally published benchmarks can provide some of the data, but custom benchmarks that perform work similar to that being modeled are the most desirable. Published benchmarks do not always stress the systems under test in the manner that the modeled system will, and they frequently do not provide the data necessary to determine unpublished but important values. Additionally the output from the benchmarks becomes a source of validation and calibration of the model. DASD and network I/O are the two most obvious needs for benchmark data. When designing benchmarks one needs to structure the programs to access data with and without cache hits. Additionally one should use multiple configurations and multiple load levels and test as many operations as possible. Where possible use different data layouts and access patterns. This may appear to be a lot of work, but in the process much of the behavior of the systems being considered will be revealed, providing guidance in what is and is not feasible to do with the new system. It can save considerable modeling effort, if the data is extensive.

The values to be obtained include CPU utilization, both system and user, I/O response time, interrupt counts, average I/O rates, I/O sizes, memory use, and the configuration used during the benchmark.

4.2 Putting It Together

Modeling systems in the design and development phase of a project requires many iterations of the process, each time refining some piece or pieces of the model in light of better information. It is important to start as simply as possible. The model WILL become more complex, faster than desired usually. There will be times when some important value or a group of values will not be reliable. The answer usually lies in one of two places, the developers have not adequately described what they are doing, or there is some important feature of the operating system or the hardware that is not correctly modeled. The former is a communication problem and requires patient, careful questioning. The latter requires research and going over all the available information, such as operating system descriptions, manufacturers literature, and published data, often with very intense analysis to obtain the values desired. A more detailed discussion of working with developers is in McBeath and Keezer(1993).

The first thing to do is some standard systems engineering and analysis, estimating overall loads for a configuration. This includes processor utilizations and I/O rates, as well as network throughputs. A good guide

in this is Jain(1991). Many designers have not done this before starting development. Remember, the cheapest, fastest simulation is the one that doesn't need to be built. Many designs implicitly assume that resources are unconstrained until proven otherwise. Find the obvious constraints the easy way.

Once the overall numbers indicate the system may be able to perform, the first iteration of the model may be started. This iteration should be as simple as possible, with no details on consumption of resources. The load can even be one or two general transactions. The idea is to obtain a first feel for the behavior of the system. If there are surprises at this stage, GOOD! Everybody stands to win once they are analyzed; early problems are far easier to solve than later ones.

The overall structure of the model should be as modular as possible. Each system in the model should be represented by either a sub-model or a generic system model with a table of parameters for each system being studied. Within a system, each function should be as independent as possible. The whole purpose of this is to allow for easy change as designs change. It also makes the job of trying various alternatives easier. One can even start as simply as a single system, with the interactions with other systems modeled as distributed delays. Later, the various systems can be combined for a more detailed look at the overall combination. The internal divisions of a system are a load generator, the CPU, memory, DASD, and a network connection.

Unless it is definitely known ahead of time, one can model the dynamics of the various components using distributions as approximations. Exponential arrivals, log-normal processing and network times, and triangular DASD responses will provide a good high level approximation for overall dynamics. Other than a reasonable estimate of the mean and dispersion, the inputs do not have to be created yet, but some first answers can be obtained. If various functions are modeled as sub-models, then as more detail is needed and the necessary supporting input data is generated those functions can be replaced easily.

The outputs of interest include resource utilization, throughput-response curves for various systems, and overall transaction response times. Throughput-response curves are very useful and sensitive for validating and calibrating components of a model against benchmark data, and the process of making the model reproduce the benchmark results can lead to insights into the operation of the systems that are not necessarily documented.

5 CONCLUSION

Just as in manufacturing, simulation studies of computer systems during the design phase and throughout development can provide considerable assistance to the developers, revealing potential problems and finding better ways to distribute work. It is important to keep everything as simple as possible. The model should never be more complex than the development concepts it supports, and generally does not need to be as specific as the development efforts. In this area, knowledge of computer systems is as important as simulation technique; the simulations flow in a straightforward manner once the systems are understood.

REFERENCES

Alexander, T.B., et al 1994. Corporate Business servers: An Alternative to Mainframes for Business Computing. Hewlett-Packard Journal 45 No. 3: 8-30.

Bach, M.J. 1986. The Design of the UNIX® Operating System. Englewood Cliffs, N.J: Prentice-Hall, Inc.

Jain, R. 1991. The Art of Computer Systems Performance Analysis. New York: John Wiley and Sons.

Keezer, W.S., in preparation. Simulating Relational Databases.

Keezer, W.S., Fenic, A.P., and Nelson, B.L. 1992. Representation of User Transaction Processing Behavior with a State Transition Matrix. *Proceedings of the 1992 Winter Simulation Conference*, Vol. 25, ed. Swain, J.J., Goldsman, D., Crain, R.D., Wilson, J.R. 1223-1231. Baltimore, MD: Association for Computing Machinery.

McBeath, D.F., and Keezer, W.S., 1993. Simulation Support of Software Development. *Proceedings of the 1993 Winter Simulation Conference*, Evans, G.W., Mollaghasemi, M., Russell, E.C., and Biles, W.E., eds., IEEE, Piscataway, NJ: p. 1143.

Nelson, B. L., Keezer, W. S., and Schuppe, T. F. 1996. A Hybrid Simulation-Queueing Module for Modeling UNIX I/O in Performance Analysis. *Proceedings of the 1996 Winter Simulation Conference*, Charnes, J. M., Morrice, D. M., Brunner, D. T., and Swain, J. J., eds., IEEE, Piscataway, NJ: p. 1238.

Ruemmler, C., and Wilkes, J. 1992. UNIX Disk Access Patterns. USENIX Winter 1993 Technical Conference Proceedings, San Diego, CA, January.

Ruemmler, C., and Wilkes, J. 1993. Modeling Disks. HP Laboratories Technical Report HPL-93-68 revision 1, December.

S. J. Leffler, M. K. McKusick, M.J. Karels, and J.S. Quarterman 1990, The Design and Implementation of the 4.3SD UNIX Operating System, Addison-Wesley, NY.

SPARCcentera 2000, Technical White Paper, Sun Microsystems, Inc., November, 1992.

AUTHOR BIOGRAPHY

WILLIAM S. KEEZER is currently a Senior Systems Engineer focusing on mainframe storage management and has been with LEXIS-NEXIS for over ten years. Before coming to LEXIS-NEXIS, he was an in-house consultant on OLTP system performance problems for the Data Pathing Division of NCR. He holds B.S. and Ph.D. degrees from the University of Oklahoma, and is a member of the ACM.