GBENLE SHERIF AYOKUNLE

16/SCI01/015

DEPARTMENT OF COMPUTER SCIENCE

COLLEGE OF SCIENCE.

CSC 410 ASSIGNMENT 2

THE PREVIOUS ASSIGNMENT

1. The concepts of operational laws as applied to computer and network system performance evaluation.

A number of laws are derived which establish relationships between throughput, response time, device utilization, space-time products and various other factors related to computer system performance. These laws are obtained by using the operational method of computer system analysis. The operational method, which differs significantly from the conventional stochastic modeling approach, is based on a set of concepts that correspond naturally and directly to observed properties of real computer systems. Operational methods simply refer to methods of physical observation and measurement, in other words, they are directly measurable or directly measured. Except for measurement errors, the operational laws apply with complete precision to all collections of observational data, and they are similar to fundamental laws found in other areas of engineering and applied science.

1. Eight Operational Laws that are widely employed in computer system performance evaluation
2. Utilization Law

If we know the amount of processing that each job requires at a resource then we can calculate the utilisation of the resource. Let us assume that each time a job visits the ith resource the amount of processing, or service, time it requires is Si . Note that service time is not necessarily the same as the residence time of the job at that resource: in general a job might have to wait for some time before processing begins. The total amount of service that a system job generates at the ith resource is called the service demand, Di :

Di = SiVi

The utilisation of a resource, the percentage of time that the ith resource is in use processing to a job, is denoted Ui . The utilisation law states that

Ui = XiSi = XDi

The utilisation of a resource is equal to the product of the throughput of that resource and the average service requirement at that resource.

1. Forced Flow Law

In discussing Little’s law, we allowed our field of view to range from an individual resource to an entire system. At different levels of detail, different definitions of “request” are appropriate. For example, when considering a disk, it is natural to define a request to be a disk access, and to measure throughput and residence time on this basis. When considering an entire system, on the other hand, it is natural to define a request to be a user-level interaction, and to measure throughput and residence time on this basis. The relationship between these two views of a system is expressed by the forced flow law, which states that the flows (throughputs) in all parts of a system must be proportional to one another. Suppose that during an observation interval we count not only system completions, but also the number of completions at each resource. We define the visit count of a resource to be the ratio of the number of completions at that resource to the number of system completions, or, more intuitively, to be the average number of visits that a system-level request makes to that resource. If we let a variable with the subscript k refer to the k-th resource (a variable with no subscript continues to refer to the system as a whole), then we can write this definition as:

Vk, the visit count of resource k: Vk =

If during an observation interval we measure 10 system completions and 150 completions at a specific disk, then on the average each system-level request requires 150/10 = 15 disk operations. If we rewrite this definition as Ck = VkC and recall that the completion count divided by the length of the observation interval is defined to be the throughput, then the throughput of resource k is given by:

The Forced Flow Law: Xk = V,X

An informal statement of the forced flow law is that the various components of a system must do comparable amounts of work (measured in “transaction’s worth”) in a given time interval.

Fundamental Laws Little’s law becomes especially powerful when combined with the forced flow law.

1. Little’s Law

Little’s Law is a theorem that determines the average number of items in a stationary queuing system based on the average waiting time of an item within a system and the average number of items arriving at the system per unit of time.

The law provides a simple and intuitive approach for the assessment of the efficiency of queuing systems. The concept is hugely significant for [business operations](https://corporatefinanceinstitute.com/resources/knowledge/finance/what-is-corporation-overview/) because it states that the number of items in the queuing systems primarily depends on two key variables, and it is not affected by other factors such as the distribution of the service or service order.

Little’s Law can only be used in queuing systems. Almost any queuing system and even any sub-system can be assessed using the law. In addition, the theorem can be applied in different fields, from running a small coffee shop to the maintenance of the operations of a military airbase.

The Formula for Little’s Law:
Mean Number in the device = Arrival Rate Mean Time in the device

Which can be denoted as Qi = λi Ri

If the job flow is balanced, the arrival rate is equal to the throughput and we can write:

Qi = Xi Ri

1. General Response Time Law

One method of computing the mean residence or response time per job in a system is to apply Little’s law to the system as a whole. However, if the mean number of jobs in the system, N, or the system level throughput, X, are not known an alternative method can be used. Applying Little’s law to the ith resource we see that Ni = XiWi , where Ni is the mean number of jobs at the resource and Wi is the average response time of the resource. From the forced flow law we know that Xi = XVi . Thus we can deduce that

The total number jobs in the system is clearly the sum of the number of jobs at each resource, i.e. N = N1 + · · · + NM if there are M resources in the system. We know from Little’s law that W = N/X and from this we arrive at the general residence time, or general response time law:

The average residence time of a job in the system will be the sum of the product of its average residence time at each resource and the number of visits it makes to that resource.

1. Interactive Response Time Law

The name of this law dates back to the time when most of the systems which were being modeled were mainframes processing both interactive jobs and batch jobs. The think time, Z, was quite literally the length of time that a programmer spent thinking at his terminal before submitting another job. More generally interactive systems are those in which jobs spend time in the system not engaged in processing, or waiting for processing: this may be because of interaction with a human user, or may be for some other reason. For example, if we are studying a cluster of PCs with a central file server to investigate the load on the file server, the think time might represent the average time that each PC spends processing locally without access to the file server. At the end of this non processing period the job generates a fresh request.

The key feature of such a system is that the residence time can no longer be taken as a true reflection of the response time of the system. The think time represents the time between processing being completed and the job becoming available as a request again. Thus the residence time of the job, as calculated by Little’s law as the time from arrival to completion, is greater than the system’s response time. The interactive response time law reflects this: it calculates the response time, R as follows:

R = N/X – Z

The response time in an interactive system is the residence time minus the think time. Note that if the think time is zero, Z = 0 and R = W, then the interactive response time law simply becomes little’s law.

1. Bottleneck Analysis

From forced flow law, the formula is Ui α Di

According to the Bottleneck Analysis,

* The device with the highest total service demand Di has the highest utilization and is called the bottleneck device.
* Note: Delay centers can have utilizations more than one without any stability problems. Therefore, delay centers cannot be a bottleneck device.
* Only queueing centers used in computing Dmax.
* The bottleneck device is the key limiting factor in achieving higher throughput.
* Improving the bottleneck device will provide the highest payoff in terms of system throughput.
* Improving other devices will have little effect on the system performance.
* Identifying the bottleneck device should be the first step in any performance improvement project.

Throughput and response times of the system are bound as follows:

And

 Here, D = is the sum of total service demands on all devices except terminals.

These are known as asymptotic bounds

Bottle Neck Analysis Proof:

The asymptotic bounds are based on the following observations:

The utilization of any device cannot exceed one. This puts a limit on the maximum obtainable throughput.

The response time of the system with N users cannot be less than a system with just one user. This puts a limit on the minimum response time.

The interactive response time formula can be used to convert the bound on throughput to that on response time and vice versa.

For the bottleneck device b we have: Ub = XDmax

Since Ub cannot be more than one, we have: XDmax 1

 X

With just one job in the system, there is no queueing and the system response time is simply the sum of the service demands:

R(1) = D1 + D2 + … + DM = D

Here, D is defined as the sum of all service demands. ! With more than one user there may be some queueing and so the response time will be higher. That is:

Applying the interactive response time law to the bounds:

R(N) D

Combining these bounds we get the asymptotic bounds

1. Space-Time Product Throughput Law

The Space-Time Product Throughput Law states that the throughput is equal to the average amount of memory in use are divided by average space-time product.

The formula is:

Where X = Throughput (i.e. number of job completions per unit time)

 M = Average amount of memory in use during the observation interval

 Y = Average space-time product (i.e. space-time product completed per job

1. Space-Time Product Response Time Law

The Space-Time Product Response Time Law is obtained by a similar relationship between average space-time product and average response time in the asymptotic case.

The formula is:

 Where R = response time (i.e. average amount of time in system state per interaction)

N = Number of interactive terminals. It is assumed that N is constant throughout the observation interval.

Y = Average space-time product (i.e. space-time product completed per job

M = Average amount of memory in use during the observation interval

Z = Average think time (i.e. average amount of think time per transition from think state to system state).

1. Differences between the Forced Flow Law and the Residence Time Law from a systems perspective

Forced Flow Law

Forced Flow Law relates the system throughput to individual device throughputs.

According to the Forced Flow law, the following is proven

* In an open model, System throughput is equal to the number of jobs leaving the system per unit time
* In a closed model, System throughput is equal to the number of jobs traversing out to in link per unit time.
* If observation period T is such that Ai = Ci, then the Device satisfies the assumption of job flow balance.
* Each job makes Vi requests for ith device in the system.
* Ci = C0 Vi or Vi = is called visit ratio
* System Throughput (X) = =
* Device Throughput (Xi) = = which can be simplified to Xi = X Vi

Where Ci and C0 = The Jobs completed

T = Total Time

Ai = Number of arrivals

Vi =

Residence Time Law

According to Residence Time Law, there is one terminal per user and the rest of the system is shared by all users.

When applying Little's law to the central subsystem, the formula is Q = X R

Where, Q = Total number of jobs in the system

 R = system response time

 X = system throughput

The Residence Time Formula is R = X1R1 + X2R2 + … + XMRM

When dividing both sides by X and using forced flow law, the formula is

R = V1R1 + V2R2 + … + VMRM

1. Basic queuing models and basic queuing disciplines

M/M/1 queue represents the queue length in a system having a single server, where arrivals are determined by a [Poisson process](https://en.wikipedia.org/wiki/Poisson_process) and job service times have an [exponential distribution](https://en.wikipedia.org/wiki/Exponential_distribution). The model name is written in [Kendall's notation](https://en.wikipedia.org/wiki/Kendall%27s_notation). The model is the most elementary of queuing models and an attractive object of study as [closed-form expressions](https://en.wikipedia.org/wiki/Closed-form_expression) can be obtained for many metrics of interest in this model. An extension of this model with more than one server is the [M/M/c queue](https://en.wikipedia.org/wiki/M/M/c_queue)

* The (M/M/1) system This is a queuing model in which the arrival is Marcovian and departure distribution is also Marcovian, number of server is one and size of the queue is also Marcovian, no. of server is one and size of the queue is infinite and service discipline is 1st come 1st serve (FCFS) and the calling source is also finite.
* M/M/1 queue is a useful approximate model when service times have standard deviation approximately equal to their means.

M/M/c/∞/∞ queue:

c servers operating in parallel

* Arrival process is Poisson with rate λ
* Each server has an independent and identical exponential service time distribution, with mean 1/μ.
* To achieve statistical e q, (equilibrium, the offered load (λ/μ) y must satisfy λ/μ < c, where λ/ (cµ) = ρ is the server utilization.

M/M/c queue (or Erlang–C model)

This is a multi-server [queuing model](https://en.wikipedia.org/wiki/Queueing_model). In [Kendall's notation](https://en.wikipedia.org/wiki/Kendall%27s_notation) it describes a system where arrivals form a single queue and are governed by a [Poisson process](https://en.wikipedia.org/wiki/Poisson_process), there are *c* servers and job service times are exponentially distributed It is a generalisation of the [M/M/1 queue](https://en.wikipedia.org/wiki/M/M/1_queue) which considers only a single server. The model with infinitely many servers is the [M/M/∞ queue](https://en.wikipedia.org/wiki/M/M/%E2%88%9E_queue).’

M/M/M/s/N/N queue:

here the arrival distribution of customers follows Poisson distribution and the distribution for service time follows exponential distribution with s number of parallel serves. The number of population and the queuing capacity is limited to N. this situation often happens in queuing for machine repair system where the number of population is equal to the number of machine = N’

M/M/s/s/n queue:

Consider the loss system (no waiting places) in the case where the arrivals originate from a finite population of sources: the total number of customers is n.

* To be specific, think of the customers being telephone users.
* Assume that the time to the next call attempt by the customer, so called thinking time (idle time) of the customer obeys the distribution Exp (γ).
* Blocked calls are lost – does not lead to reattempts – starts a new thinking time: again, the time to the next attempt ∼ Exp (γ) – the holding time X ∼ Exp (µ)

Queuing Disciplines

1. First come first services (FCFS)
2. First in First out (FIFO)
3. Last in First out (LIFO)
4. Service in Random order (SIRO)

**First Come First Serve (FCFS)** is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.

As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue

## Characteristics of FCFS method

* It supports non-pre-emptive and pre-emptive scheduling algorithm.
* Jobs are always executed on a first-come, first-serve basis.
* It is easy to implement and use.
* This method is poor in performance, and the general wait time is quite high.

## Example of FCFS scheduling

A real-life example of the FCFS method is buying a movie ticket on the ticket counter. In this scheduling algorithm, a person is served according to the queue manner. The person who arrives first in the queue first buys the ticket and then the next one. This will continue until the last person in the queue purchases the ticket. Using this algorithm, the CPU process works in a similar manner.

 First in First out (FIFO)

Stands for "First In, First Out." FIFO is a method of processing and retrieving data. In a FIFO system, the first items entered are the first ones to be removed. In other words, the items are removed in the same order they are entered.

To use a real world analogy, imagine a vending machine where the items are loaded from the back. When someone selects a Milky Way bar from row E5, the machine churns out the candy bar closest to the front. The next Milky Way in line then moves to the front. Therefore, using the FIFO method, the candy bars are dispensed in the order they were placed in the machine.

Computers often implement the FIFO system when extracting data from an array or [buffer](https://techterms.com/definition/buffer). If the first data entered into the buffer must be extracted first, the FIFO method is used. The opposite of FIFO is [LIFO](https://techterms.com/definition/lifo), in which the last data entered is the first to be removed.

Last in First out (LIFO)

Stands for "Last In, First Out." LIFO is a method of processing data in which the last items entered are the first to be removed. This is the opposite of LIFO is [FIFO](https://techterms.com/definition/fifo) (First In, First Out), in which items are removed in the order they have been entered.

To better understand LIFO, imagine stacking a deck of cards by placing one card on top of the other, starting from the bottom. Once the deck has been fully stacked, you begin to remove the cards, starting from the top. This process is an example of the LIFO method, because the last cards to be placed on the deck are the first ones to be removed.

The LIFO method is sometimes used by computers when extracting data from an array or data [buffer](https://techterms.com/definition/buffer). When a program needs to access the most recent information entered, it will use the LIFO method. When information needs to be retrieved in the order it was entered, the FIFO method is used.

Service in Random order (SIRO)

When the system chooses one of the data to execute at random. Such a system is known as **service in random order (SIRO).**

1. How to resolve some basic queuing problems

Firstly, queuing problems are the problems that occur when the service rendered doesn’t match the level of demand i.e. waiting for some time till the service is rendered to a customer. These customers might be humans at a bank on a queue or even airplanes ready to take off/ land or jobs waiting to be processed. An example could be when a supermarket doesn't have enough cashiers on a busy morning or when requests reach a system faster than it can process them.

Generally, no two businesses’ queuing problems are the same but some basic queuing problems that exist include; more customers entering the queues than leaving, when the queues are too long and strenuous, where queues are idle and not moving due to many queues and limited service providing stands to ease the queues etc.

We can then solve these various problems by:

1. Implement Digital Queuing Software:

Long queues can inspire customer’s irritation even disgust. But anyone can learn how to reduce queues the use of a nifty technology called a queue management system (QMS).Automating the queuing process creates more labor efficient customer lines, decreases the overall amount of walkaways as well as ultimately reducing queue times. When it’s their turn, a teller calls them to the counter to be served. They can see where they are in line by observing HDTVs hung on the walls of the organization and therefore customers are free to sit or wander and maybe grab a coffee across the street as they wait. They’re not corralled into the line like sheep. By giving customers back their time (their autonomy) one enable customers to wait in leisure. Now that’s effective queue management.

1. Design Your Environment To Be Able To Accommodate Queues:

Studies have shown that one of the most common issues found in lines is queue anxiety. A well-designed queuing area can help organize waiting lines, remove the possibility of queue jumpers and generally ease customer flow management.

1. Reduce Response times**:**

So when it comes to providing service, be quick as possible. It's not possible to solve every problem immediately, but customers don’t expect that from you. What they do expect is that you give them some kind of response quickly. Having all information at your fingertips is the next step as these steps will help improve the flow of the queues and have less waiting times.

1. Occupy Customers in The Queue:

Boredom in the queue can often lead to longer perceived waiting times. Queue solutions is to provide a distraction to people in the queue and help them continue shopping while waiting, easing up frustrations etc. Display entertaining programming on HDTVs. Prompt customers to answer surveys to report on their experience. Engaging customers is the best way to reduce the tension inherent in queueing. Because it’s typically the psychology behind queueing rather than the queues themselves that makes queues feel unbearable.

1. Keep The Rules Of Queuing Fair And Consistent:

One of the most important characteristics of any queue problem solving method is the queuing discipline used. Simply put, the queuing discipline is the rule used to decide who goes next in a queue.

Two of the most commonly used rules are:

First in, First out.

Last in, First out.

Bottom line, people expect queues to be fair. It’s not like they’re happy to be stuck waiting in line, to begin with. But when everyone abides by the same rules, we can’t help but follow them too.

1. Let Customers Know How Long The Wait Is:

The uncertainty of how long it will take to wait is often the cause of queue anxiety. Because of this the customers are impatient and this is a major cause of queuing problems as people want to jump the queues or altogether leave the queue.

1. Assess and improve your queue management strategy:

How do you currently handle a long line of customers? Think about what works well and what doesn’t. Assessing the tactics used to manage the queue in the particular organization will really help solve the queuing problems being encountered.

1. **The** specific motive will you consider imperative in general is **Innovation**
2. [The](https://medium.com/%40sammyabdullah/successful-saas-companies-spend-23-of-revenue-on-r-d-3602e9dc2de) best companies ensure their products stay ahead of the game when it comes to features, user-friendliness, security, flexibility, and interoperability. In answering the question of meeting future needs, don’t just consider what a tool or technology offers today, extrapolate from its innovation track-record to anticipate what it will offer in the future. To evaluate innovation, you’ll want to know how many releases per year to expect, what the current high-level roadmap looks like, how the roadmap is prioritized, and what kind of influence you’ll have (as a customer) on the roadmap.
3. I would choose the technique that carries the lowest risk. A technique that can be further upgraded depending on the project.
4. The main action is to start expanding further on the technique to make sure it is capable of handling the project given.
5. I would choose Simulation Technique. Simulation typically uses statistical and computer modeling to investigate the performance of a business process either for a new situation or to improve an existing set of processes. By modeling different process scenarios and outcomes, companies can minimize the traditional risks associated with change management initiatives without having to make changes in a 'live' business environment where performance could adversely be affected.

THE CURRENT ASSIGNMENT

The three main performance evaluation techniques are:

1. Performance measurement

2. Analytic performance modelling

3. Simulation performance modelling

Using the following performance measurement technique, we can achieve an accurate performance report:

• On-chip Performance Monitoring Counters: All state-of-the-art high performance microprocessors including Intel's Pentium III and Pentium IV, IBM's POWER 3 and POWER 4 processors, AMD's Athlon, Compaq's Alpha, and Sun's UltraSPARC processors incorporate on-chip performance monitoring counters which can be used to understand performance of these microprocessors while they run complex, real-world workloads. This ability has overcome a serious limitation of simulators, that they often could not execute complex workloads. Now, complex run time systems involving multiple software applications can be evaluated and monitored very closely. All microprocessor vendors nowadays release information on their performance monitoring counters, although they are not part of the architecture. For illustration of on-chip performance monitoring, we use the Intel Pentium processors. The microprocessors in the Intel Pentium contain two performance monitoring counters. These counters can be read with special instructions (eg: RDPMC) on the processor. The counters can be made to measure user and kernel activity in combination or in isolation. A variety of performance events can be measured using the counters

• Off-chip hardware measurement: Instrumentation using hardware means can also be done by attaching off-chip hardware, two examples of which are described in this section.

Speed-Tracer from AMD: AMD developed this hardware tracing platform to aid in the design of their x86 microprocessors. When an application is being traced, the tracer interrupts the processor on each instruction boundary. The state of the CPU is captured on each interrupt and then transferred to a separate control machine where the trace is stored. The trace contains virtually all valuable pieces of information for each instruction that executes on the processor. Operating system activity can also be traced. However, tracing in this manner can be invasive, and may slow down the processor. Although the processor is running slower, external events such as disk and memory accesses still happen in real time, thus looking very fast to the slowed-down processor. Usually this issue is addressed by adjusting the timer interrupt frequency. Use of this performance monitoring facility can be seen in Merten and Bhargava.

Logic Analyzers: Poursepanj and Christie use a Tektronix TLA 700 logic analyzer to analyze 3D graphics workloads on AMD-K6-2 based systems. Detailed logic analyzer traces are limited by restrictions on sizes and are typically used for the most important sections of the program under analysis. Preliminary coarse level analysis can be done by performance monitoring counters and software instrumentation. Poursepanj and Christie used logic analyzer traces for a few tens of frames which covered a second or two of smooth motion.

• Software Monitoring: Software monitoring is often performed by utilizing architectural features such as a trap instruction or a breakpoint instruction on an actual system, or on a prototype. The VAX processor from Digital (now Compaq) had a T-bit that caused an exception after every instruction. Software monitoring used to be an important mode of performance evaluation before the advent of on-chip performance monitoring counters. The primary advantage of software monitoring is that it is easy to do. However, disadvantages include that the instrumentation can slow down the application. The overhead of servicing the exception, switching to a data collection process, and performing the necessary tracing can slow down a program by more than 1000 times. Another disadvantage is that software monitoring systems typically only handle the user activity.

• Microcoded Instrumentation: Digital (now Compaq) used microcoded instrumentation to obtain traces of VAX and Alpha architectures. The ATUM tool [14] used extensively by Digital in the late 1980s and early 1990s uses microcoded instrumentation. This is a technique lying between trapping information on each instruction using hardware interrupts (traps) or software traps. The tracing system essentially modified the VAX microcode to record all instruction and data references in a reserved portion of memory. Unlike software monitoring, ATUM could trace all processes including the operating system. However, this kind of tracing is invasive, and can slow down the system by a factor of 10 without including the time to write the trace to the disk.

Using the Analytical performance modelling, we can achieve an accurate performance report: Performance measurement as described in the previous section can be done only if the actual system or a prototype exists. It is expensive to build prototypes for early stage evaluation. Hence one needs to resort to some kind of modeling in order to study systems yet to be built. Performance modeling can be done using simulation models or analytical models.

Using the following Simulation performance modelling techniques, we can achieve an accurate performance report:

• Trace-driven simulation: consists of a simulator model whose input is modeled as a trace or sequence of information representing the instruction sequence that would have actually executed on the target machine. A simple trace driven cache simulator needs a trace consisting of address values. Depending on whether the simulator is modeling a unified instruction or data cache, the address trace should contain addresses of instruction and data references. Cachesim5 and Dinero IV are examples of cache simulators for memory reference traces. These simulators are not timing simulators. There is no notion of simulated time or cycles, only references. They are not functional simulators. Data and instructions do not move in and out of the caches. The primary result of simulation is hit and miss information. The basic idea is to simulate a memory hierarchy consisting of various caches. The various parameters of each cache can be set separately (architecture, mapping policies, replacement policies, write policy, statistics). During initialization, the configuration to be simulated is built up, one cache at a time, starting with each memory as a special case. After initialization, each reference is fed to the appropriate top-level cache by a single simple function call. Lower levels of the hierarchy are handled automatically. One does not need to store a trace while using cachesim5, because Shade can directly feed the trace into cachesim5

• Execution Driven Simulation: There are two meanings in which this term is used by researchers and practitioners. Some refer to simulators that take program executables as input as execution driven simulators. These simulators utilize the actual input executable and not a trace. Hence the size of the input is proportional to the static instruction count and not the dynamic instruction count. Mis-predicted branches can be accurately simulated as well. Thus, these simulators solve the two major problems faced by trace-driven simulators. The widely used Simple-scalar simulator is an example of such an execution driven simulator. With this tool set, the user can simulate real programs on a range of modern processors and systems, using fast execution-driven simulation. There is a fast-functional simulator and a detailed, out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.

• Complete system simulation: Many execution and trace driven simulators only simulate the processor and memory subsystem. Neither I/O activity nor operating system activity is handled in simulators like Simplescalar. But in many 9 workloads, it is extremely important to consider I/O and operating system activity. Complete system simulators are complete simulation environments that model hardware components with enough detail to boot and run a full-blown commercial operating system. The functionality of the processors, memory subsystem, disks, buses, SCSI/IDE/FC controllers, network controllers, graphics controllers, CD-ROM, serial devices, timers, etc are modeled accurately in order to achieve this. While functionality stays the same, different microarchitectures in the processing component can lead to different performance. Most of the complete system simulators use microarchitectural models that can be plugged in and out. For instance, SimOS, a popular complete system simulator provides a simple pipelined processor model and an aggressive superscalar processor model. SimOS and SIMICS can simulate uniprocessor and multiprocessor systems.

• Stochastic Discrete Event Driven Simulation: It is possible to simulate systems in such a way that the input is derived stochastically rather than as a trace/executable from an actual execution. For instance, one can construct a memory system simulator in which the inputs are assumed to arrive according to a Gaussian distribution. Such models can be written in general purpose languages such as C, or using special simulation languages such as SIMSCRIPT. Languages such as SIMSCRIPT have several built-in primitives to allow quick simulation of most kinds of common systems. There are built-in input profiles, resource templates, process templates, queue structures, etc. to facilitate easy simulation of common systems. An example of the use of event-driven simulators using SIMSCRIPT may be seen in the performance evaluation of multiple-bus multiprocessor systems in Kurian et. Al

• Program Profilers: There are a class of tools called software profiling tools, which are similar to simulators and performance measurement tools. These tools are used to generate traces, to obtain instruction mix, and a variety of instruction statistics. They can be thought of as software monitoring on a simulator. They input an executable and decode and analyze each instruction in the executable. These program profilers can be used as the front end of simulators. A popular program profiling tool is Shade for the UltraSparc.