CHARACTERIZING USER WORKLOAD FOR CAPACITY PLANNING

By

JEE FUNG PANG

B.Sc., University of British Columbia, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1986

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 10/21/86

DE-6 (3/81)

ABSTRACT

With the widespread use of computers in today's industry, planning system configurations in computer sites plays an increasingly important role. The process of planning system configurations or determining hardware requirements for new or existing systems is commonly known as *capacity planning* among performance researchers and analysts.

This thesis presents a refined capacity planning process for centralized computing system, with special attention to characterizing user workload for capacity planning. The objective is to make the entire process simpler for the computer user community, while relieving the capacity planner or performance analyst from having to rely on guesswork for the user workload performance factors.

The process is divided into four phases; namely, data collection, data reduction, workload/user classification and, modeling and performance analysis. The second and third phases are collectively known as user workload characterization.

The main objective of our workload characterization is to avoid any guess work on the performance factors that cannot be easily measured. The results of the workload characterization process are specifically meant to be used in analytic and simulation modeling. Three software tools required for the data reduction, workload/user classification and performance analysis phases have been developed and are discussed in the thesis.

CONTENTS

INTRODUCTION

With the widespread use of computers in today's industry, planning system configurations in computer customer sites plays an increasingly important role. The process of planning system configurations or determining hardware requirements for new or existing systems is commonly known as *capacity planning* among performance researchers and analysts.

This thesis presents a refined capacity planning process for centralized computer systems, with special attention to characterizing user workload for capacity planning. In the past, researchers and analysts have devoted much time and effort in refining performance projection techniques. Little effort has been spent in finding an accurate and systematic way of representing the workload used by these projection techniques. To date, the most common ways of obtaining user workload are through the use of sampling tools and benchmarks. These tools usually do not provide enough information to represent user workload and often introduce unnecessary and non-negligible overhead.

The main objective of our workload characterization is to avoid any guess work on the performance factors that cannot be easily measured. The proposed alternative to collecting user workload is through the use of a system event-

driven software monitor. The advantages of such a monitor are that it is relatively easy to develop, it introduces only about 2%-3% overhead and it can provide almost any data desired. The tools required to process the event data will also be presented here. Each of these tools corresponds to a phase of our refined capacity planning process.

The results of the workload characterization process are specifically meant to be used in analytic and simulation modeling. An analytic modeling tool based on the Linearizer Approximation Method[4] will be used as an illustration on how the characterized workload can be used for performance validations and projections.

CHAPTER 2
THE CAPACITY PLANNING PROCESS

In general, capacity planning serves to answer the typical "what if" type questions regarding computer system performance. The most common of such questions are:

(1) What is the system bottleneck? In other words, which device(s) is limiting system performance.

(2) When will the system saturate? In other words, based on projected rate of growth of the workload, when will the system reach its capacity?

(3) What is an appropriate initial system configuration for a new installation site?

(4) What is the impact of changing user workload? Examples are by increased number of users or different kinds of user workload.

(5) How can one tune an existing system to obtain better performance?

(6) How much workload can a system support without significant degradation in performance?

Note that the answers to all of the above questions require system performance projection or prediction. In fact, capacity planning consists of two stages; namely, user workload characterization and performance projection. In the first stage the user workload is quantified and used as input to the second stage. The results of the projections help the analyst decide on the best solution for his capacity planning task.

2.1 BACKGROUND

Over the past years, several approaches for performance projection have been used for capacity planning. They range from guessing to using real systems. Brief descriptions of these approaches are given below:

(1) **Guessing.**

Here, the performance analysts guess the performance of a projected system based on past experience. Although the very experienced analyst might be able to come up with a good estimate, this approach is very often inaccurate and it is difficult to justify the results.

(2) **Linear Projection.**

The performance is projected linearly based on the performance factors or workload parameters obtained from a single user environment. Linear projection typically over-estimates projected performance because it does not take resource contention into consideration. Today, it is still used in certain pre-sales configuration planning where performance under-estimation is necessary. In general, it is inaccurate and the scope covered is very limited. Specifically, this approach cannot be used to project response times, or when response times are used as part of performance guidelines.

(3) **Analytic Modeling.**

In this approach, mathematical models and queuing theory or operational analysis [8] are used to project system performance. Compared to linear projection, more workload factors are taken into consideration. Also, the workload parameters are typically obtained from a real multi-user system, rather than a synthetic single user environment. A lot of research has been done in this area with promising results. The models developed can be specific to a particular system in which case the results are usually quite accurate, or can be general for a wide range of systems. In general,

the error given by analytic models is seldom worse than 20% (e.g. BEST1, CADS and MAP).

(4) **Simulation Modeling.**

Here, the system is simulated using software, typically with the use of a simulation language. This approach requires more information on user workload than that required by analytic models. As a result, this method is comparatively more accurate. The error introduced is usually less than 10%. Simulation models usually require much longer execution time than analytic models to produce a set of results.

(5) **Remote Terminal Emulation (RTE).**

In a RTE environment, a simulator machine emulates actual user workload by communicating with the simulated machine via terminal lines. The emulated workload is typically captured from a real system and stored in a disk file in the simulator machine. The error introduced by this method rarely exceeds 5%. The main disadvantage, however, are that it requires the actual hardware, human resources to set up the hardware and a long run time. Also, the simulator machine used should be at least 1.5 times faster than the system under study; otherwise, the simulator may become a bottleneck during the emulation.

(6) **Real System.**

Although this is the most accurate approach, it is hardly ever used for performance projection. Here, the real system is actually set up with the actual users when the system is calibrated. Because of the difficulties in controlling the workload and the costs of changing the system, this approach is very often considered not feasible.

Our objective in capacity planning is to use an approach that requires reasonable run time and hardware, and produces results of acceptable accuracy.

From the descriptions above, the best approaches are the use of modeling or remote terminal emulation.

2.2 THE CAPACITY PLANNING PHASES

The approaches towards capacity planning that are used most commonly by capacity planners and performance analysts today are analytic and simulation modeling. To date, much research has been dedicated towards refining these techniques to produce higher degrees of accuracy. The advantages of using modeling are that the hardware requirements and the response time in obtaining the results are comparatively less than those for remote terminal emulation. Specifically, modeling does not require any dedicated hardware to be set up. The accuracy of the results provided by modeling techniques is within acceptable range.

The present capacity planning process is divided into three phases; namely, data collection, workload characterization and performance modeling. In the data collection phase, special tools are used to collect system performance statistics. The measurement period varies from two hours to a few days depending on the objective of the study. The workload characterization phase involves manipulation of the measured data to a representation form usable by the modeling tools. This phase typically takes more than one hour (depending on the amount of collected data). The duration of the modeling phase depends on the modeling technique used. As mentioned earlier, analytic modeling requires only a few seconds of execution time, while simulation time usually takes more than one hour (depending on the length of simulated time specified).

Most capacity planning questions require several iterations of the process before answers can be provided. Specifically, either the workload characterization

or the performance projection process may have to be repeated several times. For example, if jobs are not grouped properly into job classes, the validation results can be inaccurate. In this case, the analyst has to try different combinations of grouping until acceptable results are obtained. As another example, to project the maximum workload supported by a system, the analyst needs to increase the workload gradually for each projection until the projected results show system saturation.

In our refined capacity planning process we elect to use both analytic and simulation modeling. As a result, our workload representation must fulfill both of their requirements, as well as the general requirements of workload characterization. Most important of all, the workload must be completely characterized and represented based on measured data so that no guess work on any part of the workload is necessary. Details of the requirements will be given in subsequent chapters.

The refined capacity planning process is divided into four phases. This process is faster when compared to the traditional capacity planning because the time required for any repetitions required in the process is reduced. The phases are data collection, data reduction, user workload classification and performance analysis. The purpose of dividing workload characterization into data reduction and workload classification is that classifying reduced data merely requires several seconds of CPU time, while analyzing raw measured data can take over one hour. The choice of using either analytic modeling or simulation modeling is left to the analyst. A guideline would be to use analytic modeling to narrow to a target configuration, (for example, finding the amount of memory required for a targeted response time), and then use simulation modeling to obtain more accurate statistics. This is because the response time of analytic modeling tools is typically in a matter of seconds, while the response time of simulation is typically hours.

Data are pipelined from each phase to the next buffered by disk files. An illustration of the data flow is given below:

Event Data ==*condenser*==> Reduced Data ==*user_class*==> Model Data

where the tools that manipulate the respective data are embedded in the arrows (in italics). The event data is collected and dumped by the system event monitor, typically onto a tape. *Condenser* processes and reduces the raw event data into reduced data. *User_class* manipulates the reduced data to produce model data useful for modeling tools. The modeling tools use the model data to do performance validations and predictions.

2.2.1 DATA COLLECTION

During this phase, a system event-driven software monitor is used to record traces of selected events whenever they occur. These traces, known as *event data* are usually dumped onto a tape because its size is normally very large. Prior to data collection, *software probes* must have been placed at pre-selected locations in the system. Each event has two probes to respectively indicate the start and end of the event. Each event record includes the CPU clock and the elapsed time clock so that statistics on the event can be calculated offline during the next phase. To minimize the monitor's overhead, all the information associated with an event record must be readily available in the system.

Note that this technique of data collection is far more accurate and introduces less system overhead than other measurement tools such as sampling tools and benchmarks. Sampling tools often do not provide enough information required by modeling techniques. Examples are CPU burst time and page fault service time. This is because such tools merely examine existing system meters and counters at the end of each sampling period. In order to obtain more detailed information, they have to perform computations during the sampling period;

thus, introducing extra overhead. Most benchmarks merely consist of command script files. During the benchmark runs, extra overhead results from reading and timing the scripts (e.g. using the UNIX's *time* command). In general, benchmarks are only useful for single-user calibration. They cannot be used for multi-user environment, nor for I/O bound environment.

There are also hardware measurement tools that can be used for data collection (see [3] for details). The most common of them are hardware monitors. Although they are more accurate and efficient than software monitors, they lack flexibility. Also, the placement of hardware probes is restricted to hardware level (i.e. difficult to interact with the operating system to obtain all the required user statistics).

2.2.2 DATA REDUCTION

Event data recorded by the monitor is typically very large. Each analysis through the raw event data typically takes one or more hours. The reason for this is due to the slow speed of tape reads (or even disk reads). As mentioned earlier, performance analysts often need to go through the data several times for analyses. It is, therefore, desirable that the data is condensed so that each analysis can be done in a few minutes. A tool known as *condenser* is introduced in our capacity planning process to perform the task of data reduction; the resulting data is known as the *reduced data.*

The data reduction process involves computing and aggregating all the statistics associated with all the monitored events. Details of the data reduction phase will be given in the subsequent chapters. The resulting reduced data is dumped into a binary file to be used for workload classification. An ASCII readable form of the reduced file is also printed for the analyst's convenience. Note that all the aggregate statistics given by *condenser* are on a per user basis.

Appendix A gives a complete description of *condenser*.

2.2.3 WORKLOAD/USER CLASSIFICATION

The purpose of this phase is to provide workload data that fulfills the input requirements of both the analytic and simulation modeling tools. A tool known as *user_class* (user classification) takes the reduced data from *condenser* and groups the workload of users or jobs into classes. The objective here is to group users of the same workload characteristics into the same class. A common example is to group users running the same application program into the same class. The performance analysts are given the option of classification by selecting specific users (identified by their user numbers) or letting *user_class* classify the users workloads according to a prespecified formula.

Each run through the reduced data typically takes several seconds, depending on the number of users and the effective speed of the input/output operations. The *model data* produced by *user_class* contains representation of the user workload derived from the measured data. A tabular readable form of the model data is also given by *user_class* for the analyst's convenience.

Appendix B contains a complete description of *user_class*.

2.2.4 MODELING AND PERFORMANCE ANALYSIS

During this phase, the model input data from the user classification phase is used by the analysis tools for validation and projection. The input data is divided into two categories, namely, *configuration data* and *workload data*. The configuration data consists of information on the measured and projected system configurations, while the workload data is a representation of the user workload.

During the validation process, the analyst merely sets the configuration data for both the measured and projected systems to be identical. The results given by the modeling tools should be reasonably close to those given by *user_class* (i.e. the measured results) before the validation process is considered successful. Although the validation process may not be necessary for a well-known and proven model of a particular system, it is often carried out to ensure that the classification of the users has been done properly.

In the projection process, the configuration data for the projected system is set accordingly. The results given by the analysis tools is then analyzed and compared to the capacity planning objective. If the objective is not met, the model data is modified and the projection process is repeated.

Note that during either the validation process or the projection process, the workload data need not be modified nor adjusted. The values of the workload parameters required by the modeling tools are obtained from the measured data. As a result, the analyst need not do any guess work or use published results of other installations as part of workload data.

In general, the choice of the modeling approach is left to the capacity planner. Depending on the objective and scope of his analysis, he can use existing modeling tools (simulation or analytic tools, or both), or he can develop his own tools based on existing algorithms/methodologies, or he can implement a new model entirely from scratch. As a simple example, an analytic modeling tool known as *qnets*, based on the linearizer algorithm, is described in chapter 5.

CHAPTER 3
WORKLOAD CHARACTERIZATION

The phases of data reduction and workload classification in our capacity planning process is collectively called workload characterization. In general, workload characterization is the quantitative representation of the hardware and software resources utilized by users in a computer system. This chapter discusses the requirements of workload representation and how they can be fulfilled using data reduction and workload classification.

3.1 REQUIREMENTS

For our capacity planning process, the requirements of workload characterization will be geared towards the requirements of both analytic and simulation modeling. These requirements are:

(1) Elapsed Time Independence.

This means that the workload representation should remain relatively invariant regardless of the measurement period, provided that the system is in steady state and the measuring period is not too short (e.g. more than 1 hour). Elapsed time independence can be achieved by representing workload on a per transaction per user basis. For example, the average CPU demand per transaction by a single user remains constant regardless of the duration of the measurement period. This assumes that the system is in, or near steady state.

(2) Representativeness of All Resources.

The resources used in the system must be properly and accurately represented. Note that this requires accurate measured data on the resources to be represented. Data on the resources used can be measured

using an event-driven monitor. Software probes are inserted into properly pre-selected areas. Workload data can then be derived from statistical information measured by these probes. The representativeness of the workload data can be verified from the results of the modeling tools.

(3) Independence on the Number of Users.

In this case, the workload characteristics of a single user should not vary regardless of other users in the measured system. By representing workload on a single user basis, the dependence on the number of users is removed. This also assumes that the workload representation does not contain statistics due to contention.

(4) Linearly Dependence on Hardware Speeds.

In other words, it should be possible to linearly extrapolate the workload data collected on one system configuration to that of another configuration based on the relative speeds of hardware (see [3] for details). As long as the hardware speeds are provided to the modeling tools, the extrapolation is simple and can be done by the modeling tools. For example, if the CPU service time for a measured machine is n milliseconds, it will be n/2 on a machine that is twice as fast.

(5) Flexibility.

The workload representation should allow for easy modification to reflect variations in the real system. For our purpose, we will restrict the flexibility. Our workload representation will be divided into two parts. The first part can easily be modified based on changes in the system configuration. The second part is the representation of the invariant user workloads.

(6) Compactness.

The degree of detail that a workload is represented. A more compact

model is usually less detailed and less representative. In general, compactness is dictated by the availability of information from the measured data. Software probes are placed to monitor all the resources utilized. Because each event is monitored, it is easy to obtain detailed information on the workload data. Our objective is to collect and represent workload to fulfill the requirements of modeling tools.

3.2 WORKLOAD REPRESENTATION

The data produced by the first three phases of our capacity planning process, namely, data collection, data reduction and workload classification, consist of three categories. They are *measured statistics*, *configuration data* and *workload data.* However, only *condenser* and *user_class* provide the user with a readable form of these data. The configuration data and workload data is collectively known as *model data.*

3.2.1 MEASURED STATISTICS

The measured statistics serve as a calibration of the measured systems. In other words, they help the analyst assess the system performance. They are dependent on system configuration and workload. The analyst can also use them directly to validate modeling tools by comparing the measured statistics with the output from the modeling tools.

The measured statistics are given in two forms: system wide, and per user/class per transaction type basis. Explanation for transaction type will be given in section 3.3. The system wide or *global* statistics are as follows:

(1) **System throughput**

This is equivalent to the *system arrival rate* for a system in steady state.

It is the rate at which jobs or transactions are being serviced at the system.

(2) **Device utilizations**

The percentage of time each device is busy during the measurement period.

(3) **Page fault rate**

The rate at which page faults occur in the system.

The second form of statistics which are on a per user/class per transaction basis are as follows:

(1) **Response times**

On the average, the amount of real time it takes to complete a transaction.

(2) **Throughputs**

The rate at which transactions are being serviced at each service center. For terminals (which is essentially a delay service center), this is the rate at which transactions are being generated.

3.2.2 CONFIGURATION DATA

The configuration data is a representation of the system configuration. It is divided into two parts, namely the measured system configuration data and the projected system configuration data. The former set of data is for the system where data measurement was previously done. The latter set is for the system whose performance we wish to project.

The components of each set of configuration data are:

(1) **CPU type**

The name of the processor or alternatively, the processor speed.

(2) **Memory size**

The size of the memory in the system.

(3) **Disks**

The number of disk controllers, and the number of disks associated with each controller.

(4) **User classes**

The number of job classes, and the number of users in each class.

3.2.3 WORKLOAD DATA

The workload data is the actual representation of the user workload in the measured system. It is also made up of two parts. The first part represents the resource demands of the users or job classes. The second part describes the behaviour of the service centers (except the CPU processor).

The demand on a resource by a user or a job class is represented by the average resource service time, and the rate of demand. Note that the representation is on a per user basis (actually on a per transaction type basis as well). The resource demands represented are:

(1) **CPU demand**

The average CPU burst time and the number of CPU bursts per transaction.

(2) **I/O demand**

The average I/O service time and the number of I/Os per transaction.

(3) **Page fault CPU demand**

This is for CPU used to service page faults. The representation is the

average page fault CPU burst time and the number of page fault CPU bursts per transaction.

(4) **Think time**

The average interval time at which jobs are generated at the terminal. This can also be viewed as the average time a user spend "thinking" before he generates a transaction.

The resource behaviour is also represented by the resource service time and the rate the resource is used. However, the representation is on a per center per transaction basis.

(1) **True disk I/O**

A true disk I/O operation is any I/O that is not due to page fault. The representation is the average I/O service time and the rate of I/O requests per disk per transaction.

(2) **Page fault disk I/O**

A page fault disk I/O operation is any I/O that is caused by a page fault. The representation is the average page fault I/O service time and the rate of page fault I/O per disk per transaction.

3.3 TRANSACTION CLASSES

In the industrial world, computer users' perception of response time is sometimes different from that reported by performance analysts and capacity planners. As an illustration, consider an environment where there are terminal users running simple commands and large compilations simultaneously. For the analysts, the response time is often expressed in terms of the average response times of the commands and the large compilations. It may be difficult to assess the system performance based on the average response time alone. Also, from the performance point of view, the users are usually less concerned with the response times of large compilations, and are more interested in the response times of these simple commands.

The inadequacy of average response times was addressed in E. Lazowska's thesis [10]. For our purpose, we will use the following simple illustration. An average response time of 10 seconds, for example, could mean that the simple commands take an average of 10 seconds to complete when there are few large compilations. This would indicate that the system is performing poorly. On the other hand, this response time could also indicate excellent system performance if there are many large compilations. In this case, the response time for the simple commands would be very small.

As a solution to the above problem, it is necessary to classify user transactions into different classes based on their resource demands. This should not be confused with the user workload classification mentioned in the capacity planning process. Transaction classification is essentially a subclassification of the user workload. At the moment, the capacity planning tools subclassify transactions into three types, namely, *micro transaction*, *normal transaction* and *large transaction*. Details of these transaction types are given below, assuming a 1MIP machine.

A micro transaction is any transaction that utilizes less than 10 milliseconds of pure CPU usage. Examples are keystrokes commands of visual editors (e.g. *EMACS*, *vi*), and trivial *UNIX* commands such as *date* and *echo*.

A normal transaction is any other transaction that utilizes less than 100 milliseconds of pure CPU time. Most commands and small programs in any operating system fall into this type. Typical examples are *UNIX* commands *ls* and small compilations.

A large transaction is any transaction that uses more that 100 milliseconds of CPU time. Most commercial packages, large compilations and scientific applications constitute large transactions.

CHAPTER 4
RESOURCE DEMAND REPRESENTATION

As described in the last chapter, the resources used in a system will be represented by the service times and the frequency of use. This representation is used most commonly by performance analyst. The only resource that cannot be represented using this representation is the system memory. Description of memory representation will be deferred to the next chapter. This chapter describes how resource utilization data are manipulated and represented during the data reduction and workload classification phase.

4.1 EVENT TRACE

In order to produce meaningful data to the *condenser*, the software monitor must record specific information. This information includes:

(1) **Event Group**

This is used to distinguish the system events. Each event group has its own characteristics, usually very distinct from other groups. Examples of event groups are transactions, page faults and I/O events.

(2) **Event Type**

For the purpose of *condenser*, this is primarily used to define the occurrence and duration of any event group. The two possible event types are *start event* and *end event*. The former type denotes the actual start of an event group, while the latter denotes the end of an event group. Note that, in general, software monitor typically have more than two event types for various uses.

(3) **Real Clock**

The system real time clock that gives the elapsed time.

(4) **CPU Clock**

The CPU processor clock that gives the total processor time used. Most systems maintain a CPU clock for each user.

(5) **Auxiliary Information**

This varies depending on the event group. For example, for an I/O event, the auxiliary information should include a drive number that identifies the disk where the I/O is taking place.

In order to record the occurrence of events, software *probes* are placed in appropriate places in the system software. At the occurrence of each event, the probe will result in a subroutine call to record all the information related to the probe, i.e. the event group, the event type, the system real time clock, and the CPU clock for the user, and some auxiliary information. More detailed information on the format and contents of an event record are given in Appendix A.

For an easier understanding of the process, an example of a typical trace will be used. Consider the following trace for a particular user:

	Event Group	Event Type	Real Clock (ms)	CPU Clock (ms)	Aux
1	Trans	start	2179	30	
2	PF	start	2182	32	
3	I/O	start	2185	35	Disk1
4	I/O	end	2190	38	Disk1
5	I/O	start	2190	38	Disk1
6	I/O	end	2193	41	Disk1
7	PF	end	2198	46	
8	I/O	start	2220	60	Disk2
9	I/O	end	2225	63	Disk2
10	Trans	end	2300	85	
11	Trans	start	6015	85	
12	I/O	start	6050	100	Disk2
13	I/O	end	6055	103	Disk2
14	Trans	end	6207	241	

where both the clock values are given in units of milliseconds.

4.2 DATA REDUCTION

The primary function of *condenser* is to gather and accumulate statistics in between start and end events of all the desired event groups. Details of the required statistics was given earlier in Section 3.2.

The basic operation used to calculate most statistics is to simply compute the difference in the clock values given in a corresponding start and end event of a transaction group. From the trace given above, for example, the first page fault event group (given by event 2 and 7) uses 14 milliseconds of CPU and takes 16 milliseconds of real time. Note that the 14 milliseconds of CPU time also include 6 milliseconds used to perform two page fault disk operations.

A revised approach is used to calculate the statistics in fragments and attribute them to the appropriate event groups. For simplicity, we will only consider the CPU statistic. First, we define a *cpu slice* to be the CPU time used between **any** *two events.* In the above example, the page fault event has 5 CPU slices. Two slices were used to perform the corresponding page fault disk operations. As a result, the page fault event uses 8 milliseconds of CPU time to perform the page fault and 6 milliseconds to perform the associated disk operations. The elapsed time (or response time) can also be calculated in a similar manner.

Note that we are also interested other statistics such as the number of other events within a particular event group. Examples are the number of I/Os for a page fault and the number of page faults per transaction. These statistics can easily be calculated by counting the occurrences of those events within an event group.

Aggregate statistics are then calculated by adding and averaging all the above statistics on a per transaction basis. Some of the average statistics of the two transactions given in the above table is listed below:

Number of CPU bursts = 4

Average CPU burst Time = 51.25 ms

Number of CPU bursts per transaction = 2

Number of page faults = 1

Number of I/Os per transaction = 2

Average page fault burst time = 3.0 ms

Number of I/Os per page fault = 2

Average disk service time per disk = 3.0 ms

Total visits to disks = 2

Average paging disk sevice time per disk = 2.75 ms

Total visits to paging disk = 2

Average Think time = 3715 ms

Average Response Time = 0.1565 seconds

Throughput = 0.4965 transactions per second

CPU utilization = total CPU used / elapsed time = 5.238%

A more detailed example of the output statistics given by *condenser* is given in Appendix A.

NOTE: A *CPU burst* is defined as the CPU time used between true I/O events (i.e. excluding I/Os due to page faults), or between the start of a transaction and the start of a true I/O event, or between the end of a true I/O and the end of a transaction event. As an example, a transaction with two true I/Os will have three CPU bursts. In general, a transaction with n true I/Os will have $n+1$ CPU bursts.

Similarly, a *page fault CPU burst* is defined as the CPU time used between page fault I/Os, or between the start of a page fault and the start of a page fault

I/O event, or the end of a page fault I/O event and the end of a page fault. Hence, a page fault with *n* page fault I/Os will have *n+1* page fault CPU bursts.

4.3 WORKLOAD CLASSIFICATION

To perform workload classification, *user_class* has to group the statistics of all the users in a particular class. The process of grouping the statistics is very simple and straightforward. It involves adding the corresponding statistics for all the users and recalculate the average statistics.

As an example, let us assume that from the reduced data provided by *condenser*, there are two users in a class, and there is only one class. For simplicity, let us consider only the CPU statistics. User A requires an average of 8.2 milliseconds of CPU burst time, and has a total of 200 CPU bursts and 20 transactions. User B requires an average of 9.0 milliseconds of CPU burst time, and has a total of 170 CPU bursts and 21 transactions. By grouping them into a class, they used up a total of (8.2*200) + (9.0*170) or 3170 milliseconds of CPU. Also, they have a total of 370 CPU bursts and 41 transactions. The resulting average statistics will be an average of 8.57 CPU burst time, and 9.02 CPU bursts per transaction.

All other statistics, except memory statistics, can be obtained in a similar manner. The average statistics calculated using this grouping and averaging process are used directly to represent the workload of all the user in a class, as described in Chapter 3. Detailed output and model data provided by *user_class* are given in Appendix B.

CHAPTER 5
MEMORY REPRESENTATION

The representation of memory demand by users have always been a problem mainly because very few tools can accurately and efficiently collect data on memory demand. In our capacity planning process, we elect to use Chamberlain's lifetime equation [1] that has long been proven to approximate the lifetime behavior. This equation is given below:

$$L = \frac{2b}{1+\frac{c^2}{m^2}}$$

where L is the lifetime, m is the active memory held by a user, and b, c are constants of the equation. The lifetime is defined as the mean CPU time used between successive page faults in a user transaction.

5.1 DATA COLLECTION

The system monitor should provide the lifetime values L_i and the active memory m_i for each user at every page fault. For systems where the active memory held by each user is not easily obtainable (i.e. without causing unnecessary overhead), one can set m_i to be the average available memory. This can be done by dividing the total memory by the number of active users during a particular page faults. Note that this requires the monitor to register the number of active users at every page fault event and it usually under-estimates m. At the moment, *condenser* indirectly calculates the m values based on information given in the event data. Details on the computation will be given later.

5.2 CURVE FITTING

For each user, given the values of L_i and m_i (where i ranges from one to the number of page faults), our objective is to obtain the values of b and c that fit the lifetime function as closely as possible. Because the sample size or the number of page faults during a data collection period is typically very large, an approximation technique that uses iterative approximation is needed. This means that we cannot use the standard Least Square Approximation because the algorithm requires all the sample data to be available first. After some research, the best possible solution is to apply the absolute deviation technique. A few simple experiments showed that the difference in the results produced by the absolute deviation is within 2% when compared to those by of the least square approximation method.

The second problem is that the lifetime function is not a linear equation, making any method to solve a matrix of equations (required for approximation techniques) non-trivial. However, it is possible to translate the lifetime function to a linear equation and the b, c values obtained using substitution.

The entire derivation process is given below, using the absolute deviation approximation:

Given a set of data L_i and m_i where i=1,...,n and n is the total number of page faults, we want to minimize the absolute error of the following:

$$\sum_{i=1}^{n} \left| L_i - \frac{2b}{1+\frac{c^2}{m_i^2}} \right|^2 .$$

Using the substitutions:

$$u_i = \frac{-L_i}{m_i^2}, \quad v_i = L_i, \quad C=c^2, \quad B=2b,$$

the above lifetime equation becomes

$$v_i - Cu_i = B, \text{ or } v_i = Cu_i + B$$

Since u_i and v_i can be directly obtained from the measured values of L_i and m_i, our objective now is to minimize the error:

$$\sum_{i=1}^{n} \left| v_i - (Cu_i + B) \right|^2$$

For the absolute deviation approximation, the conditions below must hold:

$$0 = \frac{d}{dC} \sum_{i=1}^{n} \left[v_i - Cu_i - B \right]^2 = 2 \sum_{i=1}^{n} (v_i - Cu_i - B)(-u_i)$$

and

$$0 = \frac{d}{dB} \sum_{i=1}^{n} \left[v_i - Cu_i - B \right]^2 = 2 \sum_{i=1}^{n} (v_i - Cu_i - B)(-1)$$

These simplify to the following normal equations:

$$C \sum_{i=1}^{n} u_i^2 + B \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} u_i v_i \quad \text{———(1)}$$

$$C \sum_{i=1}^{n} u_i + Bn = \sum_{i=1}^{n} v_i \quad \text{———(2)}$$

From equation (2),

$$B = \frac{\sum_{i=1}^{n} v_i - C \sum_{i=1}^{n} u_i}{n} \quad \text{----- (3)}$$

Substituting (3) into (1), we have:

$$C \sum_{i=1}^{n} u_i^2 + \left[\frac{\sum_{i=1}^{n} v_i - C \sum_{i=1}^{n} u_i}{n} \right] \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} u_i v_i$$

or,

$$nC\sum_{i=1}^{n} u_i^2 + \sum_{i=1}^{n} v_i \sum_{i=1}^{n} u_i - C\left[\sum_{i=1}^{n} u_i\right]^2 = \sum_{i=1}^{n} u_i v_i$$

or,

$$C\left[n\sum_{i=1}^{n} u_i^2 \left(\sum_{i=1}^{n} u_i\right)^2\right] = n\sum_{i=1}^{n} u_i v_i - \sum_{i=1}^{n} u_i \sum_{i=1}^{n} v_i$$

This simplifies to a solution for C:

$$C = \frac{n\sum_{i=1}^{n} u_i v_i - \sum_{i=1}^{n} u_i \sum_{i=1}^{n} v_i}{n\sum_{i=1}^{n} u_i^2 - \left(\sum_{i=1}^{n} u_i\right)^2} \quad \text{———(4)}$$

Substituting C into the equation (2), we have:

$$\left[\frac{n\sum_{i=1}^{n} u_i v_i - \sum_{i=1}^{n} u_i \sum_{i=1}^{n} v_i}{n\sum_{i=1}^{n} u_i^2 - \left(\sum_{i=1}^{n} u_i\right)^2}\right]\sum_{i=1}^{n} u_i + Bn = \sum_{i=1}^{n} v_i$$

or,

$$n\sum_{i=1}^{n} u_i \sum_{i=1}^{n} u_i v_i - \left(\sum_{i=1}^{n} u_i\right)^2 \sum_{i=1}^{n} v_i + Bn\left[n\sum_{i=1}^{n} u_i^2 - \left(\sum_{i=1}^{n} u_i\right)^2\right] = n\sum_{i=1}^{n} u_i^2 \sum_{i=1}^{n} v_i - \left(\sum_{i=1}^{n} u_i\right)^2 \sum_{i=1}^{n} v_i$$

giving,

$$B = \frac{\sum_{i=1}^{n} u_i^2 \sum_{i=1}^{n} v_i - \sum_{i=1}^{n} u_i \sum_{i=1}^{n} u_i v_i}{n\sum_{i=1}^{n} u_i^2 - \left(\sum_{i=1}^{n} u_i^2\right)^2} \quad \text{———(5)}$$

Since the cumulative sums $\sum_{i=1}^{n} u_i$, $\sum_{i=1}^{n} v_i$, $\sum_{i=1}^{n} u_i v_i$, and $\sum_{i=1}^{n} u_i^2$ in equations (4) and (5) can easily be calculated iteratively, with n=1, 2, 3, ..., none of the previous values of u_i and v_i (i.e. the samples) need to be stored. Furthermore, at each iteration, only the new values of the cumulative sums in the previous iteration are needed. At the end of the iterations (i.e. n = sample size), B and C can be computed

using equations (4) and (5).

5.3 DATA REDUCTION

The tool *condenser* simply uses the results presented in the previous section. At every page fault, the values of u_i, u_i^2, u_iv_i, and v_i are computed from the measured values of L_i and m_i, using the substitution:

$$u_i = \frac{-L_i}{m_i^2}, \qquad v_i = L_i$$

The computed values are added respectively to obtain their cumulative sums. When *condenser* has gone through all the event data, we will have the values of $\sum_{i=1}^{n} u_i$, $\sum_{i=1}^{n} u_i^2$, $\sum_{i=1}^{n} u_iv_i$, and $\sum_{i=1}^{n} v_i$. We can now compute the values of B and C using the equations (4) and (5) given in the previous section. Finally, the constants b and c of the lifetime function can easily be computed using the following substitutions:

$$c = \sqrt{C}, \qquad b = \frac{B}{2}$$

The values of L and m are not directly available from the event data. The lifetime between two page faults, i.e. L_i, is calculated by adding all the true CPU burst times that occur during that interval.

The computation for the m values requires more data from the system monitor. First, the system monitor should dump the page table at the beginning of the event data. The page table dumped is merely an array of pages with the owners' user numbers. From this table, *condenser* can easily determine the number of pages owned by each user. When the *condenser* is processing the data, it keeps track of the number of active pages held by each user at each page fault. In other words, at the end of every page fault, *condenser* will decrement the number of pages held by the owner of the page replaced, while incrementing

that of the owner who has just paged in the page. Note the data for each page fault end event should include the page number, and the owner of the page replaced. If the owner cannot be easily determined by the event monitor, a copy of the page table should be included in the header of collected data to be used by *condenser* to determine page owners at page fault events.

5.4 USER CLASSIFICATION

The workload/user classification process of the lifetime function b and c parameters involves using the absolute deviation approximation. The main purpose is to determine one value of b and c for each user class from all the b and c values of all the users in that class.

The algorithm used by the tool *user_class* for each user class is as follows: First, values of m_i, with i=1,..,10000, are generated. These values of m_i range from 0 to the maximum memory. Using the Chamberlain's lifetime function and the b and c constants for each user in the class, values of L are generated. This results in 10000*j pairs of L_{ij} and m_i, where j is the number of users in the class. The absolute deviation approximation technique is then applied to these values. The final values of b and c are finally calculated using substitutions from the solutions to the equations (4) and (5) given earlier. In other words, we use only 10,000 samples to calculate the values of b and c for each user.

CHAPTER 6
MODELING AND PERFORMANCE ANALYSIS

The main task of modeling is to represent a system using an abstract model. The model should contain only factors that are essential to the system's performance behavior. Statistics known as *performance indices* are produced by models to describe system performance.

The modeling and performance analysis phase (or *modeling cycle* as described in [9]) involves two steps. These steps are *model validation* and *performance projection.* During model validation, a system is carefully parameterized and the model is evaluated to ensure that the performance indices given by the model are within acceptable range from the actual measured statistics. During performance projection, the model is used to project the performance of another system with different configuration. If the model is under development, the analyst should also validate the projected statistics. In this case, the projected system is first configured and measured. The measured statistics are then compared to the projected statistics given by the model. Details of these steps are given in later sections in this chapter.

The efforts of research scientists have contributed to a vast variety of performance models. Some of these models are fairly general and some are customized for specific systems. Understandably, the general models are comparatively less accurate than the specific models. To preserve the generality of our refined capacity planning process, we leave the choice of the models to the capacity planner.

6.1 MODELING APPROACH

As mentioned in chapter 2, the two most common modeling techniques used today are simulation modeling and analytic modeling. Both of these techniques have their advantages and disadvantages. The choice of these techniques is also left to the capacity planner; *condenser* and *user_class* work for either modeling techniques.

We will use an analytic model to illustrate how the characterized workload produced by *user_class* can be used. The analytic modeling tool used is known as *qnets* and is based on Linearizer [4].

6.1.1 SIMULATION MODELING

The main purpose of this approach is to simulate the system's behavior in the time domain using software. The software is typically implemented using a special simulation language such as SIMULA and SIMPL. The main advantages of simulation modeling are that it is flexible and accurate. In general, the accuracy provided by simulation models is within 10%. The main disadvantage is that it is expensive (i.e. it requires a lot of CPU time and memory).

6.1.2 ANALYTIC MODELING

Analytic modeling involves representing a system using a set of mathematical equations or a *mathematical model.* The derivation of the model can be based on queueing theory or operational analysis[8]. The solutions to the equations can be exact or approximate. In general, exact models are less common and more expensive than approximation techniques.

Compared to simulation modeling, analytic modeling is less accurate because it does not take as many system parameters into consideration. On the other hand, it is very inexpensive (usually requires much less CPU time than

simulation). The accuracy given by analytic models is typically within 10% for utilizations and throughputs, and 10-30% for response times.

The tool *qnets* is based on an approximation technique known as linearizer. The reason for choosing linearizer is that it is fairly general and efficient. The two drawbacks are that it does not model system memory and it assumes processor sharing discipline on all service centers.

In memory modeling, we need to know the page fault rate and subsequently, the resulting visit ratios to disk based on a given amount of memory. To do this, *qnets* first calculates the lifetime using the b and c values provided by *user_class* and the lifetime function described in chapter 5. From the page fault rate (which is the reciprocal of the lifetime value) and disk visit rate due to page fault (from *user_class*), we can easily calculate the new visit ratios to disks.

To support other kinds of service disciplines, such as first-come-first-serve, we merely need to modify equation (1) described [1] that computes the wait times at a server queue. The solutions to various disciplines are available in [6] and [7] and will not be dealt with here.

6.2 VALIDATION

In our capacity planning process, we assume that a chosen model has already been proven to represent the system under study. In this case, the validation step involves setting the configuration data (in the model data given by *user_class*) to correspond to those of the measured system. The model is then evaluated using this model data and the performance indices are compared to the measured statistics. If the difference is not acceptable, the performance analyst will repeat his re-classification of the workload data (using *user_class*) and repeat the model evaluation. This is done until acceptable results are obtained from the model.

6.2.1 ERROR ANALYSIS

For the examples shown below, three systems with different workload environments were measured with a system monitor. The respective workload data were reduced using *condenser* and classified using *user_class*. The model data were then used by *qnets* as input and the performance indices (output) were compared with the measured statistics.

(1) Development environment, with distinct users running EMACS, Pascal, Fortran and Basic.

Environment:	Development
Users:	30
CPU Speed:	4 MIPS
Memory:	8 Mb

UTILIZATION (%)

	Measured	*Qnets*	%Error
CPU	83.8%	96.5%	+15.1%
Disk 0	15.7%	15.2%	-3.2%
Disk 1	4.94%	4.90%	-0.8%
Disk 2	30.4%	30.0%	-1.3%

THROUGHPUTS (transactions/second)

	Measured	*Qnets*	%Error
Class 1	0.399	0.401	+0.5%
Class 2	6.035	6.142	+1.8%
Class 3	7.625	7.867	+3.2%

Class 4	0.398	0.400	+0.5%
Class 5	0.261	0.245	-6.1%
Class 6	7.841	8.057	+2.8%

RESPONSE TIMES (seconds)

	Measured	*Qnets*	%Error
Class 1	1.879	1.820	-3.1%
Class 2	0.226	0.211	-6.6%
Class 3	0.147	0.127	-13.6%
Class 4	1.793	1.741	-2.9%
Class 5	11.47	12.66	+10.4%
Class 6	0.149	0.132	-11.4%

(2) In the commercial environment, the users executed MIS tools and data base queries.

Environment:	MIS Commercial
Users:	16
CPU Speed:	1.7 MIPS
Memory:	4 Mb

UTILIZATION (%)

	Measured	*Qnets*	%Error
CPU	98.9%	95.9%	-3.0%
Disk 0	3.64%	3.70%	+1.6%

THROUGHPUTS (transactions/second)

	Measured	*Qnets*	%Error
Class 1	32.88	33.57	+2.1%

RESPONSE TIMES (seconds)

	Measured	*Qnets*	%Error
Class 1	0.163	0.149	-8.56%

(3) Mixed environment, consists of distinct development users, commercial users and office automation users.

Environment:	Mixed
Users:	24
CPU Speed:	1.7 MIPS
Memory:	4 Mb

UTILIZATION (%)

	Measured	*Qnets*	%Error
CPU	89.1%	86.7%	-2.69%
Disk 0	81.3%	75.8%	-6.77%

THROUGHPUTS (transactions/second)

	Measured	*Qnets*	%Error
Emacs	2.452	2.526	+3.0%
Fortran	0.165	0.174	-5.5%
Pascal	1.900	1.979	+4.2%
OA 1	0.367	0.374	+1.9%
OA 2	0.629	0.640	+1.7%
MIS	14.45	14.44	+0.0%

RESPONSE TIMES (seconds)

	Measured	*Qnets*	%Error
Emacs	0.491	0.450	-8.35%
Fortran	9.513	8.501	-10.64%
Pascal	0.796	0.730	-8.30%

OA 1	3.306	3.002	-9.20%
OA 2	0.788	0.711	-9.77%
MIS	0.095	0.093	-2.11%

6.3 PROJECTION

During performance projection, the performance analyst merely changes the configuration data in the model data file to correspond to the projected system. The performance indices provided by the modeling tool are then analyzed to determine the projected system's performance. If the system performance does not fulfill the capacity planning objective, the analyst will modify the configuration data and repeat the projection until the objective is met.

In the following examples, we used the model data provided in the last section to do performance projection. Actual system configurations that correspond to those of the respective projected systems are also set up and measured. The purpose is to ensure that *qnets* projects system performance with acceptable results.

6.3.1 ERROR ANALYSIS

(1) In the scientific environment, the total number of users are increased from 30 to 60, while the rest of the system configuration remains the same.

Environment:	Scientific
Users:	60
CPU Speed:	4 MIPS
Memory:	8 Mb

UTILIZATION (%)

	Measured	*Qnets*	%Error
CPU	98.5%	99.8%	+1.3%
Disk 0	59.3%	59.3%	+0.0%
Disk 1	13.9%	14.1%	+1.4%
Disk 2	65.9%	66.4%	-0.8%

THROUGHPUTS (transactions/second)

	Measured	*Qnets*	%Error
Class 1	0.517	0.535	+3.5%
Class 2	7.070	7.189	+1.7%
Class 3	13.09	12.60	-3.7%
Class 4	1.955	2.017	+3.2%
Class 5	0.120	0.131	+9.2%
Class 6	14.41	13.87	-3.7%

RESPONSE TIMES (seconds)

	Measured	*Qnets*	%Error
Class 1	8.479	7.826	-7.71%
Class 2	0.769	0.746	-3.00%
Class 3	0.313	0.343	-9.60%
Class 4	2.142	1.983	-7.42%
Class 5	71.05	64.44	-9.30%
Class 6	0.285	0.312	+9.47%

(2) In the commercial environment, the number of users are increased to 32. Also, the CPU is replaced by a faster 2 MIPS CPU. The amount of memory in the system remains the same.

Environment:	MIS Commercial
Users:	32
CPU Speed:	2 MIPS
Memory:	4 Mb

UTILIZATION (%)

	Measured	*Qnets*	%Error
CPU	99.0%	99.9%	+0.9%
Disk 0	5.07%	5.20%	+2.6%

THROUGHPUTS (transactions/second)

	Measured	*Qnets*	%Error
Class 1	26.21	28.01	+6.87%

RESPONSE TIMES (seconds)

	Measured	*Qnets*	%Error
Class 1	0.813	0.735	-9.60%

(3) In the mixed environment, the total number of users is increased to 24. At the same time, the amount of memory is increased from 4Mb to 6Mb.

Environment:	Mix
Users:	24
CPU Speed:	1.7 MIPS
Memory:	6 Mb

UTILIZATION (%)

	Measured	*Qnets*	%Error
CPU	83.3%	89.9%	+7.9%
Disk 0	97.2%	97.2%	+0.0%

THROUGHPUTS (transactions/second)

	Measured	*Qnets*	%Error
Emacs	2.911	2.740	-3.9%
Fortran	0.163	0.153	-6.1%
Pascal	1.101	1.040	-5.54%
OA 1	0.369	0.379	+2.7%
MIS	11.77	13.34	+13.4%

RESPONSE TIMES (seconds)

	Measured	*Qnets*	%Error
Emacs	0.996	01.104	+10.8%
Fortran	28.26	30.55	+8.10%

Pascal	2.635	2.901	+10.1%
OA 1	9.864	9.296	-5.76%
MIS	0.365	0.285	-21.9%

CHAPTER 7
FUTURE ENHANCEMENTS

The tools described earlier assumed that the systems under study are single-processor systems. Enhancements are required to expand the scope of these tools. The following are some of the possible enhancements:

7.1 VIRTUAL PAGE FAULTS

In UNIX 4.1BSD or later systems, it is possible to have virtual page faults. A *virtual page fault* or a *soft page fault* is any page fault that does not result in actual physical I/O. This happens when the page missing from the working set is already in physical memory. Note however, compared to a *true page fault* or *hard page fault* (where a page has to be physically brought in from a secondary paging device), a virtual page fault uses more CPU time to perform the memory to memory page copy.

Virtual page faults are mostly caused by the spawning of child processes. When a child process is being *fork*ed, the child must have a copy of the parent's workspace. Since the parent process is active during the forking process, most of its workspace are still active in memory. As a result, numerous virtual page faults will likely to occur during the workspace copying.

To date, there are no modeling solutions for virtual page faults. It should be pointed out that virtual page faults must NOT be considered as CPU bursts. This is because the number of virtual page faults depends on the memory size, whereas the number of CPU bursts does not. A possible approach towards modeling virtual page faults is to determine the ratio between virtual and true page faults. This ratio depends on the memory size, pre-paging capacity (i.e. the number of pages prepaged), and the number of wired pages per process (i.e. those

pages of a process that one always resident in memory, e.g. file unit table).

7.2 TRANSACTION SUBCLASSES

All analytical modeling solutions today assume that all transactions of a user or job class are identical. As a result, the response times provided by these models are expressed as the average response times of all these transactions. Because transactions need to be subclassified (as described in Chapter 3), analytical solutions are required to model transaction classes.

7.3 CHARACTERIZING SEMAPHORES

Delays due to semaphore waits can contribute significantly to response times, especially if there is much contention on semaphores. For systems such as Berkerley's UNIX systems, semaphore operations are emulated using system calls *sleep()* and *wakeup()*. Semaphore contention is very common in data base management environment.

Our workload characterization process can easily be modified to represent semaphore contention. The representation will be CPU time used by a user while holding the semaphore, and the rate at which he waits on semaphores. This representation can easily be used by simulation modeling tools or analytical models.

7.4 MULTIPLE PROCESSORS

All the capacity planning tools described earlier assume that the systems under study are single-processor systems. The tools can be readily enhanced to characterize workload for multiple-processor system provided that the system event monitor associates each event with the responsible processor. In other

words, each event record should include a number indicating the associated processor.

7.5 NETWORKS and DISTRIBUTED SYSTEMS

Characterizing networks and distributed operating systems workload is an area that requires much work. In order to use tools such as *condenser* and *user_class* to characterize such workload, it is essential to have a network event monitor that does not introduce excessive overhead. A few issues regarding network event monitoring are addressed by Lamport [11] and Chen [10]. Because the events monitored in a distributed environment is different from those in a stand-alone system, separate tools similar to *condenser* and *user_class* have to be developed.

7.6 CLUSTERING TECHNIQUES

New clustering techniques are required to take into consideration the memory demand representation of user workload. Clustering is especially useful during the workload classification phase when the analyst is not able to classify the workload by mere selection of user numbers. The clustering process involves grouping users with similar resource demands into the same job class.

If the workload characterization phase of our capacity planning process is enhanced to take semaphore usage into consideration, a new clustering technique is also required to include semaphore demand.

REFERENCES

[1] D.D. CHAMBERLAIN, S.H. FULLER, and L. Y. LIU, *An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory.* IBM J. Res. Dev. 17, 15 Sept., 1973, 404-412.

[2] DOMENICO FERRARI, *Computer Systems Performance Evaluation.* Prentice-Hall, Inc. 1978.

[3] D. FERRARI, G. SERAZZI and A. ZEIGNER, *Measurement and Tuning of Computer Systems.* Prentice-Hall, Inc. 1983.

[4] CHANDY, K. MANDY and NEUSE, DOUG, *Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems.* CACM 25, 2 (April 1982), 126-134.

[5] R.L. BURDEN, J.D. FAIRES and A.C. REYNOLDS, *Numerical Analysis, 2nd Edition.* Prindle, Weber & Schmidt. 1978, p.318-344.

[6] REISER, M., and LAVENBERG, S.S. *Mean-value analysis of closed multichain queueing networks.* J. ACM 27, 2 (April 1980), p.313-322.

[7] FERRARI, DOMENICO, *Considerations on the Insularity of Performance Evaluation.* IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.

[8] DENNING, P. J. and BUZEN, J. P., *The operational analysis of queuing network models*, ACM Computing Surveys, Vol. 10, No. 3, September 1978, pp. 225-261.

[9] LAZOWSKA, E.D., ZAHORJAN, J, GRAHAM, G. S., and SEVCIK, K.C., *Quantitative System Performance*, Prentice-Hall, 1984.

[10] CHEN, Y.F., PRAKASH, A. and RAMAMOORTHY, C.V., *The Network Event Manager*, Computer Science Division, University of California, Berkeley, CA 94720, Report No. UCB/CSD 86./299, June 1986

[11] LAMPORT, L., *Time, Clocks and the Ordering of Events in a Distributed System*, CACM Vol. 21, No. 7, pp. 558-565, July 1978.

1. Proposal

1.1 The Problem

The modeling tools of any Capacity Planning process/package (namely the simulation and analytical tools) require specialized statistics to be abstracted from raw event data. These statistics include resource demands on cpu, I/O and the paging device. This document describes the design of a tool that is used as part of the Capacity Planning process.

The next paragraph gives a very high level and brief description of the Capacity Planning process and how *condenser* fits into the process. The reader should be familiar with system event monitoring and general performance analysis before reading this document.

In general, the performance analyst must first collect workload data on an active machine using a system event monitor. When the data collection phase is done, the analyst will use *condenser* to process and to reduce this data for simpler manipulation by *user_class*. The analyst can now use *user_class* to classify/group users and/or to omit certain set of users. Finally, the output of *user_class* can be used by modeling tools to perform validations and performance projections.

Note that it is necessary to have *condenser* because we do not want to use *user_class* to repetitively go through the raw event data. This is because the size of the event data is typically very large. The introduction of *condenser* to condense the event data for *user_class* can subsequently reduce *user_class*'s response.

2. Program Function

2.1 Terminology

Below are brief descriptions of some of the most commonly used terms in this document.

Event Data — The raw binary data produced by a system event monitor. The data is typically in buffers of records, with each record corresponding to an event.

Aggregate Statistics — These are accumulated statistics calculated from all the related event events obtained from the event data.

Transient Statistics — Statistics that are caused by unmatched events. This happens when the event monitor is shutdown and there are events with statistical data still being collected.

Micro-Transaction — A user transaction that uses less than 10 milliseconds of CPU. Examples are *EMACS* or *vi* single character transaction and ***date*** command. The limit of 10 milliseconds may be changed in the future.

Normal Transaction — A user transaction that uses more than 10 milliseconds but less than 100 milliseconds of CPU. This includes most PRIMOS commands. The range of 10 to 100 milliseconds may be changed in the future.

Large Transaction — A user transaction that uses more than 100 microseconds of CPU. A good example is a large compilation run. The limit of 100 milliseconds may be changed in the future.

Window — The user can select a contiguous block of event data to be processed by *condenser*. The window range can be provided in terms of elapsed times or buffer numbers. If both the time and buffer windowing are used, the intersection of the two window ranges will be used.

Reduction — The actual process of calculating aggregate statistics from raw event data, and then the compression of the statistics for *user_class*.

Transaction — Anything that *UNIX* inputs from a terminal and replies. This can be a command line (e.g. *UNIX* command) or a single character (i.e. *emacs* transaction).

Response Time — This is defined as the amount of real time that *UNIX* takes to reply upon receiving a transaction (either a character or a command line).

Think Time — This is defined as the amount of real time between the last *UNIX* response/reply and the next time when *UNIX* actually receive another transaction. This can also be viewed as the *idle* time plus

the typing time.

CPU Burst — The number of times that the user's job uses the CPU during a transaction.

I/O Burst — The number input/output operations done by a user's job during a transaction.

Virtual Page Fault

For *UNIX* 4.1BSD (or above) systems, it is possible to have a virtual page fault. Here, a page fault event occurs but there was no I/O operation because the required page is still in memory.

Physical Page Fault

The actual page fault event that have one or more I/O operations. This is also known as the true page fault.

Login/Logout Transaction

This includes login/logout of terminal, child users. Note that *condenser* will treat login/logout exactly as a normal terminal transaction.

Lifetime Function — An empirical formula that gives the inter-page fault time (actually CPU time) for a given memory demand. The formula of the function is $L(m) = 2*b \ / \ (1 + (c*c)/(m*m))$, where m is the memory demand (or active memory), L is the inter-page fault time, and b, c are the lifetime function parameters to be approximated for each user.

2.2 User Interfaces

Condenser requires several input parameters from the user before it can commence going through event data. The user input requirements are given in detail in the next section. Also, all collected and calculated statistics are written in a user-specified (in user input) output file. Details on the the user output is given in the following subsection below.

2.2.1 User Input

At the moment, the interaction between *condenser* and the user will be on-line. In the near future, the interaction will be screen-oriented provided that the user uses a supported terminal. Note that before running *condenser*, the user must have the event data ready to be used (either on tape or disk).

The command line options for *condenser* are as follows (with abbreviations in boldface):

condenser *[-input <file>] [-output <file>] [-reduce <file>] [-help] [-version] [-windowbuffer] [-windowtime] [-fullhelp]*

Condenser takes event data as input and reduces the data to be used by the Capacity Planning user-classification utility *user_class*. The input may be from a tape or from a disk file. The reduced file will be in binary form, but an ASCII form of the reduced file will be written in the output file.

If no options are given, *condenser* will prompt the user for the necessary set of input options.

Condenser supports the following options:

-input, -i	*-output, -o*
-reduce, -r	*-help, -h*
-version, -v	*-fullhelp, -fh*
-windowbuffer, -wb	*-windowtime, -wt*

The following paragraphs summarize all the *condenser* options, which can be selected in any order.

-input, -i

Takes a file name as an argument (for input file). If this option is omitted, *condenser* will prompt the user for an input file name. Also, *condenser* will always continue to prompt the user if it fails to open the associated file. Currently, the event data on tape is assumed to be at the beginning of the tape.

-reduce, -r

Takes a file name where the reduced data is to be stored. The user will be queried before an existing file is overwritten. If this option is omitted, *condenser* will prompt the user for a reduced file name. The reduced file must be a disk file.

-output, -o

Takes an output file name as an argument. If the output file exists the user will be queried before it is overwritten. If this option is omitted, *condenser* will prompt the user for an output file name. The output file must be a disk file. Note that the output produced by *condenser* is essentially a readable form of the reduced data.

-help, -h

Prints the command line format on how to invoke *condenser.*

-windowbuffer, -wb

Turns on buffer windowing option. The user will be prompted for the starting buffer number and the number of buffers to be processed.

-windowtime, -wt

Turns on time windowing option. The user will be prompted for the starting time in the data and the duration desired. This time is the real time in the event data.

-version, -v

Prints *condenser* version stamp plus the date and time that it was built.

-fullhelp, -fh

Prints this full help information on how to use *condenser.*

A sample run is given below (with user's input in bold face):

```
OK, condenser
[CONDENSER Rev. 1.0 - 1986]
Enter input file name          ? EVENT_DATA
Enter output file name         ? EVENT_OUTPUT
Enter reduced file name        ? EVENT_REDUCE

Monitor started on  05/25/84 15:55:03.512
Monitor Version:  1   on 4.2BSD
Monitor User Name: root   User Number: 30
CPU: VAX 11/750  Memory: 2048 pages  Maximum users: 32
REMARK:

Elapsed time = 1969.285 seconds
Number of events processed = 13310
Total blocks/buffers read = 93
OK,
```

2.2.2 User Output

Each item of the statistics printed by *condenser* in the output file is always made up of two values; namely, the total count and the total usage. For I/O statistics, for example, the total count is the total number of I/O operations and the total usage is the total I/O time used. A description of all the statistics is given below:

Response times — Response times for all interactive users on a per user per transaction basis.

Think times — Think times for all interactive users on a per user per transaction basis.

True I/O — The pure I/O operations excludes any I/O caused by page faults. Any disk queueing statistics are also excluded. This is given on a per user per transaction basis.

Page Fault I/O — The I/O operations caused by page faults only. Any disk queueing statistics are excluded. This is given on a per user per transaction basis.

True CPU — The pure CPU usage (i.e. excludes any CPU time for page faults). This is given on a per user per transaction basis.

Page Fault CPU — The CPU usage for handling page faults only. This is given on a per user per transaction basis.

Physical Page Fault — Page faults that actually cause one or more I/O operations, given on a per user per transaction basis. It should be pointed that the usage statistics for physical page faults are average elapsed times and not service/virtual time.

Virtual Page Fault — Page faults that do not cause any I/O operation at all. This is given on a per user per transaction basis. The usage statistics for virtual page faults are also average elapsed times and not service/virtual time.

Disk I/O — The disk true I/O operations on a per user per disk basis. Queueing at the disks are excluded in the statistics.

Disk PF I/O — The disk page fault I/O operations on a per user per disk basis. Queueing statistics at the disks are excluded.

Login/Logout — The login and logout of terminal users and child processes.

Lifetime Function — This is unlike all the above statistics. The approximated b and c parameters will be printed for each user.

Transient — This is due to the event monitor being shutdown before any end events are encountered (i.e. for those events that have been "started").

A sample output file is given below:

```
Monitor started on  05/25/84 15:55:03.512
Monitor Version:  1   on 4.2BSD
Monitor User Name: root   Monitor User Number: 30
CPU: VAX 11/750    Memory: 2048 pages  Maximum users: 32
REMARK:

Elapsed time = 1969.285 seconds
Number of events processed = 13310
Total blocks/buffers read = 93
Number of users = 7
```

RESPONSE TIMES (seconds)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0	0.000	1	0.039	1	0.033	2	0.036
2	69	1.093	86	7.021	57	12.270	212	6.503
3	1	0.000	19	0.046	42	0.616	62	0.432
TOTAL	70	1.077	106	5.705	100	7.253	276	5.092

THINK TIMES (seconds)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0	0.000	1	5.567	1	2.000	2	3.783
2	69	1.783	86	3.291	56	2.885	211	2.690
3	1	0.052	18	22.449	42	30.794	61	27.828
TOTAL	70	1.758	105	6.597	99	14.717	274	8.295

TRUE I/O WITHOUT QUEUE STATISTICS (ms)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0	0.000	0	0.000	21	31.025	21	31.025
2	0	0.000	0	0.000	253	17.439	253	17.439
3	0	0.000	1	12.121	244	26.391	245	26.333
TOTAL	0	0.000	1	12.121	518	22.207	519	22.187

PAGE FAULT I/O WITHOUT QUEUE STATISTICS (ms)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
2	0	0.000	5	22.424	68	16.488	73	16.895
3	0	0.000	0	0.000	112	18.534	112	18.534
TOTAL	0	0.000	5	22.424	180	17.761	185	17.887

TRUE CPU BURST STATISTICS (ms)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0	0.000	1	36.864	21	5.071	22	6.516
2	69	5.921	91	28.132	569	55.685	729	47.535
3	1	1.024	20	38.810	415	33.597	436	33.761
TOTAL	70	5.851	112	30.117	1005	45.506	1187	41.716

PAGE FAULT CPU STATISTICS (ms)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
2	0	0.000	10	1.229	328	1.861	338	1.842
3	0	0.000	0	0.000	241	1.287	241	1.287
TOTAL	0	0.000	10	1.229	569	1.618	579	1.611

PHYSICAL PAGE FAULT STATISTICS (ms)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
2	0	0.000	5	26.061	68	20.410	73	20.797
3	0	0.000	0	0.000	112	24.729	112	24.729
TOTAL	0	0.000	5	26.061	180	23.098	185	23.178

VIRTUAL PAGE FAULT STATISTICS (ms)

	MICRO		NORMAL		LARGE		OVERALL	
USER	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
2	0	0.000	0	0.000	192	2.857	192	2.857
3	0	0.000	0	0.000	17	3.030	17	3.030
TOTAL	0	0.000	0	0.000	209	2.871	209	2.871

DISK I/O WITHOUT QUEUE STATISTICS (ms)

	DISK 8		OVERALL	
USER	N	AVERAGE	N	AVERAGE
1	116	29.363	116	29.363
2	253	17.439	253	17.439
3	248	26.075	248	26.075
23	724	14.494	724	14.494
24	330	14.105	330	14.105
25	4	23.485	4	23.485
30	118	34.361	118	34.361
TOTAL	1793	18.729	1793	18.729

DISK PF I/O WITHOUT QUEUE STATISTICS (ms)

	DISK 8		OVERALL	
USER	N	AVERAGE	N	AVERAGE
2	73	16.895	73	16.895
3	112	18.534	112	18.534
TOTAL	185	17.887	185	17.887

LOGIN STATISTICS (seconds)

USER NO	NO. TRANS	AVERAGE

LOGOUT STATISTICS (seconds)

USER NO	NO. TRANS	AVERAGE

LIFETIME FUNCTION PARAMETERS

USERS	B	C	AVG MEM
1	4.2064	762.8945	1679.0278
2	0.2990	38.0003	46.5000
3	4.7094	113.9048	167.4931

TRANSIENT STATISTICS (ms)

	I/O		CPU	
USER	N	TOTAL	N	TOTAL
1	95	2754.5455	94	368.6400
3	3	15.1515	3	145.4080
23	724	10493.9394	740	32059.3920
24	330	4654.5455	347	20831.2320
25	4	93.9394	3	63.4880
30	118	4054.5455	117	1562.6240

TRANSIENT TIMES (seconds)

USER	RESPONSE	THINK
1	0	247270
2	17185	1
3	176	11

TRANSIENT LOGIN/LOGOUT (seconds)

USER	LOGIN	LOGOUT
2	7361	0

2.3 Program Interfaces

The most important ones are the layouts of the raw data that *condenser* reads and writes. These data are the event data read by *condenser* and the reduced aggregate that *condenser* writes for *user_class*.

2.3.1 Event Data

The layouts event data, which can either be on a tape or disk, must be made up of fixed size buffers. Each of these buffer is of fixed size (usually 4096 bytes). Every buffer is made up of event records. The format of each record is as follows:

NAME	SIZE	DATA TYPE
Length	2 bytes	binary short
Event Group	1 bytes	binary
Event Type	1 byte	binary
User Number	2 bytes	binary short
CPU time	4 bytes	binary long
Real time	4 bytes	binary long (microsecs)
Auxiliary Information	Length-18	Variable

The first one or two records of each set of event data must contain header information. All header records must have an associated event group of 0. The layout of the header (i.e. auxiliary information of the header event record) is given as follows:

NAME	SIZE	DATA TYPE
Event record	18 Bytes	See above for layout
Date	6 bytes	ASCII (MMDDYY)
Minutes	2 bytes	binary short
Seconds	2 bytes	binary short
Ticks	2 bytes	binary short
Tick Rate	2 bytes	binary short
Monitor user number	2 bytes	binary short
Monitor user name	32 bytes	ASCII
UNIX version length	2 bytes	binary short
UNIX version	16 bytes	ASCII
Memory size	2 bytes	binary short
Number of users	2 bytes	binary short

Following the header buffer will be the contents of *UNIX* page map. Hence, the first buffer will only contain header information.

2.3.2 Reduced Data

Condenser also writes all its statistics to a file to be read in by *user_class*. All data are written in binary format. The statistics is first preceded by *condenser*'s header and the header format is as follows:

NAME	SIZE	DATA TYPE
Month	2 bytes	binary short
Day	2 bytes	binary short
Year	2 bytes	binary short
Hour	2 bytes	binary short
Minutes	2 bytes	binary short
Seconds	2 bytes	binary short
Ticks	2 bytes	binary short
Monitor version	2 bytes	binary short
Monitor user no	2 bytes	binary short
Monitor user name	32 bytes	ASCII
CPU timer	2 bytes	binary short
Real timer	2 bytes	binary short
UNIX version stamp	16 bytes	binary short
CPU type/name	16 bytes	binary short
Memory size	2 bytes	binary short
Number of users	2 bytes	binary short
Number of event events	2 bytes	binary short
Number of event buffers	2 bytes	binary short
Length of remark	2 bytes	binary short
Elapsed time	4 bytes	double
Maximum user number	2 bytes	binary short
Maximum disk number	2 bytes	binary short
Remark	varying	ASCII

Following the header are all the statistical matrices and arrays. Their sizes are dependent on the maximum user number and maximum disk number (which are given in *condenser*'s header). The details of the statistical data are given below:

STATISTIC	DIMENSION	DATA TYPE	DESCRIPTION
recorded	max_user	short	flags for recorded users
resp_tot	NTRANS by max_user	double	total response times
resp_n	NTRANS by max_user	long	number of transactions
think_tot	NTRANS by max_user	double	total think times
think_n	NTRANS by max_user	long	total idle transactions
io_noq_tot	NTRANS by max_user	double	total I/O without queue usage
io_noq_n	NTRANS by max_user	long	total I/Os without queue
pf_io_noq_tot	NTRANS by max_user	double	total PF I/O usage
pf_io_noq_n	NTRANS by max_user	long	total PF I/Os
cpu_tot	NTRANS by max_user	double	total true CPU burst time
cpu_n	NTRANS by max_user	long	total no. of true CPU bursts
pf_cpu_tot	NTRANS by max_user	double	total PF CPU burst time
pf_cpu_n	NTRANS by max_user	long	total no of PF CPU bursts
disk_noq_tot	max_drive by max_user	double	total true disk I/O usage

disk_noq_n	max_drive by max_user	long	total no of true disk I/Os
pf_disk_noq_tot	max_drive by max_user	double	total PF disk I/O usage
pf_disk_noq_n	max_drive by max_user	long	total no of PF disk I/Os
lftb	max_user	double	*b* parameter
lftc	max_user	double	*c* parameter

3. Program Design

3.1 Design Overview

The general algorithm of *condenser* is to match *start* and *end* event types of the associated event group and calculate the appropriate statistics. An overview of the algorithm is as follow:

```
Process command line
User Input from terminal
Get event data header
    Get page map
    Compute active memory size for each user
Allocate storage
while more event records
    Classify Event_Group
    Classify Event_Type for each Group
        If Start_Event store timer values
        If End_Event calculate statistics by
             current timer values - stored timer values
Spread transient statistics
Compute aggregate statistics
Print event header and all statistics
Reduce statistics
Cleanup transient statistics for user_class
```

3.2 Internal Data Structures

3.2.1 Constants

The following constants are used to define the dimensions of matrices and arrays.

```
#define MAXUSRPLUS     257      /* maximum number of users + 1        */
#define MAXDRIVESPLUS   17      /* maximum number of disk drives + 1  */
#define NTRANSPLUS       4      /* number of transaction classes + 1  */
```

3.2.2 Types

The structures below are used to store special statistics such as memory.

```
typedef struct lt_struct                /* lifetime function structure   */
   {
      double    lftp0,                  /* below 5 are cumulative        */
                lftp1,                  /*   used to compute the final   */
                lftq1,                  /*   two parameters.             */
                lftr0,
                lftr1,
                lftb,                   /* "b" parameter                 */
                lftc;                   /* "c" parameter                 */
   } LFTSTRUCT;
```

3.2.3 Data Structures

For each set of statistical data collected (e.g. CPU usage), there are always two associated values; namely, the total usage (e.g. total CPU usage) and the total number of transactions (e.g. total number of CPU bursts). Except for login/logout statistics, all other statistics are given on a per user per transaction basis (or per user per disk basis for disk statistics). The login/logout statistics are given merely on a per user basis.

```
 /*  cumulative/aggregate statistics per user per transaction */
double **cpu_tot,           /* cumulative CPU usage             */
       **resp_tot,          /* cumulative response times        */
       **think_tot,         /* cumulative think times           */
       **io_noq_tot,        /* cumulative I/O without queue     */
       **pf_io_noq_tot,     /* cumulative PF I/O without queue  */
       **pf_cpu_tot,        /* cumulative PF CPU usage          */
       **pf_tot,            /* cumulative physical page faults  */
       **vpf_tot;           /* cumulative virtual page faults   */
long   **cpu_n,             /* total CPU bursts                 */
       **resp_n,            /* total number of command line     */
       **think_n,           /*    transactions                  */
       **io_noq_n,          /* total no. of I/Os without queue  */
       **pf_io_noq_n,       /* total PF I/Os without queue      */
       **pf_cpu_n,          /* total CPU bursts for PF          */
       **pf_n,              /* total physical page faults       */
       **vpf_n;             /* total virtual page faults        */

 /* aggregate disk statistics on per user per disk basis      */
double **disk_noq_tot,      /* aggregate disk usage             */
       **pf_disk_noq_tot;   /* aggregate PF disk usage          */
long   **disk_noq_n,        /* total disk I/Os without queue    */
       **pf_disk_noq_n;     /* total PF disk I/Os "     "       */

double  *login_tot,         /* aggregate login elapsed time     */
        *logout_tot;        /* aggregate logout elapsed time    */

long    *login_n,           /* total times logged in            */
        *logout_n;          /* total times not logged in        */
```

The following are used to store temporary accumulative statistics on a per user basis.

```
 /* temporary statistics on a per user basis only             */
double  *tcpu_tot,          /* accumulative CPU usage           */
        *tio_noq_tot,       /* accumulative I/O no queue usage */
        *tpf_io_noq_tot,    /*   " PF I/O no queue usage        */
        *tpf_cpu_tot,       /* page fault CPU usage             */
        *tpf_tot,           /* page fault service time          */
        *tvpf_tot;          /* virtual page fault service time */

long    *tcpu_n,            /* no. of CPU bursts                */
        *tio_noq_n,         /* no. of I/O without queue         */
        *tpf_io_noq_n,      /* no. of Pf I/O without queue      */
        *tpf_cpu_n,         /* PF CPU bursts                    */
        *tpf_n,             /* no. of physical page faults      */
        *tvpf_n;            /* no. of virtual page faults       */
```

The event header consists of the date and time structure and other system

information. These are all mapped into the following variables:

```
char    month[2],                       /* month of date                */
        day[2],                         /* day of date                  */
        year[2];                        /* year of date                 */
short   min,                            /* minutes of date              */
        sec,                            /* seconds of date              */
        tick,                           /* ticks of date                */
        tic_rate,                       /* tick rate in ticks/second    */
        userno;                         /* event user number              */
char    username[32];                   /* event user name                */
    /* END OF TIMEDAT structure */
short   vlen;                           /* UNIX version stamp length    */
char    version[16];                    /* UNIX version stamp           */
short   cpuid,                          /* index to CPU names           */
        memory,                         /* memory size in pages         */
        nusers;                         /* number of users              */
        mon_version,                    /* monitor version number       */
        cpu_tic_rate,                   /* processor tick rate (Rev 3)  */
    /* END OF event HEADER FORMAT */
```

Each event record will be mapped into the following:

```
short   event_group,                    /*  event event group number      */
        event_type,                     /* event event type number        */
        user_no;                        /* User number caused the event  */
double  cpu_time,                       /* CPU usage in milliseconds     */
        real_time;                      /* elapsed time in milliseconds  */

short   aux_length;                     /* length of the auxiliary info  */
char    *aux_info;                      /* Auxilliary information        */

short   *pagemap;                       /* contains UNIX HMAP page map */
```

3.3 Module Design

For the modules below, a design/execution level number is included to indicate the modules position in the *condenser*'s hierarchical algorithm. A brief description of each module's algorithm is also included.

main(argc,argv) - Level 0

The main program. Does command line processing and calls level 1 routines.

```
int  argc;
char **argv;

Process command line options
User_Input();      /* to process user's terminal input */
Get_Header();      /* Get event's header information    */
Post_Init();       /* Post initialization               */
Driver();          /* Process event data here           */
Transient();       /* spread transient statistics       */
Aggregate();       /* Compute aggregate statistics      */

Print_Output();    /* Print out all statistics          */
Reducer();         /* Reduce all statistics for user_class     */
Cleanup();         /* Clean up transient statistics     */
```

User_Input(ifname, ofname, rfname) - Level 1

Simply prompts the user for the appropriate file names. Also, prompts for window range if the appropriate flags are turned on.

```
char *ifname, *ofname, *rfname;

if input file name (ifname) not given
    prompt the user
if output file name (ofname) not given
    prompt the user
if reduced file name (rfname) not given
    prompt the user
if window_time, prompt for time range
if window_buffer, prompt for buffer range
```

Get_Header() - Level 1

Reads in event header information and stores them in memory. The number of header records read depends on the event revision. The first record is common for all event records. Subsequent records only have useful information in the auxilliary field.

```
read in first record      /* common for all event revs */
get size of page map      /* size of page map dumped   */
discard current buffer    /* for compatibilty only     */
read in page map          /* raw form of page map      */
```

Post_Init() - Level 1

Does any initialization that requires event header information. Specifically, allocate storage for all statistical structures, and initialize them to default values.

Driver() - Level 1

The actual high level driver that classify event event groups.

```
while more event records
  case (event_group) of
     1 : Group1();     /* User command level transaction  */
     2 : Group2();     /* Login/Logout  event             */
     3 : Group3();     /* Page fault event                */
     4 : Group4();     /* Disk I/O event                  */

  end_case;
```

Aggregate() - Level 1

Compute all aggregate statistics by adding all rows and columns of every statistical matrices and arrays. The algorithm is straightforward.

Print_Output() - Level 1

Prints all statistics into the output file. The algorithm is straightforward.

Reducer() - Level 1

Reduce all statistics to minimize space.

```
Prepare and write condenser header
for every matrices
   convert them into an array which can be indexed
      using (i*no_of_columns)+j; where i, j are indices
      of the matrices
write out all converted matrices and arrays.
```

Cleanup() - Level 1

Handles events that were still active when event was shutdown. The algorithm at this stage is to simple dump all the transient statistics for *user_class* to process.

Group1() - Level 2

The user command level event group. The following is done for the user causing the event.

```
if start_event
  reset temporary statistics
else if end_event
  classify transaction according to CPU usage
  copy temporary statistics to global statistics
```

Group2() - Level 2

The event group is user login/logout. This includes login/logout of terminal users, remote users, phantoms and child processes. The following is done for the event user only.

```
call Group1()   /* treat login/logout events as transactions */
if start_event
  reset login temporary statistics
  copy logout temporary statistics to global statistics
else if end_event
  reset logout temporary statistics
  copy login temporary statistics to global statistics
```

Group3() - Level 2

Page fault event group. Under 4.1BSD or higher systems, we can have either a virtual or physical page fault event. Note that the *pf_on* flag for each user can have on of the three values *TRUE*, *FALSE*, and *TRUE_PF*. We only do the following for the event user.

```
if start_event
  set pf_on flag to TRUE
  reset temporary statistics
else if end_event
  if (pf_on is TRUE_PF)
    copy temporary statistics to physical PF statistics
  else copy temporary statistics to virtual PF statistics
  reset pf_on to FALSE
```

Group4() - Level 2

The disk I/O event group. An I/O event can be caused by a page fault or be a simple I/O operation. For the user responsible for the event, we do the following.

```
if start_event
  if (pf_on is TRUE) set pf_on = TRUE_PF
  reset temporary IO statistics
else if end_event
  copy temporary statistics to global statistics
```

Transient() - Level 2

Spreading transient statistics into uniform transactions.

```
for each recorded user
   if temporary cpu usage < 2 * LARGE_LIMIT
      set no_of_trans to 1
   else  /* have a transaction every 100 seconds */
      set no_of_trans to elapsed_time / 100.0
   spread out all other statistics into global statistics
      namely, cpu, pf_cpu, io, pf_io
   add to resp_tot the sum of cpu_tot+pf+cpu_tot+io_tot
   add to think_tot any remaining idle time
   set resp_n and think_n to no_of_trans
```

3.4 Design Issues

The major concern of *condenser*'s design is the memory usage. Because the event header gives the maximum number of configured users, this parameter is used for dynamic storage allocation. The storage for all the statistical matrices and arrays should not be allocated until any user interaction has been completed in order to minimize *condenser*'s startup time. As a result, the storage is allocated in the procedure *Init_Stats*().

For login and logout events, *condenser* will treated them as normal terminal transactions. For example, terminal logins/logouts are treated identically as Group 1 events. Also, a child process (including login through logout) is treated as a user command (i.e. Group 1 event).

3.5 Standards

Condenser is developed to be used by a system monitor on 4.2BSD. If the program is to be used by monitors developed on other systems, it is important that the event data should conform to the format described in earlier sections.

3.6 Implementation Language

The language used to develop *condenser* is the C programming language. The main reasons for using this language is the need for bit and byte manipulation, and the need for address manipulation, and it is well supported in UNIX systems.

1. Proposal

Before reading this document, the reader must be familiar with the functionalities of *condenser* and the requirements of the CAPP package.

1.1 The Problem

All modeling tools require measurable representation of workload as input. The tools *condenser* and *user_class* of CAPP serve to extract workload statistics from event data and produce the input data using workload characterization and classification. In other words, *condenser* primarily deals with workload characterization while *user_class* mainly performs workload classification.

In general, the size of the data produced by a system event monitor is usually too large to allow the user to repetitively go through the data to collect different sets of data. Each set of this data is typically made up of a cluster of users. The process of forming such a cluster is known as user classification.

1.2 Goals and Non-Goals

The primary goal of *user_class* is to allow the user to perform user classification without having to go through the raw event data again. This can easily be accomplished with the use of *condenser* (which can actually be viewed as an event data pre-processor as well). In other words, it is essential that the user can do user classification in a short amount of time.

As for the user classification process, we should allow the user to do arbitrary classification. This allows the user to select specific users (by their user numbers) to different groups. *User_class* will also allow other kinds of user classification, such as automatic classification by workload.

The third goal of *user_class* is to automate the modeling phase of the capacity planning process as much as possible. To accomplish this, *user_class* will *pipeline* data to the modeling tools. In other words, the output from *user_class* can readily be used as input for modeling tools without user intervention or modification to the pipelined data.

Note that *user_class* is not a product by itself. It will only accept data from *condenser*.

2. Program Function

2.1 Terminology

User class — A group of users that share some common characteristics. The most typical characteristic is the user's type of workload or application (e.g. EMACS users or DBMS users).

User classification — The process where users are grouped into different classes. The criteria of the assignment of users to classes can be by user numbers or by the users' workload. Each user can be in at most one class. Note that this is essentially the same as workload classification.

Model data — The data that *user_class* produces to be used as input data by modeling tools. This data is essentially made up of parameter names and parameter values.

Workload characterization — The manner in which we represent workload. In general, workload characteristics include CPU, I/O and memory demands.

2.2 User Interface

To invoke *user_class*, the command line must conform to the following format (with abbreviations in boldface):

user_class *[-**input** ‹file›] [-**output** ‹file›] [-**reduce** ‹file›] [-**help**] [-**version**] [-**model** ‹file›] [-**fullhelp**] [-**classtype** [**user**|**workload**]]*

If no options are given, *user_class* will prompt the user for the necessary set of input options.

User_class supports the following options:

-input, -i	*-output, -o*
-reduce, -r	*-help, -h*
-version, -v	*-model, -m*
-fullhelp, -fh	*-classtype, -ct*

The following screens summarize all the *user_class* options.

-input, -i

Takes a file name as an argument (for input file). The input file should be saved by a previous *user_class* run, and should contain lists of user for all classes. If this option is omitted, *user_class* will prompt the user for classification information.

-reduce, -o

Takes a reduced file name as an argument. This file must be the produced by *condenser*. *User_class* will prompt the user for a reduced file name if this option is omitted.

-model, -m

Takes a file name where the model data is to be stored. The user will be queried before an existing file is overwritten. *User_class* will prompt the user for a model file name if this option is omitted.

-output, -o

Takes an output file name as an argument. If the output file exists the user will be queried before it is overwritten. *User_class* will prompt the user for an output file name if this option is omitted. Note that the output file merely contains a tabular form of the model data, plus some global system statistics.

-help, -h

Prints the command line format on how to invoke *user_class*.

-fullhelp, -fh

Prints this full help information on *user_class* usage.

-version, -v

Prints *user_class* version stamp plus the date and time that it was built.

-verbose

Tells *user_class* to print traces of its operations on the terminal.

-force

If this option is used, the user will not be prompted for confirmation before an output file is overwritten. Such files are model file and output file.

-classtype, -ct

Allows the user to do user classification. If the argument is *WorkLoad*, then the users will be classified according to workload. Otherwise, the user can do arbitrary classification by user number. The default is classification by user number.

A sample terminal session is given below, with user's typed input in boldface:

```
user_class -classtype user
[USER_CLASS Rev. 1.0 - 1986]
Enter reduced file name       ? ev60-3.red
Enter output file name        ? out
Enter model output file name  ? mod

                          CLASSIFYING USERS:
  1   2   3   6   7   8   9  10  11  12  13  14  15  16  17
 18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
 33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55  56  57  58  59  60  61  62
 63  64 126 128 130

Enter number of user classes? 6
  NOTE: for the following, terminate user number list with '$'
Enter name for class 1? CLASS1
Enter user numbers for class   1? 6 12 18 24 30 36 42 48 54 60 $
Enter name for class 2? CLASS2
Enter user numbers for class   2? 7 13 19 25 31 37 43 49 55 61 $
Enter name for class 3? CLASS3
Enter user numbers for class   3? 8 14 20 26 32 38 44 50 56 62 $
Enter name for class 4? CLASS4
Enter user numbers for class   4? 9 15 21 27 33 39 45 51 57 63 $
Enter name for class 5? CLASS5
Enter user numbers for class   5? 10 16 22 28 34 40 46 52 58 64 $
Enter name for class 6? CLASS6
Enter user numbers for class   6? 11 17 23 29 35 41 47 53 59 $
Enter save file name          ? sav
Reading data from 'ev60-3.red'...
```

NOTES

1. If a user number appears in more than one class, *user_class* will assign it to the class that it was first assigned. In other words, each user can belong to at most one class. A warning message will be printed whenever a class re-assignment is attempted.

2. If there are users without any class assigned, *user_class* will automatically create a dummy class for them.

2.2.1 User Input

User_class allows the user to save user classification input data into a file. This data informs *user_class* on how to classify the users in the reduced data. The format of the input/saved file is made up of parameter names followed by 1 or more parameter values. Each parameter name must be on a line. A /* delimits the start of a comment. The parameter names supported at the moment are:

class_type specifies how the user classification is going to be done. Possible values are **user** and **workload**. In the former case, *user_class* will classify the users in the data according to their user numbers. In the latter case, the classification is done using weighted functions of the their workloads.

no_class the number of classes desired.

class_name a user-specified name used to identify a class (apart from the class number chosen by *user_class*).

user_class used to enumerate a list of user numbers belonging to a class.

work_load used to specify the upper limit of a class' weighted workload.

A sample copy of a saved file is given below:

```
class_type       user     /* user classification type
no_class            6     /* number of user classes
/* User classification by user numbers
class_name    1 CLASS1
user_class          1   6  12  18  24  30  36  42  48  54  60
class_name    2 CLASS2
user_class          2   7  13  19  25  31  37  43  49  55  61
class_name    3 CLASS3
user_class          3   8  14  20  26  32  38  44  50  56  62
class_name    4 CLASS4
user_class          4   9  15  21  27  33  39  45  51  57  63
class_name    5 CLASS5
user_class          5  10  16  22  28  34  40  46  52  58  64
class_name    6 CLASS6
user_class          6  11  17  23  29  35  41  47  53  59
```

2.2.2 User Output

User_class's output is essentially identical to that of *condenser*, except that the statistics are given on a per class per transaction basis. However, *user_class* also gives system wide statistics that are not given by *condenser*. They are:

System Throughput
: This is the system's throughput in the number of transactions per second.

CPU utilization
: The percentage of CPU time used for true CPU usage, page faults and the total usage, assuming that the system has only one CPU.

Page Fault Rate
: The number of true and virtual page faults per second, as well as the total page fault rate per second.

Disk Utilization
: The percentage of the time when each disk is busy doing true I/Os and page fault I/Os. The total disk busy time is also given.

Throughputs
: The throughputs or arrival rates of all job classes in the number of transactions per second.

A description of all other statistics is given below:

Response times
: Response times for all interactive users on a per class per transaction basis.

Think times
: Think times for all interactive users on a per class per transaction basis.

True CPU
: The pure CPU usage (i.e. excludes any CPU time used for page faults). This is given on a per class per transaction basis.

Page Fault CPU
: The CPU usage for handling page faults only. This is given on a per class per transaction basis.

True I/O
: The pure I/O operations excludes any I/O caused by page faults. Any disk queueing statistics are also excluded. This is given on a per class per transaction basis.

Page Fault I/O
: The I/O operations caused by page faults only. Any disk queueing statistics are excluded. This is given on a per class per transaction basis.

Disk I/O
: The disk true I/O operations on a per class per disk basis. Queueing at the disks are excluded in the statistics.

Disk PF I/O
: The disk page fault I/O operations on a per class per disk basis. Queueing statistics at the disks are excluded.

Lifetime Function
: This is unlike all the above statistics. The approximated b and c parameters will be printed for all users.

A sample copy of *user_class*'s output file is given below:

```
                        USER_CLASS  OUTPUT

Monitor started on 03/17/86 14:20:21.163
Monitor Version:  1   on 4.2BSD
Monitor User Name: root   Monitor User Number: 75
CPU: VAX 11/750    Memory: 4096 pages  Maximum users: 78
Number of user classes: 3
REMARK:

Elapsed time = 8812.739 seconds
Number of events processed = 122770
Total blocks/buffers read = 817

                        GLOBAL SYSTEM STATISTICS

SYSTEM THROUGHPUT      = 2.744 transactions/sec

STATISTIC                              TRUE        VIRTUAL        TOTAL

CPU UTILIZATION                     38.6038%       0.3493%     38.9532%
PAGE FAULT RATE                       0.0044       0.0111       0.0156
DISK 0                              4.1738%       0.1612%     4.3350%
DISK 1                              0.1385%       0.2287%     0.3672%
DISK 8                              0.2788%       0.2142%     0.4930%
DISK 9                              0.0064%       0.0000%     0.0064%

                        THROUGHPUTS (transactions/sec)

CLASS          MICRO          NORMAL          LARGE          OVERALL

   1           1.316           1.060          0.114           2.491
   2           0.044           0.025          0.004           0.072
   3           0.000           0.050          0.131           0.181
TOTAL          1.360           1.135          0.249           2.744

                        CLASSIFICATION OF USERS

USER CLASS    USER NUMBERS

      1        4  10  19  20  21  24  26  27  34  46

      2       49  50

      3        1  59  60  61  62  63  64  65  66  67  69  71  72  73  74
      3       75  76  77  78

                        RESPONSE TIMES (seconds)

            MICRO              NORMAL              LARGE              OVERALL
CLASS      N   AVERAGE        N   AVERAGE        N   AVERAGE        N   AVERAGE

  1    11600     0.159     9344     0.445     1007     7.554    21951     0.620
  2      387     0.005      217     0.029       31     1.164      635     0.070
  3        0     0.000      440     0.032     1155     9.705     1595     7.037
TOTAL  11987     0.154    10001     0.418     2193     8.597    24181     1.029
```

THINK TIMES (seconds)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	11601	3.165	9344	3.036	1007	9.356	21952	3.394
2	387	7.475	217	51.862	31	110.782	635	27.687
3	0	0.000	440	47.491	1155	109.533	1595	92.417
TOTAL	11988	3.304	10001	6.052	2193	63.550	24182	9.904

TRUE CPU BURST STATISTICS (msec)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.528	3.465	0.427	21.918	0.217	137.473	1.172	35.018
2	0.609	4.294	0.381	21.974	0.235	166.458	1.225	40.851
3	0.000	0.000	0.314	24.482	14.690	104.858	15.004	103.175
TOTAL	0.496	3.491	0.418	22.047	1.172	110.670	2.086	67.442

PAGE FAULT CPU STATISTICS (msec)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.002	2.432	0.003	5.798	0.068	8.694	0.073	8.384
2	0.000	0.000	0.003	48.128	0.069	18.223	0.072	19.567
3	0.000	0.000	0.065	3.659	0.199	50.667	0.263	39.139
TOTAL	0.002	2.475	0.007	5.013	0.077	16.117	0.086	14.872

TRUE I/O WITHOUT QUEUE TIMES (msec)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.000	0.000	0.000	20.152	0.171	19.252	0.172	19.257
2	0.000	0.000	0.039	23.758	0.186	22.393	0.225	22.632
3	0.000	0.000	0.038	22.553	13.966	14.712	14.004	14.733
TOTAL	0.000	0.000	0.004	22.384	1.082	15.400	1.086	15.428

PAGE FAULT I/O WITHOUT QUEUE TIMES (msec)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.002	21.212	0.003	20.952	0.090	18.268	0.095	18.425
2	0.000	0.000	0.003	18.182	0.151	19.287	0.154	19.264
3	0.000	0.000	0.065	19.153	0.352	19.149	0.416	19.149
TOTAL	0.002	21.212	0.007	19.861	0.109	18.492	0.118	18.622

TRUE PAGE FAULT TIMES (msec)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.002	23.864	0.003	25.668	0.068	30.071	0.073	29.698

2	0.000	0.000	0.003	19.697	0.069	51.791	0.072	50.395
3	0.000	0.000	0.065	22.183	0.199	43.055	0.263	37.937
TOTAL	0.002	23.864	0.007	23.524	0.077	32.814	0.086	31.830

VIRTUAL PAGE FAULT TIMES (msec)

	MICRO		NORMAL		LARGE		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.000	1.515	0.003	3.328	0.147	2.303	0.150	2.321
2	0.002	3.030	0.076	2.399	0.447	2.113	0.524	2.157
3	0.000	0.000	0.045	2.483	2.524	4.385	2.569	4.352
TOTAL	0.000	1.818	0.007	2.746	0.311	3.409	0.319	3.392

TRUE DISK I/O WITHOUT QUEUE TIMES (msec)

	DISK 0		DISK 1		DISK 8		DISK 9	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.114	19.278	0.025	18.053	0.033	19.919	0.000	39.394
2	0.192	21.908	0.000	0.000	0.033	26.840	0.000	0.000
3	13.661	14.537	0.077	19.537	0.260	23.152	0.006	36.700
TOTAL	1.010	15.061	0.028	18.327	0.048	21.199	0.000	37.778

	OVERALL	
CLASS	N	AVERAGE
1	0.172	19.257
2	0.225	22.632
3	14.004	14.733
TOTAL	1.086	15.428

PAGE FAULT DISK I/O WITHOUT QUEUE TIMES (msec)

	DISK 0		DISK 1		DISK 8		OVERALL	
CLASS	N	AVERAGE	N	AVERAGE	N	AVERAGE	N	AVERAGE
1	0.033	16.810	0.033	18.779	0.030	19.796	0.095	18.425
2	0.002	51.515	0.099	19.240	0.054	18.360	0.154	19.264
3	0.078	16.679	0.176	19.573	0.162	19.960	0.417	19.180
TOTAL	0.035	16.832	0.044	19.017	0.040	19.789	0.118	18.629

LIFETIME FUNCTION PARAMETERS

CLASS	B	C
1	2.7500e+02	1.3860e+02
2	2.2753e+02	6.4103e+00
3	8.2088e+02	1.0695e+02

AVERAGE LIFETIMES

CLASS	AVG. LTIME
1	5.618351e+02
2	6.909106e+02
3	5.878784e+03

2.3 Program Interfaces

User_class interfaces with *condenser* and modeling tools via files. The interface with *condenser* is a binary file containing all the reduced and aggregate statistics. The interface with modeling tools is a text file containing parameter names and values required by these tools. A detail description of these interfaces is given in the next two subsections.

However, the knowledgable user can also use the output from *user_class* for input to any other modelling tools. In this case, the user has to know how to interpret this output and translate to the input of any modelling tool that the user may be using.

2.3.1 *Condenser* Interface

The reduced file written by *condenser* must begin with header information necessary for *user_class* to determine the sizes of all following data within. The header also contains a simplified form of the event data header that *condenser* obtains from the event data. The format of the header is given below:

NAME	SIZE	DATA TYPE
Month	2 bytes	binary short
Day	2 bytes	binary short
Year	2 bytes	binary short
Hour	2 bytes	binary short
Minutes	2 bytes	binary short
Seconds	2 bytes	binary short
Ticks	2 bytes	binary short
Monitor version	2 bytes	binary short
Monitor user	2 bytes	binary short
Monitor user name	32 bytes	ASCII
UNIX version stamp	16 bytes	binary short
CPU type/name	16 bytes	binary short
Memory size	2 bytes	binary short
Number of users	2 bytes	binary short
Number of events	2 bytes	binary short
Number of buffers	2 bytes	binary short
Length of remark	2 bytes	binary short
Elapsed time	4 bytes	double
Maximum user number	2 bytes	binary short
Maximum disk number	2 bytes	binary short
Remark	varying	ASCII
Recorded users	2 * maxuser bytes	binary short

Following the header are all the statistical matrices and arrays. Their sizes are dependent on the maximum user number and maximum disk number (which are given in *condenser*'s header). The details of the statistical data are given below:

STATISTIC	DIMENSION	DATA TYPE	DESCRIPTION
recorded	max_user	short	flags for recorded users
resp_tot	NTRANS by max_user	double	total response times
resp_n	NTRANS by max_user	long	number of transactions
think_tot	NTRANS by max_user	double	total think times
think_n	NTRANS by max_user	long	total idle transactions
io_noq_tot	NTRANS by max_user	double	total I/O without queue usage
io_noq_n	NTRANS by max_user	long	total I/Os without queue
pf_io_noq_tot	NTRANS by max_user	double	total PF I/O usage
pf_io_noq_n	NTRANS by max_user	long	total PF I/Os
cpu_tot	NTRANS by max_user	double	total true CPU burst time
cpu_n	NTRANS by max_user	long	total no. of true CPU bursts
pf_cpu_tot	NTRANS by max_user	double	total PF CPU burst time
pf_cpu_n	NTRANS by max_user	long	total no of PF CPU bursts
disk_noq_tot	max_drive by max_user	double	total true disk I/O usage
disk_noq_n	max_drive by max_user	long	total no of true disk I/Os
pf_disk_noq_tot	max_drive by max_user	double	total PF disk I/O usage
pf_disk_noq_n	max_drive by max_user	long	total no of PF disk I/Os
pf_n	NTRANS by max_user	long	total true PFs
vpf_n	NTRANS by max_user	long	total virtual PFs
lftb	max_user	double	b parameter
lftc	max_user	double	c parameter

2.3.2 Modeling Tools Interface

The model data file from *user_class* to be used by the modeling tools consists of parameter values and names. These parameters are the union of the input parameter requirements of these tools. The contents of the file is line-oriented, and each line must conform to the following format:

Parameter Name Parameter Value(s) ; Comments

Note that each parameter name can have one or more values. A sample subset of the model data file is given below:

```
/*************************************************************/
/*      P R O J E C T I O N     P A R A M E T E R S          */
/*************************************************************/
sim_cpu_type     P850 /* CPU Type (projection)
sim_memory_size  4095 /* Memory size (projection) in pages
sim_time         8813 /* Simulation time (seconds)
/*************************************************************/
/*      G E M     D A T A       P A R A M E T E R S          */
/*************************************************************/
cpu_type         P850 /* CPU Type (measured data)
memory_size      4095 /* Memory size (measured data) in pages
n_cont              2 /* No. of disk controllers
n_disk              0     2 /* No. of disks in controller 0
n_disk              2     2 /* No. of disks in controller 2
n_class             3 /* Number of user classes
/*************************************************************/
/*    U S E R   C L A S S   1    P A R A M E T E R S         */
/*************************************************************/
user_class   1 TERMINAL
n_users     10            /* number of users in class
lftb       2.7500e+02
lftc       1.3860e+02
cpu          1     3.4646 /* avg CPU burst time per micro transaction
cpu_n        1     0.5285 /* no of CPU bursts for micro transactions
cpu          2    21.9185 /* avg CPU burst time per normal transaction
cpu_n        2     0.4266 /* no of CPU bursts for normal transactions
cpu          3   137.4730 /* avg CPU burst time per large transaction
cpu_n        3     0.2173 /* no of CPU bursts for large transactions
pf_cpu       1     2.4320 /* avg PF CPU burst time per micro transaction
pf_cpu_n     1     0.0022 /* no of PF CPU bursts for micro transactions
pf_cpu       2     5.7976 /* avg PF CPU burst time per normal transaction
pf_cpu_n     2     0.0031 /* no of PF CPU bursts for normal transactions
pf_cpu       3     8.6937 /* avg PF CPU burst time per large transaction
pf_cpu_n     3     0.0678 /* no of PF CPU bursts for large transactions
io           1     0.0000 /* avg I/O service time per micro transaction
io_n         1     0.0000 /* no of I/Os for micro transactions
io           2    20.1515 /* avg I/O service time per normal transaction
io_n         2     0.0009 /* no of I/Os for normal transactions
io           3    19.2520 /* avg I/O service time per large transaction
io_n         3     0.1714 /* no of I/Os for large transactions
pf_io        1    21.2121 /* avg PF I/O service time per micro transaction
pf_io_n      1     0.0022 /* no of PF I/Os for micro transactions
pf_io        2    20.9524 /* avg PF I/O service time per normal transaction
pf_io_n      2     0.0032 /* no of PF I/Os for normal transactions
pf_io        3    18.2677 /* avg PF I/O service time per large transaction
pf_io_n      3     0.0901 /* no of PF I/Os for large transactions
think        1     3.1652 /* avg Think time per micro transaction
```

```
think_n      1      11601 /* No. of trans for micro transactions
think        2    3.0363 /* avg Think time per normal transaction
think_n      2       9344 /* No. of trans for normal transactions
think        3    9.3563 /* avg Think time per large transaction
think_n      3       1007 /* No. of trans for large transactions
disk                 0     19.2780 /* mean I/O service time for disk 0
disk_n               0      0.1143 /* total(ratio) I/Os for disk 0
disk                 1     18.0535 /* mean I/O service time for disk 1
disk_n               1      0.0247 /* total(ratio) I/Os for disk 1
disk                 8     19.9188 /* mean I/O service time for disk 8
disk_n               8      0.0330 /* total(ratio) I/Os for disk 8
disk                 9     39.3939 /* mean I/O service time for disk 9
disk_n               9      0.0003 /* total(ratio) I/Os for disk 9
pf_disk              0     16.8102 /* mean I/O service time for disk 0
pf_disk_n            0      0.0327 /* total(ratio) I/Os for disk 0
pf_disk              1     18.7786 /* mean I/O service time for disk 1
pf_disk_n            1      0.0326 /* total(ratio) I/Os for disk 1
pf_disk              8     19.7955 /* mean I/O service time for disk 8
pf_disk_n            8      0.0301 /* total(ratio) I/Os for disk 8
pf_disk              9      0.0000 /* mean I/O service time for disk 9
pf_disk_n            9      0.0000 /* total(ratio) I/Os for disk 9
```

3. Program Design

3.1 Design Overview

The main objective of *user_class* is to collect groups of users and recompute the associated statistics for each group. An general view of the top level algorithm is given below:

```
Process command line
Classify users to groups
For each group of users:
   add each of their total usage statistic
   add each of their total count statistic
   Resulting class statistics is total usage / total counts
Compute aggregate statistics on a per class basis
Compute averages
Output all class statistics in output file
Output model data in the model file
```

3.2 Internal Data Structures

The following is the header of the reduced data produced by *condenser*:

```
typedef struct headertype {
   short month,                    /* month of date                */
         day,                      /* day of date                  */
         year,                     /* year of date                 */
         hours,                    /* hours of time (millitary)    */
         minutes,                  /* minutes of time              */
         seconds,                  /* seconds of time              */
         ticks;                    /* ticks of time                */
   short monitor_version,          /* Monitor version number       */
         monitor_user;             /* Monitor user number          */
   char  monitor_uname[32];        /* MOnitor user name            */
   char  os_version[16],           /* UNIX  version  stamp         */
         cputype[16];              /* CPU name/type                */
   short memory,                   /* Memory size                  */
         nusers,                   /* No. of configured users      */
         nevents,                  /* total no. of  events         */
         nbuffers,                 /* total no. of  buffers        */
         rlen;                     /* length of monitor remark     */
   double elapsed;                 /* elapsed time                 */
   short maxuser,                  /* maximum user no. in data     */
         maxdisk;                  /* maximum disk no. in data     */
   }   REDUCED_HEADER;
```

3.3 Module Design

main(argc,argv) - Level 0

The main program. Does command line processing and calls level 1 routines.

```
int  argc;
char **argv;

Process command line options
User_Input();     /* to process user's terminal input */
Get_Data();       /* Input data from condenser          */
Classify();       /* Classify the users               */
Aggregate();      /* recompute on a per class basis   */
Average();        /* calculate average values         */
Model_Output();   /* Print out model data             */
Print_Output();   /* Print user_class output statistics */
```

User_Input(ifname, ofname, rfname) - Level 1

Simply prompts the user for the appropriate file names. Also, prompts for window range if the appropriate flags are turned on.

```
char *ifname, *ofname, *rfname;

if input file name (ifname) not given
    prompt the user
if output file name (ofname) not given
    prompt the user
if reduced file name (rfname) not given
    prompt the user
if window_time, prompt for time range
if window_buffer, prompt for buffer range
```

Get_Data() - Level 1

Read in all the reduced data from the reduced file.

Classify() - Level 1

Does the actual classification here. The resulting statistics will be in separate structures given above. During classification, the statistics of users in a common class are summed. For lifetime function parameters, however, the average values of a single class users are used to generate numerous sets of data points for the orthogonal approximation to produce a resulting set of parameters.

Aggregate() - Level 1

Recompute the aggregate statistics on a per class basis.

Average() - Level 1

Compute average values of all statistics, replacing the total values stored.

Model_Output() - Level 1

Print the average statistics to the model data file, together with model configuration data.

Print_Output() - Level 1

Calculate and print global system statistics. Print all average

statistics to the output file.

3.4 Design Issues

The major concern in the design of *user_class* is the size of the structures for storing all statistics. Because *condenser* can provide information on limits of these structures in the reduced file header, *user_class* can easily allocate the required space for the structures dynamically. This avoids the use of static structures (which increases the startup time of *user_class*) and minimizes memory requirements.

During user classification, the *b* and *c* parameters provided by *condenser* for each user will be averaged for each of the 100 data points generated using the lifetime curve function.

3.5 Implementation Language

The language used to develop *user_class* is the C programming language. The main reasons for using this language is the need for bit and byte manipulation, the need for address manipulation, and that it is highly portable among UNIX systems.

1. Proposal

Before reading this document, the reader must be familiar with the model data provided by *user_class*. Also, a *qnets* user should also be familiar with general capacity planning techniques.

This document describes the development of *qnets*, an analytical modeling tool that accepts model data from *user_class* and provides performance statistics of the system to be modeled.

1.1 Goals and Non-Goals

The goal of *qnets* is to be able to model as many systems as possible. Besides being general, the tool should provide results with sufficient accuracies. The input to *qnets* is the model data provided by *user_class*.

The analytical algorithm used by *qnets* is Linearizer (see *Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems*, CACM 25, 2 (April 1982), 126-134 by Mandy Chandy et. al). Memory modeling is also added to *qnets* (note that linearizer does not model memory).

Because *qnets* is based on linearizer, the user should be aware of the algorithm's restrictions. In particular, he should know whether or not *qnets* can be used to model the system under study.

2. Program Function

2.1 Terminology

Performance indices
: The set of statistics that serves to calibrate a system's performance. Examples are utilizations, throughputs and response times.

Model validation
: The purpose of model validation is to ensure that a model representation (for example, a mathematical model) of a system correctly represents the system. The process involves validating the performance indices given by the model with the measured statistics of the system.

Workload characterization
: The manner in which we represent workload. In general, workload characteristics include CPU, I/O and memory demands.

Performance Projection
: A validation model is evaluated using a representative workload to determine the performance indices of the projected system.

2.2 User Interface

To invoke *qnets*, the command line must conform to the following format (with abbreviations in boldface):

qnets *[-**input** <file>] [-**output** <file>] [-**help**] [-**full**help] [-**version**] [-**force**]*

If no options are given, *qnets* will prompt the user for the necessary set of input options.

Qnets supports the following options:

-input, -i	*-output, -o*
-help, -h	*-version, -v*
-fullhelp, -fh	*-force*

The following screens summarize all the *qnets* options.

-input, -i

Takes a file name as an argument (for input file). The input file should be saved by a previous *qnets* run, and should contain lists of user for all classes. If this option is omitted, *qnets* will prompt the user for classification information.

-output, -o

Takes an output file name as an argument. If the output file exists the user will be queried before it is overwritten. *Qnets* will prompt the user for an output file

name if this option is omitted.

-help, -h

Prints the command line format on how to invoke *qnets*.

-fullhelp, -fh

Prints this full help information on *qnets* usage.

-version, -v

Prints *qnets* version stamp plus the date and time that it was built.

-force

If this option is used, the user will not be prompted for confirmation before an output file is overwritten. Such files are model file and output file.

A sample terminal session is given below, with user's typed input in boldface:

```
qnets
[QNETS Rev. 1.0 - 1986]
Enter input file name          ? model_data
Enter output file name         ? outfile
```

2.2.1 User Input

The input file to *qnets* must be produced by *user_class*. Details of the input file and format are given in Appendix B.

2.2.2 User Output

The output produced by *qnets* consists of statistics identical to the measured statistics produced by *user_class*. They are as follows:

System Throughput
: This is the system's throughput in the number of transactions per second.

Page Fault Rate
: The number of page faults per second.

Utilizations
: The percentage of the time when each service is busy servicing jobs.

Throughputs
: The throughputs or arrival rates of all job classes in the number of transactions per second.

A sample copy of *qnets*'s output file is given below (device 0 is the CPU):

```
CPU: VAX 11/750     Memory: 4096 pages  Maximum users: 78
Number of user classes: 3
Number of disks: 4

                              SERVICE TIMES(msec)

CLASS   DEVICE 0      DEVICE 1      DEVICE 2      DEVICE 3      DEVICE 4

  0      35.0174       18.5243       18.5392       19.8468       39.3939
  1      40.8475       22.2774       19.2400       20.8149        0.0000
  2     103.1754       14.5929       19.5699       20.7756       36.7003

                                  VISIT RATIOS

CLASS   DEVICE 0      DEVICE 1      DEVICE 2      DEVICE 3      DEVICE 4

  0       1.1724        0.1646        0.0748        0.0793        0.0003
  1       1.2251        0.1945        0.1506        0.1143        0.0000
  2      15.0038       14.0266        0.8979        1.0161        0.0056


                             PROJECTED STATISTICS

SYSTEM THROUGHPUT    = 3.217966 trans/sec
PAGE FAULT RATE      = 0.019027 per sec

CLASS       THINK(sec) RESPONSE(sec) THRUPUT(/sec)

    0          3.3943        0.0062        2.9407
    1         27.6868        0.0098        0.0722
    2         92.4175        0.2550        0.2050

CENTER      UTILIZATION

    0          44.1724%
    1           5.1243%
```

2	0.7890%
3	0.9126%
4	0.0077%

QUEUE LENGTHS

CLASS	DEVICE 0	DEVICE 1	DEVICE 2	DEVICE 3	DEVICE 4
0	9.9818	0.0094	0.0041	0.0047	0.0000
1	1.9993	0.0003	0.0002	0.0002	0.0000
2	18.9477	0.0442	0.0036	0.0044	0.0000

3. Program Design

3.1 Design Overview

The modeling algorithm used in *qnets* is known as Linearizer. The overall design of *qnets* is as follows:

```
Process command line
Read in input file
Calculate page fault rate from lifetime function
Include disk visits due to page fault to disk visit ratio
Invoke Linearizer
Output statistics
```

3.2 Internal Data Structures

The following is the header of the reduced data produced by *condenser*:

```
#define MAXCENTER 30
#define MAXCLASS  10

typedef double MATRIX[MAXCLASS][MAXCENTER];
typedef double UCMAT[5][MAXCLASS];
typedef double DARR[MAXCLASS];
typedef long   LARR[MAXCLASS];

MATRIX  Ser_t,              /* service times per class per device */
        Vst_r,              /* visit ratios per class per device  */
        Res,                /* response times per class           */
        Q,                  /* queue lengths per class per device */
    /*  input parameters */
        disk, disk_n, pf_disk, pf_disk_n;
UCMAT   cpu, cpu_n, io, io_n, pf_cpu, pf_cpu_n, think, think_n;
DARR    Thk_t,              /* think times per class              */
        Thpt,               /* throughputs per class              */
        pfs,                /* page fault rates per class         */
        N_usr,              /* size of each job class             */
        lftb, lftc;
```

3.3 Module Design

main(argc,argv) - Level 0
The main program. Does command line processing and calls level 1 routines.

```
int  argc;
char **argv;

Process command line options
Restore();          /* Read in input file                */
Simplify();         /* Compute page fault and disk VR    */
Linzr();            /* Invoke Linearizer                 */
Print_Output();     /* Print all statistics              */
```

Restore() - Level 1
Read in all the input parameters from the input file. Store all data into arrays.

Simplify() - Level 1
Aggregate transaction subclass statistics. Compute page fault rate from lifetime function. Compute disk visits due to page fault. Recompute all disk visit ratios.

Print_Output() - Level 1
Print all output statistics given by Linzr().

3.4 Design Issues

Unlike *user_class* and *condenser*, all the data structures used in *qnets* are static arrays. This is because the size of the arrays are comparatively smaller. If the user requires to model more service centers or more job classes that *qnets* currently supports, the constants *MAXCENTER* and *MAXCLASS* should be increased accordingly.

3.5 Implementation Language

The language used to develop *qnets* is the C programming language. The main reason for using this language is that it is portable among UNIX systems.