THE PREDICTION AND EVALUATION OF THE PERFORMANCE OF SOFTWARE FROM EXTENDED DESIGN SPECIFICATIONS

by

CONNIE UMLAND SMITH, B.A., M.A.

August, 1980 TR-154

This report constituted the author's Ph.D. dissertation in Computer Sciences at the University of Texas at Austin.

DEPARTMENT OF COMPUTER SCIENCES
THE UNIVERSITY OF TEXAS AT AUSTIN

THE PREDICTION AND EVALUATION OF THE PERFORMANCE OF SOFTWARE FROM EXTENDED DESIGN SPECIFICATIONS

Publication No.

Connie Umland Smith, Ph.D.
The University of Texas at Austin, 1980

Supervising Professor: James C. Browne

A methodology is defined that can be used to quickly and easily predict computer performance attributes of a software system prior to its implementation. It can be used during design and implementation, and throughout the life of the software to analyze the effect of modifications to the software, host computer system, or both.

There are usually several designs that are functionally equivalent, but which exhibit different execution characteristics. It is always far less expensive and more productive to select an appropriate design initially than to tune and patch systems with inappropriate designs after implementation. Performance crises are thus prevented and efficiency gains are substantially greater.

A basic methodology is presented that is sufficient for designs without complex interrelationships of components and for those designs whose performance is independent of the environment in which they will execute. Additional techniques are given for analyzing the effect of environmental factors: data dependency, competitive effects, and memory contention. The modeling of design complexities (internal concurrency, synchronization, mutual exclusion, and blocking) are explained. Specifications are included for a comprehensive performance prediction tool. A prototype demonstrating its feasibility is presented.

The methodology embodies a number of new proposals that are essential for software performance prediction:

1. The specification of performance determining factors
2. The graphical representation of the significant structural elements of software systems including hierarchical structuring and recursion, typical control representations, concurrency and synchronization, and blocking for mutual exclusion
3. Enhancements to static graph analysis techniques for data dependent execution characteristics, hierarchical structuring, and the spectrum of nodes and arcs
4. The uniting of graph analysis techniques and queueing network models: the algorithms for the computation of elementary model parameters as well as swapping, memory, synchronization, and blocking model parameters
5. Queueing network model formulations for analyzing synchronization and blocking.

The proposals are consolidated to produce the desired performance prediction and evaluation capabilities.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.0 Performance Prediction Requirements

A methodology is presented that can be used to quickly and easily predict computer performance attributes of a software system prior to its implementation. It can be used during design, implementation, and throughout the life of the software to analyze the effect of modifications to the software, host computer system, or both. The effectiveness of this information during design is far greater than after implementation.

The methodology is specifically intended for use by software designers rather than by performance analysis specialists. They can apply it to:

1. Select suitable designs
2. Identify critical components
3. Quantify resource requirements and constraints
4. Scrutinize systems throughout implementation
5. Follow the evolution of the software
6. Analyze subsequent functional enhancements
7. Answer workload and performance attribute queries.

It provides the basis for the integration of the techniques of software development, computer performance management, and capacity planning tasks. The application of the methodology produces information valuable not only to the software designers but also to performance analysts for host system tuning functions, during the design process, such as:

1. Global analysis of computer system impact
2. Configuration analysis
3. Planning of operating system or other support system enhancements
4. Investigation of file placement strategies.

The information can also be used for long-range capacity planning tasks. The accuracy of forecasts of growth and resource usage is improved. Hardware and software requirements can be ascertained and acquisitions scheduled prior to the implementation of critical software. The increased information can be used for cost justification and accounting.

There are usually several designs that are functionally equivalent, but which exhibit different execution characteristics. It is always far less expensive and more productive to select an appropriate design initially than to tune and patch systems with inappropriate designs after implementation. Performance crises are thus prevented and efficiency gains are substantially greater. Tradeoffs on the ease of implementation, capabilities provided, resulting performance, and computer system impact can be effectively considered in the evaluation and selection of the software design.

A precise prediction of performance is not feasible in the design stage since the values of many critical variables are uncertain and must be approximated. Nevertheless, reasonable evaluations can be obtained indicating whether the design is, is not, or is marginally satisfactory with respect to performance requirements or goals. The components critical to meeting the goals ("bottlenecks" or rate determining processes) are usually clearly isolated. Optimization efforts can then be concentrated for maximal impact. This precludes devoting excessive time to less important components.

This methodology does not purport to automate the process of selecting optimal designs. It uses the limited information of execution characteristics associated with software designs to maximal effect. The result is the identification of feasible or satisfactory designs, not the selection of optimally performing designs. Thus, unsatisfactory or potentially disastrous portions of the design space can be ruled out very early in the development process before significant work is begun.

The design is reevaluated as components are implemented and actual execution data is available. Better, more precise predictions of performance can then be obtained, as well as constraints on the resource requirements of the remaining components. Once measurement data is available for the critical components, the prediction rapidly converges to the actual behavior. Likewise, the resource requirements become more accurate.

It is unusual for a software system to be implemented exactly as initially designed. Modifications are made during the development phase to add features and to correct problems. Information on these changes can be incorporated and used to assess their feasibility, desirability, and performance impact. The characteristics of the evolving software are reconciled, so predictions do not become obsolete. Upon conclusion of the project, the information is current and complete and can be used to analyze subsequent major revisions and functional enhancements. Only specifications for the revisions need be input to obtain information about the entire revised system.

Another useful feature is the availability of data for queries about expected workload and execution characteristics. It is helpful to know which fundamental operations will be required most often, at the time that algorithms are designed. A report showing their relative frequency of occurrence can easily be derived from the collected data. Similarly, it is desirable to have information about the expected execution time of components. General queries of this type are easily supported.

Data availability is particularly beneficial for computer performance management tasks. A global analysis can be performed to determine the impact of new software on existing work. It may be that a particular design results in a satisfactory response time, but it significantly degrades the response time of other, more important work. This situation is easily detected.

The hardware configuration necessary to support the new, combined workload can be determined. For example, it may be that an additional disk channel is all that is necessary to correct a problem. Alternatively, revisions to the scheduling algorithm may be necessary to ensure that incompatible jobs are not run simultaneously. These types of software configuration adjustments can also be analyzed. File placement is often a critical performance factor. Various strategies can be studied, with little effort, using the constituent models. This is superior to previous trial and error techniques.

1.1 Motivation

Software performance is rarely considered early in the development process when the design is malleable and improvements can achieve the greatest impact. Emphasis is typically placed on functionality and expediency of the completion of the project. Often, when software is implemented, it performs so poorly that new computer resources are required to run it, it is necessary to immediately begin redesigning it, or it is never used. It is not wrong to emphasize functionality and expediency, but the usability of the final product and its impact on the host computer system should also be considered. The rapid growth of on-line and interactive systems and the rapid integration of computer system processing into human work procedures makes performance an integral part of functionality. A system which unnecessarily wastes human time increases costs and wastes the principal resource of an organization.

There are many reasons for neglecting efficiency. In most organizations responsibility for host system performance is distinct from software development responsibility. Performance analysts lack the necessary data on new software designs and systems analysts lack the expertise to analyze performance in the complex host environment. Systems analysts have previously received little feedback on resource requirements of software systems they have developed and therefore are unable to provide accurate data on new systems. As a result, they have a false idea of the quantity of information necessary for prediction and the time and effort required to make reasonable estimates. They also lack confidence in the results of studies based on these "inaccurate" estimates. It is argued that it is cheaper to acquire additional hardware to alleviate performance problems than to expend additional time and effort to develop efficient software.

It is certainly preferable to resolve these performance prediction problems than to allow crises to develop and to solve them by acquiring additional hardware. It is difficult to obtain computing hardware on short notice. Most companies expect advanced planning and justification for hardware purchases. Often a lengthy RFQ process is required. Expansion is still bounded by maximum amounts of memory and peripherals that can be added. The maintenance of large computer systems becomes difficult to manage as they approach this maximum size.

This methodology alleviates the above problems. Once implemented, it can be used by software designers as an interactive tool to produce feedback on design quality. Minimal specifications, time and effort are required from the designer. Specifications include estimates of expected values as well as upper bounds to compensate for the uncertainty of resource requirements. (If the performance goal is satisfied when upper bounds are used in calculations, the design is likely to be suitable.) The evaluation is repeated, as components are completed and actual data becomes available, to obtain more accurate forecasts. Since feedback is provided to the designer, future resource estimates also become more accurate. The designer is motivated to use the tool since many additional benefits (mentioned initially) are newly attainable. The time and effort required to use the methodology are offset by eliminating time spent on undesirable designs and focusing attention on critical modules.

A pragmatic approach was used in this research. This provides a means of validating methods and focusing attention on solutions to problems typically encountered during software development projects. Initially, the scope of the problem was reduced, by limiting design and environmental complexities, to a class of software designs whose performance could be predicted. The resulting "basic methodology" and its application are described in previous papers [SMI79a,SMI79b]. Next, more complex software systems were studied throughout design and implementation stages and the techniques applied. Typical evaluation and prediction difficulties were examined and solutions incorporated to resolve them. Another previous paper contains some of these resolutions [SMI80]. This document consolidates the resulting comprehensive methodology which encompasses features to handle most, general software design problems.

1.2 Overview of the Methodology

The "basic methodology" consists of capturing performance specifications, mapping them onto a graphical representation, and using the graphs to perform static analysis procedures to derive the mean response time, as well as expected best and worst cases.

The specifications provide information on the performance goals, execution characteristics, linking information, and the execution environment. The performance goal is a criterion against which measurement data can be compared to determine whether or not the software system is acceptable. An example of a reasonable goal is a maximum response time of eight seconds for a specified data base query scenario. Performance is unacceptable if the response time is longer.

A top-down approach is used in the specification of the execution characteristics of the system. Estimates are first provided for resource requirements (such as CPU time) for software components that can reasonably be predicted at the top level of the design. Processing details of the remaining components are resolved into sub-components; resource requirements can then be estimated for some of the sub-components, others will need further resolution. This process continues until sub-components, whose resource requirements can be estimated, are defined and specifications are complete. This explicit hierarchical structuring can be "collapsed" into higher levels of abstraction as required.

For example, a data base query scenario consists of parsing the query into standard data base commands then invoking these commands. Estimates can be provided for the parsing component, however, the details of the invocation must be refined and a sub-component defined for each type of data base command that could be invoked. Either specifications are provided for these sub-components, or the processing details of the commands are resolved, until estimates are provided for all processing that occurs. Structural information is still available for the top level (two components). Application of the analysis procedure ("collapse") then yields resource requirements for the top level component that was originally specified through resolution of processing details.

The linking information defines the inter-relationships of components. All possible execution paths are defined along with the type of connection and the frequency of occurrence. For example, component A contains _procedure calls_ to B and C. B is called _each time_ A is executed, but C is only called _50%_ of the time.

The execution environment specifications are collected once, and used in the evaluation of all software executing in that environment. They include information on the host computer system configuration and the operating system overhead and can easily be provided by a performance analyst.

The data provided above is collected and the software system structure is extracted. A graphical representation is used to depict the structure. Functional components are considered nodes and links are considered arcs. The structure is hierarchical with the lowest level containing complete information on estimated resource requirements. The performance analysis then proceeds with the computation of the elapsed time for each function at the lowest level. This is equivalent to collapsing a set of nodes and arcs into a node in the next higher level of design. The elapsed time and resource requirements for that node are then known and can be used to collapse that level into a single node. Ultimately, the elapsed time and resource requirements for the top level of the design are known. The results are then compared to the performance goal and evaluated accordingly.

The analysis is an iterative process. As measured data from implementation executions is obtained, the specification data is updated and the design is reevaluated. It is possible for the initial evaluation to indicate that performance is satisfactory, but for the execution characteristics of functional components, after they are implemented, to vary enough from their specifications that bottlenecks appear at later stages of the software development. The importance of identifying these bottlenecks prior to final software implementation cannot be over-emphasized. Alternate design selections or additional hardware acquisitions performed prior to implementation can prevent disasters.

Extensions to the above "basic methodology" are incorporated to handle three environmental complexities: data dependency, competitive effects, and memory contention. It is often impossible to provide precise specifications of execution characteristics since they depend upon the data processed by the software. Design structures often involve looping where the number of passes through the loop is dependent, for example, upon the number of records in a file of a particular type. Other possible dependent factors are CPU time, I/O operations, and execution paths. The data dependency problem is resolved by introducing parameters to represent the data objects upon which performance depends. Conditional performance goals are also introduced. An example of one is a requirement that response time be less than 10 seconds when 20 type A records are in a file and less than 15 seconds when there are 100 type A records.

The second environmental complexity is the effect of other software that executes on the host system and competes for the computer resources. It introduces queueing delays into a scenario, for example, when a job is ready for CPU processing but must wait until another job has finished using the CPU. The problem is resolved by using queueing network models. Parameters necessary to run these models can be derived directly from the execution graphs in a straightforward manner.

The third is the effect of contention for primary memory. An extension of the analytic memory model of Brown, et.al.[BRO77], formulated by Keller, is used as the vehicle for the analysis [IRA79]. Once again, the execution graphs provide the information for the derivation of model parameters.

The next extension to the "basic methodology" resolves the effects of concurrent processing. Queueing network models are used to reflect the processing of multiple, simultaneous users. Specialized models are needed when synchronization of processing is involved. For example, program A and B can execute in parallel, but A needs the output from B before it can complete final processing and terminate. There is no known analytical solution to this problem; a satisfactory approximation is developed. Likewise, an approximate analytical solution is used to incorporate the impact of blocking. Blocking occurs when a process requires exclusive use of a resource (such as a data base index) and causes other processes desiring access to it to wait until it has been freed.

Details of each of these topics follow. Chapter 2 contains a complete description of the basic model. The extensions to accommodate the environmental factors are given in Chapter 3. Chapter 4 covers the effects of concurrent processing. The implementation of these features is discussed in Chapter 5. A prototype system is presented that demonstrates the feasibility of implementing such a tool. Conclusions and suggestions for future research are in Chapter 6.

Simple examples are included throughout to illustrate and validate the models. There is no single application to demonstrate all features. A comprehensive example is presented in the appendix to illustrate several design iterations. It contains many, but not all, of the design problems addressed. The remainder of this chapter contains a review of the previous and contemporaneous related work.

1.3 Related Work

The need for software development aids that provide feedback to the designer prior to the implementation of software has long been recognized. To date, emphasis has been on specification languages, data bases and retrieval methods for design details; project management aids; and issues such as verification, reliability, and testing. Stavely discusses the state of the art in the field of software design aid systems [STA78].

Few research efforts have considered performance directly. Most design aid systems that use performance data are primarily concerned with software behavior with respect to verification of proper sequences of events in time. One exception is POD, a software engineering tool for performance oriented design by BGS Systems, Inc. [BGS79a,BGS79b]. Specifications of performance requirements are used in conjuction with operational analysis methods [BUZ76, DEN78] to produce performance evaluation data for the system design. Reports can then be examined interactively. Facilities are provided for altering the hardware configuration or the specifications and repeating the analysis.

The POD concept is similar to this one. The major differences are as follows:

1. It is oriented to use by performance analysts primarily for capacity planning purposes
2. The analysis is not based on a graphical representation
3. Techniques for the derivation of model parameters are not documented
4. No provisions are included for handling design issues such as data dependency, blocking, or synchronization of concurrent processes.

POD demonstrates the viability of an interactive approach to performance prediction and evaluation of software designs.

Earlier systems combined functional and performance specification, verification, and evaluation techniques. The first was proposed by Graham, Clancey, and Devaney [GRA73]. Their performance evaluation is based on a graphical representation of the design. Some techniques are given to do a static analysis of the performance with respect to CPU requirements. They recommend simulation of software systems prior to implementation.

Another more comprehensive system is the Design Realization, Evaluation and Modeling (DREAM) System designed by Riddle and the Reliable Software Systems Group at the University of Michigan [RID78a, RID78b]. Sanguinetti incorporates simulation as the primary vehicle for the determination of system performance characteristics using the DREAM methodology [SAN77, SAN78, SAN79]. The Software Tool for Evaluating System Designs (STESD), designed by Baker, Chester, and Yeh of the University of Texas provides system performance and cost estimates using hierarchical simulation models that are constructed along with the implementation of the software [BAK78].

The designers of these systems have considered the use of graphical analysis techniques for computing execution time; however, they have placed emphasis on the use of simulation for complex software systems. Environmental effects are not considered.

This methodology is based on the philosophy that suitable approximations of performance can be derived, without simulation, prior to software implementation. Since it is desirable to perform the analysis interactively, and because the specification data used is imprecise, the extra time required to model and simulate the software to obtain more accurate performance predictions is usually not justified early in software development projects.

Other related research has been done on the use of graphical analysis techniques to study program performance. Algorithms to derive mean, variance, and distribution of CPU time are presented by Kelly [KEL74]. Sholl and Booth describe similar techniques and include program size considerations [SHO75, BOO79]. The concept of cost-oriented flows in network flow problems is related to program execution time by Kodres [KOD78]. Graphs are used as the basis for analysis of programs to determine optimal overlay structures by Baer and Caughey [BAE72], Van Hoep[VAN71], Lowe [LOW70], and Kernighan [KER71].

These techniques are incorporated and extended here to include systems of programs. This introduces additional computational structures and techniques. Additional algorithms are presented for the derivation of queueing network model parameters from graphs.

Research has been done on other graphical representations of software systems; however, emphasis is on depicting the system structure rather than performance analysis. Hebalkar and Zilles develop an interactive system for graphically representing designs and maintaining an associated data base of information [HEB78]. It depicts execution flow and data constructs at the module level but does not provide features for a more refined level of detail. Ng discusses a means of generating source code from Nassi-Sneiderdam charts [NG78]. Similar techniques could be combined with this type of analysis to yield a powerful software design aid tool. Haranda and Kunii describe another interesting graphical representation of software [HAR79]. Recursive graphs and recursive graph operations are suggested as a basis for specification languages. Performance analysis is not addressed.

Other approaches to the analysis of predicted performance have been proposed by Allen [ALL79] and Shaw [SHA79], but are considered ineffective as described due to the complexity of the analysis. Allen uses flow graphs as the basis for an analysis which consists of transforming the graphs to a first order differential equation and solving it for the expected value of the response time. It is a rather complex analysis not suitable for interactive evaluation. Environmental factors are not considered.

Shaw developed formal techniques for specifying and verifying program performance. They are extensions to verification techniques for functional properties of programs. The complexity and level of detail preclude application of these techniques to general prediction problems.

Other research is related to the extensions to this basic model; however, general analysis methods are addressed rather than specific performance-related design issues. A discussion of the relevant work is included as each model extension is presented.

CHAPTER 2
BASIC METHODOLOGY

2.0 Introduction

The basic methodology comprises the elements necessary for the prediction and analysis of designs for all types of software. It is sufficient for designs without complex interrelationships of components and when the performance does not depend upon the environment in which it will execute.

Concurrent processing adds complexity to software designs. It may involve multiple users: one user delays the processing of another due to sharing of or having exclusive access to data. It may also involve parallel execution of software components for a single user: communication or synchronization between components causes one of them to wait on another.

Environmental effects may be unimportant when the software executes on a dedicated machine or at a higher priority than all other software. They are important when the execution characteristics of the software depend upon the data to be processed or when other software executing on the host computer system impacts performance of the software. In this case, a lower bound for the actual response time can be obtained from the basic methodology.

Six fundamental steps are necessary to satisfy the performance prediction requirements:

1. Extending functional design specifications to include performance goals and execution characteristics.

2. Mapping these design level specifications onto an appropriate representation.
3. Analyzing the structure of the system and deriving performance metrics.
4. Evaluating the software design with respect to the performance goals.
5. Iterating the preceding steps as the software is implemented and actual execution characteristics are obtained.
6. Updating and retaining the execution characteristics for more accurate capacity planning.

The specific procedures were derived by studying the design specifications and performance of actual software systems. The information necessary for response time prediction was deduced and performance specifications were formulated. They include information on the expected execution characteristics of the software and on the average time required for (some of the) operating system functions.

An appropriate representation was chosen that supports the following design features of large software systems:

1. Hierarchical structuring
2. Conditional execution
3. Looping
4. Recursion
5. Nesting

Elementary graph theory is used as the basis for the representation and the analysis [BER58]. Each functional component is a node in a graph, where a functional component is a collection of program statements, procedures, subroutines, modules, or programs that perform a logical function with respect to the software design. The arcs represent paths between the components. Traversal of an arc implies some type of protection domain switch such as a procedure call or a supervisor call. A probability associated with each arc reflects the likelihood of traversing it.

Standard analysis techniques are used to find the shortest and longest paths in the graph. They represent the minimum and maximum response time. A technique similar to those of Sanguinetti [SAN77] and Kelly [KEL74] is used to derive the mean response time.

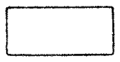
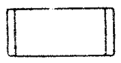
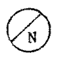
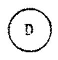
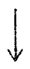
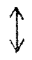
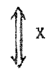
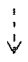
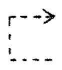
The basic methodology was applied to the National Software Works (NSW) system design [FOR78]. The response times calculated were remarkably close to measured results [SMI79a]. The bottlenecks and critical components were easily identified. This indicates that these specification, modeling, and analysis techniques can reasonably be used as the foundation for the methodology. It also indicates that the essential elements with respect to software response time are included. The representation is appropriate for supporting the analysis and is adequate for depicting the structure of the system. The algorithms yield accurate results and are computationally tractable. The model is suitable for interactive evaluation.

The remainder of the chapter gives the details necessary for understanding and using the basic methodology. The graphical representation is presented first, and is followed by the specifications, the response time analysis, the evaluation of the results, and an example.

2.1 Graphical Representation

Four types of nodes are used, as shown in Table 2.1. Basic nodes and collapsed nodes both represent functional components. The distinction is that the function performed by a collapsed node is defined by another graph at the next level of detail. It is called the associated sub-graph. Multiple levels of detail are allowed.

TABLE 2.1. GRAPH NOTATION

Symbol	Name	Description
	Basic node	Represents a functional component whose execution characteristics are defined at this level.
	Collapsed node	Represents a function whose execution characteristics are included in a graph at the next level of detail.
	Repetition node	Defines the beginning of a loop that will be repeated N times. The last node in the loop has a dummy arc back to the repetition node.
	Dummy node	No processing is associated with the node.
	Arc	Shows a transfer of control or a protection domain switch.
	Bi-directional arc	Shows that control will return to the origin node when processing is completed at the destination node.
	Double bi-directional arc	Same as above except that control returns to the driver, X.
	Dummy arc	No processing time is associated with the arc. They may be bi-directional.
	Self loop	Shows that processing may be completed at this node when there are additional nodes below it in the graph.

Collapsed nodes represent hierarchical and recursive structures. Recursion is represented by placing a collapsed node within its own sub-graph.

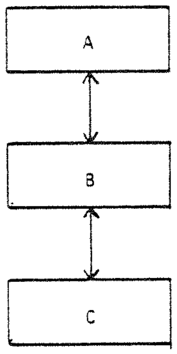
Repetition nodes indicate that some of the functions following the node are repeated one or more times, as indicated by the repetition factor. A special arc connects the last node in the repetition loop to the repetition node. Dummy nodes are occasionally needed to represent special software structures.

There are five types of arcs, as shown in Table 2.1. Standard arcs represent a transfer of control. Bi-directional arcs are used when control will eventually return to the origin node, as in a procedure call. Nested procedure calls are represented by a series of bi-directional arcs.

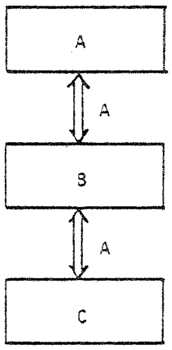
A double bi-directional arc (DBA) is used when one component functions as a driver: it calls many other components and regains control between each of the calls. Note that any of the components called by the driver could be represented by a collapsed node and thus may contain nested procedure calls. A sequence of DBA´s is used as a shorthand notation for ordering the components called by drivers.

A dummy arc is used to illustrate precedence between nodes; no processing is associated with it. An example of its use is a dummy arc between a repetition node and the last node in the repetition loop. A self loop is a special dummy arc. It is used in conjunction with bi-directional arcs to show that a node may either call a component below it in the graph and return to the origin node when that component completes, or return without issuing a call.

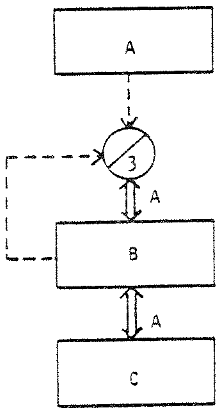
Figure 2.1 contains four examples that illustrate how these conventions are used to represent typical software structures. Figure 2.1a is the representation of nested procedure calls. Figure 2.1b is the representation of a driver. A is the driver; the execution of component B precedes the execution of C, but there is no direct connection between B and C. Figure 2.1c is an extension of



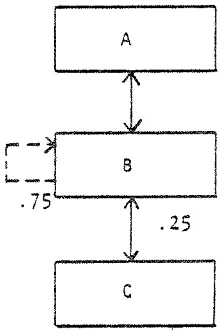
(a) Nested procedure calls:
A calls B, then B calls C.



(b) Sequence of procedure calls (driver)
A calls B then after completion of B, A calls C.



(c) Repetition loop:
A calls B 3 times then A calls C.



(d) Self loop:
A calls B, B calls C 25% of the time then returns. 75% of the time B returns without the call.

FIGURE 2.1. EXAMPLES OF SOFTWARE REPRESENTATIONS

2.1b in which component B is called 3 times by A before C is called. Note that the arc that represents the procedure call is included in the loop. A dummy arc connects A to the repetition node since no corresponding processing takes place at that point. Figure 2.1d illustrates the use of a self loop. Collapsed nodes are not shown in the example; however, any of the graphs shown could be a sub-graph associated with a collapsed node. For example, Node B in Figure 2.1a could be a collapsed node represented by the sub-graph in Figure 2.1d.

There is an additional convention that is not explicit in the notation; the nodes can be or-nodes. An or-node is the origin of two or more arcs, each of which has a probability associated with traversing it; the sum of the probabilities must be one. Only one arc is traversed each time the node is reached. Node B in Figure 2.1d is an or-node.

An initial node in a graph represents the first functional component executed in the graph. Initial nodes can only be the origin of arcs, not the destination. Node A is the initial node in all graphs in Figure 2.1. It is necessary (for the analysis) to impose a restriction that only one initial node is allowed in a graph or a sub-graph. This is a minor restriction, since a dummy or-node can be inserted with an arc to each component that could be executed first.

A terminal node in a graph is the last functional component executed. A special interpretation of "last" is required when bi-directional arcs are involved; that is, a terminal node is the last function called. It is possible that processing may actually continue upon the return from the component; however, the last function called is still considered the terminal node. This is because, in the analysis, the total execution time is computed for each node, disregarding the internal ordering of procedure calls and processing. By this definition, a node with a self loop is also considered a terminal node. There may be multiple terminal nodes in a graph or a sub-graph. Node C is a terminal node in all graphs in Figure 2.1. Node B in Figure 2.1d is also a terminal node.

A path is a sequence of arcs between an initial node and a terminal node in a graph. There may be multiple paths in a graph. For this analysis, if a path includes a repetition loop, the arcs within the loop only appear once in the path regardless of the value of the repetition factor.

A restriction is required that all loops in the graph must be either repetition loops or self loops. Again, this does not reduce the modeling power of this graphical representation since equivalent representations can be used that contain only these types of loops, as shown in Figure 2.2.

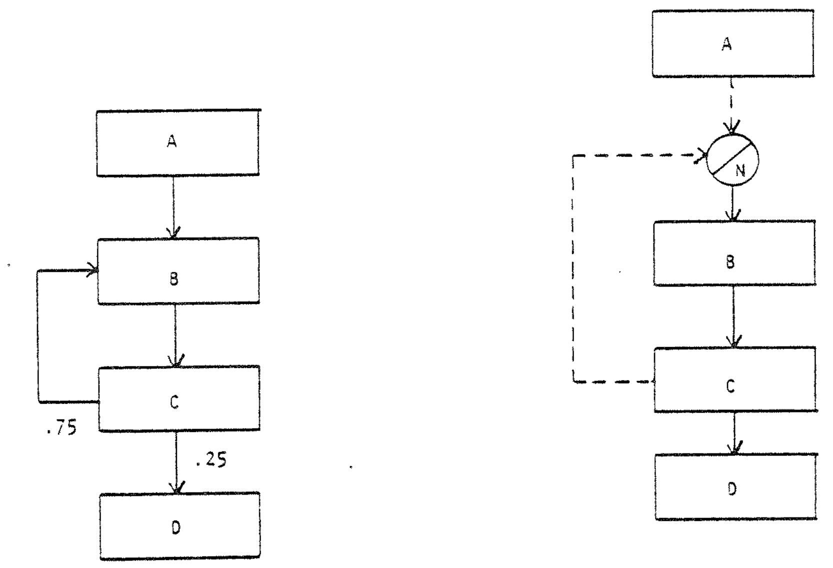
A sub-path is a segment of a path. There is a sub-path associated with each arc out of a node that is either an origin or a destination of multiple arcs. The sub-path terminates with the arc into a terminal node or a node that qualifies as an origin of another sub-path. A self loop is a trivial example of a sub-path. The paths and sub-paths associated with the graphs in Figure 2.1 are shown in Table 2.2.

The following example illustrates the graphical representation of a software system. Figure 2.3 shows the top level of an execution graph of a query against a data base containing computer performance data. The nodes represent the functional components of the query language and the data base management system. The query is "display programs run between November 15 and December 1, 1979 that required more than one hour of CPU time."

In the graph, "interpret command" is an or-node that transfers control to "parse" for new queries and to "send message" for a continuation of a previous query, that is, to retrieve more of the data that satisfied the original request. "Get data" is repeated once for each data base record that satisfies the request until enough information has been retrieved to fill the screen or the screen is full. "Process request" and "get data" are collapsed

TABLE 2.2. EXAMPLES OF PATHS

Graph	Paths	Sub-paths
1a.	1. (A,B) , (B,C)	1. (A,B) , (B,C)
1b.	1. (A,B) , (B,C)	1. (A,B) , (B,C)
1c.	1. (A,R) , (R,B) , (A,C)	1. (A,R) , (R,B) , (A,C)
1d.	1. (A,B)	1. (A,B)
		2. (B,B)
	2. (A,B) , (B,C)	1. (A,B)
		2. (B,C)



(a) "Illegal" loop

(b) Equivalent Repetition Loop where N is geometrically distributed with mean = 4:

$F(N) = (1/4)(3/4)^N$

FIGURE 2.2. AN EQUIVALENT GRAPH USING A REPETITION LOOP

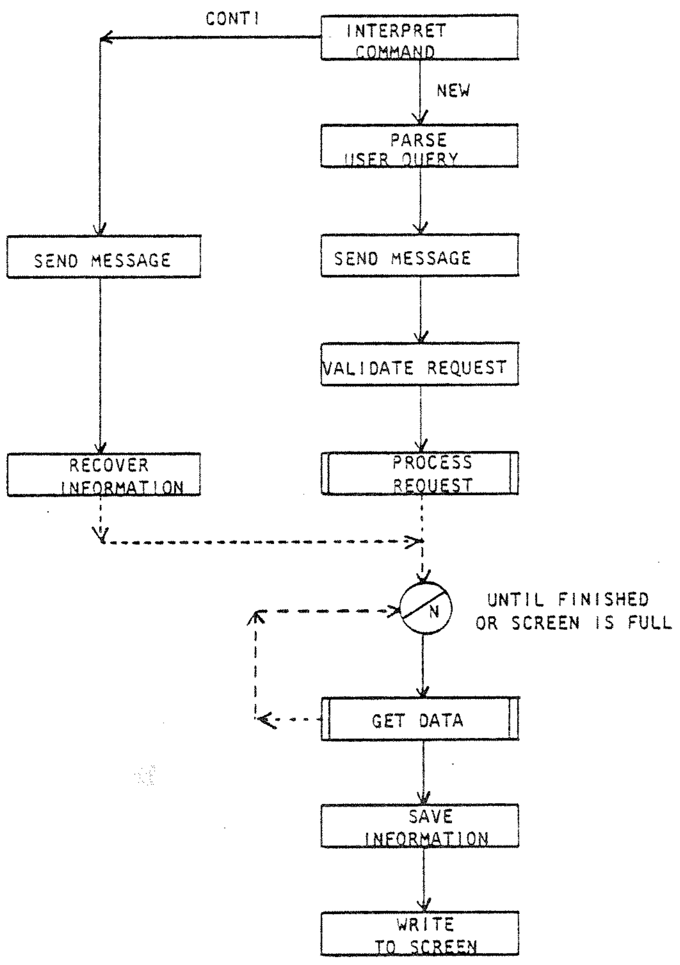


FIGURE 2.3. LEVEL 1: DATA BASE QUERY

nodes. The details of each function are in level 2 and are shown in Figure 2.4. The level 2 graphs also contain collapsed nodes.

The level 3 graph for the level 2 collapsed node, "get record," is shown in Figure 2.5. In it, "check memory" is a driver that first checks to see if the desired record is already available. If it is not, a component is called to allocate a buffer and issue a read. In both cases, another routine is then called to set a pointer to the desired record within a block. A self loop is required to show that some of the processing below the "check memory" node may not be done. Since there is a probability associated with the self loop, its associated node must be an or-node. By convention, drivers are not also or-nodes. The processing is more clearly visualized by explicitly representing the conditional execution of the allocate and read components. Therefore, a dummy node and arc have been included to represent the two possibilities resulting from the memory check. A dummy arc leads to it since there is no actual link to the dummy node. There is a double bi-directional arc between the "read" node and the "set pointer" node. It indicates that there is actually no connection between the two nodes, but that when the "read" is needed it is executed prior to the "set pointer."

2.2 Performance Specifications

Generally, functional design specifications contain insufficient data to analyze the performance of the finished software. Emphasis in this section is on the additional data required, rather than the format for the actual specifications. For example, an estimate is required for the CPU time: the estimated lines of code could be specified and mapped into an estimate for CPU

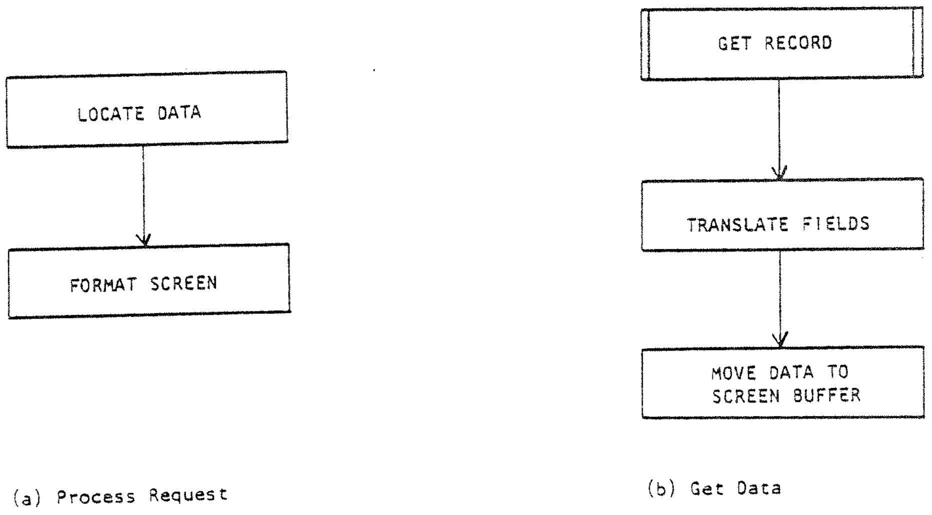


FIGURE 2.4. LEVEL 2

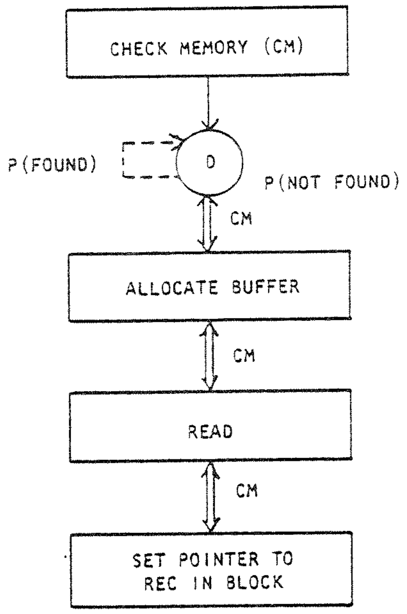


FIGURE 2.5. LEVEL 3: GET RECORD

time. It is assumed that the data can be derived in a reasonable manner, thus the consideration of the best initial format for the specifications and the mapping into the necessary data is not addressed.

The performance goal is the first specification necessary. It is used as the basis for comparison in the evaluation of the resulting performance predictions. The software design is satisfactory if the performance goal is met.

The specification of the performance goal for software designs is not only necessary for this analysis, but generally desirable. Once the expectations are clearly stated, the choice of appropriate designs and algorithms to be implemented is much easier. Several people are usually involved in the design and implementation of large software systems. It is important for them to realize what the overall performance goals are, so that decisions made on implementation issues are compatible. This methodology helps to analyze the impact of an implementation decision upon the rest of the software system.

The type of goal selected depends upon the nature of the software. A natural choice for an on-line transaction oriented system, eg., an airline reservation inquiry program, is the response time. For the software controling terminal activities, eg., polling and transmission, throughput is appropriate. For batch programs, the host system impact is important. The goal may be conditioned upon environmental factors. For example, the response time depends upon the number of terminal users. Information of this type must be included in the specifications: the response time must be less than 8 seconds with up to 10 users, and less than 12 seconds with up to 20 users.

The performance analysis techniques depend upon the type of goal selected. The analysis described in this chapter is appropriate for response time and throughput goals. Since competition for host system resources is not modeled here, the effects of multiple terminal users cannot be analyzed using these basic techniques. The other types of goals are accommodated in the extensions in chapters 3 and 4.

Response time goals can be derived quite easily for interactive jobs based on the type of user interaction required and the amount of processing necessary. Martin gives some guidelines for acceptable response times for various types of interactions [MAR67].

In order to determine if the goal can be met, information is required about the expected execution characteristics of each of the functional components. This information is shown in Table 2.3.

TABLE 2.3. SPECIFICATIONS FOR FUNCTIONAL COMPONENT EXECUTION CHARACTERISTICS

Factors	Units	Example
Component Identifier	Identifier	LOCATE
CPU time	ms	45
Module Size	bytes or words	20K words
I/O requirements: (1)		
File Name	Identifier (2)	ACCESS RIGHTS
Number of opens	integer	0
Type of I/O	identifier (2)	Read
Number of I/O's	integer	25
Information transferred	characters (K)	2
Operating System Calls: (1)		
Type of call	Identifier (2)	CLOCK
Frequency	Integer	2
Linkage: (1)		
Destination component	Identifier	FORMAT SCREEN
Type of link	Identifier (2)	dummy
Frequency	probability	1

(1) Information is repeated for each type (and file).
(2) Reference to information in Table 2.4.

The factors are selected because of their impact on the elapsed execution time of a software component. Since these factors are not easily estimated prior to implementation, two figures are used for each of them: an expected value and an upper bound. The expected value is the best guess for each of the requirements, while the upper bound is a number that can be met with high certainty, For example, a component will probably require 8 ms. of CPU time, but it will "certainly" require less than 30 ms. The upper bound for each component is used in the analysis to determine a worst case. If the goal is met by the worst case and the upper bounds are chosen judiciously, performance should be satisfactory.

Of course, the better the estimates of expected values and the tighter the upper bounds, the more meaningful are the results of the initial evaluation. As designs are detailed and as components are implemented, revised figures can be used for each of the factors. The upper bound and the expected value will then be equal. Thus, the accuracy of the evaluation will improve with successive iterations.

The linking information in Table 2.3 defines all functional components that could receive control from the component being specified, the probability that they will actually receive control, and the type of link. Examples of types are procedure calls, subroutines calls, and forks (for concurrent execution).

Information is also required on the execution environment of the software. Data must be supplied for all factors that affect a component´s total execution time (elapsed time). The operating system overhead influences the component execution time even when no other work is run. The specific data necessary to define the operating system overhead is shown in Table 2.4.

Clearly, the time required for operating system functions depends upon the workload on the system. Specifications are averages under a representative workload for the new software. They can be derived from measurements of the host system or from specifications for new systems. Default values for various environments are easily established.

TABLE 2.4. EXECUTION ENVIRONMENT SPECIFICATIONS: OPERATING SYSTEM OVERHEAD

Factors	Units	Example
Number of processors	Integer	1
Memory available	bytes or words	400K words
Process initialization (1)	ms	< 1 ms
I/O processing: (2)		
Type of I/O	Identifier	Read
Open/close time	ms	100
Supervisor time	ms	28
I/O completion time	ms	1 ms/K
Linkage: (2)		
Type of link	Identifier	Procedure | Dummy
Processing time	ms	1 ms | 0
Operating system calls: (2)		
Type	Identifier	CLOCK
Elapsed time	ms	1 ms
Process termination	ms	< 1 ms

(1) Includes resource allocation and activation time
(2) Information is repeated for each type.

TABLE 2.5. PROCESSING TIME COMPUTATION

Terms	Example
Process initialization	.5
I/O processing:	
Number of opens/closes x open/close time	+ 0
Number of I/O's x supervisor time	+ 700
Number of I/O's x information transferred x I/O completion time	+ 50
Operating system calls: Number of calls x time	+ 2
CPU time	+ 45
Process Termination	+ .5

Total processing time	798 ms.

2.3 Performance Analysis

The performance specifications are used, first, to evaluate the proposed software execution in isolation in order to determine a lower bound for the performance. Clearly, if the goals are not met in this environment, performance will be unsatisfactory.

The first step in the analysis is to use the specifications to construct a graph of the software execution profile. Next, the processing time for each node is computed as shown in Table 2.5. It is the sum of the times for process initialization, operating system calls, I/O processing, process termination, and the CPU time for that component. The computation is done twice, once using the expected values for the factors defined in Table 2.3, and again using the upper bounds. This gives (an estimate of) the expected value and the worst case elapsed processing time for each component.

These values can then be used to compute an elapsed time for the execution of all components. The computation differs depending on whether the performance goal is for maximum, minimum, or expected response times. The basic procedure for computing the expected response time is described next. Variations of it follow.

The analysis begins at the top level of the design. Table 2.6 contains a list of rules to be applied to each node in the graph. The initial node is evaluated first; the appropriate rule for it is selected and applied. After the application of the rule, the successor of the node in the sub-path is evaluated according to the appropriate rule. This analysis continues until the end of the sub-path is reached.

When a collapsed node is encountered, its corresponding sub-graph is evaluated recursively, using the same rules, to obtain the total for the collapsed node to be used in the calculations. Nested repetition loops are handled by first multiplying the repetition factors then using the product to calculate the elapsed

TABLE 2.6. BASIC RULES FOR OBTAINING EXPECTED ELAPSED TIME FOR EXECUTION GRAPHS

Rule	Origin Node Type	Procedure
1	Basic node Dummy node	Compute the node and arc cost and add to the current total.
2	Collapsed node	Evaluate the corresponding subgraph to obtain the node cost. Add it and the arc cost to the current total.
3	Repetition node	Save the repetition factor. Use it to compute each node and arc cost within the repetition loop.
4	Or-node	Select one arc out of the node and push the others onto a stack. Compute the node and arc cost and add the sum to the current total. Save the probability and use it to compute each node and arc cost on the sub-path out of the or-node.
5	Terminal node or Any node with multiple entrance arcs	Processing of the sub-path terminates. Pop an arc off the stack. Add the arc cost to the current total.

Note: arc cost in rules 1-4 refers to the (selected) arc originating at the node being evaluated. If it is a terminal node, the arc cost is zero. In rule 5 the arc cost corresponds to the arc from the stack.

time for each node in the nested loop. An or-node on a sub-path out of another or-node is handled by first multiplying the probabilities associated with the sub-paths then applying the appropriate product to each node on the sub-path out of the subsequent or-node. Both repetition factors and probabilities may apply to nodes.

When a node is encountered that has multiple arcs leaving it, evaluation of a new sub-path begins. One of the arcs is selected, the rest are pushed onto a stack. Evaluation continues with the new sub-path. When one of the following is encountered:

1. A terminal node
2. A node with multiple arcs into it (one or more paths join at the node)

processing of the current sub-path stops. If the node is a terminal node or if one or more of the other sub-paths leading to this node have not been evaluated, another arc is popped off the stack and the evaluation of a new sub-path begins. If all sub-paths into the node have been evaluated, an arc out of the node is selected and the evaluation of a new sub-path begins. If there are multiple arcs leaving it, the others are pushed onto the stack.

This process continues until all sub-paths have been evaluated. This is indicated by an attempt to pop an item off an empty stack. The expected response time is the value contained in the "current total".

When sub-paths join at a node, the branching probability of the sub-path below the node is the sum of the probabilities of the sub-paths that join. If the node is an or-node, the branching probabilities of each new sub-path are multiplied by this sum. The sum of the probabilities should never be greater than 1 since the entire sub-graph originates at a single initial node.

An example of the application of the rules to the graph in Figure 2.6 is shown in Table 2.7. The times shown for the components in the example are estimated elapsed times. They are computed using

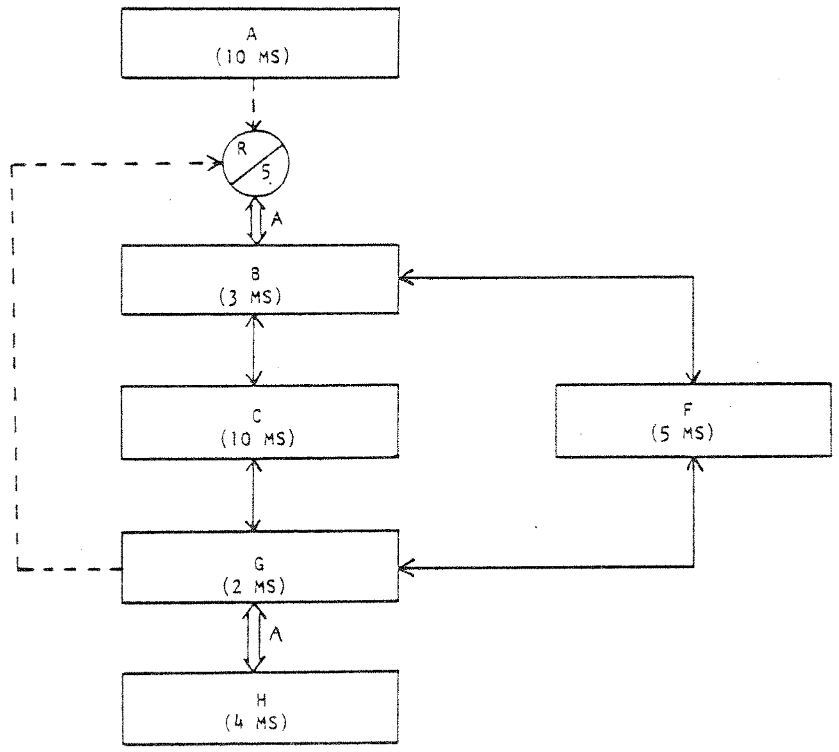


FIGURE 2.6. ILLUSTRATION FOR SUB-GRAPH EVALUATION

TABLE 2.7. EVALUATION OF A SUB-GRAPH

Step	Current Node	Current Total	Stack Contents	Repetition	Probability	Rule
Init	A	0 +10 ----- 10	empty	1	1	|| 1 || \/
1	R	+(5*1) --------- 15	empty	5	1	|| 3 || \/
2	B	+(5*3) +(5*.3*1) ---------- 31.5	(B,F)	5	.3	|| 4 || \/
3	C	+(5*.3*10) +(5*.3*1) ----------- 48	(B,F)	5	.3	|| 1* || \/
4	G	+(.7*5*1) ----------- 51.5	empty	5	.7	|| 5* \/
5	F	+(.7*5*1) +(.7*5*5) ----------- 72.5	empty	5	(.7+.3)=1	|| 1 || \/
6	G	+(5*2) +1 -------- 83.5	empty	5	1	|| 5* || \/
7	H	+4 ----- 87.5	empty	1	1	|| 5* || \/
End	none	87.5	empty	1	1	

Note: 5* indicates that processing of a sub-path is complete the procedure described earlier. All arc times in this example are 1 millisecond (ms.) with one exception: dummy arc times are zero ms.

The initial node, A, is considered first. By rule 1, arc (A,R) is evaluated first. The current total is updated with the cost of the node and arc (10). Node R is evaluated next. By rule 3, the current total is updated. In step 2, the repetition factor is saved and applied to all nodes and arcs through node G. Steps 2 through 5 involve probabilities as well. The resulting total for the graph is 87.5.

This computation is based on that of Kelly [KEL74]. He develops the graph theoretical proofs for the algorithms. There are three main differences between the two approaches:

1. The addition of a technique for handling hierarchical graphs, various types of nodes and control representations.
2. The introduction of a different order of evaluation.
3. The application of the loop repetition factor and execution probability to each component individually.

The first extension makes the specification and analysis procedure symbionic with a top-down design approach. The second and third are to provide a straightforward interface for the extension to handle data dependency. The order of evaluation facilitates the calculations, since repetition factors and branching probabilities are easily managed, and the range of nodes and arcs to which they apply is easily identified.

A variation of the above procedure can be used to obtain estimated minimum and maximum response times. Finding the maximum response time is equivalent to finding the longest path in the graph. In this case, the cost of a collapsed node is the cost of the longest path in its corresponding sub-graph. Repetition loops are evaluated as in the previous analysis. Or-nodes need no special processing; the longest sub-path is desired regardless of its probability of execution. It is not necessary to distinguish between parallel and sequential processes in this computation since the single path that dominates is sought regardless of its characteristics. The procedure for finding the minimum response time is analagous.

The throughput can be obtained from the calculated response time by applying Little's law [LIT61]. Of course, a specification for the arrival rate is required.

2.4 Performance Evaluation

These analysis techniques are employed for both the expected value and worst case specifications. If the worst case elapsed time is less than the response time goal, the performance of the software should be satisfactory. If the worst case is higher, but the expected value is lower, the performance is marginally satisfactory. If it is marginal, or if neither the expected value nor the worst case response time is less than the goal, an analysis is needed to determine the critical components and possible bottlenecks. Possible bottlenecks include excessive CPU time, I/O processing, operating system calls or linkage time.

Sufficient information is contained in the graphs for the automatic generation of a frequency distribution showing, for example, each component of the software and its total CPU time requirement. Reports of this type are generated for all the specification factors. The ratio of each to the elapsed time is computed; the largest ratio indicates the most critical resource. A ratio significantly higher than the others indicates a bottleneck. The components with high requirements for that resource are the critical components.

Other useful reports show the number of times each component is invoked, and the types of links and their frequency of occurrence.

Components that are invoked many times are also critical since small changes in execution characteristics have a significant impact.

Possible solutions for bottlenecks are:

1. Reducing the resource requirements by revising the design and thus the specifications
2. Reducing the system overhead by upgrading hardware and thus revising environment specifications
3. Restructuring the processes (combining them) to eliminate or reduce linkage time

The visual nature of the graphs often reveals structural type optimizations to the design, such as removing components from loops whose function is loop invariant. Other similar compiler-type optimizations are equally applicable to software designs.

Note that these are passive resolutions to the bottlenecks since the problem is not resolved by the methodology. Nevertheless, the means for identifying the problem and evaluating possible solutions is provided. An example of the application of these techniques is described next.

2.5 Example

Consider the software execution graph in Figure 2.7. Only the top level of the processing is illustrated here. The CPU time and I/O requirements for each component are shown in Table 2.8.

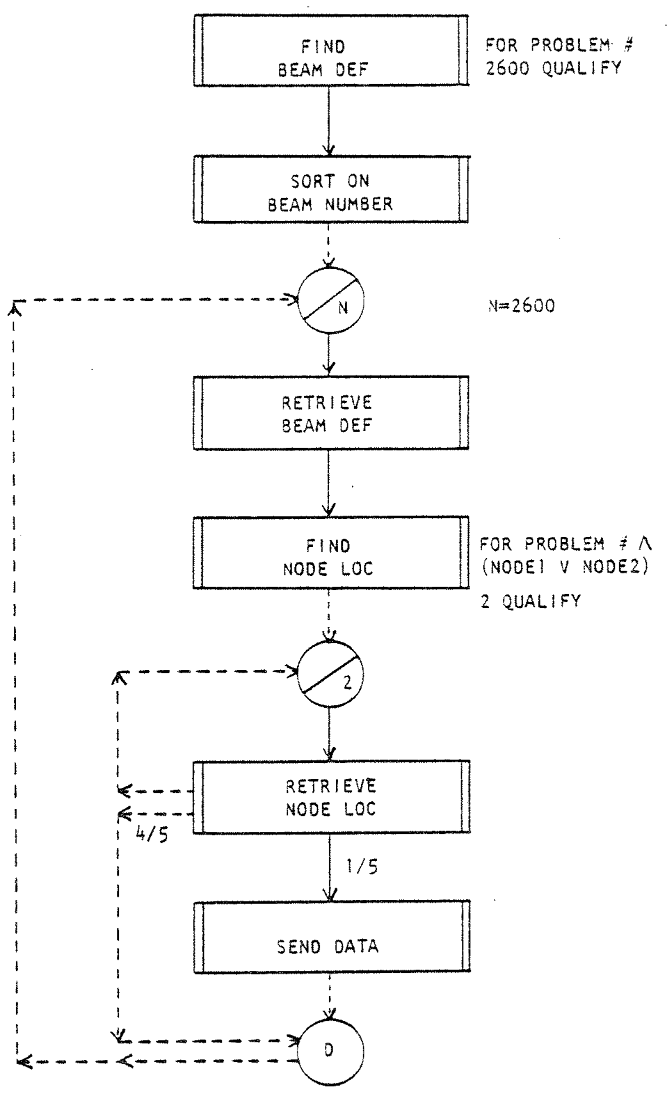


FIGURE 2.7. OPTIMIZATION EXAMPLE

TABLE 2.8. RESOURCE REQUIREMENTS FOR OPTIMIZATION EXAMPLE

Function	Disk Accesses	CPU Time (ms)
Find beam definition	7	111
Sort on beam number	72	32,644
Retrieve beam definition	72	88,832
Find node locations	21	3,018,726
Retrieve node locations	36	177,016
Send data	0	2,600
	---	------------
Total	208	3,319,929 ms.

The elapsed time to complete an I/O is assumed to be 30 ms. Other specifications are unimportant in this example.

The average response time for this scenario is 3326 seconds (55.4 minutes). This is clearly unacceptable for an interactive transaction. The bottleneck analysis indicates that the CPU is the critical resource since it has a higher ratio to the elapsed time than the I/O ratio. Furthermore, the "find node location" component is the critical component.

The processing details of this collapsed node are not shown; however, close examination of them indicates that it invokes a "find" data base command once for each of the three keys then takes the intersection of the records that qualify. Also, the result of the "find" for the problem number key is invariant throughout the loop and need not be repeated. A knowledge of the nature of the problem leads to the observation that most of the time (85%) the "find" on the node 1 key yields the same result as the "find" on the node 2 key from the previous pass through the loop, and need not be repeated.

These optimizations are reflected in the execution graph in Figure 2.8. This graph is more complex; however, the total processing requirements are reduced, as shown in Table 2.9.

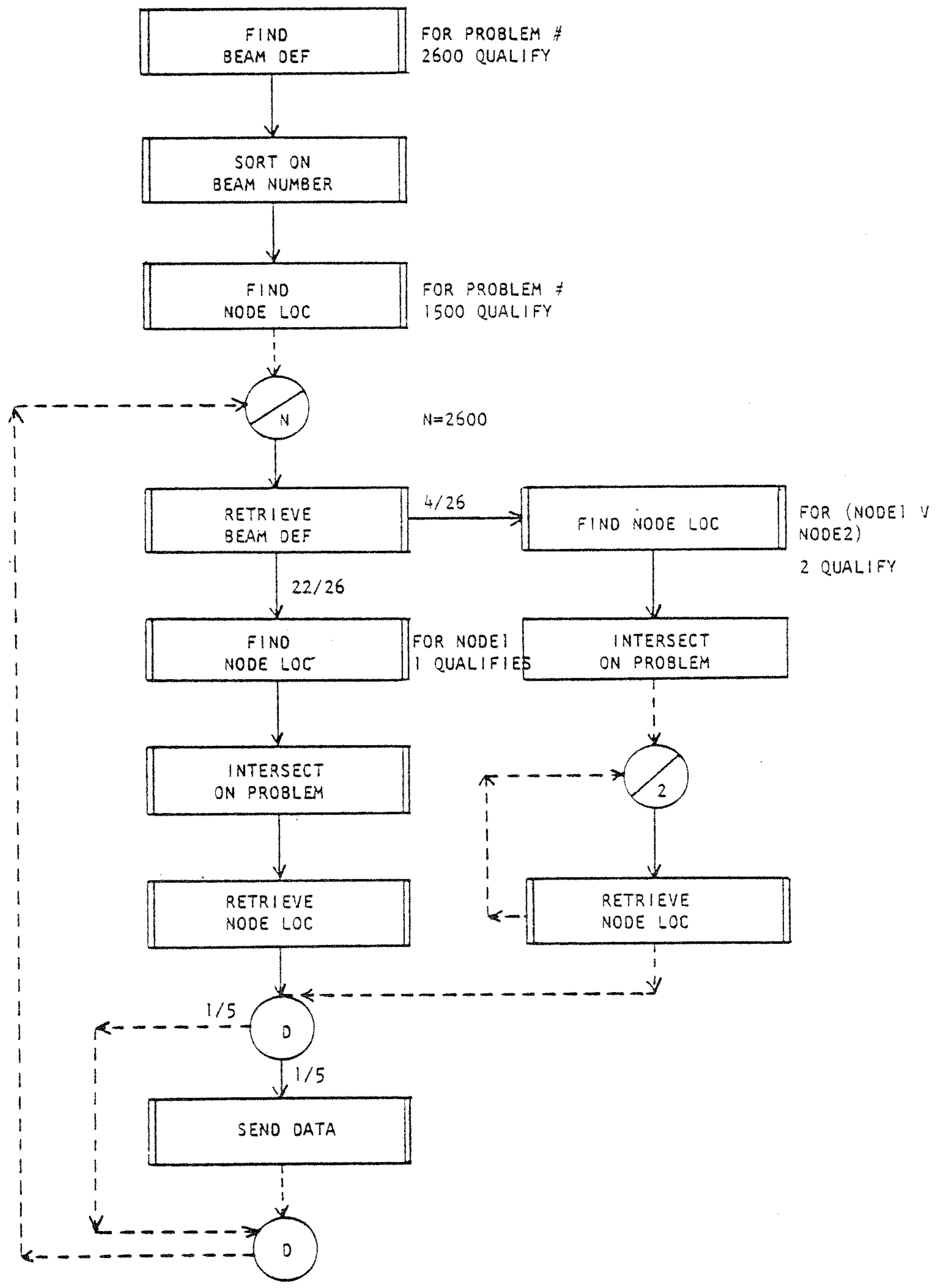


FIGURE 2.8. REVISED OPTIMIZATION EXAMPLE

TABLE 2.9. RESOURCE REQUIREMENTS FOR REVISED OPTIMIZATION EXAMPLE

Function	Disk Accesses	CPU Time (ms)
Find beam definition	7	111
Sort beam number	72	32,644
Find node location	4	1,075
Retrieve beam definition	72	88,832
Find node location:		
B-tree I/O	17	102
Find 2 nodes	--	44,000
Retrieve 2 nodes	--	27,200
Find 1 node	--	26,000
Retrieve 1 node	--	74,800
Record I/O	36	216
Send data	0	2,600
	---	-------
Total	208	297,580 ms.

The response time has been <u>reduced</u> by 3023 seconds, a substantial savings!

The response time (303 seconds) is still unacceptable for most on-line applications. Another optimization, storing the "beam def" data in beam number sequence, precludes the sort. The resulting response time is 269 seconds. This process continues until a resulting response time of 82 seconds is obtained.

The performance is still only marginally acceptable, but it is a dramatic improvement over the original design. The bottlenecks are detected and corrected prior to actual coding, therefore, the modifications require minimal effort.

Having demonstrated the basic methodology, the next step is to incorporate the environmental effects. The specification and analysis of data dependent execution characteristics are next, followed by the inclusion of contention for memory and the effect of other software competing for host system resources.

CHAPTER 3
ENVIRONMENTAL FACTORS

3.0 Overview

The basic methodology incorporates a static analysis to derive estimated response times for software execution in an environment without competition for resources. An extension to the basic methodology is required when the performance impact of environmental factors is to be considered. It embodies dynamic analysis techniques to manage the following:

1. data dependent execution characteristics
2. external competition for host system resources
3. memory contention and overhead.

Additional performance metrics are obtained from these dynamic analysis techniques. The performance goals are expanded to include consideration of resource utilization, host system impact, conditional goals, and response time distributions and variances.

Resource requirements of software systems may vary substantially between executions against different sets of data. This variability is primarily due to changes in execution paths. The number of times that components are executed is typically dependent upon data characteristics while the individual component execution characteristics are often nearly constant.

An example is a data base retrieval module that is repeated many times. Data is retained in buffers and the retrieval can sometimes (conditionally) be satisfied with data in buffers, thereby eliminating processing steps to initiate I/O and thus reducing resource requirements.

Variability may appear within a component at a given level of resolution. For example, the amount of CPU time required by a compiler component depends upon the size and structure of the program to be compiled. In this case, however, when the processing details of each component are resolved, the execution paths of sub-components are once again observed to be the primary cause of the variability.

A possible solution to the data dependency problem is to devise a representative benchmark workload and specifications of expected values and upper bounds for that workload. The results, however, are still not generally applicable to other workloads.

The preferable solution is to identify the data objects upon which performance depends and provide specifications in terms of those objects. The specifications contain either parameters or random variables to represent the data objects. The response time algorithm is then modified to accommodate these parameters. Additional algorithms are added for the calculation of the variance and distribution of response time when random variables are used in specifications. Approximation techniques are included that greatly simplify these calculations with little effect on the usefulness of the results.

The second environmental factor addressed is the external competition for host system resources. Other software that executes on the host competes for shared resources such as the CPU and I/O devices. This introduces queueing delays when a job is ready to use the resource but must wait until it is free. External competition is important not only because of its effect on the software design of interest but also because the new software may exhibit satisfactory response time characteristics at the expense of the response time of the other, perhaps more important, work on the host. It is also important to know how many users of the software can be supported without a significant degradation of response time.

A solution to this problem is to construct a queueing network model of the host system to reflect its performance with the competing work [KLE75,KLE76]. Graph analysis algorithms are applied to yield data for the model parameters necessary to include the new software. Solution of the resulting queueing network model, the elementary model, yields the revised response times for existing work, response time for new software, and additional performance indicators such as resource utilization, wait (queue) time for each resource, and throughputs. The additional metrics are useful for the evaluation of other performance goals as well as the identification of potential bottlenecks.

The third environmental factor is the contention for executable memory on a non-paged host system. When a host system with non-paged memory is shared among multiple users, there is usually contention for the available memory. This limits the number of users that can execute in parallel and adds additional overhead for swapping, that is, replacing inactive users (who hold memory while waiting for resources) with users who can execute.

For example, a transaction arriving to a busy system may experience a delay until enough memory becomes available for the necessary programs to be loaded. During execution, if a program has a long wait for the results of a called sub-program, it may be swapped out of memory. When the results are available, the job is delayed until the program is swapped in again. The delay experienced by the job depends upon the host system configuration and its workload.

The solution to this problem is to first insert special swap in and swap out nodes into the execution graphs to represent the additional resource requirements. The elementary model is augmented to produce a model of the system with the swapping activity incorporated: the swapping model. Then, the queueing network modeling techniques of Brown, et.al., as extended by Keller, are employed to quantify the memory wait and its impact on response time:

the memory model [BRO77,IRA79]. Additional algorithms are introduced to obtain the memory model parameters from the execution graphs.

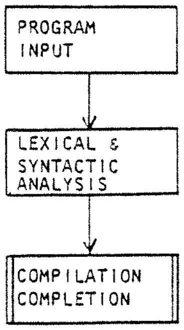
The remainder of this chapter elaborates on these dynamic analysis techniques and the necessary performance specifications. The data dependent execution characteristics are discussed next, followed by the external competitive effects, and memory analysis.

3.1 Data Dependency

When the resource requirements of software vary considerably with the data to be processed, specifications for the expected value and upper bound provide insufficient information to adequately characterize performance. This problem is resolved by using variables in an expression of the relationship between the behavior determining data objects and the variable resource requirements, execution probabilities, or loop repetition factors.

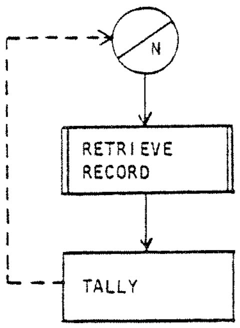
Consider the examples shown in Figure 3.1. In 3.1a, the CPU time required for the "lexical and syntactic analysis" component depends upon the number of statements in the source program. A variable loop repetition factor is shown in 3.1b and variable execution probabilities in 3.1c. The calculation of the average, minimum or maximum response time proceeds using the algorithms in Chapter 2; the calculations now include variables as well as constants. Figure 3.1 illustrates some average response time calculations involving variables.

Values are assigned to the variables during the evaluation as soon as sufficient information is available. Consider the example in Figure 3.1c. The value of P depends upon the access method, the number of records in a buffer, the total number of records, and the number of buffers retained in memory. The relationship is defined as



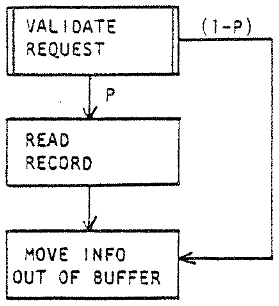
CPU Requirements	Computation
10	10
	+ 1
.03 S	+ .03 S
	+ 1
.75 S	+ .75 S
	12 + .78 S

(a) Resource requirements: S represents the number of statements in the source program.



CPU Requirements	Computation
	N x 1
20	+ N x 20
	+ N x 1
5	+ N x 5
	N x 27

(b) Loop repetition factor: N is the number of requests in the data stream.



CPU Requirements	Computation
5	5
	+ p x 1
5	+ p x 5
	+ p x 1
	+ (1-p) x 1
2	+ 2
	8 + 6p

(c) Execution Probability: P is the probability that the data is not in memory.

FIGURE 3.1. VARIABLE EXECUTION CHARACTERISTICS

Random access: $P = 1 - (NB \times RB) / RT$

Sequential access: $P = 1 / RB$

where NB = number of buffers

RT = total number of records

RB = number of records in a buffer

Since the response time is calculated for a specific scenario, values for each of these particular variables are provided as part of the specification process. Only one of these variables, the total number of records, will change from one execution to the next. The rest are actually design decisions and will probably remain constant after implementation. Using variable specifications at this point in the design process allows the analysis of alternate design decisions.

This example illustrates a problem in the application of previous graphical analysis techniques [KEL74]. The algorithm for replacing a repetition loop by a single node with an equivalent execution time (the repetition factor times the elapsed time per repetition) requires an assumption that the time for each repetition of a loop be independent of the time for the previous repetition and the number of repetitions. The assumption is violated in this example for the sequential access method since the number of times the data is in memory is dependent on the number of repetitions of the loop.

The answer to the problem is to defer assigning a value to the variable, P, until the value of the loop repetition factor, N, is known. The total elapsed time is then calculated for all loop repetitions. This resolves another problem, that is, the actual number of times that the "read record" component is executed must be an integer. The ceiling function (next higher integer) is added to the expression for P resulting in

$$P = \lceil N / RB \rceil$$

Note that the ceiling function can only be added to the expression for all loop repetitions. Otherwise P would always be one, incorrectly indicating that "read record" is always executed.

In the previous example, reasonable values for all of the data dependent variables could be provided with the scenario specifications. This is not always the case. For example, the value of the loop repetition factor, N, in Figure 3.1b may depend on data characteristics not known a priori, such as the number of employees earning less than \$1000. When this happens, the variables are retained throughout the computation. The result is a response time function in terms of the dependent variables. A conditional performance goal is then introduced, that is, the goal depends on the value assigned to the variable in the response time function. For example, if the loop in Figure 3.1b is repeated once for each employee who earns less than \$1000, the conditional response time goal could be 8 seconds if 25 or less employees qualify, 12 seconds if up to 100 qualify, 20 seconds if there are more.

The response time analysis of software with data dependent execution characteristics is best demonstrated by an example. Consider the software described in Chapter 2 and depicted in the graphs in Figures 2.3, 2.4, and 2.5. Suppose the elapsed time for each basic node has been calculated; the results are in Table 3.1.

TABLE 3.1. ESTIMATED PROCESSING TIME OF BASIC NODES

Name	Processing time (ms.)
Level 1:	
Interpret Command	2
Parse	50
Send Message	5
Validate Request	250
Recover information	2
Save information	2
Write to screen	20
Level 2:	
Locate	45 + (36 * #reads)
Format screen	5
Translate fields	5
Move data	1
Level 3:	
Check memory	2
Allocate buffer	5
Read	30
Set pointer	1

The arc cost is one millisecond (ms), except the dummy arcs which are zero ms. The variables will temporarily remain undefined. Note that the time required for the terminal I/O is excluded from the model, so the calculated response time is the elapsed time in the host system.

The calculation of the average response time is carried out using the variable names rather than substituted values. The result is a function for the response time in terms of the variables. Table 3.2 illustrates the calculation. The assumption that p(not) equals .9 is obtained by assuming there are 10 blocks in the file, retrievals are random, and there is only one buffer. Thus, there is one chance in ten that the desired block is in memory.

The average response time is evaluated with respect to the scenario and the specified value of each variable. The branching probabilities, p(conti) and p(new), are implicitly defined by the scenario. The number of reads and the number of programs that qualify are specified as part of the scenario. In Table 3.3, scenarios 1, 2, and 3 illustrate three such calculations.