The use of performance models in systematic design

by K. M. CHANDY, J. MISRA, R. BERRY, and D. NEUSE

University of Texas
Austin, Texas

ABSTRACT

The paper describes a top-down methodology for evaluating the performance of computer/communication systems and describes tools that help in implementing the methodology. It also deals with performance analysis during the design of new hardware and/or software systems. The goal of the methodology is to detect and correct performance problems early in the design cycle.
INTRODUCTION

Much has been written about systematic design.1,2 This paper describes tools that aid in designing to meet performance goals. We briefly review top-down performance design methodology as expounded in Smith and Browne and elsewhere and then show how the methodology is implemented by the use of appropriate tools.

In a data processing system, entities called transactions consume resources. At different stages in the design cycle it is convenient to view transactions and resources at different levels of detail, incorporating more detail as the design proceeds. For instance, at an early stage in a software design, a transaction may be “opening a bank account” or “sending a reminder about accounts past due.” At a later stage in the design cycle the “send reminder” transaction may be refined into a sequence of smaller transactions such as:

- If amount owed exceeds AMOUNT-LIMIT, start transaction to cancel credit.
- If account is past due for a period exceeding TIME-LIMIT, start transaction to send warning letter and notify collection agency.
- Send reminder to customer.

Note that the manner in which a transaction is refined may be conditional; i.e., the refinement may depend on certain conditions (such as “amount owed exceeds time limit”). The refinement must also indicate which subtransactions may be executed in parallel and which must be executed sequentially. We will discuss a formal method for specifying the refinement of transactions later.

It is convenient to view resources as logical resources, physical resources, and a mapping from logical to physical ones. For instance, a transaction specification might state that a transaction needs to execute 10,000 lines of instructions; here the resource specification is logical, because the identity of the CPU to be used (if there are several) is not specified, and neither is the duration of time for which the CPU is held. Of course, the duration of time for which the CPU is held will depend on CPU speed. It is convenient to separate the hardware (physical-resource) specification from the software (transactions requesting logical resources). An application program will have the same specification no matter what hardware it runs on. A physical computing system, computers (CPUs, channels, controllers, disks), and message communication links will be specified in the same way, no matter what application programs run on them. Changing the way an application program runs (its priority, the specific site at which it runs, the allocation of databases to a different set of disks) only changes the mapping from logical resources to physical resources.

If both hardware and software are being designed, the specifications of the physical resources are also refined as the design proceeds. If the hardware is in place and only the software is being designed, the specifications of the physical resources are available in detail and no refinement is necessary. At the start of a design, a physical resource may be a computer; as the design progresses, this physical resource may be refined into a memory resource, CPU resources, and an I/O resource. At even later stages in the design the I/O resource may be refined into channels, controllers, and disks.

The refinement of transactions and resources may not proceed in a synchronized fashion. Either the application programs may be already given while a new hardware system is being designed, or the hardware system may be given while a new application is being designed. To allow a decoupling of application program design and hardware design, the logical mapping must be capable of mapping any level of transaction definition to any level of hardware definition. The mapping is discussed later.

In the next section we describe a tool called Performance Analyst's Workbench System (PAWS) that can be used to predict performance at various stages in the design cycle, given various levels of definition of transactions and hardware. PAWS is derived from other languages, notably RESQ.3 This discussion also applies to RESQ.

PAWS

PAWS is a tool developed by Information Research Associates.4 We will not describe PAWS in depth, but we will describe it in enough detail that the reader can understand how PAWS can be used to predict performance within the framework of a systematic design methodology.

PAWS is a very high-level simulation language designed specifically to model computer/communication-office systems. The specificity of the problems PAWS was designed to handle is both its strength and weakness: computer/communication systems can be modeled easily, whereas GPSS, SIMULA, or SIMSCRIPT are probably preferable for general-purpose simulation. Resources modeled by PAWS include memory, buffers, CPUs, and I/O devices. Scheduling disciplines to handle memory in PAWS include first-fit and best-fit. New disciplines and new resource features are being added. Thus it is easy to study computer design problems such as the memory fragmentation problem in PAWS. A variety of scheduling disciplines are available for resources such as CPUs.

Entities that use resources in PAWS are called transactions (see Figure 1). There are several facilities to control the manner in which transactions use resources; facilities include branching on condition (to simulate if-then-else, while loops, go tos), probabilistic branching (to simulate nondeterminism), interrupts (to simulate one transaction's being inter-
interrupted by some external action) and forks/joins (to simulate parallelism). Transactions have local variables, and the system has global variables that may be accessed by all transactions. Transactions acquire and release resources at resource-manager nodes. For instance, a transaction may go to a memory management node and request a block of 100 words of contiguous memory. The transaction will have to wait at the memory management node until its request is satisfied or it is interrupted. Thus the sequence in which transactions request resources can be represented by a graph with resource-management nodes, decision nodes such as forks/joins (to control the flow of transactions), and edges showing how transactions use resources.

Though the specific sets of resources and transaction control facilities vary from one computer modeling language to another, the key ideas are common.

In the next section we show how computer modeling languages can be used in systematic design. To help focus our discussion, we shall consider one specific language, PAWS, though the discussion applies equally well to languages such as RESQ.

**Design Methodology**

Performance modeling in the design of computer/communication systems occurs in three categories: modeling of (1) application programs (transactions), (2) hardware (resources), and (3) the map between application programs and hardware. We now consider each of the three categories in turn.

**Application Programs**

The first step is to identify all transactions at a coarse level of detail—the business level. An example of a transaction at this level of detail is "adding to a checking account." The logical resources used by the transaction are identified next, also at a coarse-grained, or business, level of detail. Examples of logical resources for the adding-to-checking-account transaction are drive-in-windows, bank tellers, communication lines, and computing facilities. At this point in the design it may not be appropriate to describe the computer facilities resource at a finer level of detail; thus all resources of bank computing—terminals, CPUs, disks, databases—are lumped together into a single entity. If a bank is considering setting up small suburban drive-in branch offices, resources such as real estate and personnel may be more important than issues such as the number of disk accesses. At early stages in the design, the designer may be concerned with questions such as these: How many square feet, drive-in windows, and bank tellers will I need? What is the overall additional load on computing facilities? If answers to these general questions suggest that the new application is cost-effective, the design should proceed further.

We shall now study the problem of specifying demand for a composite resource such as a computing facility. An adequate level of detail for specifying the load placed by a new application on computing resources can be derived from accounting data. The amount charged for running a transaction on a system is a function of the amounts of resources consumed. In many cases, computer centers use linear functions such as service units, with weights attached to consumption of the different resources—CPUs, I/Os, memory. Our goal at the first stages in design is to estimate the service units (or some other composite accounting entity) required by each transaction in the new application. Using service units or an accounting function is a very imprecise way of estimating load, because a computing system is not a homogeneous entity, but consists of very different parts. However, this level of detail may be sufficient at the first stage in design. The level of detail used in modeling real estate (drive-up windows), personnel, and computing depends on their relative incremental costs in the new application.

The amount of service units required by a transaction in a new application is estimated by comparing the new application with an existing one. No attempt is made to analyze the application in detail. It is quite common to hear an analyst say, "In Company X they handle about 20,000 transactions of an application very similar to our new application, and their application takes about 30 percent of their machine." It is from such imprecise statements that the first models should be built, because such statements give one a ballpark estimate of load. As a first guess, we might estimate that each transaction in the new application takes 0.3/20,000 hour of a system equivalent to Company X's.

An initial model for a new drive-in-banking facility may have the form shown in Figure 2.

Figure 2 has also set the level of detail for the resources. For instance, we assume $M$ bank-tellers, all of whom are equivalent, though in reality the bank manager or supervisor may be the only one who can handle special transactions. We have also assumed that there is enough bandwidth in the communication lines linking the tellers to the computers that communication delays may be ignored.

The mapping of logical resources to physical resources is straightforward if the computing system is centralized: all logical computing system requests for all transactions map into the same centralized computer. When the computing system is distributed, the map from logical to physical states where each transaction will be processed. This map may be dynamic; however, as a first pass, it is often sufficient to assume that the

![Diagram of a business transaction](image-url)
Performance Models in Systematic Design

Refinement

As the design proceeds into greater detail, the application/hardware/mapping models become refined. The assumptions made in the earlier stages of design should be checked against the results of the refined models. If these latter results show that the assumptions made in the earlier models are grossly in error, the decisions made in the earlier stages of the design cycle must be reexamined.

Refining a transaction model consists of (1) describing a transaction in terms of more detailed and smaller transactions and (2) specifying the logical resources at a greater level of detail. For instance, a transaction to process accounts past due that is specified as using $x$ seconds of a standard computing system may be partitioned into several transactions, as shown in Figure 3.

In this example a single transaction at a coarse level of detail is partitioned into four smaller transactions. If the amount due is less than LIMIT, the single business transaction manifests itself as two transactions, (1) figure amount owed and (4) write letter. If the amount due is greater than LIMIT, the single business transaction manifests itself as four transactions: after completing 1, 2 and 3 are executed in parallel; after both 2 and 3 are complete, 4 is executed. Resource demands for the smaller transactions are computed in the usual way. Branch points such as those shown in Figure 3 are normally modeled as probabilistic branches; some fraction of all transactions takes one branch, and some fraction takes the other. The modeling languages have explicit facilities for models of fork/join and routing, so transaction refinement is straightforward.

In addition to specifying a business transaction in terms of more detailed transactions, the refinement step may also specify logical resources in greater detail. For instance, instead of specifying a transaction's demand for data processing resources in terms of $x$ seconds of system time, we may specify the demand in terms of $x_1$ units of memory, $x_2$ units of CPU time, and $x_3$ I/O accesses to data sets. The specification of which data sets are accessed and the relative frequency of access to each of the data sets is not given here; but it will be given, with further refinement, in later stages of design.
As the design proceeds, the refinement of the specification of the physical resources will continue as well. Thus a computing system may be refined into memory, CPU, and I/O devices. If the refinement of transactions and physical resource models proceed hand in hand, the logical-physical mapping is straightforward. Let us consider the case where the hardware design lags behind the application program design. Suppose the hardware model is still that of a composite computing system. In this case the logical-physical map must map a relatively refined application program onto a relatively gross hardware model. In the initial design stages, the load placed on data processing resources by transactions is in terms of a single metric (processing time or service units) on a “standard” or benchmark computing system. The refined transaction model will result in better estimates of resource demand, though once again the estimates will be made with respect to the standard system. As long as the hardware model remains at the composite computer system stage, resource demand must be estimated as a single metric on a standard system and the proposed hardware must be defined in terms of its speed relative to that of the standard system. Now consider the case where the hardware model has also been refined into distinct physical resources—memory, CPU, and I/Os. The logical-physical map does not change, because once a transaction is assigned to one computing system, all logical-resource requests (memory, CPU, and I/O) will refer to physical resources within that computing system. Thus the map could continue to be in the form of an array L. The use of the physical resources by transactions in this more detailed model is also easily represented (Figure 1).

In the detailed model of the data-processing system, the other aspects of the business should not be modeled at all or should be modeled in an extremely approximate fashion. For instance, we considered a problem in which, at the first stage in the design, tradeoffs were made between real estate (drive-in windows), personnel (bank tellers), and data processing, using approximate models of the data processing system. At the next stage we construct more detailed models of each of the subsystems, including data processing. However, at this second stage we will ignore the other components, such as drive-in windows and bank tellers, and focus primarily on the data processing system. The load offered to the data processing system in terms of transactions per hour is obtained from previous models, but no other aspects of the previous model are brought into the current model. The objective is to keep each model at a manageable size. Only if the results of the detailed model show that the assumptions made in the coarse model are grossly erroneous do we go back to the coarse model.

The use of modeling languages in the design process in a systematic, top-down refinement procedure is extremely helpful in catching performance problems early. Moreover, the methodology is manageable, and the tools to implement the methodology exist.

SUMMARY

We have shown how modeling languages such as PAWS and RESQ can be used with performance design methodologies such as those in Smith and Browne 1 and Le Mer. 2 Our approach consists of building separate models for application programs, hardware, and the map from programs to hardware. At each step in the design the models for application programs and hardware are refined. We have shown that these models can be represented naturally in modeling languages.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the U.S. Air Force, AFOSR-81-0205.

REFERENCES