Performance Modelling of Concurrent and Parallel Software

C. M. Woodside
Telecommunications Research Institute of Ontario
Dept. of Systems and Computer Engineering
Carleton University
Ottawa, Canada K1S 5B6

Abstract

Software made up of communicating tasks on multiple processors exhibits behaviour which is substantially different from software on a uniprocessor. In particular it exhibits the phenomenon of software bottlenecks. Performance models must incorporate the new effects, which go beyond more conventional models such as queueing networks. This paper deals with systems in which software tasks communicate by passing messages, rather than by shared variables (although the message-passing may be implemented in shared variables, as in Mach). Three basic classes of model have been proposed: extensions of queueing networks, program flow models, and models of interacting objects. This paper describes their different strengths and their relationships.

Résumé

Les logiciels composé de plusieurs tâches qui communiquent sur des processeurs multiples démontrent un comportement qui est bien différent de ceux sur un processeur unique. Le phénomène démontré le plus notamment est l'embouteillage. Les modèles de performance doivent incorporer les nouveaux effets, qui sont bien au delà des modèles plus conventionnels tels les réseaux composés de files d'attente. Dans cet article, nous discutons des systèmes où les tâches communiquent au moyen de messages, plutôt qu'au moyen de variables en commun (bien que l'envoi de messages puisse être fait à travers de variables en commun, p.ex. en Mach). Nous proposons trois classes de modèles : extension des réseaux de files d'attente, des modèles de flots, et des modèles d'objets intéractifs. Nous décrivons leurs différents avantages et les relations entre-eux.

Introduction

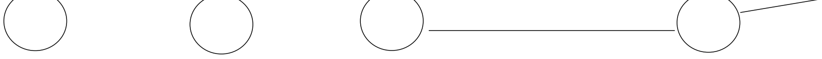
Computer system performance is limited by contention for resources such as processors, memory, and disks. In concurrent systems there are more complicated causes of delay, with their origins in the software structure and behaviour and also in the more complex architecture of the hardware.
We consider models which may be based either on simulations or on analytic techniques, and which are used to obtain insight into performance limitations, or to compare alternatives for systems that have not yet been built. In the case of distributed and parallel systems the model must describe the behaviour of the software in more detail than is usual in, for instance, conventional data center modelling (as described for instance in one of the chapters in [1]).
This paper describes three major approaches to modelling software effects on performance, namely

- extensions to queueing models originally designed for modelling hardware,
- execution flow models (activity sequence models),
- models of communicating objects, both hardware and software.

Each approach has a particular strength. For example, extending the classical model makes sense when the system is dominated by independent application processes, but also includes some particular software effect, such as lock contention or fork-join parallelism. Execution flow models are appropriate for critical-path type problems where there is only one program, but it has special internal parallel structure, as is common in signal or image processing. Models of communicating objects are well-adapted to client-server architectures with coordinated components or server task heirarchies, such as are frequently used in networks (using remote procedure calls) or in real-time systems. A hazard of such systems, which the model can expose, is the software bottleneck. The three approaches are related to each other and the models can usually be transformed from one form to another.
The systems to be modelled may be implemented in various kinds of distributed environments, as illustrated in Figure 1, including bus-based multiprocessors, local networks, and wide-area nets. An important restriction in scope is a focus on message-passing systems, and an exclusion of shared memory systems. This is for purely practical reasons of length.
This paper describes the three approaches and their relationships to each other. Space limitations and the size of the very extensive literature prevent a complete survey of the relevant literature, but more reference material is available from the author in [2].

Extending the Classical Queueing Network Model

The classical queueing network model (eg., Sauer and Chandy [3]) captures much of the behaviour of such systems, especially when it is adapted in a few simple ways as shown by



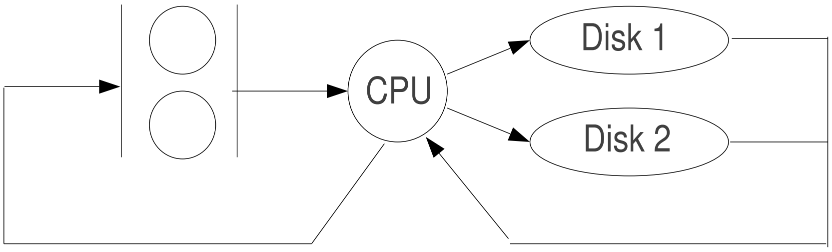


Figure 1: Various Hardware Environments for Concurrent and Parallel Software with Message Passing (buses, local and wide area networks)

Figure 2: The 'Classical' Model of a Shared Computer

these authors or Lazowska et al [1]. The basic queueing network model of a computer is illustrated in Figure 2 for a simplified timesharing computer.

The arrows define paths from one resource to another, followed by "customers" representing user tasks which are independent of each other and only interact through contention for the resources. The resources are all hardware devices, managed by schedulers. There may be multiple classes of customers representing, for instance, batch and timesharing jobs. The key to the popularity of this model is the existence of a relatively simple solution in product form for a restricted but interesting class of systems [3]. Mean Value Analysis, as described eg. in [1] is commonly used to compute solutions.

In practice certain advanced model features, outside the range of simple analytic "product-form" solutions, are required– for example to represent memory contention, or combinations of execution time distributions and queue disciplines which cause difficulty for the standard model. Approximate extensions in the form of equivalent servers to represent these effects have been developed and are surveyed and explained in [1] and in [4].

An important and very simple analysis may be carried out to find the bottleneck server of a closed model like Figure 2 (a closed model has a constant population of users). The bottleneck is the server with the largest workload per response, and it limits the throughput of the entire system to a rate which saturates it. When there are several independent workload classes there may be more than one bottleneck. Such saturation bounds and refinements of them were studied in [5].

A second general type of performance-limiting effect may be termed a path length bound, and is due to the total work done in one average response. In light loads this limits the rate of operation to less than $(\text{total work})^{-1}$.

The extended model features mentioned above have also been used to analyse software effects such as critical sections and synchronization delays, however they have weaknesses for our purposes. The software entities and design decisions tend to be stripped away, and the "customers" in the model, which now represent parts of a parallel program, are still assumed to be independent. We will next examine two models which emphasize two different aspects of the software structure and design, namely flow models and object models. The former emphasizes the sequential and parallel execution constraints in the software, and the latter emphasizes software structural effects.

Flow or Task Graph Models

The classical flow model or "task graph" is essentially a flow chart, but including the feature of parallel branches; it closely resembles a critical path or pert chart in showing predecessor relationships among different activities which must be completed to execute the program. Figure 3 shows a version which we use, called "Activity Sequence Networks" or ASNs, but there are in fact many such models defined in the literature, including the UCLA graph model of behaviour, computation flow graphs, and Petri Nets, and some versions of data flow graphs (those with implicit control flow semantics). Some references may be found in [2].

Figure 3: An 'Activity Sequence Net' (the rectangles are activities that take time)

Flow models have the advantage of revealing potential parallelism, and critical paths. They give information which may be useful for reducing the completion time, in a system which is not saturated by competing responses – for example a system which executes one parallel program at a time on many parallel processors.

Perfomance limitations due to a program having a long critical path, with insufficient program parallelism to keep all the processors busy, are essentially the same as the path length bound mentioned above. Here the path length is the effective critical path, including scheduling constraints. Resource bottlenecks are not revealed by this model until the resource schedule is included in the critical path.

The calculation of performance from a flow model is carried out on the basis of certain asumptions about the scheduling and execution of the activities. The most primitive assumption is that each activity can be executed as soon as all its predecessors are completed, as in the basic critical path or pert model; this is equivalent to each activity having a private processor, and there being no conflicts for resources such as data access. On such a basis the critical path algorithm can be applied for deterministic processing times, or a Timed Petri Net resolution algorithm for deterministic or (exponentially distributed) stochastic times. Stochastic times may be a good representation of data-dependent execution times but care is needed; greater randomness tends to reduce the predicted throughputs. In general if either of mean delay or throughput can be found, the other can be derived from it. Distributions of delay however are difficult to obtain analytically; simulation may be necessary.

Models of Communicating Objects

Software is structured from modules which may be called procedures, tasks, modules, or processes depending on the language or operating system. It is natural to suppose that a model which represents these directly would offer more insight than one which represents the software differently. Such a model directly addresses the matter of software architecture which must be settled early in the design process.

In parallel systems those modules with a separate thread of control are important – in UNIX they are processes, in Ada or MVS, tasks. We will use the term task. The task is a system resource in that requests to it must queue; it in turn may use other resources, including using processors to execute itself, and data resources or semaphores or other capabilities for periods of time.

A simple point of view considers all of these resources to be similar objects, which may sometimes act as servers and sometimes as clients or customers requesting service. They have been termed "active servers" or "mobile servers". In their role as servers they introduce contention delays; in their role as clients they are in turn delayed by other servers. This is the point of view of the "rendezvous network" model for task-to-task interactions, for instance. Of course some objects are purely clients or originators of requests, while others may be purely servers– a CPU, for instance– just as in a queueing network. Figure 4 shows a model with client tasks and server tasks; an arrow represents a request/reply interaction directed from an object with the role of client, to an object with the role of server. Processors can also be represented.

The service time of a server task has two aspects, the time until it generates a response (call this "phase 1") and the remaining time until it is ready to accept its next request (phase 2). These (average) times will be denoted by x_1 and x_2 sec respectively; they include execution times by the server (denoted s_1 and s_2 sec.) and also the delay required for any requests made by the server to another server– call this average delay t_j sec. for server j, for each request. Then if phase 1 includes y_{j1} requests to server j the service time is:

$$x_1 = s_1 + \sum_j y_{j1} t_j$$

in which the sum is the average total time the server under study is blocked (not executing, but still technically "busy" and not able to accept a new request). It is this blocking time that creates software bottlenecks. More generally, when studying server task i in phase p, we may write:

$$x_{ip} = s_{ip} + \sum_j y_{ijp} t_{ijp}$$

An algorithm for finding the bottleneck rates has been given in [6].

Even if a certain resource is quite passive – a semaphore or hardware device, for instance – the model may represent it as an active object like a task, which inherits the behaviour

Figure 4: A Model with Communicating Software Objects (processors and locks can also be represented)

of its customers. In this way a critical section may be represented by a pseudo-task which, when acting for a certain client, executes the client's messages and processing.

The Software Bottleneck

A software bottleneck is a software task or other logical object whose service time limits the system throughput. It is possible to saturate a parallel system with no device being saturated, due to a software bottleneck. Essentially this happens because one task is idle waiting for a response from another, but cannot accept further requests because it has not completed the current one. Thus from the point of view of its clients it is busy; a queue builds up for this task, and the system is throttled back.

The system illustrated in Figure 4 may have a software bottleneck at the task $S1$. According to the parameters, each client task performs a cycle in which it executes for an average of one second, and then makes on average one request from $S1$. To serve a request, $S1$ executes 0.2 sec in phase 1 (with one request to each of $S2$, $S3$) and 0.1 sec in phase 2 (with one more request to each of $S2$, $S3$). $S2$ and $S3$ are similar, and each executes x seconds in phase 1 and the same in phase 2. Each task has a private processor which is not shown.

If x becomes large we find that $S1$ saturates even though no processor is fully utilized. Figure 5 shows the effect of x ranging from 0 to 1.0. At $x = 1$, $S1$ is fully saturated at 99.9%, while its processor is only busy 4.5% of the time; tasks $S2$, $S3$ (and their processors) are 61% utilized, and the clients' utilization is 5%.

Reflection reveals that the action required to alleviate a software bottleneck may require changes to other parts of the system. This is a significant difference from the action required to alleviate a hardware bottleneck. In this case, speeding up $S2$ or $S3$ will relieve the bottleneck at $S1$.

Relationship of the Three Models

The three models described in this paper are intimately related, and a unified modelling framework appears to be both useful and possible. For example, it is possible to model the communicating software objects by giving a state-machine for each one (indeed this is the approach of several real-time software development methods), and then to model performance by assigning processing times to transitions in these state machines. Closer examination shows that the resulting model contains the flow model embedded in it, and similar analysis methods can be applied, for instance timed Petri Nets. However practical difficulties arise at this point; the resulting model is typically very large and detailed, and such large Petri Nets are often impractical to analyse. Approximations are needed, or simulations.
The objects model is at a higher level, with less detail, and may be more useful for design analysis. It is constructed by adapting the classical queueing model, and can probably be unified with it, giving a unified hardware-software model with compact solutions.

Acknowledgements

The collaboration of John Neilson, Shikharesh Majumdar, and Dorina Petriu in our joint work on Rendezvous Nets contributed substantially to the ideas of this paper; conversations with George Yee, Yao Li, Elias Hagos, and Jerome Rolia also were of help. This research was supported by TRIO, the Telecommunications Research Institute of Ontario, a provincially funded research institution.

References

[1] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.G. Sevcik, *Quantitative System Performance*. Englewood Cliffs, N.J. 07632: Prentice-Hall, Inc., 1984.

[2] C.M. Woodside, "A partial bibliography of research in software performance modelling," technical report, Dept. of Systems & Computer Engineering and School of Computer Science, Carleton University. (to appear May '90.).

[3] C.H. Sauer and K.M. Chandy, *Computer Systems Performance Modeling*. Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc., 1981.

[4] S.S. Lavenberg, "A perspective on queueing models of computer performance," *Performance Evaluation*, vol. 10, pp. 53–76, 1989.

[5] D.L. Eager and K.C. Sevcik, "Performance bound hierarchies for queueing networks," *ACM Trans. on Computer Systems*, vol. 1, pp. 99–115, May 1983.

[6] C.M. Woodside, "Throughput calculation for basic stochastic rendezvous networks," *Performance Evaluation*, vol. 9, pp. 143–160, 1989.