On the Static Evaluation of Distributed Systems Performance

A. Mahjoub
Electrical Engineering Department, College of Engineering, King Saud University, Saudi Arabia

This paper examines the problem of response time computation from the system designer's point of view. It defines certain properties of system structures that lead to compile-time computation of worst-case response times. The steps of a fairly efficient algorithm are described.

1. INTRODUCTION

Real-time systems are often programmed as a collection of internal processes co-operating through a run-time support package. A signal emitted by a physical process controlled by a real-time system activates a corresponding internal process. A very challenging issue in the design of these systems is the ability to ensure that a particular system design meets the (stringent) timing requirements of the external environment it is intended to serve. The lack of providing adequate response could have the following effects on the system's behaviour:

(a) loss of accuracy in controlling the physical process because some required sampling rate could not be accommodated
(b) certain external events (or arriving data) may not be recognized on time by the computer and consequently will be lost
(c) system malfunctions resulting from slow response.

The classical approach to response time computation is based on queuing theory.[1,2] These schemes present probabilistic analyses which lead to approximations of expected response times. Such approximations, however, give little assistance in the design of systems intended to meet stringent timing requirements even in extreme situations.

Recent studies[3,4] have advocated some compiler-assisted techniques that determine if given timing constraints can be met by a system. The main idea in these studies is the partition of programs in strips scheduled for execution in a manner that suits the prespecified timing constraints. In the absence of adequate hardware support, this method incurs significant overhead due to the run-time scheduling of the code strips.

The objective of this paper is to define a systematic procedure for computing worst-case response times. It is expected that his procedure will be carried out entirely at compile-time without interfering with the code of the programs that make up a system.

The next section classifies current system organizations into two categories and explores the difficulties in response time computation with respect to each. Section 3 gives an informal outline of the response time computation, focusing mainly on delays. The basic definitions and simplifying assumptions are given in Section 4. A graph model of delays is described in Sections 5 and 6. Sections 7 and 8 present two cases where worst-case response times can be computed efficiently, and Section 9 concludes the paper.

2. SYSTEM STRUCTURES

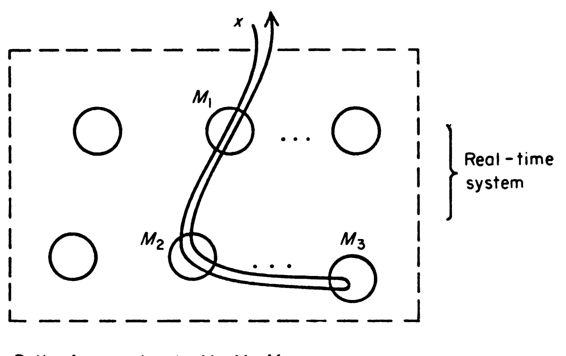
In a real-time system that supports resource sharing, response times are largely dependent on the way processes request service. We can identify at least two ways.

(1) In a monitor-based system, services (operations) that are requested by more than one process are programmed as monitor entry procedures. A process requesting a given service must first call the appropriate monitor procedure and wait until the monitor is free before starting execution. When execution of the procedure is terminated, the process exits the monitor and immediately resumes execution of its own code. The important characteristic of this system is that a service requested by a process is executed as if it were part of that process. Namely, the process resumes execution of its code immediately after completion of the service. We shall refer to systems having this characteristic as *function-oriented.* Clearly, monitor-based systems are just an example of these.
(2) In a *message-based* system, however, services are requested by the sending of a message. The requestor is generally required to wait until its message has been accepted by the receiver, but may continue afterwards. Thus, contrary to the previous case, a process requesting a service may execute in parallel with the service it requested. A process accepts a reply, which is transmitted to it via a message, when it is ready to do so. Thus, a reply transmitted while its recipient is not ready will be delayed. This delay must be accounted for in response time computations.

In a function-oriented system, the association of a service request with its corresponding reply is straightforward, as processes are required to wait for a reply after each service request. This association is however less obvious in message-based systems as a process may at any given time be waiting for many replies. Intuitively, response time gives the interval between the instant at which a service is requested and the instant at which its corresponding reply is received by the requestor. The lack of a clear request/reply association undoubtedly leads to difficulty in response time computation.

CCC-0010-4620/84/0027-0201 $04.00

A service request generated by an external stimulus will be referred to simply as a *request*. In a system, a request represents the thread of execution that takes place as part of the requested service. Thus, we can view the lifetime of a request as the tracing of a *path* consisting of all system modules that implement the various parts of the task generated by the request (Fig. 1). As a request



Path of request x is M_1, M_2, M_3

Figure 1

progresses through its path, it may encounter some delay when competing with other requests for shared resources. In the case of monitor based systems, a request may encounter a delay at each monitor in its path. In a chain of monitor procedure calls, delays are encountered during the nested calls, but no delays are experienced when returning from these calls. Thus, if the path of a request is a linear chain of monitors, then the lifetime of that request can be viewed as consisting of two phases: a first phase where the request progresses through its path, possibly encountering delay at some of the nodes, and a second phase during which it retraces (backwards) its path, but encountering no delays at any node. As a consequence, the worst case delay would be reduced to the sum of all delays encountered during the first phase. This is a significant simplifying factor that makes response time computation manageable, as will be shown in the remainder of this paper.

The same conclusion cannot be drawn with respect to message-based systems. A request received by a system of this type may encounter delays at any node in its path during either phase. This makes it very difficult to compute upper bounds on response times. We will not elaborate on this issue in this paper; for details, the reader is referred to Ref. 5. The remainder of this paper will only deal with function-oriented systems in which requests result only in linear chains of function calls.

3. RESPONSE TIME COMPUTATION

Given a request x received by a system S, we wish to compute its response time. This can be expressed as the sum of:

(a) The total delay encountered by x at all modules in its path in S.
(b) The execution time of all functions invoked during the lifetime of x.

We shall assume that each *module* in S is a *sequential process* executing on a *dedicated processor* and containing an entry point for each service it implements. Consequently, services are executed without interruption and thus take fixed times. Since these times depend only on the speed of the processor, we shall assume them to be known. This process model is inspired by Ref. 6.

The delays are of two kinds:

(a) If two or more requests require services that are implemented by the same module M, then these requests could encounter some delay if they simultaneously compete for these services. The reason for this is that M, being a sequential process, can execute only one service at a time.
(b) Once a service is being performed, its completion may be dependent on the progress of other requests in the system. Consider for instance a request x that calls for a service to acquire exclusive access to a shared peripheral. This action could be delayed if the desired peripheral is not available when requested; and can be performed only when the peripheral is voluntarily released. This delay is eventually reflected in the response time of x.

In a system consisting of processes and monitors, the first type of delay is experienced by processes attempting to enter an already occupied monitor, whereas the second type corresponds to delays occurring inside a monitor.

Delays of type (b) are difficult to compute because they are non-deterministic. Given a process that has acquired a given resource, it is generally undecidable if and when that process will release the resource. In this paper, we shall assume that we are dealing with systems that are deadlock free, and therefore we can assume that each service on which the completion of another service is dependent will, in a finite time bounded by a known constant, allow it to complete. This assumption encompasses all delays that normally occur inside a monitor.

Delays of type (a) are those that are normally associated with mutual exclusion. These delays could also be non-deterministic. Consider for instance a system consisting of a collection of distributed processes.[7] These processes implement procedural operations which can be invoked by other processes. Acceptance of multiple calls to a given procedure is however non-deterministic, which implies that one cannot compute nor bound the time a process waits upon invoking a procedure of another process. Response time computation is more manageable if this non-determinism is eliminated and replaced with a fixed order, e.g. a process implementing n services (procedures), would look for calls for them in a cyclic manner. The important implication of this is that, given a module M implementing n services $s_1, s_2, \ldots, s_n$ and a request r waiting for service s_j, the maximum delay that would be experienced by r is

$$(m+1)\sum_{i=1}^{n} T(s_i) - T(s_j)$$

where m denotes the number of requests waiting for s_j at the time of call, and $T(s_i)$ denotes the time taken by s_i to complete. In the worst case, s_j would be listed last in the cycle and all m requests waiting for s_j would go before r. These will cause r to wait for the completion of m cycles. Hence the above formula. This formula can be simplified by the assumption that each service is part of at most one request (i.e. $m \leq 1$). This assumption greatly simplifies the model. Further generalizations can be made to deal with multiple requests that require the same service by making this service available to them under different names. In the remainder of this paper, we shall deal *only* with systems for which this assumption holds.

4. BASIC DEFINITIONS

Essential to the study of response time presented in this paper is the notion of experiment.

Experiments

In order to analyse the time behaviour of systems, we shall deal with several requests simultaneously active. The response time of a given request is clearly dependent on which requests are simultaneously active with it. This leads to the notion of *experiment*. Formally an experiment represents the activity in the system resulting from (and only from) the receipt of a set of requests. We shall use the notation

$$(\langle r_1, t_1\rangle, \langle r_2, t_2\rangle, \ldots, \langle r_n, t_n\rangle)$$

to represent the activity resulting from the arrival of request r_1 at time t_1, r_2 at t_2, etc. We shall assume that before the start of an experiment, the system is in an idle state, and no requests other than those listed in an experiment are submitted until the system regains its idle state. Thus any delays that occur during an experiment are only due to the interactions of the requests submitted in that experiment. Clearly the set of all possible experiments in a given system is infinite.

Response time

Let RT(x, e) denote the response time of x in experiment e. This is the time taken by x to complete execution when x and all other requests in e are submitted at the times defined by e and no other request is submitted. The worst case response time of x, denoted WRT(x), is the maximum RT(x, e) over all possible experiments e in a system.

5. GRAPH MODELLING OF DELAYS

An important tool in the analysis of response time presented in this paper is the *delay graph*. It is a directed labelled graph depicting the interactions between requests during an experiment. The delay graph of an experiment

$$e = (\langle r_1, t_1\rangle, \ldots, \langle r_n, t_n\rangle)$$

has n nodes labelled $r_1, r_2, \ldots, r_n$, respectively. It has a directed edge $r_i \rightarrow r_j$ labelled d for each pair r_i, r_j such that r_j caused d units of delay to r_i. A request r_i is *delayed* by r_j at a module M if r_i calls for the execution of a service implemented by M, but, at the time of the call, M is performing a service s_j invoked by r_j. The *duration* of this delay is the time taken by M to complete the execution of s_j in the absence of any other request in the system. Thus any delays that r_j encounters after r_i starts waiting for it are not included in d. The sequential behaviour of modules precludes the possibility of a given request being simultaneously delayed by multiple requests. This, in conjunction with the ordering property (next section) precludes the possibility of a request causing delay to multiple requests simultaneously.

Figure 2(a) gives a system S which accepts four requests x, r_1, r_2, r_3. The path of each request in the system is labelled with the name of that request. Figure 2(b) gives a graphical illustration of experiment.

$$e = (\langle x, 20\rangle, \langle r_1, 0\rangle, \langle r_2, 0\rangle)$$

Figure 2(c) gives the delay graph of e.

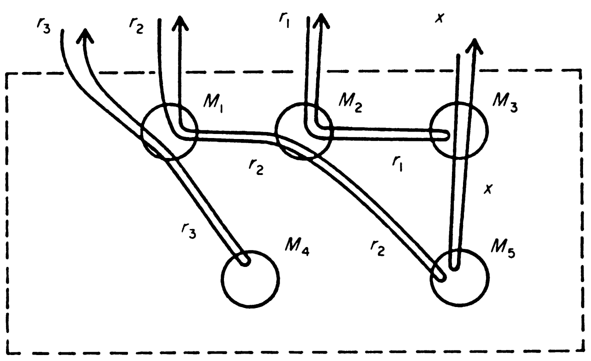


Figure 2a

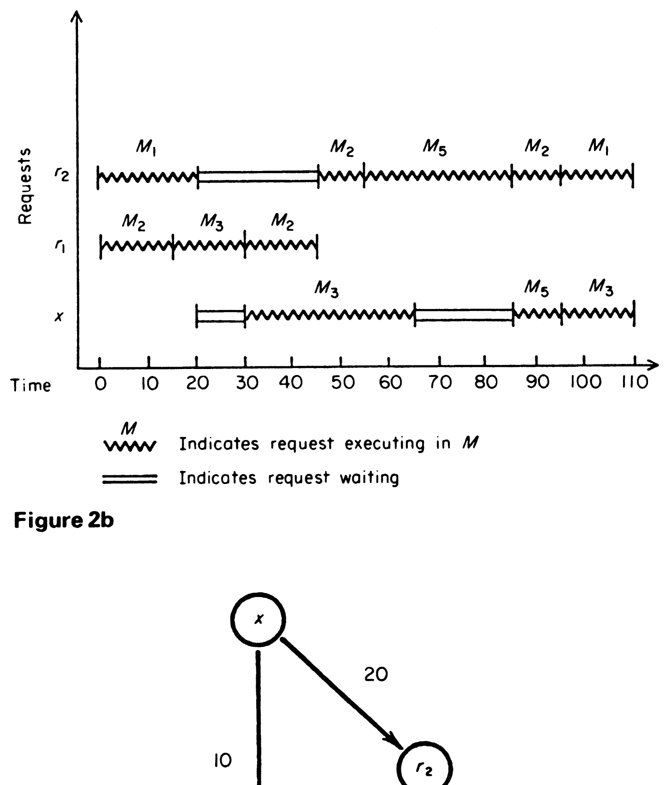


Figure 2b

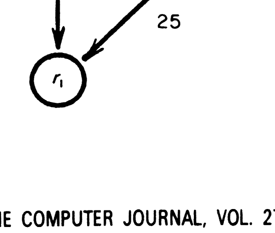


Figure 2c

Each edge in a delay graph corresponds to some delay. Further, if in a delay graph two requests are not connected then they do not delay each other. A delay graph cannot contain loops, for this would imply the existence of deadlocks. An isolated node in a delay graph of an experiment represents a request that had no interaction with other requests during that experiment. Clearly, such requests can be ignored altogether in response time computations. We shall now outline how delay graphs can be used in the computation of response time. For this purpose, we introduce the following notation:

$x <^{d}_{e,\delta} y \equiv$ In experiment e, request x is delayed by request y for a period of d units and the waiting commences at time δ.

If the context does not require specific mentioning of d or δ, then either (or both) of them may be omitted. If d is replaced by *, it implies that y causes maximum delay to x.

$x \ll^{d}_{e} y \equiv$ There exists a chain

$$x <_e z_1 <_e z_2 <_e \ldots <_e z_n <^{d}_{e} y, (n \geq 0)$$

Ordering property

An important factor in real-time analysis is the order in which events occur.

Consider an experiment e in which three requests x, y and z are submitted. Let

$$x <^{d_1}_{e,\delta_1} y \quad \text{and} \quad y <^{d_2}_{e,\delta_2} z$$

The computation of RT(x, e) is dependent on the relation between δ_1 and δ_2.

(a) $\delta_2 \leq \delta_1$
In this case, the delay caused by z to y is d_2. However, only a portion of this delay is subsequently reflected in the response time of x. This portion is $\alpha = \max(0, d_2 - (\delta_1 - \delta_2))$. Let $\beta = d_2 - \alpha$. β represents a delay which is irrelevant to x and thus must not be included in RT(x, e). That is, the delay caused to x by y and z is $d_1 + d_2 - \beta$. Since $0 \leq \alpha < d_2$, it follows that $\beta > 0$. Hence, this case shows that the delay encountered by x due to the execution of z is not optimal. This suggests the possibility that there may exist other experiments in which x would encounter more delay due to the execution of z. This is explored in case (b). In the experiment of Fig. 2(b), we have,

$$x <^{20}_{e,65} r_1 \qquad r_1 <^{25}_{e,20} r_2$$

Thus, $\alpha = 0$ and $d_2 = 25$ is totally irrelevant.

(b) $\delta_1 \leq \delta_2$
In this case z starts causing delay to y after y starts causing delay x. Thus d_2 takes place while x is waiting and consequently it should be included in RT(x, e). Note that d_2 cannot take place after y has terminated delaying x because y is then in its second phase and cannot be delayed. Hence, in this case the delay encountered by x due to the execution of z is *strictly greater* than that of case (a). This suggests that the delay encountered by x in some experiment e can be increased (if not already optimal) by proper rearrangement of the ordering of the delay events in e. This is formally shown in Ref. 5.

The important property established in Ref. 5 states that for determining the worst case response time for x, it is sufficient to consider experiments in which every request, other than x, is not delayed until it itself causes delay (i.e. case (a) above never occurs). We shall refer to this property of an experiment as the *ordering property*.

Since delays caused by x are irrelevant to x (for otherwise it would imply deadlock) they can be ignored without loss of generality. In the remainder of this paper we will treat x as the main request, and consider only experiments in which x is submitted and x does not cause any delay.

The ordering property has two implications. Let e be an experiment satisfying the ordering property, then

(a) The delay graph of e is rooted at x, connected, and contains a path from x to every other node.
(b) The total delay encountered by x during e is the sum of all labels on the arcs of the delay graph of e.

6. ENUMERATION OF DELAY GRAPHS

The *topology* of a delay graph, i.e. a delay graph without its arc labels, illustrates the delay relationship between requests during an experiment. There may exist several experiments whose delay graphs have the same topology, but possibly different labels. Because a system is capable of supplying only a finite number of distinct services, there can exist only a finite number of different topologies. Consider all experiments whose delay graphs have a given topology. We wish to construct a procedure that determines which of these experiments results in the largest response time for x. This procedure can then be combined with another procedure for enumerating all possible delay graph topologies to produce an algorithm for computing the worst case response time for x. This algorithm is dependent on static information only, and thus can be carried out *entirely* at compile time.

In general, there is no simple way of enumerating all delay graph topologies for a given system. It is shown in Ref. 5 that this can be reduced to finding a feasible solution to a linear programming problem, and that its optimal solution gives the worst case delay. Unfortunately this approach is too inefficient for practical purposes. There are however some classes of systems for which worst case response times can be derived quite efficiently and without a need for complete enumeration of all delay graph topologies. This will be proved in the following sections.

Tightness

Consider an experiment e in which a request y causes delay to n requests $z_1, z_2, \ldots z_n$ ($n > 1$) which in turn cause delay to x (see Fig. 3). That is, for all $i = 1$ to n,

$$x <_e z_i \quad \text{and} \quad z_i <_e y$$

Let z_1 be the first request that starts waiting for y, z_2 the second, z_3 the third, etc. As noted in Section 4, the intervals of time during which y delays these requests are disjoint. This implies that z_2 starts waiting for y after z_1 has finished waiting for it, z_3 starts waiting for y after z_2

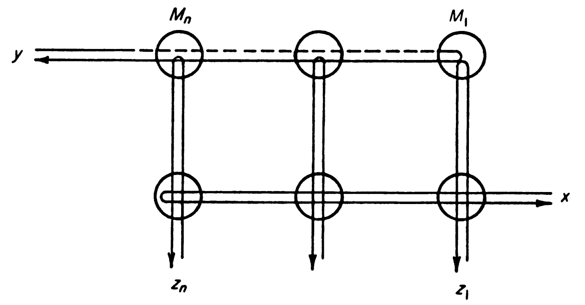


Figure 3

has finished waiting for it, etc. This has two important implications.

(a) y causes delay to $z_2, z_3, \ldots, z_n$ while it is in its second phase (retracing its path).
(b) Let M_i be the module (in the path of y) at which z_i is delayed by y. Then M_n is the first among these in path of y (Fig. 3).

(a) and (b) imply that in the experiment in question the following relation holds.

$$\text{For all } i = 2 \text{ to } n, z_i <_e y \quad \text{but not} \quad z_i <_e^* y$$

The broken lines (Fig. 3) inside the modules $M_n, \ldots, M_2$ indicate execution time intervals not added to the delay of x in e. The above relations in turn imply that

$$x \ll_e y \quad \text{but not} \quad x \ll_e^* y$$

That is, y has contributed some delay to x, but this delay is not maximum. Thus, there may exist another experiment e_1 such that

$$x \ll_{e_1}^* y$$

For instance, experiment e_1 may have

$$z_n <_{e_1}^* y$$

This delay is clearly larger than the sum of the delays caused by y to $z_1, z_2, \ldots, z_n$ in experiment e.

If for a given system we can show the existence of an experiment e_0 in which all requests that can potentially cause delay to x are submitted and for each pair of requests y, z in this experiment

$$y <_{e_0} z \Rightarrow y <_{e_0}^* z$$

then clearly $\mathrm{RT}(x, e_0) = \mathrm{WRT}(x)$. For the systems under consideration, it is possible to determine the existence of e_0.

Systems for which e_0 exists are hereafter called *free systems*. The next section gives a characterization of these systems and shows how e_0 can be constructed for each system.

7. FREE SYSTEMS

The algorithm for determining the existence of e_0, for a given system, comprises three main steps:

Step 1. Construct a directed graph **G**, the union of all delay graph topologies for the given system. Formally **G** is defined as a directed graph containing a node x, and a node y for each request y that contributes to the delay of x in some experiment e, i.e. $x \ll_e y$. For each pair of nodes y, z in **G**, there is an arc $y \rightarrow z$ in **G** if and only if there exists an experiment e such that $y <_e z$ if $y = x$, or $x \ll_e y <_e z$ if $y \neq x$.

Intuitively, the existence of node y in **G** justifies the existence of an experiment in which y contributes to the delay of x. Hence, there must exist a delay graph in which there is a path from x to y. Thus **G** is a directed graph in which there is a path from x to every other node.

Step 2. Using the graph constructed in Step 1, construct the topology G_0 of the delay graph of e_0. It can be shown[5] that G_0 is a particular spanning tree of **G**.

Step 3. Compute the times of arrival of the requests in e_0 and check that when e_0 actually takes place its delay graph topology will be that constructed in Step 2, and each request will cause maximum delay.

Step 1. Construction of G

Algorithm

(a) Initially **G** contains only the node x.
(b) For each y that intersects the path of x add node y and arc $x \rightarrow y$ to **G**.
(c) Repeat this step until no further changes can be made to **G**.

If possible, select a path $x = y_0, y_1, y_2, \ldots, y_k$ in **G** and a node y_{k+1} (which may or may not be in **G**) such that the following conditions hold:

(1) There exist distinct modules M_i at which y_i intersects y_{i-1} for all $i = 1, k+1$.
(2) The path of y_i up to M_i does not intersect the path of any other y_j up to M_j for all $i, j = 1, k+1$, $i \neq j$.
(3) M_{i-1} appears before M_i in the path of y_i, for all $i = 1, k+1$.

Then add node y_{k-1} and arc $y_k \rightarrow y_{k+1}$ to **G** if not already present.

Construction of G. **G** can be constructed from the system as described in the above algorithm. The algorithm builds **G** incrementally by adding arcs to the tail end of existing paths from x. It considers each path and examines the feasibility of an experiment in which the delay pattern corresponding to the path may occur. Condition (1) ensures that each request in the path intersects its predecessor (to cause delay). Condition (2) avoids the possibility of two requests blocking each other, thereby precluding the delay pattern indicated by the path. Condition (3) ensures that the ordering property can be satisfied. Satisfaction of these conditions guarantees the existence of an experiment whose delay graph topology is the path $x, y_1, \ldots y_k, y_{k+1}$. A systematic way of examining all the paths satisfying the conditions is given in Ref. 5.

Step 2. Generation of G_0

As noted earlier, G_0 is the topology of the delay graph of the experiment e_0 in which each request causes maximum delay. G_0 is a spanning tree of **G** such that each request $z \neq x$ in **G** has only one predecessor y, and z can cause greater delay to y than to any of its other predecessors in **G**. Thus in order to construct G_0, we need to identify, for each z, which of its predecessors should be included in G_0. In order to achieve maximum delay, we must select the predecessor that intersects the path of z earliest. If this is true of more than one, then the selection is arbitrary. Clearly for a given system, G_0 may not be unique. The graph **G** corresponding to the system of Fig. 4(a) is given in Fig. 4(b). Figures 4(c) and 4(d) show two possible constructions of G_0.

Note that each request in e_0, other than x, delays exactly one other request. This is consistent with the point made in Section 5, that the maximum delay contributed by a request is not obtained by having that request delay a number of other requests, but by allowing it to delay only one particular request.

Although G_0 is constructed primarily for the computation of WRT(x), we shall first use it to derive an upper bound on WRT(x). This upper bound may not be tight, i.e. it may not correspond to the response time obtained in an actual experiment, but can be used as an indication of the worst case response time.

An upper bound on WRT(x). For each request y in G_0, let $d(y)$ be the time taken by y to execute (without delay) in the portion of its path from the first module that it shares with its predecessor in G_0 to the end. This is the maximum delay that y can cause in the experiment e_0. Let $d(x)$ be the time taken by x to execute (excluding any delays).

Lemma 1. Let $\text{ub}(x) = \sum_{\forall z \in G_0} d(z)$, then $\text{WRT}(x) \leq \text{ub}(x)$.

Proof. Immediate.

Step 3. Definition of e_0

We wish to define an experiment e_0 such that $\text{RT}(x, e_0) = \text{WRT}(x)$. As argued before, e_0 should be such that: for all y, z in G_0

$$z <_{e_0} y \Leftrightarrow z \rightarrow y \text{ is an arc in } G_0 \qquad (1)$$

$$z <_{e_0} y \Rightarrow z <^{*}_{e_0} y \qquad (2)$$

Consider the experiment e_0 and define the following terminology with respect to e_0. Let,

$a(y)$ = arrival time of request y
$c(y, M)$ = constant execution time y takes to reach module M, without any intervening delays
$t(y, M)$ = time at which y reaches module M in e_0
$w(y)$ = total delay caused by y to its predecessor in G_0 (including delays encountered by y).

By definition, we have for all y in G_0

$$w(y) = d(y) + \sum_{\substack{\text{for all successors} \\ z \text{ of } y \text{ in } G_0}} w(z)$$

All $w(y)$ can be computed from the system definition. Let $a(x) = 0$. We now outline how $a(y)$ can be computed for all other y in G_0. The successors of each y in G_0 can be ordered according to the order in which they intersect y. If two requests intersect at the same module their order is determined by the fixed cyclic scan at the module. The tree G_0 can then be traversed in preorder and $a(y)$ can be computed for each y in the order defined by this traversal.

Consider an arc $y_1 \rightarrow y_2$ in G_0. Let M be the module at which y_2 causes delay to y_1. Equation (2) implies that y_1 and y_2 reach M at the same time and y_2 executes in M first. Then we have

$$t(y_1, M) = t(y_2, M) \qquad (3)$$

Since the ordering property ensures that y_2 cannot be delayed before M, we have

$$t(y_2, M) = a(y_2) + c(y_2, M) \qquad (4)$$

Equations (3) and (4) give

$$a(y_2) = t(y_1, M) - c(y_2, M) \qquad (5)$$

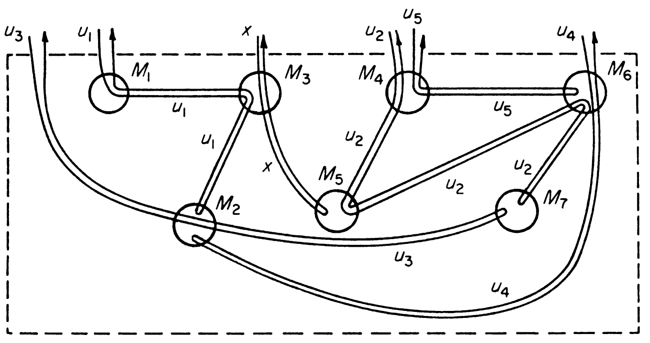


Figure 4a

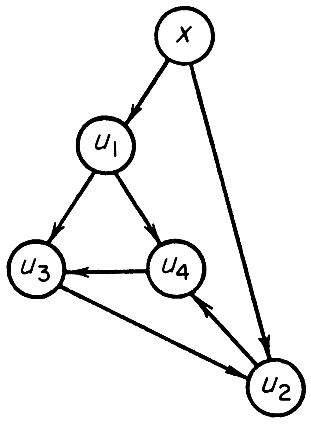


Figure 4b

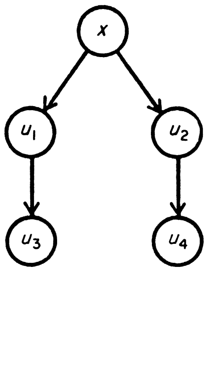


Figure 4c

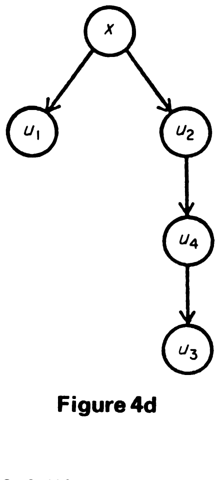


Figure 4d

$t(y_1, M)$ can be computed as follows:

$$t(y_1, M) = a(y_1) + c(y_1, M) + \sum_{\substack{\text{for all } z \text{ in} \\ v = \text{successors of } y \\ \text{in } G_0 \text{ which intersect} \\ y \text{ before } M}} w(z) \tag{6}$$

and hence $a(y_2)$ can be expressed in terms of $a(y_1)$ as follows:

$$a(y_2) = a(y_1) + k \tag{7}$$

where k is the constant:

$$k = c(y_1, M) - c(y_2, M) + \sum_{\text{for all } z \text{ in } v} w(z)$$

A tight upper bound on WRT(x)

Definition. For each request y, different from x, in G_0 the *initial segment* of y refers to the portion of its path from the beginning to the first module it shares with its predecessor in G_0. The remainder of the path is referred to as the *non-initial segment*. The initial segment is empty if the request in question shares with its predecessor the first module in its path. Each request in G_0 which is different from x has a unique initial segment.

Theorem 1

If the initial segments of all requests in G_0, other than x, are mutually disjoint then*

(1) for all y, z in G_0 such that $z \to y$ is an arc in G,

$$z <^{d(y)}_{e_0} y$$

(2) $\mathrm{RT}(x, e_0) = \mathrm{WRT}(x)$

Proof. (2) Assuming that (1) holds, then by lemma 1, $\mathrm{RT}(x, e_0) = \mathrm{ub}(x)$, and thus $\mathrm{RT}(x, e_0) = \mathrm{WRT}(x)$.

(1) By assumption, the initial segments of requests in G_0 do not have any modules in common. We show that this implies that the initial segment of each request contains modules that are completely 'private' to it and thus each request can be conveniently submitted so that it causes the desired delay.

Assume, if possible, that there exists a request y_1 in G_0, whose initial segment contains a module M that is also in the path of another request y_2 in G_0. Since the initial segments are mutually disjoint, M must be in the non-initial segments of y_2. But this implies the existence of an experiment e' in which

$$y_2 <^{d'}_{e'} y_1$$

Let z be the predecessor of y_1 in G_0. Since z does not intersect the initial segment of y_1 it must be distinct from y_2. Further $d' > d(y_1)$ as y_1 executes a larger portion of its path while y_2 is waiting for it. Thus z cannot be the predecessor of y_1 in G_0. Hence a contradiction (Q.E.D.).

8. INTERFERENCE FREE PROPERTY

The thrust of the condition of Theorem 1 is to ensure that each request can progress in its path without delay until it reaches the module at which it is supposed to delay its predecessor in G_0. Although mutual disjunction of initial segments guarantees this situation, it is too strong for this purpose. It is quite possible, for instance, that two requests that have a common module in their respective initial segments execute in that module in such a way that they do not interfere with each other. Clearly, if we can guarantee this situation, then theorem 1 would still be applicable.

The modules of interest in this case are those that are common to several initial segments. Let y_1 and y_2 be two requests in G_0 whose initial segments contain a common module M. In order that in the experiment e_0, y_1 and y_2 do not interfere with each other at M, it must be the case that one exits M before the other attempts to enter it. Let y_1 be the first to execute in M. (There is a systematic way of determining the order of execution in shared modules.[5]) We wish to define a relation that ensures that y_2 does not reach M until y_1 has left it. For this purpose we define the following terms. Let

arrivaltime (y, Q) =	the time at which y invokes a service implemented by a module Q in its path during experiment e_0. This is the same as $t(y, Q)$ defined on p. 206.
exittime (y, Q) =	time at which the execution of the service invoked by y at Q is completed.

The following condition ensures that no interference between y_1 and y_2 will occur at M:

$$\text{exittime}\,(y_1, M) \le \text{arrivaltime}\,(y_2, M)$$

We shall refer to this as the *interference free relation*. Since it depends only on y_1, y_2 and M, we shall denote it as $\mathrm{IFR}(y_1, y_2, M)$.

IFR can be verified statically knowing only the system structure and the constants giving the sequential execution times. The actual computation of exittime and arrivaltime can be derived from Eqns (6) in Section 7. Below we give a quick outline of how these terms can be computed.

arrivaltime (y_2, M) is computable from $a(y_2)$, the time at which y_2 is submitted in e_0. Since M is in the initial segment of y_2, the execution of y_2 in the portion of its path from the beginning to M is a constant.

exittime (y_1, M) depends on $a(y_1)$ and on the delays encountered by y_1 at the modules in its non-initial segment. These delays correspond to all the arcs in G_0 that form the subtree rooted at y_1. Thus the total delay that y_1 should encounter in e_0 is the sum of $w(z)$ over all successors z of y_1 in G_0.

Let c be the (constant) time taken by y_1 to complete execution (without any delay) in the portion of its path from M to the end. The reader can verify that

$$\text{exittime}\,(y_1, M) = \text{arrivaltime}\,(y_1, M) + c + \sum_{\substack{\text{for all } z \text{ successor} \\ \text{of } y_1 \text{ in } G_0}} w(z)$$

Theorem 2

$\mathrm{RT}(x, e_0) = \mathrm{WRT}(x)$ if and only if for all pairs of requests y_1, y_2 in G_0 whose initial segments contain a common module M, $\mathrm{IFR}(y_1, y_2, M)$ holds.

* $d(y)$ is defined on p. 206.

Proof. This theorem is a natural conclusion of the above discussion and its proof reduces to the proof of Theorem 1. The fact that some initial segments share some modules has been made irrelevant as far as e_0 is concerned by the assumption that the interference free property is satisfied, (Q.E.D.).

9. CONCLUSIONS

For systems which do not fall in either of the categories described in Section 2, the procedure for determining WRT(x) is complex and lengthy to describe. In Ref. 5, we have characterized two other classes of systems for which WRT(x) can be found without using linear programming techniques (which are the most inefficient in this case). The characterization was based on properties of the graph **G** which is the union of all delay graphs. For all practical purposes, however, an upper bound should be sufficient.

The problem of response-time prediction is very difficult. The classical approach to it has been based on queueing theory. These approaches, however, lead only to the expected response time and give little indication of the worst-case response time. This paper has alleviated the difficulties by considering a simple process model in which processes have a fixed behaviour. Additional restrictions have been imposed on the system structure to simplify the analysis. Clearly much remains to be investigated in this area, but we hope that this paper has provided a good first step in compile-time approaches to response-time computations.

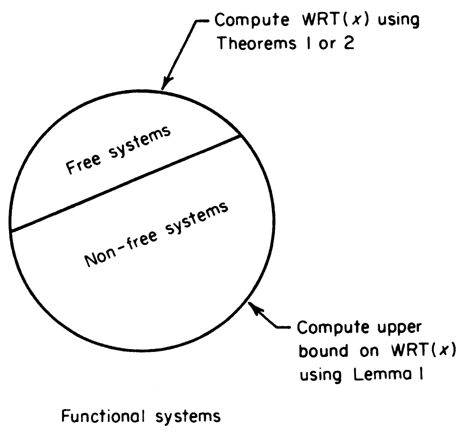

Compute WRT(x) using Theorems I or 2
Free systems
Non-free systems
Compute upper bound on WRT(x) using Lemma I
Functional systems

Figure 5

The procedure for computing worst-case response times given in this paper is shown in Fig. 5, and can be summarized as follows:

(1) Using algorithm given for Step 1, construct **G**.
(2) Using algorithm given for Step 2, construct G_0.
(3) Check if initial segments of all requests in G_0 are mutually disjoint. If so, apply Theorem 1.
(4) Otherwise, check if IFR(y_1, y_2, M) holds for each pair of requests y_1, y_2 in G_0 whose initial segments intersect at M. If so, apply Theorem 2 and derive WRT(x).
(5) Otherwise, use Lemma 1 to derive an upper bound.

REFERENCES

1. G. M. Schneider, A modeling package for simulation of computer networks. *Simulation* **31**(6), (1978).
2. L. Kleinrock, *Queuing Systems, Volume II: Computer Applications,* Wiley Interscience, N.Y. (1976).
3. J. L. Hennessy, A real-time language for small processors: design, definition, and implementation. *Ph.D. Thesis,* Dept. of Computer Science, SUNY at Stony Brook, N.Y., August (1977).
4. S. A. Ward, An approach to real-time computation. *Proc. of the Seventh Texas Conf. on Computing Systems,* Texas, November (1978).
5. A. Mahjoub, Analysis of response time in real-time systems. *Ph.D. Thesis,* Dept. of Computer Science, SUNY at Stony Brook, N.Y., May (1979).
6. C. A. R. Hoare, Communicating sequential process. *CACM* **21**(8), 666–677 (1978).
7. P. Brinch Hansen, Distributed processes: a concurrent programming concept. *CACM* **21**(11), 934–941 (1978).

Received February 1983