Finding Response Times in a Real-Time System

M. JOSEPH AND P. PANDYA
Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India

There are two major performance issues in a real-time system where a processor has a set of devices connected to it at different priority levels. The first is to prove whether, for a given assignment of devices to priority levels, the system can handle its peak processing load without losing any inputs from the devices. The second is to determine the response time for each device. There may be several ways of assigning the devices to priority levels so that the peak processing load is met, but only some (or perhaps none) of these ways will also meet the response-time requirements for the devices. In this paper, we define a condition that must be met to handle the peak processing load and describe how exact worst-case response times can then be found. When the condition cannot be met, we show how the addition of buffers for inputs can be useful. Finally, we discuss the use of multiple processors in systems for real-time applications.

Received March 1985

1. INTRODUCTION

A typical real-time system has a number of devices connected to a computer system: a real-time program running on the system reads inputs from devices, processes these inputs, and then (often) produces outputs to be sent to the devices. In the simplest cases, the devices produce inputs at regular intervals and expect outputs to be sent to them also at regular intervals. Inputs may arrive either through asynchronous interrupts, or by software polling at specified intervals. The time between the arrival of an input from a device and the completion of processing for that input is usually called the response time for the device; it may be assumed that outputs are produced only when processing is complete.

Not all devices operate in quite this fashion; for example, some devices may have minimum and maximum times between their inputs, and others may produce inputs at random times. Also, different devices will have different response time requirements. A safer characterisation of such devices is therefore to say that they each have a minimum time between successive inputs and a maximum allowable response time. To ensure that these response-time requirements can be met, the devices are usually connected to the system at the different priority levels at which their inputs will be processed.

The computer system can be characterised by its speed, i.e. the time it takes to process an input. In fact, this time is usually variable, but within bounds. A safe assumption is to take the upper bound of the processing time for each kind of input. Processing of an input will be pre-empted when another input of higher priority arrives and will only be resumed when there is no processing remaining at higher priorities.

Consider such a real-time system, in which a number of devices are connected to a single processor computer system at different priority levels. Initially, assume that an input from a device is saved in a buffer until it is overwritten by the next input from the same device.

The first problem is to determine whether for a given assignment of devices to priority levels the system will meet its peak processing load (i.e. no input from any device will be lost). A more basic problem than this is to give a method of finding an assignment of devices to priorities for which the system will meet its peak processing load. Some earlier work[4] gives conditions under which such an assignment is possible, and a method[3] of determining the system load for such an assignment. Under these conditions, there may be more than one assignment of devices to priorities that is feasible. For a priority assignment to be feasible, it is necessary that, on the average, the processing of each input be completed in a time shorter than the time between two successive inputs from the same device. But it is possible that only some subset of these feasible priority assignments will also satisfy the response-time requirements, or that no such assignment can be found.

An available method[3] can be used for finding good upper bounds to the response times, but this does not provide the least upper bound. A recent investigation[7] gives a more rigorous formal analysis of the problem, showing when it is possible for a feasible priority assignment to be found, but it does not provide a computationally efficient way of finding the exact worst-case value of the response time. It should be noted that typical statistical analyses and queuing models are not suitable for this problem because the values they can predict are bounded by probabilities: while such studies can be used in the absence of better methods, they cannot guarantee correct working of a system under worst-case conditions. And in general real-time systems must be designed to tolerate worst-case conditions, even if the probability of the occurrence of such conditions is low. In safety-critical systems, for example, there are usually severe response-time constraints, and Currie[1] has suggested that it will probably be essential for such systems to be programmed in languages with restrictions which make it possible to ensure that timing bounds are kept. Our problem is to determine whether such bounds are kept and to find the absolute values of the response times.

In this paper we shall describe how exact values can be found for the worst-case response times of a real-time system. The method, which is proved to be correct, can easily be extended to handle cases where inputs are buffered, and to find the number of buffers that will allow a given assignment of devices to priority levels to be feasible. We discuss the use of multiple processors for a real-time system and consider situations in which this can provide a solution for computationally bound systems.

2. DEFINITIONS

Let the system have n devices connected, one at each of n different priority levels. Following conventional practice, let 1 be the highest and n the lowest priority level. Let T_i be the minimum time between successive inputs from the device connected at priority level i, and let C_i be the maximum associated processing requirement.

Values of T_i can usually be specified without difficulty by the system engineers, as they are either characteristics of the devices or requirements of the real-time system. Determining C_i requires a little more analysis, but is in practice not difficult. Assume the real-time program is composed of n processes, one servicing each device. For simplicity, let these processes be independent and have no communication with each other. Then, provided the program code is available, a compiler can make a static analysis and provide an accurate estimate of the worst-case execution time of each statement in the program. However, this will not suffice to calculate the worst-case execution time for each process, as it is necessary in addition to know the number of times each statement will be executed. For this to be possible, the programmer must specify an upper bound to the number of iterations of each loop. Given a proof of total correctness for the program, such an upper bound must exist, and it is not a great additional imposition for the programmer to give it a numeric value. If a system must meet critical real-time constraints, it must be programmed with this in mind, and putting upper bounds to loop iterations is one of the first steps in this direction.

Let RL_i be the required upper bound on the response time of device i, and let RT_i be the worst-case upper bound provided by the real-time system. From the definition of RL_i, RT_i must be less than RL_i under all conditions. And a common requirement is for RT_i to be less than T_i. If this condition cannot be met, buffers can be added to store successive inputs coming from devices, or additional processors can be added to the system to reduce the total processing time.

3. WORST-CASE LOAD

The highest sustained processing load arises when all the inputs arrive periodically with the minimum interval T_i between them and each such input requires the maximum processing time C_i.

Following Liu and Layland,[4] we state the following definition and theorem.

Definition. The critical instant for level i is defined to be the instant when an input at this level will have the largest response time.

Theorem 1. A critical instant for level i occurs whenever an input at this level occurs simultaneously with the inputs of all higher-priority tasks.

Proof. As in Liu and Layland.[4] □

It then follows that the critical instant for the system as a whole is the instant when inputs from all the devices arrive simultaneously. We shall refer to the load at this instant as the worst-case load. If the worst-case load occurs at time $t = 0$, it will occur again at multiples of time $M_i = LCM(\{T_j \mid 1 \leqslant j \leqslant i\})$, where the right-hand side is the least common multiple of all values from T_1 to T_i.

If for all i the values of T_i are equal, then the largest response time RT_i is simply the sum $\sum_j C_j$, $1 \leqslant j \leqslant i$. But in general this is not the case and there may be no relation between T_j and T_k for $j \neq k$. If $j < k$ and $T_j < T_k$, there will be more than one input from device j in the time interval $[0, T_k)$.*

4. AN EXACT ANALYSIS

The fact that inputs from higher levels pre-empt processing of those from lower levels leads to the conclusion that for all $i > j$, $RT_i > RT_j$. But to find the actual value of RT_i we need to know exactly how many inputs come at the higher levels in the interval $[0, RT_i)$. For each such input at some level j, there will be a computational requirement of C_j which must be completed before the computational requirement of C_i can be completed. Note that it is not sufficient just to find the number of inputs from each higher level in the whole of the interval $[0, T_i)$ because some of these may come in the part $[RT_i, T_i)$ of the interval and will thus not contribute to the value of RT_i.

The response time RT_i is thus the sum of the computational requirements for all inputs from higher levels occurring in the interval $[0, RT_i)$, and C_i, the computational requirement for one input at level i,

$$RT_i = \left(\sum_{j=1}^{i-1} \lceil RT_i/T_j \rceil * C_j\right) + C_i. \qquad (1)$$

Unfortunately, equations of this form do not lend themselves easily to analytical solution.

There is another means of reasoning that could lead to a more tractable solution. Computation at level i cannot begin until the computations for inputs at all the higher levels have been completed. So a worst-case lower bound for RT_i will be ΣC_j, for $1 \leqslant j \leqslant i$. After it begins, it will be pre-empted for each higher-level input that comes before the completion of C_i, and each such input may again be pre-empted by a still higher-level input, and so on. From this description, one way to solve the problem would appear to be to define a set of equations that essentially 'simulate' the processing from input to input, in much the same way that a discrete event simulation program would be structured. But equations of this form would be complex, intuitively unclear and, typically, have as many terms as the number of priority levels; moreover, the solution obtained would require unnecessarily many computations (one for each input).

The analysis can be greatly simplified by the recognition of one important fact: once processing at level i is pre-empted, the point of its resumption is independent of the order of arrival of inputs at higher levels.

Theorem 2. If processing at some level i is pre-empted at time t by one or more inputs at levels higher than i, and is resumed at time t', $t' > t$, then t' depends on the number of inputs but not upon their order of arrival in the interval $[t, t')$.

Proof. Obvious, based on commutativity of integer addition. □

To make use of this theorem, we define the maximum number of inputs from device j in the interval $[t, t')$ as

$$\text{Inputs}([t, t'), j) = \lceil t'/T_j \rceil - \lceil t/T_j \rceil. \qquad (2)$$

* We use the conventional notation $[t, t')$ to indicate a closed–open interval from t up to, but not including, t'.

Each input at level j requires a computation time of C_j. Thus, in the interval $[t, t')$ the computation required for inputs at all the levels from 1 to $i-1$ will be

$$\text{Comp}([t, t'), i) = \sum_{j=1}^{i-1} \text{Inputs}([t, t'), j) * C_j. \quad (3)$$

Since Comp is defined for an interval, it distributes over interval arithmetic: for $t_1 \leqslant t_2 \leqslant t_3$

$$\text{Comp}([t_1, t_2), i) + \text{Comp}([t_2, t_3, i) = \text{Comp}([t_1, t_3), i).$$

A necessary condition for a priority assignment to be feasible is that the average processing load is always met. For $M_i = LCM(T_i)$, the load condition

$$\sum_{j=1}^{i-1} (M_i/T_j) * C_j < M_i \quad (4)$$

guarantees that if the system critical instant occurs at time = 0, there will be no incomplete processing of inputs when the critical instant recurs at time = M_i.

We now define a function Response (t, t', i) which can be used to compute the worst-case response time RT_i at level i.

Response (t, t', i) =

if Comp $([t, t'), i) = 0$ then t'

else Response $(t', t' + \text{Comp}([t, t'), i), i)$ (5)

At time t, $t' - t$ is the computation time pending at level i. The function Response finds the time when the pending computation becomes zero (under our worst-case assumption). Consider the time interval $[t, t')$. Inputs arriving from levels 1 to $i-1$ in this period require Comp $([t, t'), i)$ processor time. The remaining time $(t' - t) - Comp([t, t'), i)$ is available for processing the input at level i. Thus the pending computation at time t' (i.e. just after the interval $[t, t')$) is

$$(t' - t) - ((t' - t) - \text{Comp}([t, t'), i)) = \text{Comp}([t, t'), i).$$

On termination of the recursion, Comp $([t, t'), i) = 0$; i.e. there are no pending inputs from the higher priority levels and the entire interval $[t, t')$ has been used for computation.

The response time for level i is given in terms of the function Response by the following equation:

$$RT_i = \text{Response}(0, C_i, i), \quad (6)$$

since at time = 0 the entire processing load for level i is pending.

The function Response terminates if the load relation (4) is met, because under that condition the interval $[t, t')$ will decrease on each successive recursive call to Response.

5. PROOF OF CORRECTNESS

Equation (1) defines the response time at level i. We now show that the result in (5) satisfies this equation.

Proof (using computational induction[6]). Let INV be an invariant over the parameters of each recursive call of the function Response.

$$\text{INV: Comp}([0, t), i) + C_i = t'.$$

(1) The initial condition Response $(0, C_i, i)$ satisfies the invariant.

(2) By the induction hypothesis, Response $(t', t' + \text{Comp}([t, t'), i), i)$ satisfies the invariant.

Therefore,

$$\text{Comp}([0, t'), i) + C_i = t' + \text{Comp}([t, t'), i) \quad (7)$$

We have

$$\text{Comp}([0, t'), i) = \text{Comp}([0, t), i) + \text{Comp}([t, t'), i)$$

where $$0 \leqslant t \leqslant t'.$$

Substituting in (7) we get

$$\text{Comp}([0, t), i) + C_i = t'.$$

This proves the induction step.

(3) On termination we have

$$\text{INV} \wedge \text{Comp}([t, t'), i) = 0 \quad \text{and} \quad RT_i = t'$$

Substituting for INV we get

$$\text{Comp}([0, t'), i) + C_i = t' \wedge RT_i = t'.$$

Finally, substituting for Comp and simplifying we get (1)

$$\left(\sum_{j=1}^{i-1} \lceil t'/T_j \rceil * C_j\right) + C_i = t' \wedge RT_i = t'. \quad (8)$$

Without any further assumptions about the structure of the program, the necessary and sufficient condition that no inputs are lost is

$$\forall i,\ 1 \leqslant i \leqslant n,\ RT_i < T_i. \quad (9)$$

Equation (8) can be shown to give the average load in the interval M_i. Substituting the LCM M_i for t' and using the constraint (9) we have

$$\sum_{j=1}^{i-1} (M_i/T_j) * C_j + (M_i/T_i) * C_i < M_i$$

$$\text{or} \sum_{j=1}^{i} (M_i/T_j) * C_j < M_i.$$

6. BUFFERED INPUTS

For some real-time systems, it may be found that the load equation (4) is satisfied but equation (9) is not. This means that for these systems the average load is within limits but the response time for at least one level is larger than its input repetition time, so that some inputs will get lost. The priority assignment for such systems can be made feasible by providing buffers in which inputs are saved (in a first-come-first-served order).

Theorem 3. If equation (4) is satisfied, a finite number of buffers will ensure that no input at any level is lost.

Proof. Consider the interval $[0, M_i)$ and let the critical instant for the system occur at time = 0. Equation (4) will be satisfied iff processing of all inputs arriving in the interval $[0, M_i)$ is completed by time = M_i. For all i, the number of inputs in the interval M_i is finite. Hence the number of buffers required will also be finite. □

We know that the critical instant for the system occurs when inputs from all the devices appear simultaneously. Thus at any priority level the maximum buffering will be required for this worst-case situation. Let RT_i be the worst-case response time at level i (RT_i can be computed using Ref. 6). Then the number of buffers required for this level is the smallest integer value k such that

$$RT_i \leqslant k * T_i. \quad (10)$$

Under certain specific assumptions, it is occasionally possible to reduce the number of buffers from 2 to 1. If the process servicing a device always reads the input at the start of its execution, thereby preventing its value from being lost during the processing, it may be possible for the remaining processing of the input to be completed after the arrival of the next input from that device. The load relation (4) must still hold; but the relation bounding the response time will now be $RT_i < 2T_i$. More details of this special case can be found elsewhere.[3]

Calculating the number of buffers is a requirement in a number of real-time systems. For example, in a packet-switched communication network, providing the correct number of buffers to meet some performance criterion will reduce (or eliminate) retransmission of packets due to buffer overflows. And in small microprocessor-controlled systems, where for reasons of economy the amount of additional hardware is kept to a minimum, it is important to know the amount of buffering required to be provided in the system. Another example occurs in distributed processing systems, where information collected or generated at nodes must be propagated to other nodes in an asynchronous manner: if the processing speeds of the nodes are not matched, outputs must be buffered if they are not to be lost.

7. TWO EXAMPLES

Example 1

A real-time system has four devices connected to a computer. The input repetition times T_i, the associated computation times C_i, and the specified limits RL_i to the response times are given in Table 1.

Table 1

Device	T	C	RL
P	10	1	10
Q	12	2	12
R	30	8	40
S	600	20	30

Substituting these values in (9), it can be seen that the load relation is met and the average system load is near 55%. However, there is no feasible assignment of devices to priorities under which all the response-time limits are also met.

A solution can be found by buffering inputs for device R. First, assign the devices to priorities in the order P, Q, S, R. This will ensure that the response limits for the devices P, Q and S will be met. The response time for S (which is at priority level 3) will then be

$$RT_3 = \text{Response}(0, 20, 3) = \text{if Comp}([0, 30), 3) = 0 \text{ then } 20 \text{ else Response Comp}([0, 20), 3), 3)$$

$$\text{Comp}([0, 20), 3) = \text{Inputs}([0, 20), 1) * 1 + \text{Inputs}([0, 20), 2) * 2 = 2 + 4 = 6$$

$$\text{Comp}([20, 26), 3) = \text{Inputs}([20, 26), 1) * 1 + \text{Inputs}([20, 26), 2) * 2 = 3$$

$$\text{Comp}([26, 29), 3) = 0$$

or

$$RT_3 = 29.$$

By similar calculation, RT_4 will be seen to be 40 but T_4 is only 30. Using equation (10), it can be shown that device R at priority level 4 needs two buffers if no input is to be lost. (In fact, if the assumptions of the special case, mentioned above, hold in this case, the use of the second buffer can be dispensed with.) Note that the *average* response time for device R is around 11, but it would be very misleading to take this as any indication of what might be the worst-case response time.

Thus, though the system load does not exceed 56%, buffering is needed if the response-time requirements are to be met and no inputs is to be lost.

Example 2

A system similar to that of Example 1 has the characteristics shown in Table 2.

Table 2

Priority	T	C
1	100	40
2	140	60
3	500	80
4	1000	10
5	1000	1

Find the response times at each level and the amount of buffering required to make this priority assignment feasible.

The load equation (4) is satisfied by the system. Using the function Response, the values shown in Table 3 are

Table 3

Priority	RT
1	40
2	100
3	560
4	2490
5	6991

derived for the response times at each level. From this, it can easily be seen that the numbers of buffers needed is 2 at level 3, 3 at level 4 and 7 at level 5. The relatively large numbers of buffers are required in this case because the average system load is very high–over 99%.

8. MULTIPLE PROCESSOR SYSTEMS

When the load relation is met but the feasibility equation (9) is not, buffers can be used to average out the rate of arrival of inputs. But when the load relation is not met, the total computational load of the system exceeds that

available from the processor, and buffering cannot help. Two other strategies are commonly used in these situations: using a processor of higher speed, or using more than one processor. The first remedy simply has the effect of reducing the processing times required for each input, while the second introduces parallelism in processing which can reduce the response time at each level. When compared with the previous case, a processor which is m times faster and a system with m processors each has a load equation of the form

$$\sum_{j=1}^{i} ((M_i/T_j) * C_j)/m < M_i \qquad (11)$$

and if this is met, the system is capable of meeting its processing requirements. There is no difficulty with determining the response time in the case of the faster processor. But the same analysis does not hold for a multiple-processor system.

If the system has m processors and n devices, and $m > n$, it is no longer necessary for the computation time for an input to be less than its repetition time.

Example 3

Consider a system with one device and two processors. Let $T = 100$ and $C = 150$. Equation (11) will be satisfied, and no input will be lost if alternate inputs go to each processor.

If, as is more likely, $m < n$, the analysis is considerably more complicated. Let RT_i and RT_i^m be the response times at level i with one and m processors respectively. The following fairly obvious propositions can be proved for all such systems.

(1) For $1 \leqslant i \leqslant m$, $RT_i^m = C_i$.

(2) For $m \leqslant i \leqslant n$, processing at level i cannot start before time t, where $t = \min(\{RT_{i-m}^m, \ldots, RT_{i-1}^m\})$, and will end by RT_i.

More specific propositions are restricted to particular cases, and the number of cases that need analysis is so large that a general method of analysis is very difficult to formulate. Providing (10) is met, simulation can be used to find the worst-case values of the response times.

Given the difficulty of the general analysis, it is comforting to know that good results can be obtained by partitioning an m processor system into m separate systems and making an appropriate (but static) assignment of devices to processors. Since each separate system has one processor, the earlier analysis can be used. The additional flexibility that is available is that there is now a choice of priority level *and* processor for the connection of each device, and this can be used both to get the required response times and to average the load on each processor.

9. CONCLUSIONS

Programming for real-time systems has long been considered difficult because in addition to the problems of concurrent programming there is the need to ensure that real-time constraints are met.[8] As a result, the best efforts at such programming have usually followed careful coding practices, sometimes reinforced by discrete event simulation studies. As will be noticed from the analysis above, care in coding is not enough to guarantee that the system will meet real-time constraints, nor can they always be met by keeping the average processor load at a low level. Even for a system with low average load, inputs may be lost at some level due to the characteristics of the repetition times and the computation times of the previous levels. And with a high level of criticality, it will not do to use approximate figures for these times. Thus the system load and response times must be computed for the particular computation times of the code that will actually run on the system. Every time alterations are made in the code, this procedure must be repeated.

The analysis given here is for programs with very simple structure. Basically, it has been assumed that each device is serviced by a sequential process which does not communicate with any other process, and that the arrivals of inputs are independent of each other. This does not preclude dependencies of the form where an input from device j may arrive after, say, m inputs from some device k; such cases can be modelled by choosing the period T_j to be m times T_k.[4] A more restrictive assumption is that the upper bounds of the processing times of processes are independent of the previous history of their executions and the values received as inputs.

Typical real-time programs are designed without such constraints and so this analysis cannot easily be applied to them. However, the time-dependent structure of such programs is usually so complex that any general worst-case analysis would be extremely difficult. For such an analysis to be feasible, it appears necessary for these programs to have simple and well-defined structures, and this is one more argument in favour of Currie's general prescriptions for programming safety-critical systems.[1]

An obvious extension of our work is to extend the analysis with some of the assumptions relaxed. From preliminary work in this direction, it appears feasible for such an analysis to be automated, perhaps by the compiler for the real-time programming language. (The suggestion that static compile-time analysis can be used for performance studies has also been made by Mahjoub,[5] though for the somewhat different case of distributed systems, and without priority-based inputs.)

Harter[2] has studied a similar problem, but using a framework based on a temporal logic proof system. As he also considers procedure calls between processes at different priority levels, the worst case becomes difficult to define, and this makes comparison with other work rather involved. The advantage of our analysis is that though it is based on very simple reasoning using interval arithmetic, the results are powerful enough to be widely applicable and extend easily to handle problems like buffering. This is an outcome of avoiding the rather obvious temptation of following an operational approach which, for multi-level priorities and pre-emptive processing, can be surprisingly complex.

REFERENCES

1. I. F. Currie, Orwellian programming in safety-critical systems. In *Proceedings, IFIP Working Group 2.4 Conference on System Implementation Languages: Experience and Assessment, Canterbury (1984)*, edited P. R. King. North-Holland, Amsterdam.
2. P. K. Harter, Jr., *Response Times in Level-Structured*

Systems, Technical Report CU-CS-269-94, Department of Computer Science, University of Colorado (1984).
3. M. Joseph, On a problem in real-time computing. *Information Processing Letters* (in the Press).
4. C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM* **20** (1), 46–61 (1973).
5. A. Mahjoub, On the static analysis of distributed systems performance. *The Computer Journal* **27** (3), 201–208 (1984).
6. Z. Manna, S. Ness and J. Vuillemin, Inductive methods for proving properties of programs. *Communications of the ACM* **16** (8), 491–502 (1973).
7. A. Moitra, *Analysis of Hard-real-time Systems*, Department of Computer Science, Cornell University (1985).
8. N. Wirth, Towards a discipline of real-time programming. *Communications of the ACM* **20** (8), 577–583 (1977).