

Seminar report: “*Real time calculus for Analysis of System performance bounds*”

12.06.2006

Lingyun Li, Supervisor: Prof. Dr. M. Radetzki

1.	Background and motivation	1
1.1	Traditional design method.....	2
1.2	Many models have restrictions.....	3
1.3	Advantage of this new method.....	4
2.	Main principles	5
2.1	2 basic input parameters.....	5
2.2	The single stream, single resource	7
2.3	Multiple streams, single resource.....	9
2.4	Multiple streams, multiple resources	10
3.	Using DAG to calculate demand time $a_d(\Delta)$	16
4.	Conclusion	20
5.	References.....	22

1. Background and motivation

This report is mainly about a new mathematical model to represent the behavior of complex networks. It is also applicable for simple periodic systems. The feature of this new model is that it does not have restrictions on the system to be investigated.

There are already many previous papers in the field of timing characteristics

analyzing of embedded system designs and these papers have their own advantages: they try to weaken various kinds of restrictions on system model and do have achieved a lot. For example, [1] can give the result of process scheduling using static priority under a certain industrial bus protocol. It combines pure theoretical analysis with real applications. Besides, papers [2] and [3] already deal with multi-domain analysis. But to different extents, they still have restrictions on event models. A method to analyze heterogeneous platform-based architecture in a single coherent way is proposed. Its goal is simple and clear: remove restrictions (for example: events should be somehow periodic, or the analysis result can only be response time) on event models and answer the question that: how the whole system (which consists of several different components) behaves and what the characteristics of dynamic load on systems are. An effective algorithm to compute the request curve (in the paper the precondition is that the system is using static priority) is also proposed.

1.1 Traditional design method

Because this report is related to embedded system design, one classical and very mature design method is briefly mentioned and evaluated. Platform-based design is the most frequently used methodology. Its target is obvious: reuse available cores, IPs or blocks from 3rd party at several levels of abstraction. Everyone knows that this design methodology can avoid redundant re-design works and save unnecessary test routines thus the development cycle is greatly shortened. Because this method has been used for long time, its disadvantages have already drawn engineers' attention: just because of its advantage, the integration works tend to be out of control sometimes. Cores are usually provided without detailed description of internal structures. Lack of this usually requires additional long time to build complex "interfaces or adaptors" [8].

When all discrete components are connected, of course a formal description of the system properties, like utility of bus bandwidth, usage of memory and response time, should be done. The main problem is: because we use platform-based design (so far we have no other design methodology better than this), it is difficult to analyze the system as a whole on a relatively high abstraction level. Mature and well-developed

commercial analysis tools seem to be only available for a certain analysis area. They can not analyze a compound system.

1.2 Many models have restrictions

There are already some break-ups trying to solve this problem. [4] and [5] did some research and achieved some results although they are not very effective or efficient. Their solution in fact is quite direct: since there is no available mathematical model to model or analyze (compound) products which are developed using platform-based design, we simply use simulation. It works in many cases. The price for it is: very long simulation time. Also some significant characteristics like “corner cases and complete coverage” [8] can easily be ignored or even can not be captured by simulation. These are the main disadvantages of the most primitive but to some extents quite efficient way.

As we already mentioned before, although there are many mature and long-time-available analysis tools not for compound system but for single core, they also have many restrictions on prototypes. Many of them can only model purely periodic systems with jitter, burst or sporadic systems. Arbitrary systems are not well dealt with. In this case they are “approximated by some standard model which minimizes the error” [8]. The direct drawback resulted from this method is that: only overly conservative bounds can be achieved. In other words, because we can only achieve extremely pessimistic estimation on system performance, we will quite possibly either waste lots of hardware and software resources or erroneously judge the feasibility and schedulability.

The situation until now can be concluded: for single component, if it is periodic, or almost periodic, then we have many highly successful methods and commercial tools to analyze its timing characteristics. If we want to find out the properties of a compound system, or the system behaves in an arbitrary way, we need to find a new and efficient way besides simulation.

1.3 Advantage of this new method

The method presented is not brand new. It is derived from the methodology presented in papers [6] and [7]. They solved two main drawbacks: now it “is applicable to the more general domain of heterogeneous embedded system design” [8] and the timing bounds are “tighter compared to our previous results” [8].

Multiple input event streams (which are not necessarily periodic) with multiple processing capabilities are analyzed in a high abstract level. Additionally, this problem is solved in quite an efficient way. The computational complexity is significantly reduced. The main tasks for system analyzers are shifted to record how many events will probably happen in a certain time interval. Since how much service can be provided in a certain time interval is usually known, all necessary input parameters are available for this new method. After computation, we can get the output behavior of system as well as the load on different blocks of the system.

2. Main principles

2.1 2 basic input parameters

Different from other models which usually require the events should be periodic or periodic with jitters, two functions are proposed to describe the input event streams and the processing capacities. These two functions have no requirement on the characteristic of the input event stream. They are: request curve $\alpha(t)$ and delivery curve $\beta(t)$. These two functions are the kernel variables used in the whole model.

For request curve $\alpha(t)$, it is in fact the limit (lower and upper) of input event stream. Input event stream is described by function $R(t)$. Independent variable “ t ” means the time from 0 up to t . Function $R(t)$ stands for the number of input events (events requested) from time 0 up to t . As stated before, request curve $\alpha(t)$ is used to set the lower and upper limit for $R(t)$.

The independent variable “ t ” in request curve $\alpha(t)$ and $R(t)$ has different meaning. Letter “ t ” in request curve $\alpha(t)$ means any time interval which equals to t . But the independent variable in $R(t)$ means the time period from 0 up to t . In order to avoid this chaos, sometimes “ Δ ” is used instead of “ t ” in request curve $\alpha(t)$.

Now come the two variations of request curve $\alpha(t)$. Because the “worst case bounds on system properties” [8] is interested, lower and upper bounds are used to describe $R(t)$. On this point, it is similar with other analysis methods:

$a^l(\Delta)$ is the lower bound (“ l ” stands for lower) for $R(t)$

$a^u(\Delta)$ is the upper bound (“ u ” stands for upper) for $R(t)$

From these 2 request curve functions $a^l(\Delta)$ and $a^u(\Delta)$ we can now answer the question: why this new model discussed here does not have restriction on the behavior of input event stream? In other words, how can this model to describe both periodic

and aperiodic input event stream?

It becomes true when we apply certain conditions to $a^l(\Delta)$ and $a^u(\Delta)$. Here we have a very simple example: a pure periodic system with period “ p ”, we can easily find out: in any time interval smaller than “ p ”, if it is a pure periodic system with period “ p ”, at least none event will happen during this interval and at most one event will happen during this interval. This is how the model describes pure periodic input event stream. By setting different $a^l(\Delta)$ and $a^u(\Delta)$, it is not difficult to simulate various kinds of systems. Therefore, the input event streams with period are only special cases of this new model.

When input stream has certain characteristics, for example, it is periodic, or periodic with an estimated jitter, it is not difficult to calculate its $a^l(\Delta)$ and $a^u(\Delta)$. But when the input stream does not have any characteristic, that is to say, it is purely arbitrary, or in other words, its input behavior can not be predicted at all, is this model still feasible? The answer is still “Yes”. One method to find out correspondent $a^l(\Delta)$ and $a^u(\Delta)$ from a relatively primitive way is presented below:

We can record a complete input event stream, and then open a time window whose length is Δ . Then we slide this time window along the whole event cycle to find out, in a certain time interval Δ , at most how many events come and at least how many events come.

These two request curves $a^l(\Delta)$ and $a^u(\Delta)$ are used to describe the characteristics of input stream. Similarly, we use service curves $b^l(\Delta)$ and $b^u(\Delta)$ to describe the processing capability $C(t)$. Now the 2 basic formulas are clear as stated:

$$a^l(t-s) \leq R(t) - R(s) \leq a^u(t-s) \quad \text{for any } 0 \leq s \leq t$$

$$b^l(t-s) \leq C(t) - C(s) \leq b^u(t-s) \quad \text{for any } 0 \leq s \leq t$$

Again, these two formulas are the basic concepts proposed here. Further calculation and reasoning are always based on them.

2.2 The single stream, single resource case

It would be quite understandable to use simple and complex examples to elaborate the algorithm. For the simplest case, we can abstract the system model as: single input stream and single processing capability. In this situation, no other input stream competes for processing resources. The reason why we introduce this example is: after this example is carefully analyzed, we will get four very important derived variables: $a^l(\Delta)$, $a^u(\Delta)$, $b^l(\Delta)$ and $b^u(\Delta)$. The procedure to deduce them will only be mentioned briefly. They are also the theoretical foundations of the following parts. Compound systems like multiple input event streams and multiple processing resources can always be split into simple single stream and single resource case.

Because this model is on a relatively high abstract level, it can deal with not only response time problems. Here we suppose that: one special channel is used to transmit important data from an external sensor. The collected data is analyzed by a weather station. Then we have the following interpretation:

1. The external sensor will generate at least $a^l(\Delta)$ bytes data to be transmitted in any time interval of Δ so that the system can record a complete external temperature list;
2. On the other hand, the sensor will generate at most $a^u(\Delta)$ bytes data to be transmitted in any time interval of Δ , more data will be redundant and does not help improve the accuracy;
3. During any time interval of Δ , the channel will at least transmit $b^l(\Delta)$ bytes data in order to make sure a lower limit of sensor data can be processed;
4. Because of channel buffer capacity, at most $b^u(\Delta)$ bytes data can be transmitted during time interval of Δ .

$a^l(\Delta)$ is the lower limit of “outgoing arrival curve” [8]. It means that, after processing, at least a certain number of events will be output from processing unit. From this curve, we can analyze the output characteristic of an input event stream. But its more important usage is that, it will be connected to the next processing unit as the lower limit of input event stream. But in this section, we restrict it as single input event stream and single processing capability.

$a^u(\Delta)$ has the same principle as $a^l(\Delta)$. The only difference is that $a^u(\Delta)$ stands for the upper limit of the output event stream.

$b^l(\Delta)$ is the lower limit of “remaining service curve” [8]. Because the processing resource is (partially) occupied by input event stream, during a certain time interval, the processing capability will decrease. Its remaining capacity is characterized by this function.

$b^u(\Delta)$ is the upper limit of “remaining service curve” [8].

Four complex formulas to compute these four important functions are given without deducing or explanation. In this form, the causal relationship is still clear:

$$a^l(\Delta) = \min\left\{\inf_{0 \leq m \leq \Delta} \left\{\sup_{l > 0} \{a^l(m+l) - b^u(l)\} + b^l(\Delta - m)\right\}, b^l(\Delta)\right\}$$

$$a^u(\Delta) = \min\left\{\sup_{\Delta > 0} \left\{\inf_{0 \leq l < \Delta} \{a^u(m) + b^u(l + \Delta - m)\} - b^l(l)\right\}, b^u(\Delta)\right\}$$

$$b^l(\Delta) = \sup_{0 \leq l \leq \Delta} \{b^l(l) - a^u(l)\}$$

$$b^u(\Delta) = \min\left\{\inf_{l > \Delta} \{b^u(l) - a^l(l)\}, 0\right\}$$

For $b^l(\Delta)$, we can interpret it as: within a certain time interval Δ , we set another time interval λ , and for all $0 \leq \lambda \leq \Delta$, the difference between the lower limit of service capacity and request are checked, and the “largest” difference is selected as the lower limit of the remaining capacity limit. The theory is absolutely correct, but when this theory is implemented by computer program, how to set the reasonable step of λ is

still an open issue.

It is important know, after a certain kind of complex calculation, we will obtain the “after processing” [8] characteristics of both the input event stream and the processing capability. According to our previous assumption, we can interpret these four formulas as following:

1. We can get at least $a'(\Delta)$ bytes at the end of transmission channel during a time interval of Δ ;
2. We will receive at most $a''(\Delta)$ bytes data at the output of transmission channel during a time interval of Δ ;
3. The transmission channel will have at least $b'(\Delta)$ free bandwidth during the time interval Δ ;
4. During time interval Δ , the transmission channel will have at most $b''(\Delta)$ free bandwidth.

These data will be very helpful to analyze the behavior of system.

2.3 Multiple streams, single resource

Because the model proposed are mainly used to deal with the complex behavior of compound systems, it is necessary to extend the simplest case “single event stream, single processing resource” finally to “multiple streams and multiple resources”. As an intermediate case, we have a situation as “multiple input event stream with single processing resource” [8]. This condition is not strange to us. Imagine that, most PCs have only one CPU, and all PCs are running multiple task operating systems. We can map the “single CPU” to “single resource” and map “multiple tasks” to “multiple input event streams”. This situation is solved without adding too many parameters to the formulas. They just use a weight factor w_i for each stream. This factor stands for

the different load requirement of each input event stream. This is quite reasonable because in real cases, of course different kinds of tasks will ask for different processing time. Again, the model proposed here is applicable for static priority scheduling. It means, process with higher priority will keep on running until it finishes and then lower process will be able to execute. We assign different priority to Process i , and then we have $b_i^u = b_{i-1}^u$ and $b_i^l = b_{i-1}^l$ (the smaller i is, the higher priority the process has). For α we have similar situation. The computations are based on the above mentioned 4 formulas. Below is a picture stating the resource allocation in this case. It clearly shows that, only the remaining service capacity left from a process with higher priority can be passed to a process with lower priority as input service capacity.

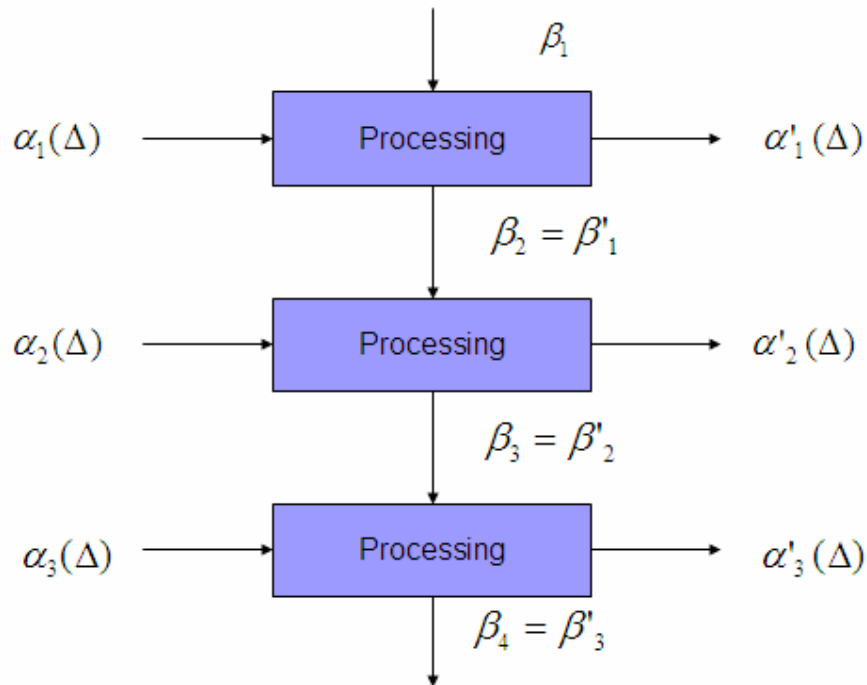


Figure 1. System allocation for multiple streams, sing resource.

2.4 Multiple streams, multiple resources

The main goal the new model is to analyze the system properties of compound systems. Because an embedded system will in most cases have multiple event queues and multiple processing units, we will now have a good look at this.

The following example is directly cited from the original paper [8].

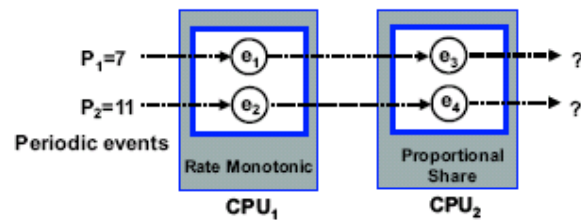


Figure 2. System description.

I put the above system into following real application:

1. P1 stands for input video data. Because video stream has much larger data volume, we assign it with higher priority and it has a period of 7.
2. P2 stands for input audio data. It is assigned lower priority and has a period of 11.
3. It is quite reasonable because in real application the sampling rate and data of video stream is much larger than audio stream.
4. CPU1 does some pre-processing work like remove spikes in the sampled data. We use "Rate Monotonic" scheduling, this is to say, P2 can not be processed when P1 is in processing.
5. After pre-processing, video and audio data are passed to CPU2 to be encoded. We use "Proportional Share" scheduling (it means 2 process will have the same processing ratio).
6. We use "Proportional Share" algorithm because we want to keep the video and audio as synchronous as possible.
7. All input streams, to CPU1 and CPU2, each event requires the same processing time: 2 time units.

The two input streams' characteristics and rescheduling mechanism are known. How

the output stream will behavior (its period, jitter) can be derived from the formulas mentioned in Section 2.2.

In order to show the data flow in a more direct way, we draw the following picture:

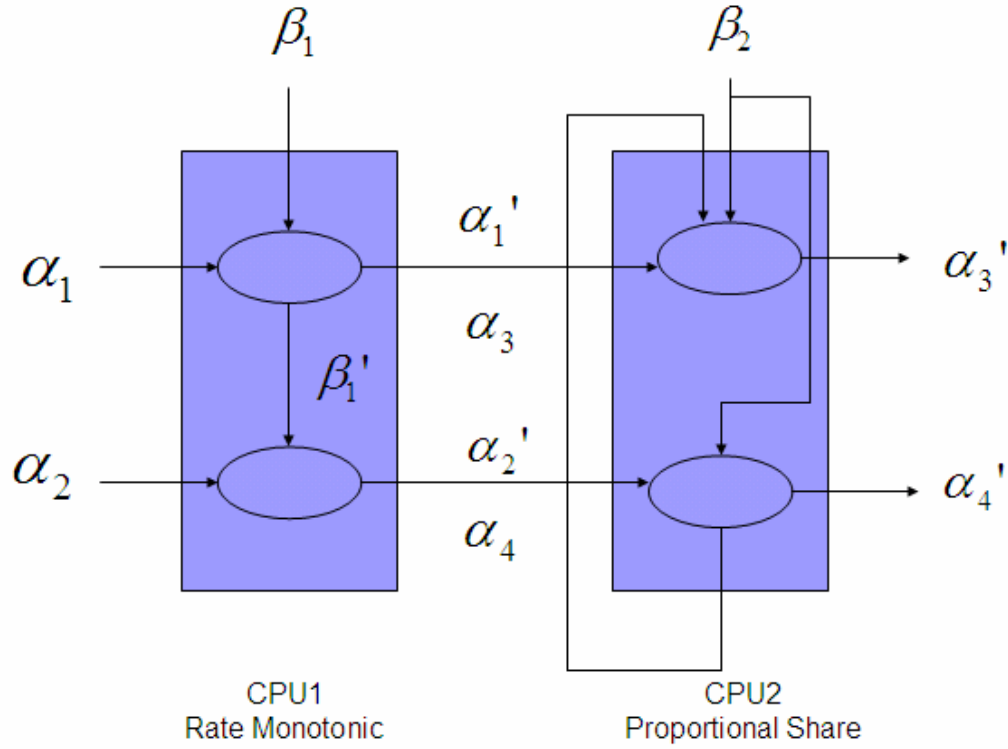


Figure 3. Multiple streams, multiple resources.

It is clearly shown by this picture, the processing capability β_1 of CPU1 is first to be occupied by input event stream α_1 because of its scheduling algorithm. Only the rest processing capabilities will be passed to α_2 . After processed by CPU1, the two streams will enter CPU2 and share its processing capability.

For the pre-processed video sample stream a_1' , we will have the following graph Figure 4. [8] computed from these 2 formulas provided previously:

$$a^l(\Delta) = \min\left\{\inf_{0 \leq m \leq \Delta} \left\{\sup_{l > 0} \{a^l(m+l) - b^u(l)\} + b^l(\Delta - m)\right\}, b^l(\Delta)\right\}$$

$$a^u(\Delta) = \min\left\{\sup_{\Delta > 0} \left\{\inf_{0 \leq l < \Delta} \{a^u(m) + b^u(l + \Delta - m)\} - b^l(l)\right\}, b^u(\Delta)\right\}$$

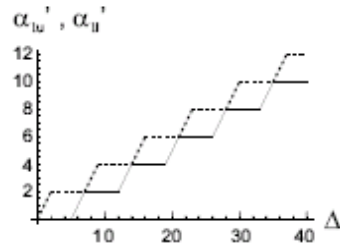


Figure 4. Arrival curve of event stream 1.

The dashed line of course shows the upper output curve because its position is above the solid line. When we draw this curve, we do not have to consider α_2 just because it has lower priority thus does not have any effect on α_1 . We can see the first corner of upper arrival curve happens at time 2, then the second corner at time 7. At time 7, the lower arrival curve also begins to rise because the period of stream 1 is 7.

After having processing α_1 , we can start to computer the remaining service curve β_1 which will be used as input service curve of α_2 . All formulas needed are already listed in Section 2.2. Below the curve is shown [8]:

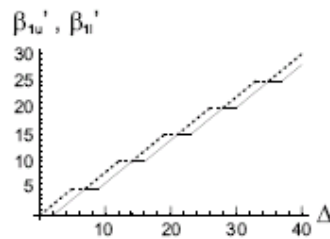


Figure 5. Remaining service curve of CPU1 after processing event stream 1.

It is easy to read the above picture. First, we do not consider the effect of α_2 because it has lower priority and according to the scheduling algorithm it can only wait until higher priority process finishes. The lower curve arises from 0 at time 2 because α_1 is being processed from time 0 to 2. Then from time 2 to 7, CPU1 is free. Again, from time 7 to 9, α_1 arrives for the second time so the curve can not rise and keeps a slope of 0. The upper limit is just the shifted version of lower curve. The offset is exactly the processing time for one event: 2 time units.

We use the four formulas repetitively to calculate all the curves on the first graph. The only difference is that: because CPU2 uses “Proportional Share” scheduling algorithm, the service curve β_2 has only half slope of service curve β_1 . Figure 3 shows clearly that: service capacities of CPU2 are equally shared by stream 1 and stream 2. In other words, CPU2 has to allocate its processing ability equally to the two processes according to its scheduling algorithm.

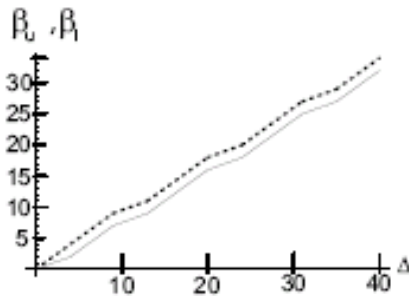


Figure 6. [8] Service curve of CPU2 to process stream 1 coming out of CPU1

From the above picture, we see that the service curve is not an absolutely straight line with slope 0.5, but it has some corners. This is because of its scheduling mechanism, if one stream does not consume all the processing capacity allocated for itself, the remaining capacity will be passed to the other stream. In this case, the variations are contributed by stream 2 coming out of CPU1.

The final result is interesting (see pictures below): the coded video stream has a jitter of 2 while the coded audio stream has a jitter of 4 (jitter can be calculated by dividing the distance of upper and lower curves by 2). And according to the final computation result, they are still periodic. From arrival curve a_3' , the period is proved to be the same, namely 7 time units, as its input period. The condition is the same for arrival curve a_4' , the second event stream keeps its period (11 time units). This model shows the phenomenon: if a processing resource has to be shared among processes, then jitter will quite possibly appear and increase even each individual input is pure periodic. But after processing, the event streams are still periodic. The reason is hidden behind the presumption of this model: all tasks are scheduable and the scheduling mechanisms (different scheduling algorithms are acceptable and have

different computation procedures respectively) are not preemptable. How to assign the proper priorities to processes will be mentioned in next chapter.

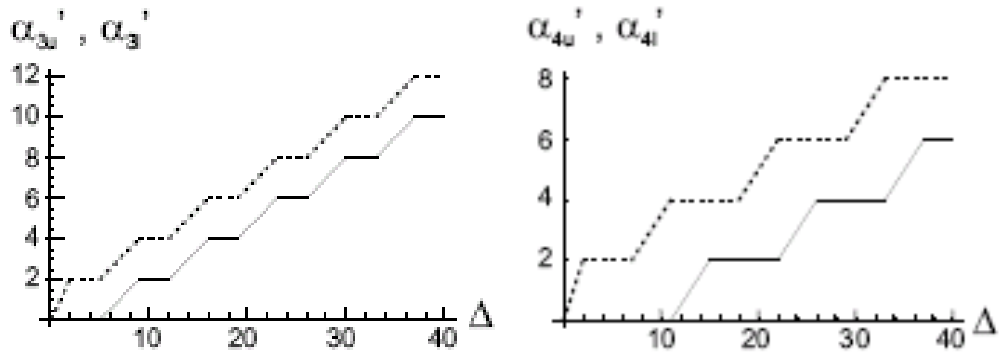


Figure 7. [8] Final arrival curves of streams 1 and 2.

3. Using DAG to calculate demand time $a_d(\Delta)$

In embedded system design, a widely considered question is: during a certain time interval Δ , what is maximal computation time $a_d(\Delta)$ (including “triggering time and deadline” [8]) required? This data is absolutely necessary and extremely important because in hard real-time system we have to satisfy the criteria otherwise severe consequence will happen.

Actually there are already many algorithms which can do this, or, almost achieve the optimal result if the problem finally turns to be an NP problem. Then what is the unique advantage of the algorithm proposed here? The answer is: this new algorithm reduced the computational efforts from exponential complexity to $O(n^3)$. To apply this algorithm, the precondition is: each task has a fixed computation time, earliest starting time and the latest finishing time. Besides, the interval between tasks should also be determined before applying this algorithm.

The following graph is an example which can use this algorithm to investigate its maximal computation time $a_d(\Delta)$:

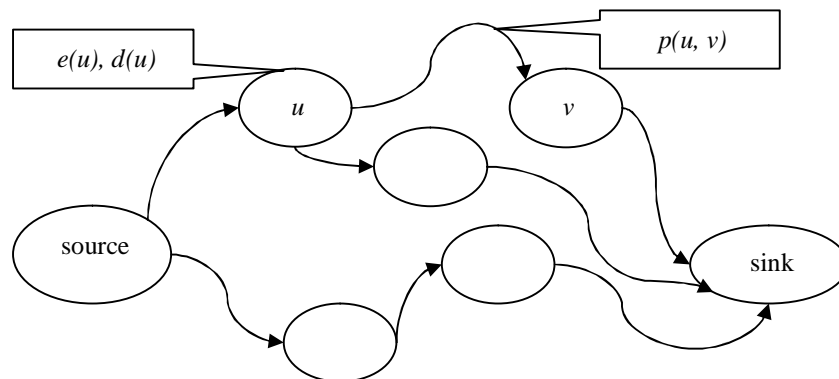


Figure 8. DAG flow chart.

Here is some explanation of the functions used in this chart and structures of the flow chart:

1. First, a task consists of several sub tasks, they are displayed by nodes in the graph. Function $e(u)$ means the execution time of sub task u . $d(u)$ is the deadline of sub task u . $p(u, v)$ stands for the “pause” time between sub tasks u and v . In other words, sub task v can not start earlier than sub task u began and time $p(u, v)$ has already passed by. “Source” is the starting point and “sink” is the end point.
 2. The whole graph should be a Directed Acyclic Graph. The reason is quite clear: otherwise there will be loop in the graph and the tasks can never be finished.
 3. If the task is periodic, just copy the graph and put the “source” node of the next graph to the position of “sink” node of the previous graph. Thus, periodic tasks can be modelled.
 4. $L(i)$ is a list of sub tasks ending at node i , and L^i is a set of sub tasks ending at node i . The difference is that: the elements of $L(i)$ are ordered.
 5. In a tuple like (f, Δ) , “ f is a lower bound in the execution demand in interval Δ ”.
- [8]

For $i = 1, \dots, j$, “One iteration of the algorithm to compute $L(j+1)$ and L^i ” [8]:

```

 $L^{j+1} = \{(e(j+1), d(j+1))\}$ 
for each edge  $(k, j+1)$  do
  for each tuple  $(f, \Delta) \in L^k$  do
     $L^{j+1} = L^{j+1} \cup$ 
       $\{(f + e(j+1), \Delta + p(k, j+1) + d(j+1) - d(k))\}$ 
 $L(j+1) = L^{j+1} \cup L(j)$ 
Reduce the sets  $L^{j+1}$  and  $L(j+1)$ 

```

For each arrow pointing to node $j+1$, we first check the arrow’s starting node, then we add all possible tuples to the set. Clearly, $f+e(j+1)$ is the new lower bound for node $j+1$. $p(k, j+1)+d(j+1)-d(k)$ is the time from the complete time of sub task k to the complete time of sub task $j+1$. After the iteration, all possible paths are put into set.

Now remove redundant set to simplify the following computation:

for each tuple (f_i, Δ_i) and (f_j, Δ_j) , we check if there are any weaker requirement. If $(\Delta_i \leq \Delta_j)$ is true and $(f_i \leq f_j)$ is also true, it means that we find a requirement, here namely tuple j, it has looser time requirement and should be removed to avoid unnecessary further comparison.

After all the redundant tuples are removed, we just find out the maximum value of f , it is the value of $a_d(\Delta)$.

The implied restriction on this system model is that, all sub tasks are already well pre-defined, their start time, end time, estimated execution time and intervals between each tasks are already fixed. Also, no dynamic execution, in other words, changing the sequence of execution, can be implemented in this algorithm.

After we have got the important parameter $a_d(\Delta)$, we can easily find out whether our design can satisfy all requirements for all incoming processes. The schedulability test runs like following:

we assume that static priority is used. As the examples mentioned before, larger priority number means lower priority. If there is higher priority process running, lower ones can do nothing but to wait.

Then the following criteria to check whether the processes with assumed priorities can be scheduled or not is given out:

for any time interval Δ ,

$$b'(\Delta) \geq a_d^i(\Delta)$$

in which $b'(\Delta) = \max_{0 \leq u \leq \Delta} \{b(u) - a_r(u)\}$

Here because we suppose the scheduling algorithm is static priority scheduling, so

$a_r(u)$, which is used to calculate $b'(\Delta)$, is the sum of all arrival curves whose priorities are higher, namely $\sum_1^{i-1} a_r^i$.

This is not difficult to interpret. The left side of the inequation stands for the remaining service ability of a certain system (here we assume the processing unit is single). The right side of the inequation is the demand curve of process i . And of course the left side should at least have the same value of the right side in order to meet process i . Otherwise the system will fail.

This could also be used to assign proper static priorities to tasks so that the system will cooperate well. First choose one task as the beginning point, then choose randomly another task, give this task the lowest priority, use $b'(\Delta) \geq a_d^i(\Delta)$ (just discussed in this section) to test whether they are schedulable. If the condition $b'(\Delta) \geq a_d^i(\Delta)$ fails for all combinations, then the tasks can not be scheduled. If it is schedulable, remove it, repeat this procedure until no more task is left. According to this iteration, the earlier one task is removed, the lower its priority is. This is because we always try to assign the possible lowest priority to one task when we test it. We can also draw the conclusion that, from this algorithm, it is possible to find out several different and correct priority assignments. The reason is, each time we pick up one task to test, the task under test is chosen randomly.

4. Conclusion

This report discusses the analysis method on different architectures and weakens the requirement and restriction on system models proposed in [8]. Besides, the timing bounds of input event streams can be computed in complexity of $O(n^3)$.

The most obvious difference which distinguishes this system model is:

the model does not use traditional quantities like starting time, execution time, latest allowed finish time, or period / jitter to describe the behavior of a system. It proposes new mathematical models, namely $a(\Delta)$, $b(\Delta)$ and their variations to represent any arbitrary system.

Because this new model only use the number of events happened in a certain time interval to characterize system behaviors, theoretically there is no precondition on the input event stream(s) compared to other algorithms which usually requires the input event stream(s) is / are at least pseudo periodic (periodic with a reasonable jitter, or burst, or not periodic: sporadic).

Another advantage of this new model is that it can simulate compound systems. Platform-based design is the dominant design methodology so far and almost all products are composed of different relatively independent blocks. By connecting the output of the previous core to the input of the next core (how to compute the output curve and remaining service curve is presented in Section 2.2 in detail), the new model is able to answer the system properties of an abstract network.

Except the above mentioned advantages, there are still some issues left open: in theory it appears to be universal because it does not have any precondition on the system model. But the computation of the 4 kernel formulas in Section 2.2 seems very abstract and complex. How to choose proper value of μ and λ used in the functions,

and how to implement the special algorithm by computer programs are still quite obscure.

Another point which is interesting is that: how sensitive is the final result, namely $a'(\Delta)$ and $b'(\Delta)$ to the input parameters $a(\Delta)$ and $b(\Delta)$? Does the mathematical model easily generate spikes? This is very important because when analyzing chip behavior, the computation complexity usually tends to be extremely large and the mathematical model should at least to some extent converge, otherwise the efficiency will be a problem. It may be necessary for this new model to first set a clear condition under which this algorithm can be applied. For example, if there is any dynamic execution or feedback, this algorithm will have difficulties to work out the final result. Computer program can only deal with finite algorithm, that is, the algorithm will definitely end after some time with an acceptable result. For computing $a_d(\Delta)$, the computation complexity is proved to be $O(n^3)$. But for computing the output behaviors and load requirements, the complexity still needs to be discussed.

5. References

- [1] P.Pop, P.Eles, and Z. Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. In *Proc. DATE*, 2000.
- [2] K. Richter and R.Ernst. Model interfaces for heterogeneous system analysis. In *Proc. DATE*, 2002.
- [3] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proc. 39th DAC*, 2002.
- [4] The Cadence Virtual Component Co-design (VCC). <http://www.cadence.com/products/vcc.html> Cadence Design Systems
- [5] Seamless Hardware/Software Co-Verification, Mentor Graphics. <http://www.mentor.com/seamless/> Mentor Graphics
- [6] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *Network Processor Design: Issues and Practices, Volume 1*. Morgan Kaufmann Publishers, October 2002.
- [7] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proc. 39th DAC*, New Orleans, 2002.
- [8] S. Chakraborty, S. Künzli, L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proc. DATE*, 2003.