First IEEE Symposium on Industrial Embedded Systems,
Antribes Juan-Les-Pins, October 2006

Analyzing the Timing Characteristics of Task Activations*

Frank Bodmann, Karsten Albers, Frank Slomka
Department of Computing Science – University of Oldenburg
{Frank.Bodmann, Karsten.Albers, Frank.Slomka}@informatik.uni-oldenburg.de

Abstract

We propose a model and method to determine the temporal distribution of task activations occurring within an event driven real time system. As real time systems even in safety critical applications are becoming increasingly complex, it becomes increasingly important to get a better understanding of the activity within these systems. The event stream model is a powerful yet efficient way to describe the temporal occurrence of events within a system. Real time analysis can be elegantly performed using given event streams, however it has so far not been investigated how the event streams themselves can be acquired. By calculating the temporal distribution of task activations within a system, a more accurate description of the timing behavior of a system is possible which in the end can lead to a save and cost efficient estimate on the hardware required for the system.

1. Introduction

Verifying and finding a cost optimal mapping of software onto hardware components for a distributed real time system requires expressive models and efficient methods to determine the systems timing behavior. One important aspect is to determine the worst case and best case execution times of the systems individual tasks. However once this information is known, the timing behavior of the interaction of these tasks has to be considered, as the activity of one task may trigger activity of other tasks which translates into load on the system. Consequently this has to be taken into account when verifying whether the selected hardware is guaranteed to always meet the desired timings.

In this paper we introduce a method that allows an efficient automatic extraction of its relevant timing dependencies from a system specification or implementation. This is achieved by taking the inner state of tasks into account, but abstracting from it by transforming the tasks control flow into its impact on the overall system within arbitrary time intervals. The result is a description of the systems timing behavior that can be analyzed efficiently using established real time analysis techniques.

This paper is organized as follows: after presenting the related work, the model used by this approach is introduced and a method is described to calculate the timing of events generated by a task during one invocation. Based on this method a generalization is offered which shows the timing of events generated by a task, which is itself triggered by events with a given timing. These information are used for an efficient real time analysis of static or dynamic priority scheduling. Finally an example is given where the results are compared to those generated by traditional means.

2. Related Work

The periodic task model [1] shown in Figure 1 assumes tasks to be activated at a constant rate T. Several extensions have been introduced to the traditional periodic task model to overcome its shortcomings, such as the sporadic task model [2] and the recurring real time task model [3]. The event stream model [4] is more expressive by describing the maximum number of events in arbitrary time intervals. It is described in detail in Section 3.1.

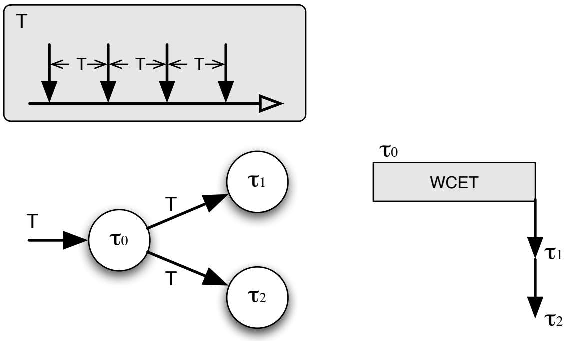


Figure 1. Simple task model

All these models assume that internal events are generated by tasks at the end of their execution. However in reality these events may occur anytime during their execution, so these models lead to overly pessimistic approximations. Consider Figure 2. As the runtime of task τ_0 may vary across activations, the density of events activating task τ_0 (ES_{in}) would differ from the density of events that activate τ_1 ($ES_{A'}$) and τ_2 ($ES_{B'}$). In order to maintain analyzability, buffers are introduced that are to restore the

*This research has been supported in part by the Deutsche Forschungsgemeinschaft under grant SL 47/1-1, SL 47/2-1

1-4244-0777-X/06/$20.00 ©2006 IEEE

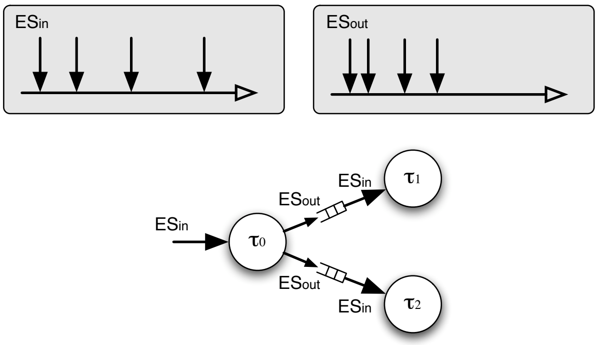


Figure 2. Task model for event streams

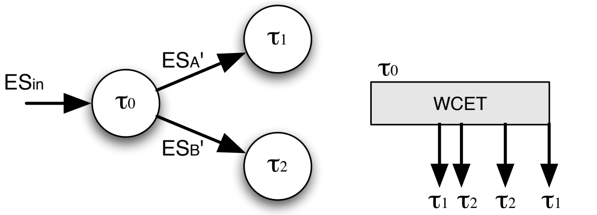


Figure 3. Event dependency graph

original event stream ES_{in} [4]. These buffers make it unnecessary to calculate the modified event stream, however additional hardware is required to make the analysis feasible.

Synchronous data flow graphs [5] are a model for signal processing real time task systems which are a simplification of Petri Nets to make their analysis feasible. However they are restricted to periodic events. Processing Graphs [6] describe the occurrence of events based on the rate-based execution theory. In rate-based execution a time interval is given for which the average number of events occurring is known. As there is no upper bound on the events that can actually occur in a given interval, the deadline has to be adjusted and only the feasibility of the system in regard to these adjusted deadlines is considered.

The SPI (*System Property Intervals*) Workbench [7] is a methodology to describe the characteristics of a heterogeneous system. However realistic systems lead to highly complex models that are hard to analyze.

3. Model

In this section the model used by this approach is described. The event stream model is used to describe the timing of events. Task graphs are used to describe the concurrent tasks of the system while the tasks themselves are specified as flow graphs. Finally the timing information yielded by this approach will be added to the task graph which produces the event dependency graph.

3.1. Event Stream Model

The event stream model is introduced in [4]. The key question this approach answers is: How many events can at most occur within any time span of a given length? This allows an abstraction from the absolute time towards a description in arbitrary time intervals. This provides a powerful way to describe the timing of events and at the same time allows an efficient analysis of the real time properties of a system.

An event stream is a set of event elements $\binom{p}{a}$. Each element represents an additional event that can occur when the observed interval has a size equal to or larger than a. The period p allows the event to be repeated every p time units.

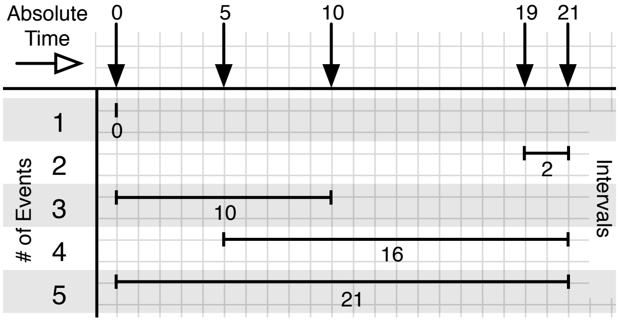


Figure 4. Event stream model

Consider Figure 4 as an example. Only one event occurs concurrently so there is a maximum of one event in any interval with a size towards zero. As the interval grows, the number of events that can occur within it increases. Two events can occur in an interval sized two, etc. So the depicted sequence of events adheres to the event stream

$$\left\{ \binom{p}{0}, \binom{p}{2}, \binom{p}{10}, \binom{p}{16}, \binom{p}{21} \right\}$$

The period p indicates that a set of events following the same constraints may occur every p time units. Note that a sequence of events that matches this event stream could be more relaxed: the event at time 10 could have occurred at 11. If it would occur at 12 however, the three events at 12, 19 and 21 lie within an interval of nine time units, but only two events would be allowed in an interval that size.

The maximum number of events that can occur in a given time interval I can be calculated by the event function (n is the number of event elements describing the event stream):

$$E(I) = \sum_{i=1}^{n} \left\lfloor \frac{I - a_i}{p_i} + 1 \right\rfloor$$

By scaling the event function by the maximum computation time c each event causes, the amount of requested computation time for intervals can be determined. This is known as the request bound function. By reducing the size of the given interval by the deadline d the computation time of those events that have to be completely processed during the interval is retrieved. So the demand bound function $C(I)$ tells how much demand for computation time is at most possible for any interval [8].

$$C(I) = E(I-d) \cdot c$$

These functions are sufficient for the analysis of static and dynamic priority scheduling using the methods proposed by Baruah [3]. For the dynamic priority case it is sufficient to test whether the required computation time is lower than the available computation time for all intervals up to a sufficient bound. So the overall schedulability can be verified. Through the reduction of the number of required test intervals the run time complexity of the real time analysis is $O(n \cdot \log n \cdot \frac{1}{\varepsilon})$ where n is the number of event elements and ε is an adjustable error [9].

Hence the accuracy of real time analysis can greatly be increased by considering the event stream of events occurring within a task system.

3.2. Task Graph

The nodes of a task graph are individual tasks of an application. An edge from one task to another indicates that the first task may activate the second by sending an internal event. An activated task may itself create events for other tasks as a result. The events triggering the task currently observed will be called external events. Figure 5 shows a task graph consisting of the tasks τ_0, τ_1 and τ_2 where τ_1 is triggered by τ_0 and τ_2 is triggered by τ_1.

3.3. Flow Graph

In order to get a deeper understanding of the behavior of an individual task, their flow graph as used in compiler theory [10] can be observed. A flow graph describes the sequential behavior of a task. Figure 5 shows the flow graph of τ_1. The *GNU Compiler Collection (GCC)* has been used to provide the necessary flow graphs in our implementation of the method introduced in this paper. *ChronEst* [11] was used to determine the execution times of the individual nodes of the graphs.

The nodes V of a flow graph are basic blocks. They are connected by edges E that represent a possible execution path. $pred(k)$ and $succ(k)$ are the sets of successors and predecessors of a node k. For every basic block the following additional properties must be known:

$event_\tau(k)$ is true iff node k generates an event for task τ. Basic blocks that trigger events can be represented as block arrows in the style of the output symbol of the SDL (*Specification and Description Language*) specification [12]. In Figure 5 nodes 0 and 4 of τ_1 activate τ_2.

$time(k)$ is the minimum execution time of node k. As the maximum number of events in a given interval is of interest it is the minimum execution time that is used as the basis for all following calculations.

Events are generated after the basic block has been processed. If need be this can easily be adjusted by splitting the node and letting the first trigger the event and the second consume the remaining time.

Any flow graph can be constructed by using two operations, *concatenate* and *merge*. A concatenation of two graphs represents the consecutive execution of the two graphs. A merge represents the conditional execution of basic blocks. The following formal definitions are specifically modeled to partition the flow graph in a way to match the requirements of the timing analysis functions that will be introduced in section 4.1.

Definition 1 (Concatenate *cat***)** *The concatenation of* (V_1, E_1) *and* (V_2, E_2) *is constructed by introducing an edge for every node of the first graph that has no successor (end) to all nodes of the second graph that have no predecessor (start).*

$$\begin{aligned} cat((V_1,E_1),(V_2,E_2)) &= (V_1 \cup V_2, E_1 \cup E_2 \cup \\ &\quad \{(k,k')|k \in end(V_1,E_1), \\ &\quad k' \in start(V_2,E_2)\}) \\ cat((V_1,E_1),k_1) &= cat((V_1,E_1),(k_1,\emptyset)) \end{aligned}$$

Definition 2 (Merge *mrg***)** *A merge is the union of all nodes and edges of two graphs. The two graphs may have common nodes, but nodes only present in one graph may not have a successor in the other graph. Therefore a merge is performed to produce a forking into alternative paths of execution that may have a common beginning.*

$$\begin{aligned} mrg((V_1,E_1),(V_2,E_2)) &= \{V_1 \cup V_2, E_1 \cup E_2\} \textit{ where} \\ &\quad succ(k \in V_1 \setminus V_2) \in V_1 \setminus V_2, \\ &\quad succ(k \in V_2 \setminus V_1) \in V_2 \setminus V_1 \end{aligned}$$

As an example the flow graph shown in Figure 5 could be constructed as

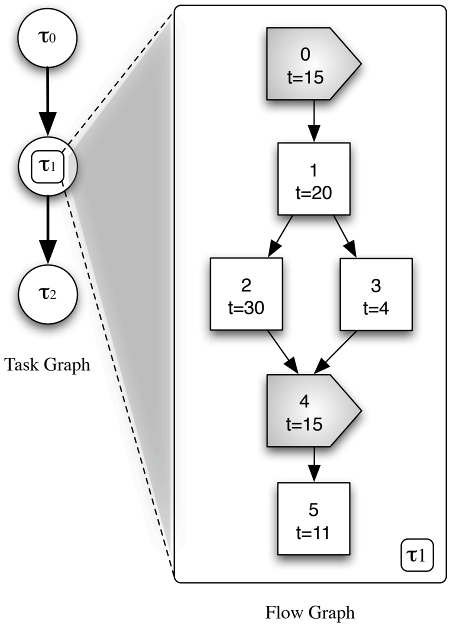
$$\begin{aligned} G_1 &= cat(cat(\emptyset,0),1) \\ G_2 &= mrg(cat(G_1,2),cat(G_1,3)) \\ G &= cat(cat(G_2,4),5) \end{aligned}$$

3.4. Event Dependency Graph

Event Dependency Graphs are based on task graphs. The edges of the graph are additionally weighted by the event stream flowing from one task to another. These event streams consist of events sent from the environment and also of events generated by the tasks themselves.

4. Retrieving the Event Stream

In order to retrieve the event stream of tasks their flow graph must be traversed. During this process the timing analysis functions introduced in the next section are used to collect the required timing information. The functions are defined recursively and determine the change in the



Node	Instruction
	procedure foo(int x);
	begin
0	send$_{\tau_2}$ (x);
1	**if** (*condition*) **then**
2	x := (*slow operation*)
	else
3	x := (*quick operation*)
	end if ;
4	send$_{\tau_2}$ (x);
5	doSomething() ;
	end ;

Figure 5. Task graph and flow graph

timing behavior of a flow graph as it is extended by attaching further nodes or graphs to it. Next an algorithm is given that will visit the nodes of the flow graph in the order required for this method. After these steps the event stream that is generated by a task if it is triggered just once is known.

4.1. Timing Analysis Functions

The following timing analysis functions are defined for subgraphs of a flow graph generated by the *cat* and *mrg* operation. They are recursively defined to decompose the graph into further subgraphs.

4.2. Maximum Number of Events Generated by a Task

$maxE(G)$ is the maximum number of events that can be generated by a task represented by its flow graph G during a single activation.

Definition 3 (maxE)

$$maxE(\emptyset) = 0$$

$$maxE(cat(G,k)) = \begin{cases} \textit{if } event_\tau(k) = \textit{true:} \\ \quad maxE(G)+1 \\ \textit{else:} \\ \quad maxE(G) \end{cases}$$

$$maxE(cat(G,H)) = maxE(G) + maxE(H)$$
$$maxE(mrg(G,H)) = \max(maxE(G), maxE(H))$$

The values of *maxE* for any flow graph can be computed by systematically rebuilding the graph and in each step use the function defined for the performed composition. The timing analysis functions introduced in the next sections are used in the same way.

The flow graph shown in Figure 5 can be constructed while calculating the values for *maxE* as follows:

$$\begin{aligned}
&G_1 = cat(\emptyset, 0)\\
&\quad maxE(G_1) = maxE(\emptyset) + 1 = 1\\
&G_2 = cat(G_1, 1)\\
&\quad maxE(G_2) = maxE(G_1) = 1\\
&G_3 = cat(G_2, 2)\\
&\quad maxE(G_3) = maxE(G_2) = 1\\
&G_4 = cat(G_2, 3)\\
&\quad maxE(G_4) = maxE(G_3) = 1\\
&G_5 = mrg(G_3, G_4)\\
&\quad maxE(G_5) = \max(maxE(G_3), maxE(G_4)) = 1\\
&G_6 = cat(G_5, 4)\\
&\quad maxE(G_6) = maxE(G_5) + 1 = 2\\
&\ G = cat(G_6, 5)\\
&\quad maxE(G) \ = maxE(G_6) = 2
\end{aligned}$$

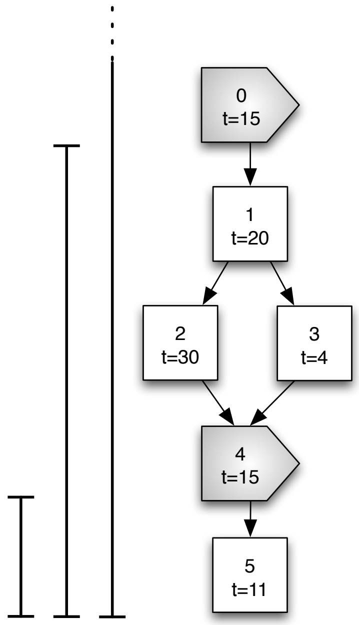
4.3. Intervals from Start to End of Graph

$totalI_n$ is the minimum amount of time the task needs from start to end while generating exactly n events. If the requested number of events can not be generated, the time is infinity, even if a higher or lower number of events is possible.

Definition 4 (totalI)

$$
\begin{aligned}
totalI_0(\emptyset) &= 0\\
totalI_{n>0}(\emptyset) &= \infty\\
totalI_0(cat(G,k)) &= \begin{cases} \textit{if } event_\tau(k) = true\textit{:}\\ \quad \infty\\ \textit{else:}\\ \quad totalI_0(G) + time(k) \end{cases}\\
totalI_{n>0}(cat(G,k)) &= \begin{cases} \textit{if } event_\tau(k) = true\textit{:}\\ \quad totalI_{n-1}(G) + time(k)\\ \textit{else:}\\ totalI_n(G) + time(k) \end{cases}\\
totalI_n(cat(G,H)) &= \min_{x\in[0,n]} (totalI_x(G) + totalI_{n-x}(H))\\
totalI_n(mrg(G,H)) &= \min(totalI_n(G), totalI_n(H))
\end{aligned}
$$

4.4. Intervals Relative to End of Graph



# of Events	1	2	3
Time	11	50	∞

Figure 6. Intervals relative to end of graph

$endI_n$ is the minimum interval from the end of the flow graph back into the graph so that n events are generated. As events are assumed to occur at the end of the nodes the minimum execution time of the first node is not added to the interval. Figure 6 illustrates the *endI* values for the given graph. One event can be triggered 11 time units before the end; 50 time units are needed to reach two events. As three events can not be generated, the time required is considered infinite.

Definition 5 (endI)

$$
\begin{aligned}
endI_0(G) &= 0\\
endI_{n>0}(\emptyset) &= \infty\\
endI_1(cat(G,k)) &= \begin{cases} \textit{if } event_\tau(k) = true\textit{:}\\ \quad 0\\ \textit{else:}\\ \quad endI_1(G) + time(k) \end{cases}\\
endI_{n>1}(cat(G,k)) &= \begin{cases} \textit{if } event_\tau(k) = true\textit{:}\\ \quad endI_{n-1}(G) + time(k)\\ \textit{else:}\\ \quad endI_n(G) + time(k) \end{cases}\\
endI_n(cat(G,H)) &= \min \begin{cases} endI_n(H),\\ \min\limits_{x\in[1,n]} (endI_x(G)\\ \qquad\qquad + totalI_{n-x}(H)) \end{cases}\\
endI_n(mrg(G,H)) &= \min(endI_n(G), endI_n(H))
\end{aligned}
$$

4.5. Intervals Relative to Begin of Graph

$startI_n$ is complementary to $endI_n$. It is the minimum interval from the start of the flow graph in which n events are generated.

Definition 6 (startI)

$$
\begin{aligned}
startI_0(G) &= 0\\
startI_{n>0}(\emptyset) &= \infty\\
startI_n(cat(k,G)) &= \begin{cases} \textit{if } event_\tau(k) = true\textit{:}\\ \quad time(k) + startI_{n-1}(G)\\ \textit{else:}\\ \quad time(k) + startI_n(G) \end{cases}\\
startI_n(cat(G,H)) &= \min \begin{cases} startI_n(G),\\ \min\limits_{x\in[0,n-1]} (totalI_x(G)\\ \qquad\qquad + startI_{n-x}(H)) \end{cases}\\
startI_n(mrg(G,H)) &= \min(startI_n(G), startI_n(H))
\end{aligned}
$$

4.6. Intervals Within the Graph

$inI_n(G)$ is the minimum interval within the flowgraph, in which n events can be triggered.

$inI_n(cat(G,H))$ finds a distribution of events on $endI(G)$ and $startI(H)$ so that the resulting interval $endI(G) + startI(H)$ is minimal. If however there is already a smaller interval for the requested number of events in G or H it remains the interval for $inI(cat(G,H))$.

```
procedure calculateIntervals(FlowGraph G);
begin
  Node k := start(G);
  k.preG := $\emptyset$;
  push(k);
  while (stack not empty) do
    Node k := pop();
    if (allPredVisited(k)) then
      if (predCount(k) $\leq$ 1) then
        analyze k.preG := cat (pred(k).preG, k);
      else if (predCount (k)) > 1) then
        analyze k.preG := cat (mrg (pred(k).preG), k);
      end if ;
      markVisited(k);
      for each k' $\in$ succ(k) do
        push (succ(k'));
      end for ;
    end if ;
  end while ;
  assert (|end(G)| > 1 or k.preG = G);
end ;
```

Figure 7. Algorithm: traversal of a flow graph

Definition 7 (inI)

$$
\begin{aligned}
inI_n(cat(G,k)) &= \min\left(inI_n(G), endI_n(cat(G,k))\right)\\
inI_{n\leq 1}(cat(G,H)) &= \min\left(inI_n(G), inI_n(H)\right)\\
inI_{n>1}(cat(G,H)) &= \min\begin{cases} inI_n(G),\\ inI_n(H),\\ \min\limits_{x\in[1,n-1]}(endI_x(G)\\ \qquad\qquad + startI_{n-x}(H))\end{cases}\\
inI_n(mrg(G,H)) &= \min(inI_n(G), inI_n(H))
\end{aligned}
$$

The event stream caused by a single execution of the task can now easily be formulated:

$$ES(G) = \bigcup_{n\in[1,maxE(G)]} \begin{pmatrix} \infty \\ inI_n(G) \end{pmatrix}$$

4.7. Flow Graph Traversal

To analyze a flow graph the operations *cat* and *mrg* can be used beginning with the start node to match the structures in the graph. A simple algorithm to traverse the flow graph is given in Figure 7. For the sake of simplicity it is assumed that loops in the graph have been unrolled. If the flow graph has no single end node, a virtual end node has to be introduced and all end nodes must be connected to that node.

The algorithm begins with the start node of the flow graph. If all predecessors of the current node have been analyzed ($allPredVisited(k)$) the node itself is analyzed. If it has a single predecessor the *cat* operation is used to merge the current node to the preceding graph. If it has more than one predecessor the node represents the end of an alternative. The alternatives are merged using the *mrg* operation and then concatenated to the current node using the *cat* operation. In the algorithm every node has a property *preG*, which represents the graph preceding the node.

When the algorithm is used in the context of an actual analysis, a copy of the graph does not actually have to be constructed. Instead the *cat* and *mrg* operations are only needed to select the matching timing analysis functions. The results of each function can be stored in a linear table that maps numbers of events to times. Only these tables have to be updated with every iteration of the algorithm.

5. Introducing External Events

Up to now the consequences of a single task activation have been considered. The intervals that can be calculated so far can now be used to retrieve the resulting event stream if the task is triggered not by a single event but itself by an event stream.

The interval function $a(n)$ is the minimum interval in which n events can occur.

$$a(n) = \min\{I \in T | n = E(I)\}$$

In the following we assume the deadline d to be smaller than $a(1)$ of the external event stream. This means a new external event can only occur after the task has completed. Therefore the deadlines are disjoint as shown in Figure 8. As a consequence the execution times of the inner activations of the task ($a(2)$ and $a(3)$ in Figure 8) become irrelevant in regard to the overall interval. Therefore the maximum number of events *maxE* may be triggered during each of these executions. Only the remaining events must be distributed among the first and last execution in a way to find the minimal overall interval.

$innerE^i$ represents the maximum number of events that can be generated when i external events are taken into account minus the maximum number of events generated by the first and last external event.

Definition 8 (innerE)

$$innerE^i = \max(0, (i-2)\cdot maxE)$$

The first and last event of an interval determine its overall size. *marginSkew* is the amount of time that has to be

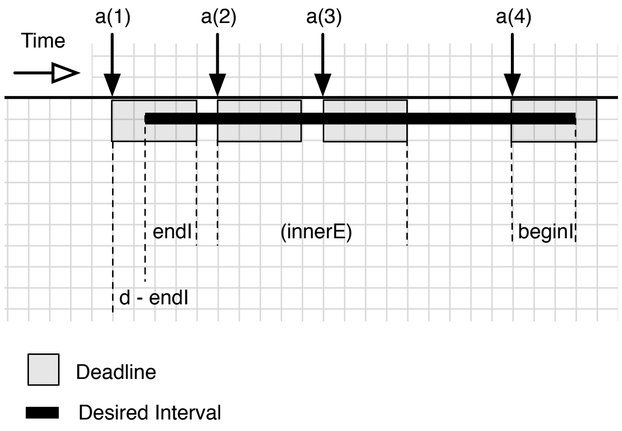


Figure 8. Interval over multiple external events

added to the last external event $a(i)$ in order to get the overall interval if the two outer invocations must generate n events. For this $a(i)$ has to be enlarged by $startI_x$ and reduced by $d - endI_{n-x}$. This may cause $marginSkew$ to become negative.

Definition 9 (marginSkew)

$$marginSkew_n = \min_{x \in [1,n-1]} (startI_x - (d - endI_{n-x})$$

inI_n^i represents the minimum interval in which n events are generated while considering i external events. All external events of a cycle taken into account this equals the desired event stream.

Definition 10 (inI for Multiple External Events)

$$\begin{aligned} inI_n^1 &= inI_n \\ inI_n^{i>1} &= a(i) + marginSkew_{n-innerE^i} \end{aligned}$$

5.1. Example: External Events

As an example the system described in Figure 5 will be used. τ_1 must be executed within the deadline $d = 90$. τ_0 activates τ_1 with the event stream

$$\left\{ \binom{350}{0}, \binom{350}{100}, \binom{350}{220} \right\}$$

In order to determine the interval in which two events can occur, inI_2^2 must be calculated:

$$\begin{aligned} inI_2^2 &= a(2) + marginSkew_2 \\ &= a(2) + startI_1 - (d - endI_1) \\ &= 100 + 15 - (90 - 11) \\ &= 36 \end{aligned}$$

Note that the interval inI_2^2 is 36 while the interval inI_2 of τ_1 alone is $time(1) + time(3) + time(4) = 39$. This worst case takes place when node 4 is executed as late as possible (11 time units before the end of the deadline at 90) while node 0 is executed as soon as possible at the next execution: 15 time units after a second start, which can be 100 time units after the first activation of τ_1.

The intervals for all other events can be determined accordingly. The resulting event stream generated by τ_1 for τ_2 is

$$\left\{ \binom{350}{0}, \binom{350}{36}, \binom{350}{75}, \binom{350}{114}, \binom{350}{195}, \binom{350}{234} \right\}$$

Figure 9 shows the event dependency graph for this example.

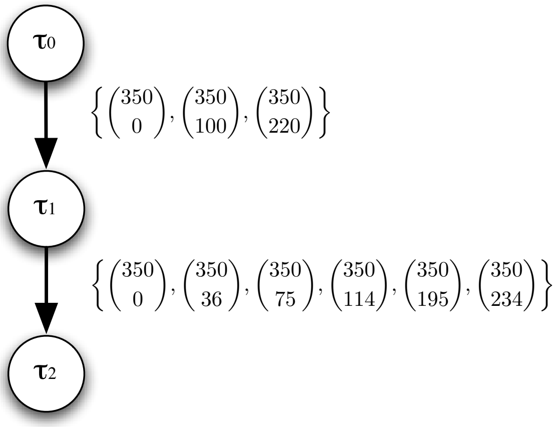

τ0
$\left\{ \binom{350}{0}, \binom{350}{100}, \binom{350}{220} \right\}$
τ1
$\left\{ \binom{350}{0}, \binom{350}{36}, \binom{350}{75}, \binom{350}{114}, \binom{350}{195}, \binom{350}{234} \right\}$
τ2

Figure 9. Event dependency graph

6. Run Time Complexity

The algorithm introduced in Figure 7 must visit every node once. For the analysis of every node the timing information of every event that may have occurred up to that point must be adjusted. So the complexity is $O(nodes \cdot events)$. Calculating the resulting event stream when multiple external events are taken into account takes a second step which has $O(ext \cdot int^2)$ where ext is the number of external events and int is the maximum number of events generated by the task within one activation.

For efficient loop unrolling the loop bodies can be analyzed as an isolated flow graph. If no events are triggered in the loop this makes the computation time of the analysis independent of the number of basic blocks or iterations of that loop. As the combinatorial aspects of branches are removed by the mrg operation, the conditional statements do not have a negative impact on the run-time.

7. Example: Real Time Analysis using Event Dependency Graphs

As an example the flow graph of the algorithm in Figure 7 itself can be used as the task that is triggered by the event stream

$$\left\{ \begin{pmatrix} 3400 \\ 0 \end{pmatrix}, \begin{pmatrix} 3400 \\ 700 \end{pmatrix}, \begin{pmatrix} 3400 \\ 1500 \end{pmatrix}, \begin{pmatrix} 3400 \\ 2200 \end{pmatrix} \right\}$$

The task is assigned a deadline of 640. Calls to the functions *push* and *pop* have been chosen to generate events for another task on a different computation unit and which has a computation time of 45 with a deadline of 700. The loops have been unrolled with an upper bound of five passes for the *while* loop and two passes for the for loop. This results in a maximum of 16 events per activation or a computation time of $4 * 16 * 45 = 2880$ for the period (p=3400).

Figure 10 shows the resulting demand bound function for the task triggered by the *push* and *pop* calls. The dashed line indicates that in an interval of size I the amount of computation time on an idle component is I. If the demand for computation time within an interval is smaller than the available computation time, the system is schedulable [4].

In the periodic approach the event stream created by the task is not known. Instead all events have to be assumed to occur periodically at the end of the task; as the events are assumed to occur at the same time, a great number of events has to be processed within the same deadline. If however more information is available about the spread of events over the execution time of the task, fewer events can be predicted to occur simultaneously and the component is known to be able to process events at an earlier stage. The example shows that our approach would identify the system as schedulable while the traditional approach suggests that a more powerful processor or a more relaxed deadline is required.

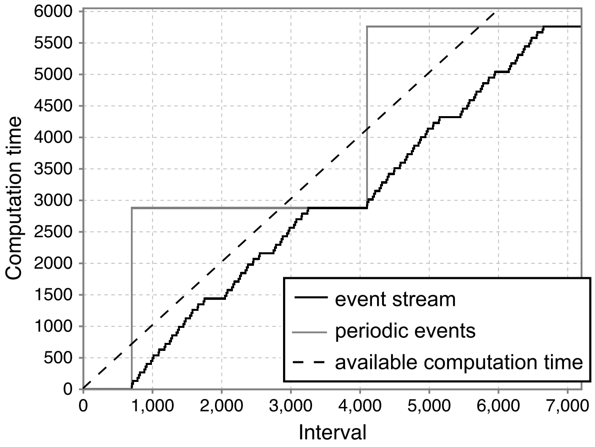


Figure 10. Demand bound function

8. Conclusions

We have proposed a methodology to extract the worst case temporal distribution of events within a task system using the information of the tasks individual control flow graphs. We have shown how to use these results for a more accurate real time analysis for static or dynamic priority scheduling, allowing an optimized estimate on the required hardware. The acceptable run time complexity, the integration of state of the art real time analysis techniques and the use of readily available data structures such as flow graphs are favorable properties of this approach.

References

[1] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, Januar 1973.

[2] A. K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*, PhD thesis, Massachusetts Institute of Technology, 1983.

[3] S. K. Baruah, “Dynamic- and Static-priority Scheduling of Recurring Real-Time Tasks”, *Real-Time Systems*, vol. 24, pp. 93–128, 2003.

[4] K. Gresser, “An Event Model for Deadline Verification of Hard Real-Time Systems”, in *5th Euromicro Workshop on Real-Time Systems*, 1993, Finland.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, volume 360 of *The Kluwer International Series in Engineering and Computer Science*, Kluwer Academic Press, 1996.

[6] S. Goddard, *On the Management of Latency in the Synthesis of Real-Time Signal Processing Systems from Processing Graphs*, PhD thesis, University of North Carolina, 1998.

[7] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich, “Hardware/Software Codesign of Embedded Systems - The SPI Workbench”, in *Proceedings of the IEEE Computer Society Workshop on VLSI’99*, 1999.

[8] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok, “Generalized Multiframe Tasks”, *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, 1999.

[9] K. Albers and F. Slomka, “An Event Stream Driven Approximation for the Analysis of Real-Time Systems”, in *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, July 2004. IEEE.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, Addison–Wesley, second edition, 1986.

[11] F. Slomka, “New Techniques for the Design of Distributed Embedded Real-Time Systems”, in *Proceedings of the Embedded World Conference*, February 2005, Nuremberg, Germany.

[12] Z.100, “Specification and description language (SDL)”, November 1999, ITU-T Recommendation Z.100.