MRTC
Report

MÄLARDALENS HÖGSKOLA

Response-Time Analysis for Dynamically and Statically Scheduled Systems

Mikael Sjödin
mikael.sjodin@mdh.se

April 2002

MRTC Report no. 55

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

www.mrtc.mdh.se

ISSN 1404-3041 ISRN MDH-MRTC--55--SE

1 Introduction

This paper describes how to perform response-time analysis on a set of tasks scheduled by a fixed priority scheduler that runs "in the background" of a static cyclic schedule. The system model contains:

- Interrupts. There may be multiple interrupt levels, so an interrupt may be interrupted by a higher level interrupt.
- A static cyclic schedule. The schedule has a major cycle and is divided into minor cycles. At the start of each minor cycle a set of functions are scheduled for execution. These functions all execute to completion and are all executed "back-to-back" as a single block of execution.

  Interrupts may preempt the execution of the static scheduled functions.
- A set of tasks that are dynamically dispatched and executed by a fixed priority scheduler in the time slots available between interrupts and minor cycle function-blocks.

Traditionally, in this kind of system the dynamic tasks are assigned to non time-critical functions. The time-critical functions are all allocated in the static schedule. The reason for this partitioning has been that no method to calculate the response-time for dynamically dispatched tasks has existed.

This paper presents a method to calculate worst-case response-times for the dynamically dispatched tasks.

2 Recapitulating Response-Time Analysis

Lets begin by revising classical response-time analysis methods.

2.1 Task Model

In the classical response-time analysis (see e.g. [BW96, ABD$^+$95])we assume a fixed priority scheduler (i.e. a dynamic scheduler that always executes the highest priority eligible task).

Each task i is assumed to be a periodic task with the following attributes:

This work has been supported by Volvo Technical Development within the EAST/EEA project.

C_i The Worst-Case Execution-Time (WCET) of the task.

T_i The period of the task.

B_i The maximum blocking time (i.e. the maximum time to wait for a lower priority task that has locked a resource).

J_i The maximum jitter (i.e. max deviation for the ideal periodicity).

D_i The deadline of the task.

Additional attributes for tasks that will be used in this paper are:

P_i The priority of the task. Priorities can be assigned with any method (e.g. rate monotonic or deadline monotonic). If a task i has higher priority than a task j, then $P_i > P_j$.

R_i The worst-case response-time (as derived by the response-time analysis).

For this paper it assumed that:

- $C_i > 0$, $T_i > 0$, $D_i > 0$.
- $B_i \geq 0$, $J_i \geq 0$.
- $P_i \neq P_j$ if $i \neq j$ (i.e. unique task priorities).
- $D_i < T_i - J_i$ (i.e. deadline less than period).

The two last assumption above could be removed using elsewhere published techniques (see e.g. [Tin94, AKA94, ABD$^+$95]. The techniques presented later in this paper could be applied also without these assumptions. However, since these assumptions significantly simplifies the response-time equations we will keep them throughout this paper.

2.2 Response-Time Equations

For the model above, the formula for calculating the response-time is:

$$R_i = B_i + C_i + \sum_{j \in \{x : P_x > P_i\}} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{1}$$

Since R_i cannot be isolated on one side of the equality the following iterative solution method is used:

$$R_i^{n+1} = B_i + C_i + \sum_{j \in \{x : P_x > P_i\}} \left\lceil \frac{R_i^n + J_j}{T_j} \right\rceil C_j \qquad (2)$$

where $R_i^0 = 0$ and $R_i = R_i^n$ when $R_i^n = R_i^{n+1}$.

A system is deemed schedulable if $\forall i : R_i \leq D_i$.

3 Modeling Static Cyclic Schedules

3.1 An Example Schedule

Lets consider an example static schedule with 4 functions A, B, C and D. The schedule has a major cycle of length 72 ms and a minor cycle of length 6 ms. The functions have the following characteristics:

Function	**WCET**	**Period**
A	1	6
B	2	18
C	1	12
D	1	24

Note that by necessity all period times are multiple of the minor cycle (6) and a divisor of the major cycle (72).

A static cyclic schedule for these functions is shown below:

```
ABBCD A     AC    ABB   ACD   A     ABBC  A     ACD   ABB   AC    A
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
0     6     12    18    24    30    36    42    48    54    60    66    72
```

In the major cycle/minor cycle paradigm the functions A, B, C and D would all be executed back-to-back a time 0. This means that if A only executes for 0.5 ms, then B would be dispatched at time 0.5 (not at time 1 as indicated in the schedule above). We call each set of functions dispatched in the beginning of a minor cycle a *function chain*. This method of using major and minor cycles is commonly used in practice. For instance the Rubus [Arc02] operating system implements this type of static cyclic scheduling.

In this model interrupts are assumed to preempt the execution of a function chain, thus delaying its execution. Thus, the above schedule is only valid if there only can be a total of 1 ms of interrupt execution during any 6 ms of time (otherwise the function chain for the first minor cycle would risk not to complete during the first minor cycle).

3.2 Combining Static Schedules and Dynamic Tasks

When a function chain is completed, the remaining time of the minor cycle can be spent executing any eligible dynamically dispatched tasks. For instance, the Rubus operating system [Arc02] supports this type of mixed static and dynamic scheduling.

In the example above, in the first minor cycle there is at least 1 ms of time available for executing dynamic tasks (assuming that no interrupts delayed the execution of the function chain). However, if some of the functions execute faster than their WCET then more than 1 ms may be left over for the dynamic tasks to execute in.

Thus, we can make the observation that the static schedule behaves like a single high priority task (from the dynamic tasks point of view). So, how do we model this in the response time equations?

First, we recognize that from the dynamic tasks point of view the actual functions executed in function chain is irrelevant. From the dynamic tasks point of view the schedule would look like this:

```
XXXXX X     XX    XXX   XXX   X     XXXX  X     XXX   XXX   XX    X
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
0     6     12    18    24    30    36    42    48    54    60    66    72
```

A naive way to model the interference caused by the static schedule would be to make it a high priority task with attributes $C_i = 5$ (where 5 is the maximum time allocated for any function chain in the schedule) and $T_i = 6$ (where 6 is the minor cycle time). This would, however, be overly pessimistic, and in essence it would allocate 5/6 (≈83%) of the CPU to the static schedule. In this example we can however see that only 29/72 (≈40%) of the CPU is allocated to the static schedule.

3.3 A New Task Model

To remedy this problem we extend the model of section ( )2.1. We will use a technique that is similar to the technique used in [SH99]. Lets change the attribute C_i to denote a *vector* of execution times, such that:

$C_i = [C_i[0], C_i[1], \ldots, C_i[\overline{C}_i - 1],]$ where each $C_i[k]$ an execution time.

$\overline{C}_i$ is the number of elements in C_i.

The elements of C_i denotes a cyclic pattern of execution times. Where each execution time would be the execution time of a function chain. In our example we would get:

- $C_i = [5, 1, 2, 3, 3, 1, 4, 1, 3, 3, 2, 1]$
- $\overline{C}_i = 12$
- $T_i = 6$

A task that always have the same WCET would have $\overline{C}_i = 1$ (that is, if $\overline{C}_i = 1$ then this new model is equivalent to the classical model).

4 A New Response-Time Analysis

Now, when we have a task model that accurately captures the behavior of a static schedule (with respect to its interference on dynamic tasks), how do we calculate the response times for our new task model?

Lets begin with *rephrasing* equation 1 on page 2:

$$\begin{aligned} R_i &= B_i + C_i + \sum_{j \in \{x : P_x > P_i\}} execution\_demand(j, R_i) \\ execution\_demand(j,t) &= occurrences(j,t) * C_j \\ occurrences(j,t) &= \left\lceil \frac{t + J_j}{T_j} \right\rceil \end{aligned} \tag{3}$$

Where *execution_demand*(j,t) denotes the maximum execution demand task j can generate in an interval t, and *occurrences*(j,t) denotes the maximum number of time a task j can arrive during a time t.

Next, we introduce the infinite array $\hat{C}_i[k]$, where $\hat{C}_i[k]$ is the maximum total execution time of k successive invocations of task i. The formal definition is:

$$\hat{C}_i[k] = \begin{cases} 0 & \text{if } k = 0 \\ \max\limits_{t \in 0 \ldots \overline{C}_i - 1} \sum\limits_{l=0}^{k-1} C_i[(t+l) \mod \overline{C}_i] & \text{if } k > 0 \end{cases} \tag{4}$$

Note specifically that $\hat{C}_i[1]$ is the maximum of the execution times in C_i, and $\hat{C}_i[\overline{C}_i]$ is the sum of execution times in C_i. Also, if $\overline{C}_i = 1$ then $\hat{C}_i[k] = C_i[0] * k$. For our example above, we would get $\hat{C}_i = [0, 5, 6, 8, 11, 14, 15, \ldots]$ (which happens to coincide with summing up the elements from the first position in C_i).

This leads us to the final formulation of the new response-time analysis. Using $\hat{C}_i$ we can now change *execution_demand*(j,t) and equation ( )3 to:

$$\begin{aligned} R_i &= B_i + \hat{C}_i[1] + \sum_{j \in \{x : P_x > P_i\}} execution\_demand(j, R_i) \\ execution\_demand(j,t) &= \hat{C}_j[\text{occurrences}(j, R_i)] \\ occurrences(j,t) &= \left\lceil \frac{t + J_j}{T_j} \right\rceil \end{aligned} \tag{5}$$

4.1 Implementational Aspects

For efficient implementation fast access to the elements of the array $\hat{C}_i$ is crucial.

An efficient implementation of equation 4 on the page before (with complexity $O(1)$ in contrast with the naive solution which has complexity $O(\overline{C}_i k)$) can be made by pre-computing and storing the first $\overline{C}_i + 1$ elements of $\hat{C}_i$. The pre-computed array, denoted $C^{\mathrm{pre}}{}_i$, is defined as:

$$C^{\mathrm{pre}}{}_i[k] = \hat{C}_i[k] \qquad \forall k \in [0 \ldots \overline{C}_i] \tag{6}$$

$C^{\mathrm{pre}}{}_i$ is computed in $O(\overline{C}_i{}^3)$ time and stored in $O(\overline{C}_i)$ space. Since $\overline{C}$ is expected to be small this overhead can be considered negligible.

Lets call a sequence of $\overline{C}_i$ task executions a *full execution*. A full execution will contain one execution of each instance in C_i, i.e., a full execution will take at most $\sum_k C_i[k]$ time. As pointed out above, $\sum_k C_i[k] = \hat{C}_i[\overline{C}_i]$ (which is stored in $C^{\mathrm{pre}}{}_i[\overline{C}_i]$). Now we can implement equation 4 on the preceding page as:

$$\begin{aligned}
&\hat{C}_i[k] = \mathit{no\_of\_full\_executions} * C^{\mathrm{pre}}{}_i[\overline{C}] + C^{\mathrm{pre}}{}_i[\mathit{no\_of\_remaining\_executions}] \\
&\mathit{no\_of\_full\_executions} = k \text{ div } \overline{C} \\
&\mathit{no\_of\_remaining\_executions} = k \text{ rem } \overline{C}
\end{aligned} \tag{7}$$

5 Putting It All Together

So, now we have a way to calculate the response times for our new task model. How do we model a system with interrupts, a static schedule and a set of dynamic tasks? Well, everything is modeled as tasks in out new task model, as follows:

- Interrupts are assigned the highest priorities. Higher priorities to higher level interrupts. Each interrupt must have a minimum interarrival time and the interrupt handler must have a known WCET. Each interrupt is modeled as a task with the following attributes:

  - $C_i = [\langle\text{the known WCET}\rangle]$
  - $\overline{C} = 1$
  - $T_i = \langle\text{the known interarrival time}\rangle$
  - $J_i = 0$
  - $B_i = \langle\text{the longest time interrupts at this level are disabled}\rangle$
  - $D_i = \infty$ or $\langle\text{the interrupts deadline}\rangle$

- Next, the task for the static schedule is assigned a priority lower than the interrupts and the following attributes:
  - $C_i = [\langle\text{Array of WCETs for function chains}\rangle]$
  - $\overline{C} = \langle\text{number of minor cycles in schedule}\rangle$
  - $T_i = \langle\text{minor cycle time}\rangle$
  - $J_i = 0$
  - $B_i = 0$
  - $D_i = \infty$
- Finally, the dynamic tasks are assigned priorities that are lower than the task for the static schedule.

Now, equation 5 on page 6 can be used to calculate R_i for each task and if $\forall i : R_i \leq D_i$ then all dynamic tasks (and interrupts) will meet their deadlines.

6 Preemptive Static Schedules

In some static scheduled systems it is allowed to for functions chains to preempt each other. That is, a function chain A may not complete before the next function chain B is released. In this case, A is preempted by B and when B completes A is resumed. The Rubus [Arc02] operating system is an example of where such schedules can be allowed.

When using the response-time analysis in section ( )4 to model the interference caused by function chains to dynamic tasks this type of preemptive static schedule need no special treatment. For instance a preemptive schedule modeled as $C_i = [7, 1, 5, 1]$ and $T_i = 4$ is perfectly all right. One need to keep in mind though, when modeling such schedules, the response time calculated for the *static schedule task* is not a valid response time (whereas the response times calculated for all other tasks *will* be valid).

7 Future Work

In the formulation of equation 5 on page 6 the term $execution\_demand(j, t)$ gives us a powerful tool to express arbitrary complex execution patterns for tasks. For instance, in the Rubus operating system, no minor cycles are needed. Instead, function chains can be scheduled at arbitrary points in time during the major cycle. To model a static schedule with arbitrary release times for function chains the function $execution\_demand(j, t)$ can be changed to properly describe such a release pattern. (For instance, by using an array $T_i[]$ with release times instead of the plain T_i that expresses the period of the minor cycle.)

On the scheduling theory side, work could be performed to formally prove that the response-time analysis is correct for any monotonically increasing function $execution\_demand(j, t)$ (monotonically with respect to t). Also, it could be shown that other task models, like the sporadically periodic tasks model [Tin94], can be modeled by the $execution\_demand(j, t)$ function.

References

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[AKA94] A.Burns, K.Tindell, and A.J.Wellings. Fixed Priority Scheduling with Deadlines Prior to Completion. In *Proc. of the 6th Euromicro Workshop of Real-Time Systems*, pages 128–142, June 1994.

[Arc02] Arcticus Systems Home Page, 2002. http://www.arcticus.se.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[SH99] M. Sjödin and H. Hansson. Analysing Multimedia Traffic in Real-Time ATM Networks. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 203–212, June 1999.

[Tin94] K. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, February 1994. Available at ftp://ftp.cs.york.ac.uk/pub/realtime/papers/thesis/ken/.