Static Priority Scheduling of Event-Triggered Real-Time Embedded Systems

Cagkan Erbas
Dept. of Computer Science
University of Amsterdam
Amsterdam, The Netherlands
cagkan@science.uva.nl

Selin Cerav-Erbas
Inst. d'Admin. et de Gestion
Univ. Catholique de Louvain
Louvain-la-Neuve, Belgium

Andy D. Pimentel
Dept. of Computer Science
University of Amsterdam
Amsterdam, The Netherlands

Abstract

Real-time embedded systems are often specified as a collection of independent tasks, each generating a sequence of event-triggered code blocks, and the scheduling in this domain tries to find an execution order which satisfies all real-time constraints. Within the context of recurring real-time tasks, all previous work either allowed preemptions, or only considered dynamic scheduling, and generally had exponential complexity. However, for many embedded systems running on limited resources, preemptive scheduling may be very costly due to high context switching and memory overheads, and dynamic scheduling can be less desirable due to high CPU overhead. In this paper, we study static priority scheduling of recurring real-time tasks. We focus on the non-preemptive uniprocessor case and obtain schedule-theoretic results for this case. To this end, we derive a sufficient (albeit not necessary) condition for schedulability under static priority scheduling and show that this condition can be efficiently tested in practice. The latter is demonstrated with examples, where in each case, an optimal solution for a given problem specification is obtained within reasonable time, by first detecting good candidates using meta-heuristics, and then by testing them for schedulability.

1. Introduction

In general, real-time embedded systems are assumed to run infinitely on limited resources and the scheduling in this domain tries to address the problem of finding a set of rules to schedule independent tasks on these limited resources. There exists a trade-off between the generality of the task model (a measure of accuracy) and the analyzability of the system modeled. As a result of this, many task models have been proposed in the past which differ in terms of their expressive power and the complexity to analyze them. In general terms, a real-time system is a collection of independent tasks, each generating a sequence of subtasks associated with a ready-time, an execution requirement, and a deadline. Different task models may specify different constraints on these parameters. For example, the multiframe model [12] permits task cycling but ignores deadlines while the generalized multiframe [4] adds explicit deadlines to the multiframe model. Furthermore, the system may be composed of one or more processors and the task execution may be preemptive or non-preemptive. The schedulability analysis of such a real-time system is to identify whether it is possible to guarantee for each task a processor time equal to its execution requirement within the time duration between its ready-time and deadline.

For many embedded systems running on limited resources, preemptive scheduling may be very costly and the designers of such systems may prefer non-preemptive scheduling despite its relatively poor theoretical results. This is mainly due to the large runtime overhead incurred by the expensive context switching and the memory overhead due to the necessity of storing preempted task states. There is also a trade-off between static (tasks are given unique priorities offline) and dynamic scheduling (tasks are given priorities online) policies. While static scheduling has a very low CPU overhead, run-time scheduling may be necessary for better processor utilization.

Conditional real-time code. Embedded real-time processes are typically implemented as event-driven code blocks residing in an infinite loop. The first step in the schedulability analysis of such real-time code is to obtain an equivalent task model which reveals the control flow information. In the following conditional real-time code, execution requirement and deadline of subtasks v (representing code blocks) are shown with the parameters e and d, respectively. This means that whenever a subtask v is triggered by an external event, the code block corresponding to that subtask should be executed on the shared processing resources for e units of time within the next d units of time from its triggering time in order to satisfy its real-time constraints.

```
while (external_event)
    execute v1            /* with (e1, d1) */
    if (X) then           /* depends on system state */
        execute v2        /* with (e2, d2) */
    else
        execute v3        /* with (e3, d3) */
    end if
end while
```

The traditional analysis of such conditional codes, which depends on identifying the branch with the worst case behavior, does not work in this case. The branch with the worst case behavior depends on the system conditions that are external to the task. Consider the situation $(e_2 = 2, d_2 = 2)$ and $(e_3 = 4, d_3 = 5)$. If another subtask with $(e = 1, d = 1)$ is to be executed simultaneously, then the branch (e_2, d_2) is the worst case, whereas if the other subtask is with $(e = 2, d = 5)$, then the branch (e_3, d_3) corresponds to the worst case.

Previous results. There is a tremendous amount of work on scheduling even if we restrict ourselves to the uniprocessor case, history of which goes back at least to [10]. While some work in the real-time embedded systems domain tried to improve modeling accuracy, in one way or another generalizing the restrictions in [10] that has very desirable theoretical results, some other tried to answer schedule-theoretic questions arising in the generalized models. Most task models assumed event-triggered independent tasks. However, there are also heterogeneous models considering mixed time/event-triggered systems [14] and systems with data and control dependencies [13]. The recurring real-time task model [2], on which we focus in this study, is a generalization of the previously introduced models, such as the recurring branching [1], generalized multiframe [4], multiframe [12] and sporadic [11] models. It can be shown that any of these task models corresponds to a special instance of the recurring task model, which in turn implies that it supersedes all previous models in terms of its expressive power. With respect to dynamic scheduling, it has been proved for both preemptive [10] and non-preemptive uniprocessor cases [5] that Earliest Deadline First (EDF) scheduling (among the ready tasks, a task with an earlier deadline is given a higher priority online) is *optimal*. The latter means that if a task is schedulable by any scheduling algorithm, then it is also schedulable under EDF. Hence, the online scheduling problem on uniprocessors is completely solved, we can always schedule using EDF. On the other hand, analysis of static priority scheduling yields to two problems [3]:

- *Priority testing.* Given a hard real-time task system and a unique priority assignment to these tasks, can the system be scheduled by a static-priority scheduler such that all subtasks will always meet their deadlines?
- *Priority assignment.* Given a hard real-time task system, what is the unique priority assignment to these tasks (if one exists) which can be used by a static-priority run-time scheduler to schedule these tasks such that all subtasks will always meet their deadlines?

However, neither of these issues could have been solved within the context of the recurring real-time task model (for both preemptive and non-preemptive cases) up to this date and no optimal solution is known.

Our contributions. The priority assignment problem can be attacked by simply assigning a priority to each task in the system, and then checking if the assignment is feasible. However, for a system of n tasks, this approach has a complexity of $n!$ which grows too fast. Therefore, it does not provide a polynomial reduction from priority assignment to priority testing. In this paper, we study static priority scheduling of recurring real-time tasks. We focus on the *non-preemptive uniprocessor* case and obtain schedule-theoretic results for this case. To this end, we derive a sufficient (albeit not necessary) condition for schedulability under static priority scheduling, and show that this condition can be efficiently tested *provided that task parameters have integral values*. In other words, a testing condition is derived for the general *priority testing problem*, and efficient algorithms with run-times that are pseudo-polynomial with respect to the problem input size are given for the *integer-valued* case. In addition, it is shown that these results are not too pessimistic, on the contrary, they exhibit practical value as they can be utilized within a search framework to solve the *priority assignment problem*. We demonstrate this with examples, where in each case, an optimal priority assignment for a given problem is obtained within reasonable time, by first detecting good candidates using simulated annealing and then by testing them with the pseudo-polynomial time algorithm developed for priority testing.

The paper is organized as follows: next section formally introduces the recurring real-time task model. Section 3 presents the schedulability condition for static priority schedulers. In Section 4, we present a simulated annealing based priority assignment search framework. Section 5 presents experimental results. Finally, concluding remarks are given in Section 6.

2. Recurring Real-Time Task Model

A recurring real-time task T is represented by a directed acyclic graph (DAG) and a period $P(T)$ with a unique source vertex with no incoming edges and a unique sink vertex with no outgoing edges. Each vertex of the task represents a subtask and is assigned with an execution requirement $e(v)$ and a deadline $d(v)$ of real numbers. Each directed edge in the task graph represents a possible control flow. Whenever vertex v is triggered, the subtask corresponding to it is generated with ready time equal to the triggering time, and it must be executed for $e(v)$ units of time within the next $d(v)$ units of time. In the non-preemptive case which we consider, once a vertex starts being executed, it can not be preempted. Hence, it is executed until its execution time is completed. Only once it is finished with execution, another vertex which has been triggered possibly from another task, can be scheduled for execution. In addition, each edge (u, v) of a task graph is assigned with a real number $p(u, v) \geq d(u)$ called inter-triggering separation which denotes the minimum amount of time which must elapse after the triggering of vertex u, before the vertex v can be triggered.

The execution semantics of a recurring real-time task state that initially the source vertex can be triggered at any time. When a vertex u is triggered, then the next vertex v can only be triggered if there is an edge (u, v) and after at least $p(u, v)$ units of time has passed since the vertex u is triggered. If the sink vertex of a task T is triggered, then the next vertex of T to be triggered is the source vertex. It can be triggered at any time after $P(T)$ units of time from its last triggering. If there are multiple edges from vertex u which represents a conditional branch, among the possible vertices only one vertex can be triggered. Therefore, a sequence of vertex triggerings $v_1, v_2, \cdots, v_k$ at time instants $t_1, t_2, \cdots t_k$ is legal if and only if there are directed edges (v_i, v_{i+1}) and $p(v_i, v_{i+1}) \leq t_{i+1} - t_i$ for $i = 1, \cdots, k$. The real-time constraints require that the execution of v_i should be completed during the time interval $[t_i, t_i + d(v)]$.

Schedulability analysis of a task system. A task system $\mathcal{T} = \{T_1, \cdots, T_k\}$ is a collection of task graphs, the vertices of which are triggered independently. A triggering sequence for such a task system $\mathcal{T}$ is legal if and only if for every task graph T_i, the subsequence formed by combining only the vertices belonging to T_i constitutes a legal triggering sequence for T_i. In other words, a legal triggering sequence for $\mathcal{T}$ is obtained by merging together (ordered by triggering times, with ties broken arbitrarily) legal triggering sequences of the constituting tasks. The schedulability analysis of a task system $\mathcal{T}$ deals with determining whether under all possible legal triggering sequences of $\mathcal{T}$, the subtasks corresponding to the vertices of the tasks can be scheduled such that all their deadlines are met. Particularly, we are interested in the non-preemptive uniprocessor case.

2.1. Demand Request Function ($T.rbf(t)$)

The results on schedulability analysis in this paper are based on the abstraction of a task T by its demand request function $T.rbf(t)$ which is defined as follows [3]: $T.rbf(t)$ takes a non-negative real number $t \geq 0$ and returns the maximum cumulative execution requirement by the subtasks of

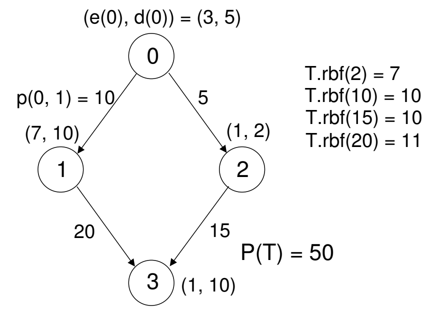


Figure 1. Demand request function for T.

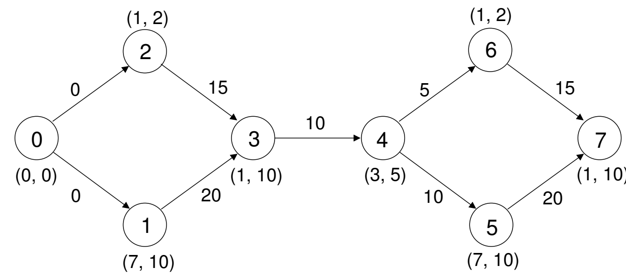


Figure 2. Transformed task graph T' for T in Figure 1.

T that have their triggering times within any time interval of duration t. In other words, demand request function $T.rbf(t)$ of task T denotes the maximum execution time asked by the subtasks of T within any time interval of length t, *however all of which is not necessarily to be completed within* t.

In Figure 1, we give an illustrative example. In this graph, $T.rbf(2) = 7$ because vertex v_1 can be triggered within 2 units of time. Similarly, $T.rbf(20) = 11$ due to a possible legal triggering sequence of v_3, v_0, v_1 at time instants $t_1 = 0$, $t_2 = 10$, $t_3 = 20$ within a time interval of $t = 20$. It can be shown by exhaustively enumerating all possible vertex triggerings of T that there exists no other sequence of vertex triggerings with a cumulative execution requirement that would exceed 11 within $t = 20$. Also notice that in the mentioned vertex triggering, the deadline requirements state that v_3 and v_0 should be completed by the time instants $t_1 + 10 = 10$ and $t_2 + 5 = 15$ which are both within t, while the deadline requirement for v_1 is at $t_3 + 10 = 30$ which is outside t.

2.2. Computing Demand Request Function

First we are going to compute $T.rbf(t)$ for small values of t in which the source vertex is either not triggered, or is triggered only once. Then using results of [3], we will provide an expression for any t. In this way, the effect of recurring behavior of the task model can be included in the calculations.

Calculating $T.rbf(t)$ for small t. To obtain all vertex triggerings, in which the source vertex is either not triggered or is triggered only once, we take two copies of the original DAG, add an edge from the sink vertex of the first copy to the source vertex of the second copy (by setting the inter-triggering separation equal to the deadline of the sink vertex of the first copy), and then delete the source vertex of the first copy. To make the resulting graph a DAG, we add a dummy source vertex to the first copy with $(e, v) = (0, 0)$. Starting from a transformed task graph which is not a DAG, [3] enumerates all paths in the task graph to compute $T.rbf(t)$ which has an exponential complexity while [5] starts from T and neglects the recurring behavior. Based on dynamic programming, we now give an incremental pseudo-polynomial time algorithm[1] to compute $T.rbf(t)$ for tasks with integral execution requirements and inter-triggering separations[2]. Let there be n vertices in T', $v_0, \cdots, v_{n-1}$. As shown in Figure 2, the vertex indices of T' are assigned such that there can be an edge from v_i to v_j only if $i < j$. Let $t_{i,e}$ be the *minimum time interval* within which the task T can have an execution requirement of *exactly* e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_0, \cdots, v_i\}$. Similarly, let $t^i_{i,e}$ be the *minimum time interval* within which a sequence of vertices from the set $\{v_0, \cdots, v_i\}$ and *ending* with vertex v_i, can have an execution of *exactly* e time units. Apparently, $E_{max} = (n-1)e_{max}$ where $e_{max} = \max\{e(v_i), i = 1, \cdots, n-1\}$ is an upper bound for $T.rbf(t)$ for any small $t \geq 0$.

Algorithm 1 computes $T.rbf(t)$ for small t in pseudo-polynomial time for tasks with integral $e(v) \geq 0$. Starting from the sequence $\{v_0\}$ and adding one vertex to this set in each iteration, the algorithm builds an array of minimal time intervals ending at the last vertex added for all execution requirement values between 0 and E_{max}, i.e. it computes $t^i_{i,e}$. Then using this result and the result of the previous calculation ($t_{i-1,e}$), it computes $t_{i,e}$ by taking their minimum. Once all vertices are processed and an array of minimal time intervals is built, the algorithm makes a lookup in the array and returns the maximum execution requirement for a given small t. It has a running time of $O(n^3 E_{max})$.

Calculating $T.rbf(t)$ for any t. Once $T.rbf(t)$ is known for small t, the following expression from [3] can be used to calculate it for any t.

$$T.rbf(t) = \max\{\lfloor t/P(T)\rfloor E(T) + T.rbf(t'), \\ (\lfloor t/P(T)\rfloor - 1)E(T) + T.rbf(P(T) + t')\}, \quad (1)$$

where $E(T)$ denotes maximum possible cumulative execution requirement on any path from the source to the sink vertex of T and $t' = t \bmod P(T)$.

Algorithm 1 Computing $T.rbf(t)$ for small t

input: Transformed task graph T', a real number $t \geq 0$
output: $T.rbf(t)$

for $e = 0$ to E_{max} **do**

$$t_{0,e} \leftarrow \begin{cases} 0 & \text{if } e(v_0) \geq e \\ \infty & \text{otherwise} \end{cases}$$

$t^0_{0,e} \leftarrow t_{0,e}$
end for
for $i = 0$ to $n-2$ **do**
for $e = 0$ to E_{max} **do**
Assume there are directed edges from the vertices $v_{i_1}, v_{i_2}, \cdots, v_{i_k}$ to v_{i+1}

$$t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} + p(v_{i_j}, v_{i+1}) \text{ such} \\ \text{that } j = 1, \cdots, k\} \quad \text{if } e(v_{i+1}) < e \\ 0 \quad \text{if } e(v_{i+1}) \geq e \end{cases}$$

$t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$
end for
end for
$T.rbf(t) \leftarrow \max\{e \mid t_{n-1,e} \leq t\}$

3. Schedulability under Static Priority Scheduling

In this section, we derive a sufficient condition for schedulability under static priority scheduling. It is based on the abstraction of a recurring real-time task in terms of its demand request function.

Theorem 1 *Given a task system $\mathcal{T} = \{T_1, \cdots, T_k\}$, where the task T_r has priority r, $0 \leq r \leq k$, and $r < q$ indicates that T_r has a higher priority than T_q. The task system is static priority schedulable if for all tasks T_r the following condition holds: for any vertex v of any task T_r, $\exists\tau$ with $0 \leq \tau \leq d(v) - e(v)$ for which*

$$e^{>r}_{max} + T_r.rbf(t - p^{T_r}_{min}) + \sum_{i=1}^{r-1} T_i.rbf(t + \tau) \leq t + \tau, \quad \forall t \geq 0 \quad (2)$$

[1] A pseudo-polynomial time algorithm for an integer-valued problem is an algorithm whose running time is polynomial in the input size and in the values of the input integers. See [9] for a nice coverage.

[2] Computing $T.rbf(t)$ remains NP-hard even if the parameters (i.e. execution requirements, deadlines and inter-triggering separations) of the recurring real-time task model are restricted to integer numbers [6].

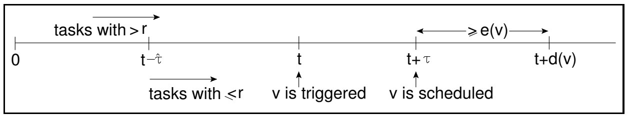


Figure 3. Scheduling scenario in Theorem 1.

where $e_{max}^{>r} = \max\{e(v') \mid v'$ *is a vertex of* $T_j, j = r+1, \cdots, k\}$ *and* $p_{min}^{T_r} = \min\{p(u,u') \mid u$ *and* u' *are vertices of* $T_r\}$.

Proof: Let v be any vertex of the task T_r with an execution requirement $e(v)$ and a deadline $d(v)$. Consider the following scenario which is also depicted in Figure 3:

Let v be triggered at time t and be scheduled at time $t+\tau$. We assume that $t-\hat{\tau}$ is the first time before time t where the processor has no task with priority $\leq r$ to execute. Hence, at this time the processor is either idle or executing a task with priority $> r$. On the other hand, $t-\hat{\tau}$ is also the time where at least one vertex of a task graph with priority $\leq r$ was triggered. Under these conditions, the *upperbound* for the total remaining execution requirement before the vertex v can be scheduled at time $t+\tau$ is composed of

- the remaining execution requirement of some task triggered before time $t-\hat{\tau}$: $e_{max}^{>r}$,
- the execution requirement of the task T_r (excluding v) during time interval $[t-\hat{\tau}, t]$: $T_r.rbf(\hat{\tau}-p_{min}^{T_r})$ where $p_{min}^{T_r}$ is the minimal inter-triggering separation in T_r,
- the total execution requirement of the tasks with $< r$ during time interval $[t-\hat{\tau}, t+\tau]$: $\sum_{i=1}^{r-1} T_i.rbf(\tau+\hat{\tau})$.

Therefore, within $[t-\hat{\tau}, t+\tau]$, the upperbound for the total execution requirement is

$$e_{max}^{>r} + T_r.rbf(\hat{\tau} - p_{min}^{T_r}) + \sum_{i=1}^{r-1} T_i.rbf(\tau + \hat{\tau}). \quad (3)$$

We define $I[t-\hat{\tau}, t+\tau]$ to be the processor idle time during time interval $[t-\hat{\tau}, t+\tau]$. If we show that the *lowerbound* for $I[t-\hat{\tau}, t+\tau]$ is non-negative, then we can conclude that the task system is schedulable. The lower bound for $I[t-\hat{\tau}, t+\tau]$ can be written as,

$$I[t-\hat{\tau}, t+\tau] \geq (t+\tau) - (t-\hat{\tau})$$
$$-(e_{max}^{>r} + T_r.rbf(\hat{\tau} - p_{min}^{T_r}) + \sum_{i=1}^{r-1} T_i.rbf(\tau + \hat{\tau})). \quad (4)$$

By the condition (2) in Theorem 1, (3) is bounded by $\tau + \hat{\tau}$. Substituting this in (4), we obtain,

$$I[t-\hat{\tau}, t+\tau] \geq 0. \quad (5)$$

Algorithm 2 Schedulability under Static Priority Scheduling

input: Task system $T_r \in \mathcal{T}$ with unique r
output: $decision$
  $decision \leftarrow yes$
  for all $T_r \in \mathcal{T}$ and **for all** $v \in T_r$ and **for all** $t \geq 0$ **do**
    $flag \leftarrow 0$
    $e_{max}^{>r} \leftarrow \max\{e(v') \mid v' \in T_i, i > r)\}$
    $p_{min}^{T_r} \leftarrow \min\{p(u,u') \mid u, u' \in T_r)\}$
    $\mathcal{T}_{<r} \leftarrow \mathcal{T} \backslash \{T_i \mid i \geq r\}$
    $\tau_{max} \leftarrow d(v) - e(v)$
    for $\tau = 0$ to τ_{max} **do**
      if $e_{max}^{>r} + T_r.rbf(t - p_{min}^{T_r}) + \sum_{T \in \mathcal{T}_{<r}} T.rbf(t + \tau) \leq t + \tau$ **then**
        $flag \leftarrow 1$
      end if
    end for
    if $flag = 0$ **then**
      $decision \leftarrow no$
    end if
  end for
  return $decision$

Hence, all tasks scheduled before vertex v meet their deadlines at $t+\tau$. The condition $0 \leq \tau \leq d(v) - e(v)$ ensures that v also meets its deadline. ■

Theorem 1 can be used to construct Algorithm 2 which solves the *priority testing problem* as defined in Section 1. Algorithm 2 simply checks if condition (2) holds for every vertex in the task system, and relies on Algorithm 1 and (1) for $T.rbf(t)$ calculations. Algorithm 2 along with Algorithm 1 is again a pseudo-polynomial time algorithm, since all other steps in Algorithm 2 can also be performed in pseudo-polynomial time. To see this, given any $T_r \in \mathcal{T}$, let $t_{max}^{T_r}$ denote the maximum amount of time elapsed among all vertex triggerings starting from the source and ending at the sink vertex, if every vertex of T_r is triggered at the earliest possible time without violating inter-triggering separations. Clearly, it is sufficient to test condition (2) in Algorithm 2 for $t_{max} = \max\{t_{max}^{T_r}, T_r \in \mathcal{T}\}$ times, which is pseudo-polynomially bounded. Therefore, Algorithm 2 is also a pseudo-polynomial time algorithm.

4. Simulated Annealing Framework

Simulated annealing (SA) can be viewed as a local search equipped with a random decision mechanism to escape from local optima. It is inspired by the annealing process in condensed matter physics. In this process, a matter is first melted and then slowly cooled in order to obtain the perfect crystal structure. In high temperatures, all the particles move randomly to high energy states. But as the temperature is decreased, the probability of such movements is also decreased.

In combinatorial optimization, the energy of a state corresponds to the cost function value of a feasible point and the temperature becomes a control parameter. We start with an arbitrary initial point and search its neighborhood randomly. If a better solution is found, then it becomes the current solution and the search continues from that point. But if it is a worse solution, then it may still be accepted with some probability depending on the difference in cost function values and the current temperature. Initially at high temperatures, the probability of accepting a worse solution is higher. The acceptance probability decreases, as the temperature is lowered. As a consequence, SA behaves like a random walk during early iterations, while it imitates hill climbing in low temperatures.

One of the strong features of SA is that it can find high quality solutions independent of the initial solution. In general, weak assumptions about the neighborhood and cooling scheme are enough to ensure convergence to optimal solutions. The key parameters in SA are temperature reduction rate and neighborhood definition. In most cases, it may require a lot of trials to adjust these parameters to a specific problem. We discuss the latter within the context of the schedulability problem in the next section. In order to utilize SA, we first formulate schedulability under static priority scheduling as a combinatorial optimization problem.

Problem formulation. Assume that we are given an instance (F, c) of an optimization problem, where F is the feasible set and c is the cost function. In our case F is the set of all possible priority assignments to tasks in $\mathcal{T}$ and c is the cost of such an assignment. Given a priority assignment f to tasks in $\mathcal{T}$, let $cond$ represent the schedulability condition in (2), i.e. $cond = e^{>r}_{max} + T_r.rbf(t - p^{T_r}_{min}) + \sum_{T \in \mathcal{T}_{<r}} T.rbf(t + \tau)$. In this assignment, we define the cost of assigning priority r to a task, $c(T_r, t)$ as

$$c(T_r, t) \leftarrow \begin{cases} 0 & \text{if } t = 0^- \\ c(T_r, t-1) & \text{if } \forall v \in T_r, \\ & \exists \tau \text{ s.t. } cond \leq t + \tau \\ |A| + c(T_r, t-1) & \text{if for some } v \in A \subseteq T_r, \\ & \nexists \tau \text{ s.t. } cond \leq t + \tau \end{cases}$$

where $0 \leq \tau \leq d(v) - e(v)$ and $0 \leq t \leq t^{T_r}_{max}$. Following this definition, the cost of a particular priority assignment to a task system becomes $c(f, \mathcal{T}) = \sum_{T_i \in \mathcal{T}} c(T_r, t^{T_r}_{max})$. The aim is to find the priority assignment $f \in F$ which minimizes $c(f, \mathcal{T})$.

Corollary 2 *A task system $\mathcal{T}$ is schedulable under static priority scheduling if $\exists f$ such that $c(f, \mathcal{T}) = 0$.*

Proof: The proof follows from the definition of $c(f, \mathcal{T})$. Clearly, for each task in $\mathcal{T}$, a violation of schedulability condition given by (2) increments the value of $c(f, \mathcal{T})$ by one. Hence a value of zero indicates that the schedulability condition is not violated. ■

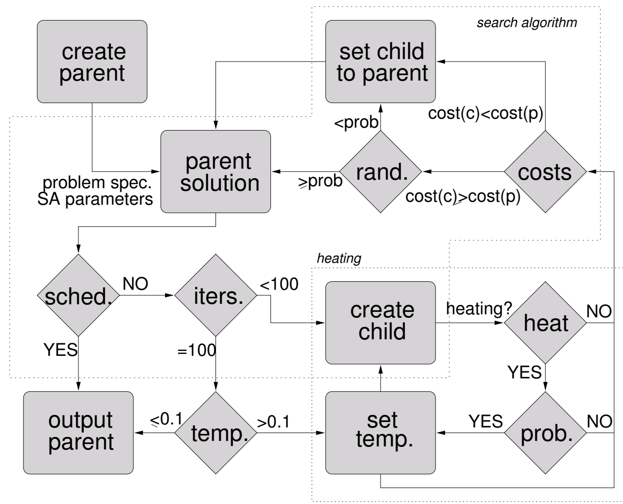


Figure 4. Overview of SA framework.

5. Experimental Results

In the previous section, we have introduced general characteristics of a simulated annealing framework. We now continue discussing those parameters of SA that are fine tuned according to the problem in hand. These parameters are used in all the experiments reported here. Figure 4 provides an overview for our framework. At first we use a heating mechanism to set the initial temperature. To do so, we start with $temp = 100$ and look at the first 10 iterations. If it is found that $prob(p \to s) < 0.5$ in one of these early iterations, then the temperature is increased such that the new solution is always accepted, i.e. current temperature is increased with $temp = |c(p, \mathcal{T}) - c(s, \mathcal{T})| / \ln(0.5)$ on that iteration. Remember that we are given an instance of the scheduling problem defined in the form (F, c) (see Section 4), and at each iteration, we search a neighborhood $N : F \to 2^F$ randomly at some feasible point $p \in F$ for an improvement. If such an improvement occurs at $s \in N(p)$ then the next point becomes s. But if $c(s, \mathcal{T}) > c(p, \mathcal{T})$ then s is still taken with a probability $prob(p \to s) = e^{-(c(s,\mathcal{T}) - c(p,\mathcal{T}))/temp}$. In our experiments, the number of such iterations at each temperature is set to 100. The control parameter $temp$ is gradually decreased in accordance with a pre-defined cooling scheme. For the latter, we use a static reduction function $temp = 0.9temp$. Finally, we stop the search either when a feasible schedule is found or when the temperature drops under a certain value ($temp = 0.1$).

To illustrate the practical usefulness of our results, we have taken task examples from the literature and construc-

Table 1. Task System Specifications

task(T)	#ver./ed.(T)	#ver./ed.(T')	$P(T)$	TS1	TS2	TS3	TS4	TS5	TS6	TS7
hou_c1	4/3	9/10	300	o	o	o	o	o	o	o
hou_c2	4/4	8/9	200	o	o	o	o	o	o	o
hou_c3	3/2	7/8	200	o	o	o	o	o	o	o
hou_c4	3/2	6/5	200	o	o	o	o	o	o	o
yen1	5/4	11/12	300	o	o	o	o	o	o	o
yen2	4/3	9/10	300	o	o	o	o	o	o	o
yen3	6/5	14/18	400	-	o	o	o	o	o	o
dick	5/5	11/14	400	-	-	o	o	o	o	o
hou_u1	10/13	21/30	700	-	-	-	o	o	o	o
hou_u2	10/16	20/33	500	-	-	-	-	o	o	o
hou_u3	10/15	22/41	600	-	-	-	-	-	o	o
hou_u4	10/14	22/36	700	-	-	-	-	-	-	o

Table 2. Experimental Results

				ES (secs.)		SA (secs.)		
task system	#tasks	$(t_{min}, t_{mid}, t_{max})$	sol. density	ES1	ES2	search	test	total
TS1	6	$(10, 100, 189)$	2.5%	$4,511$	333	22.15	6.25	28.40
TS2	7	$(10, 100, 188)$	2.14%	-	$2,856$	32.75	7.90	40.65
TS3	8	$(10, 100, 184)$	0.89%	-	$23,419$	64.40	9.30	73.70
TS4	9	$(10, 100, 428)$	-	-	-	88.60	37.80	126.40
TS5	10	$(10, 100, 429)$	-	-	-	170.75	53.40	224.15
TS6	11	$(10, 100, 427)$	-	-	-	311.05	62.80	373.85
TS7	12	$(10, 100, 430)$	-	-	-	331.45	86.45	417.90

ted new task systems using their different combinations. The first column in Table 1 refers to tasks used; tasks starting with hou_c and hou_u are Hou's clustered and unclustered tasks [8], respectively. Tasks starting with yen are Yen's examples on p.83 in [15]. Task dick is from [7]. These tasks were originally defined in different task models and do not posses all characteristics of a recurring real-time task. Topologically, some tasks have multiple source and/or sink vertices. In such cases, we have added dummy vertices (with null execution requirements) when necessary. In most cases, originally a deadline for each task was defined, contrary to recurring real-time task model in which a deadline is defined for each vertex (subtask). Therefore, we have defined deadlines for all vertices in all tasks (same in all task sets) using a random generator. Then in all task systems, we have tried to give maximal execution requirements to vertices in order to minimize the number of feasible priority assignments. We have achieved this by gradually increasing execution requirements until a small increase yielded an unschedulable task system.

In the appendix, Figure 7 and Table 3 provide original task graphs and values of task parameters in our experiments, respectively. What is more, the details of transforming task graphs with multiple source and/or sink vertices is explained on illustrative examples given in Figure 8. We provide a summary of most important parameters in columns 2, 3 and 4 of Table 1. The former two columns give the number of vertices and edges in task and transformed task graphs, while the latter provides task periods. As already stated, run-times of Algorithms 1 and 2 are pseudo-polynomially bounded with task sizes. The rest of the columns in Table 1 give task system specifications.

For numerical results, we have integrated Algorithms 1 and 2 as C functions into the introduced SA framework which was also implemented in C. The experiments reported here have been performed on a Pentium 3 PC with 600 MHz CPU and 320 MB RAM running Linux OS. For each task system scenario, we have performed 20 runs (with seeds from 1 to 20 for the random generator) searching for feasible schedules using the SA framework. All results given in Table 2 are arithmetic means of 20 runs. During the experiments, we observed that schedulability condition is more sensitive to small values of t, and in most cases, it is enough to test up to the first t_{min} times to find a majority of non-optimal solutions. Therefore, in order to decrease search run-times by means of spending less time on non-optimal solutions, we have used a combination of t_{min} and t_{mid} values instead of the actual value t_{max}. In most cases, it was enough to test for the first t_{min} times to find a majority of non-optimal solutions. In the search, if a feasible solution for t_{min} was found, it was further tested for times up to t_{mid}. Only if the solution also passed this second test, it was output as a candidate for an optimal solution and the search was stopped. We call this CPU time spent on

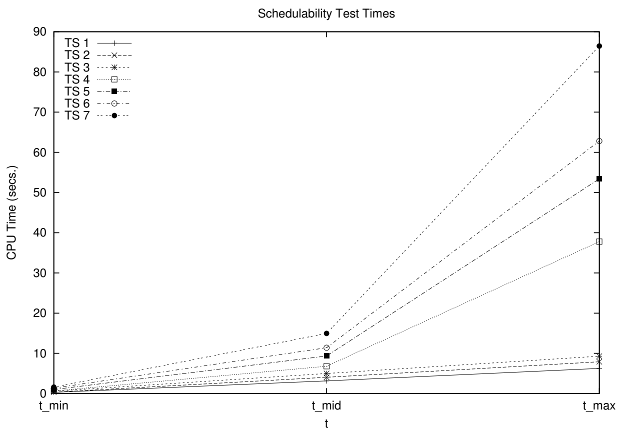


Figure 5. Schedulability test times.

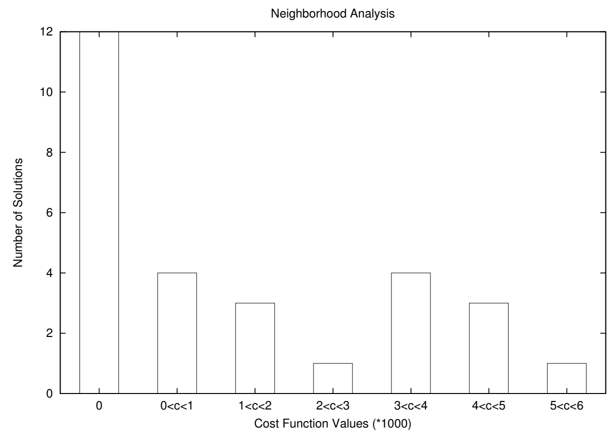


Figure 6. Neighborhood analysis.

search as *SA search* and report their averages in column 7 of Table 2. Since SA tests many non-optimal solutions until it reaches an optimal one, using t_{min} instead of t_{max} at each iteration dramatically decreased run-times. In Figure 5, we plotted CPU times of schedulabilility tests with t_{min}, t_{mid} and t_{max} in all task system scenarios. If we compare CPU times of t_{min} and t_{max}, the difference lies between 15 to 25 times for TS1, TS2 and TS3, while for relatively larger task systems TS4, TS5, TS6 and TS7, it is between 45 to 60 times. Finally, all candidate solutions found has been tested once with t_{max}. We call the CPU time spent on this last step as *SA test*, and similarly report their averages in column 8 of Table 2. If a candidate solution fails in the last step, SA is started again and the next solution found is taken as the new candidate. However, it is interesting to note here that in our experiments, all first candidate solutions passed the last test, and therefore none of the SA search or test steps were repeated.

We have also performed exhaustive searches (ES) in cases where the size of task systems permitted to do so. The main reason behind this was to find out solution density, i.e. the number of optimal solutions in the feasible set. As a result of ES runs (given in column 4 in Table 2), we found out that solution densities in TS1, TS2 and TS3 scenarios are less than or equal to 2.5%. Two exhaustive searches ES1 and ES2 are given in columns 5 and 6 of Table 2. In ES1, all points in the feasible set were tested with t_{max} which took $4,511$ secs. for TS1. In ES2, we first tested all points with t_{min}, then only tested those points which passed the first test with t_{max}. Using this method, ES CPU time for TS1 decreased from $4,511$ to 333 secs. and we could also perform ES runs for TS2 and TS3.

Finally, we have taken one optimal solution for TS3 and examined all its 28 neighbors. The costs of these points are plotted in Figure 6. Despite a significant number of other optimal solutions around this solution, there are also some points with moderate to very high costs. This shows us that the distance from an optimal point and the value of the cost function are not strongly correlated. The latter may substantially increase SA run-times.

6. Conclusion

In this paper, we have derived a sufficient (albeit not necessary) condition to test schedulability of recurring real-time tasks under static priority scheduling. It was shown that this condition can be tested in pseudo-polynomial time, provided that task execution requirements and inter-triggering separations have integral values. Furthermore, these results were not too pessimistic and had also practical value. The latter was demonstrated in terms of experiments performed with different task systems, where in each case, an optimal solution for a given problem specification could be reported within reasonable time.

7. Acknowledgement

This work is partly supported by the Dutch Technology Foundation STW under grant AES 5021.

References

[1] S. K. Baruah. Feasibility analysis of recurring branching tasks. In *Proc. of the Euromicro Workshop on Real-Time Systems*, June 1998.

[2] S. K. Baruah. A general model for recurring real-time tasks. In *Proc. of the Real Time Systems Symposium*, Dec. 1998.

[3] S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1), 2003.

[4] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1), 1999.

[5] S. Chakraborty, T. Erlebach, S. Künzli, and L. Thiele. Schedulability of event-driven code blocks in real-time embedded systems. In *Proc. of the Design Automation Conference*, June 2002.

[6] S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. In *Proc. of the 7th Int. Workshop on Algorithms and Data Structures*, LNCS 2125. Springer-Verlag, 2001.

[7] R. P. Dick and N. K. Jha. MOCSYN: Multiobjective core-based single-chip system synthesis. In *Proc. of the Design, Automation and Test in Europe*, Mar. 1999.

[8] J. Hou and W. Wolf. Process partitioning for distributed embedded systems. In *Proc. of Int. Workshop on Hardware/Software Codesign*, Mar. 1996.

[9] J. Hromkovič. *Algorithmics for Hard Problems (Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics)*. Springer-Verlag, 2002.

[10] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

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

[12] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Trans. on Software Engineering*, 23(10), 1997.

[13] P. Pop, P. Eles, and Z. Peng. Schedulability analysis for systems with data and control dependencies. In *Proc. of Euromicro Conference on Real-Time Systems*, June 2000.

[14] T. Pop, P. Eles, and Z. Peng. Schedulability analysis for distributed heterogeneous time/event-triggered real-time systems. In *Proc. of Euromicro Conference on Real-Time Systems*, July 2003.

[15] T. Yen and W. Wolf. *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. Kluwer Academic Publishers, 1996.

A. Task Systems

Original task graphs for tasks used in the experiments are given in Figure 7. In all task graphs, a vertex with no incoming edge is a source vertex, and similarly a vertex with no outgoing edge is a sink vertex. In the recurring real-time task model, tasks have single source and sink vertices in their task graphs and transforming such a task graph was already shown in Figure 2. However as seen in Figure 7, a number of tasks taken from the literature were defined in earlier task models and they have multiple source and/or sink vertices. In Figure 8, we show how these task graphs are transformed so that the transformed task graphs have single source and sink vertices. There exist three different cases: (1) tasks with multiple sink vertices, (2) tasks with multiple source vertices, and (3) tasks with multiple source *and* sink vertices. In Figures 8(a), 8(b) and 8(c), one example of a transformed task graph is given for each case.

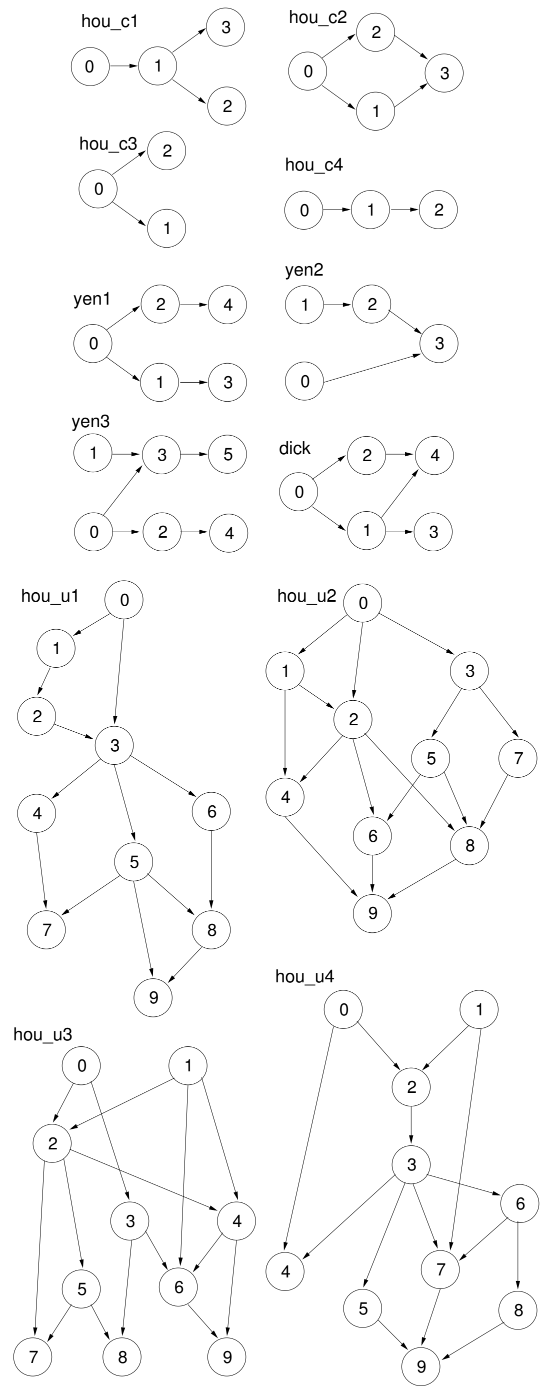
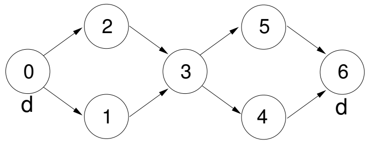
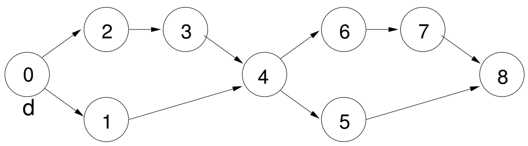


Figure 7. Original task graphs taken from the literature. Some of the task graphs have multiple source and/or sink vertices.

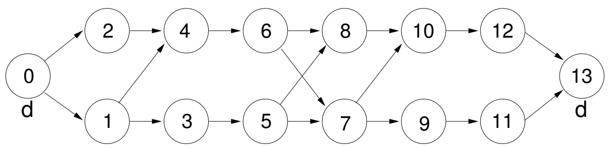
Alternative to the examples in Figure 8, we could also add dummy vertices to original task graphs and subsequently use the standard procedure (explained in Sec-



(a) Case 1: Task with multiple sink vertices.



(b) Case 2: Task with multiple source vertices.



(c) Case 3: Task with multiple source *and* sink vertices.

Figure 8. Three example transformed task graphs for tasks with multiple source and/or sink vertices. In all transformed task graphs, dummy vertices added are labeled with "d". (a) Transformed task graph for hou_c3 which has multiple sink vertices. (b) Transformed task graph for yen2 which has multiple source vertices. (c) Transformed task graph for yen3 which has multiple source and sink vertices.

tion 2.2) to transform them. However, this would unnecessarily increase the number of dummy vertices in the transformed task graphs, which in turn would increase run-times for $T.rbf(t)$ calculations.

Although not explicitly mentioned previously, it should be clear from its input that Algorithm 2 actually operates on the original task graphs. Hence, the schedulability condition is tested only once for all vertices (i.e. subtasks) on each iteration of the algorithm.

Finally, in Table 3 we provide values used in the experiments for each task system that we have synthesized using tasks in Figure 7. The values are given with respect to original task graphs rather than transformed task graphs. In addition, we should also note that during all experiments, inter-triggering separations were set equal to deadlines, i.e. $p(u, v) = d(u)$ for all $u, v \in T$ in all task systems.

Table 3. Experimental Data

		$e(v)$							
T	v	TS1	TS2	TS3	TS4	TS5	TS6	TS7	$d(v)$
hou_c1	0	7	3	6	5	5	5	5	71
	1	9	5	6	5	5	4	5	56
	2	10	6	6	5	5	4	5	63
	3	4	1	1	1	3	1	3	50
hou_c2	0	5	4	2	2	3	4	3	80
	1	8	7	6	5	5	4	5	99
	2	10	9	6	5	4	6	5	99
	3	10	9	5	4	3	2	2	46
hou_c3	0	6	5	8	5	4	5	4	64
	1	8	6	5	4	6	2	3	93
	2	10	9	8	5	4	4	4	53
hou_c4	0	4	4	5	4	5	6	4	78
	1	8	8	5	4	5	4	4	74
	2	8	7	5	3	4	3	4	57
yen1	0	1	6	6	5	5	5	4	73
	1	1	1	1	1	3	1	3	72
	2	6	4	6	5	5	5	4	82
	3	8	6	6	5	4	4	4	82
	4	8	6	6	5	4	5	4	77
yen2	0	9	7	6	6	5	4	2	80
	1	9	7	6	6	4	3	3	57
	2	8	6	5	5	4	4	3	66
	3	8	6	6	6	4	3	3	61
yen3	0	-	4	4	4	4	2	4	53
	1	-	5	5	5	4	4	4	61
	2	-	2	2	2	3	1	5	89
	3	-	6	6	5	5	4	5	82
	4	-	1	1	1	2	1	5	97
	5	-	4	4	4	3	2	3	32
dick	0	-	-	4	4	4	2	3	96
	1	-	-	5	5	2	3	3	48
	2	-	-	7	6	3	4	4	59
	3	-	-	7	6	3	4	4	85
	4	-	-	3	3	1	1	1	36
hou_u1	0	-	-	-	4	4	4	3	58
	1	-	-	-	6	5	4	4	98
	2	-	-	-	2	2	2	3	52
	3	-	-	-	1	3	1	3	59
	4	-	-	-	1	3	1	1	64
	5	-	-	-	1	1	1	1	62
	6	-	-	-	4	5	4	4	88
	7	-	-	-	4	4	4	2	63
	8	-	-	-	4	4	4	3	68
	9	-	-	-	5	5	4	3	54
hou_u2	0	-	-	-	-	4	4	4	85
	1	-	-	-	-	3	1	1	89
	2	-	-	-	-	3	3	2	66
	3	-	-	-	-	4	4	4	78
	4	-	-	-	-	4	4	3	97
	5	-	-	-	-	3	1	1	80
	6	-	-	-	-	3	3	2	74
	7	-	-	-	-	4	4	4	82
	8	-	-	-	-	4	4	4	56
	9	-	-	-	-	6	4	4	95
hou_u3	0	-	-	-	-	-	4	4	72
	1	-	-	-	-	-	1	1	62
	2	-	-	-	-	-	4	4	92
	3	-	-	-	-	-	2	1	88
	4	-	-	-	-	-	4	4	81
	5	-	-	-	-	-	4	3	85
	6	-	-	-	-	-	4	4	86
	7	-	-	-	-	-	2	1	95
	8	-	-	-	-	-	3	2	70
	9	-	-	-	-	-	4	4	77
hou_u4	0	-	-	-	-	-	-	1	71
	1	-	-	-	-	-	-	4	80
	2	-	-	-	-	-	-	4	96
	3	-	-	-	-	-	-	4	72
	4	-	-	-	-	-	-	2	78
	5	-	-	-	-	-	-	1	96
	6	-	-	-	-	-	-	4	82
	7	-	-	-	-	-	-	3	68
	8	-	-	-	-	-	-	4	97
	9	-	-	-	-	-	-	3	91