Off-line (Optimal) Multiprocessor Scheduling of Dependent Periodic Tasks

Julie Baro  
ONERA, Toulouse, France

Frédéric Boniol  
ONERA, Toulouse, France

Mikel Cordovilla  
ONERA, Toulouse, France

Eric Noulard  
ONERA, Toulouse, France

Claire Pagetti  
ONERA, Toulouse, France

ENSEEIHT, Université de Toulouse, France

ABSTRACT
This paper addresses the global scheduling of constrained deadline periodic dependent task sets on multiprocessor platforms composed of identical processors. We propose two orthogonal approaches: (1) an off-line computation of a valid fixed priority assignment (2) a computation of an off-line schedule. The method in both cases is based on the efficient exploration of a finite automaton encoding all the possible executions. Even if the problems are NP-complete, we obtain rather reasonable performances as illustrated in the benchmarks.

Keywords
Multiprocessor scheduling, off-line computation

1. INTRODUCTION
The future of microprocessors tends to multiprocessor architectures forcing designs to adapt the specifications to this new paradigm. Critical embedded systems are faced to the issue of embedding safely such hardware. As far as scheduling is concerned, the specification, usually given as a dependent periodic task set, must be scheduled in a way that respects the real-time attributes of each task and the precedence constraints between the tasks. This paper addresses the problem of scheduling a real-time task set with a global preemptive fixed priority policy or a static sequencer on a symmetric multiprocessor system consisting of m identical processors.

1.1 Real-time scheduling
We consider periodic task sets \( \mathcal{S} = \{\tau_i\}_{i=1,...,n} \) where the temporal behaviour of a task \( \tau_i \) is defined by the four real-time attributes \( (T_i, C_i, O_i, D_i) \). \( T_i \) is the period of repetition, \( C_i \) is the worst case execution time estimation (wcet), \( O_i \) is the first arrival time and \( D_i \) is the relative deadline. The task set is said synchronous if all offsets are equal to 0, and asynchronous otherwise. The task set has implicit deadlines (resp. constrained deadlines) if for all \( i \), \( D_i = T_i \) (resp. \( D_i \leq T_i \)). Each repetition of a task is called job or instance. The hyper-period designates the least common multiple of all the task periods, i.e. \( H = \text{lcm}_{1 \leq i \leq n}(T_i) \). This model is a standard adaptation of the Liu and Layland model [16]. We consider dependent task set where tasks are subject to precedence constraints.

The scheduling of periodic task sets on multiprocessor has been studied since the seventies and numerous results have been provided since, part of which is summarised in [11]. In this paper, we consider preemptive (a task can be interrupted while executing by another task) and global (a suspended instance may be resumed on any processor, we then speak of migration) scheduling policies.

Fixed priority policy consists in assigning a distinct static priority to each task. At run-time, the access is given to the \( m \) highest priority tasks on the processors. An off-line schedule is computed statically before run-time. The obtained order is stored in a table used at run-time to guide the execution. The scheduler is in fact a dispatcher which allocates the processors to the tasks at predefined dates and does this in a constant time. These kinds of policy have the favour of industrial designers since they are easy to implement, analyse and evaluate.

1.2 Off-line computation
Both activities, of computing a fixed priority assignment and an off-line schedule, occur before run-time and simplifies the designer work.

Priority assignment.
How to (efficiently) assign safely the priority to the tasks is a well-known problem in uniprocessor case. Rate Monotonic (priorities are assigned according to the period) is optimal for synchronous implicit deadline independent task sets, Deadline Monotonic (priorities are assigned according to the deadline) is optimal for synchronous constrained deadline independent task sets and Audsley’s algorithm [2] is optimal for asynchronous constrained deadline independent periodic tasks. These policies have been extended for dependent task sets in [12].

Unfortunately, none of these policies remain optimal on multiprocessor platforms. A brute force solution, proposed in [9], is to search among all the possible assignments which complexity is at worst \( n! \times c \) where \( n \) is the number of task and \( c \) is the cost of the verification of the schedulability of
Sequencing vs scheduling.

A fixed priority policy requires the scheduler to take online decisions. This solution, even if very simple, is sometimes excluded for highly critical applications. In that case, industrial designers prefer choosing to compute an off-line static sequencing which executes indefinitely. This provides several interesting properties: (1) predictability. All the costs (preemptions, migrations and context switches) are assessable before run-time. (2) simplicity. The use of semaphore or any other synchronisation mechanisms can be avoided. The management of external constraints, such as the use of specific drivers which require the application to behave in certain manner, is simplified. (3) efficiency. Off-line schedules are tractable and provide a very simple and efficient scheduler implementation. In multiprocessor SoC domain, authors of [7] advocate the use of pre-determined schedule in order to minimize scheduler overhead. (5) optimality. Moreover, off-line schedule search [14] allows theoretically to find a valid schedule if any exists.

1.3 Contribution

The purpose of the paper is to help the designer to find an adequate solution for scheduling a task set using either a fixed priority policy or an off-line schedule.

Priority assignment.

We first implement the brute force exploration of priority assignment proposed by [9]. As expected, the applicability of the method quickly encounters the state space explosion problem. We then propose a sub optimal heuristic with good results in the sense that it finds solution for non trivial task sets (in size and in “hardness”). It is a modified Audsley-like algorithm based on a branch and bound solution with efficient cuts using schedulability results. Both methods have been implemented in a C code and are detailed in section 3.

Sequencing vs scheduling.

Computing an off-line sequence requires to have a cycle of repetition. In uniprocessor platforms, the feasible window is known to be \([0, H]\) in case of independent constrained deadline synchronous task set and \([0, \max(O_i)+2H]\) in case of independent constrained deadline asynchronous task set. As soon as there are precedences, the window is more complex to determine [15]. In the multiprocessor case, Cucu and Goossens [9] have proved the feasible window to be \([0, H]\) in case of synchronous independent constrained deadline sets.

Xu and Parnas [13,19] have worked on pre-run-time scheduling which is equivalent to off-line feasible schedule. They propose an optimal scheduling method based on a branch and bound approach for synchronous dependent periodic task sets on uni and multiprocessor platforms with additional constraints such as mutual exclusion. Shepard and Gagné [17] extend the results of Xu and Parnas for multiprocessor but with no migration: it is a bin packing problem which consists in finding an adequate repartition of the tasks on the processors and then applying a monoprocessor sequencing. In [3], the authors use priced timed automata for searching an optimal scheduling for a set of jobs related by precedence constraints. They apply the model checker UPPAAL [2] for the effective search. Their modelling relies on several automata and states which is not efficient.

We define an optimal search of off-line global preemptive schedule for asynchronous periodic dependent task set. The general problem of finding such a schedule is NP-hard in the strong sense [9]. We have implemented the search in UPPAAL because the configuration graph is very complex and model checkers are dedicated for efficient exploration of such data structure. The method is detailed in section 4. Section 5 presents the benchmarks and performances for both approaches. The heuristic manages to schedule numerous task sets and is comparable to gEDF and gLLF. From the generated task sets, we do not manage to exhibit feasible task sets not schedulable by any on-line policy.

2. PROBLEM ENCODING

We establish two preliminary results: (a) the problem of finding a schedule can be considered as a search problem among a space of configurations, and (b) this configuration space can be reduced to a finite space, making the search problem decidable.

Encoding of the space of configurations.

The scheduling of the system can be represented as a sequence of configurations. At time \(t\), the state of the system is represented by \(conf(t) = \{conf(\tau_1, t), \ldots, conf(\tau_n, t)\}\) where for \(i \in [1, n]\), \(conf(\tau_i, t) = (T_i^\tau(t), C_i^\tau(t), O_i^\tau(t), D_i^\tau(t))\). \(O_i^\tau(t) = \max(O_i - t, 0)\) is the remaining time since the first awakening of the task \(\tau_i\). If \(O_i^\tau(t) > 0\), all the other parameters are null. \(O_i^\tau(t) = 0\) means that \(t\) is greater than the release time and the other parameters have a value. \(T_i^\tau(t) = T_i + ((t - O_i) \mod T_i)\) is the remaining time till the next activation of \(\tau_i\). \(D_i^\tau(t) = \max(0, T_i(t) - (T_i - D_i))\) is the remaining time till the next deadline and \(C_i^\tau(t)\) is the remaining execution time for current instance of \(\tau_i\). This encoding is close to the one proposed in [10]. Because of the lack of space, we do not detail the encoding of the precedence but the idea is the same: a variable \(P(\tau_i, \tau_j, t) = \tau_i \rightarrow \tau_j \wedge C_i(t) > 0\) formalised that \(\tau_j\) is free or not to execute.

We only consider integer times since all the decisions are taken at integer dates. This is the same assumption as Guan et al. [14] who have shown that a discrete model checking is sufficient.

Example 1. Figure 7 shows the real execution as a Gantt diagram or a sequence of configurations of a task set and a fixed priority policy on two processors.

Finiteness of the configuration space.

The first lemma shows that the behaviour is repetitive once all the tasks are awake.

Lemma 1. \(\forall i \in [1, n]\) and \(\forall t \geq \max_{\leq n}(O_i)\), we have:

\[ T_i^\tau(t + H) = T_i^\tau(t) + H = D_i^\tau(t) \]

where \(H = lcm_{\leq n}(T_i)\) is the hyper-period.

Proof. This is straightforward from the definition of \(T_i^\tau\).

Let \(t \geq \max_{\leq n}(O_i)\) and \(i \in [1, n]\), we have \(T_i^\tau(t) = T_i + ((t - O_i) \mod T_i)\). Thus, \(T_i^\tau(t + H) = T_i + ((t + H - O_i) \mod T_i) = T_i^\tau(t)\) since \(H \mod T_i = 0\). Moreover \(D_i^\tau(t) = \max(0, T_i^\tau(t) - (T_i - D_i))\).

Property 1. There is a finite number of states. It is sufficient to study at most the window \([0, \max_{\leq n}(O_i) + p \times H]\) where \(p = \prod_i(C_i + 1)\) to analyse the task set feasibility.
PROOF. It is simply an enumeration question. We count the number of possible states at time $t = \max_{i \leq n} (O_i)$ since we know that $O_i^c(t) = 0$, $T_i^c$ and $D_i^c$ are constant at these times. The only parameters that can change are $C_i^r \in [0, C_i]$. Thus there are $C_i + 1$ possible states for each task $\tau_i$ and $p = \prod C_i (C_i + 1)$ states for the executing system.

This ensures that the exploration will always conclude. However, it may fail in practice because of the state space size.

3. OPTIMAL FIXED PRIORITY ASSIGNMENT

In multiprocessor case, no simple algorithm is known for assigning the priorities. For instance, rate monotonic is not optimal for implicit deadline synchronous task set as illustrated in example where task $\tau_0$ has necessarily one of the 2 highest priority. Audsley’s algorithm cannot be translated, indeed the original idea is to test if a task accepts the lowest priority by applying a schedulability analysis which does not require to assign a priority to the other tasks. No such test is known for multiprocessor platforms and we necessarily need to impose the complete priorities to test the schedulability. This is the reason why Cucu and Goossens have proposed the exploration of the fixed priority assignments.

Implementation of brute force exploration.

A direct implementation is: (1) choose a complete priority assignment (2) apply an exact test. We use a complete simulation of the execution on the window $[0, H]$ for synchronous task set and for $\{S_h, S, + H\}$ for asynchronous independent task set. $n$ is the number of tasks and $S_h$ is inductively obtained when the tasks are ordered according to their priority with:

$$
\begin{align*}
S_1 &= O_1 \\
S_i &= \max\{O_i, O_i + \frac{S_i-1-O_i}{T_i} T_i\}, \forall i \in \{2, 3, ..., n\}
\end{align*}
$$

For asynchronous dependent task set, we save the configuration every max $O_i + kH$ and stop the simulation once the schedule repeats. Note that we must enforce the wct at execution to avoid scheduling anomalies and lack of predictability.

Efficient sub optimal heuristic.

The brute force method has a complexity of $n! \times c$ where $n$ is the number of task and $c$ is the cost of the verification of the schedulability of an assignment. For a task set of more than 15 tasks, the brute force method does not conclude if it has to explore the complete space. Therefore, we have encoded a suboptimal heuristic. Let us consider a periodic task set $S = \{\tau_1, \ldots, \tau_n\}$. The idea is the following:

1. the lowest priorities $n, n-1, \ldots, n-k+1$ have been given to the tasks $S' \subseteq S$.
2. consider task $\tau_k \in S \setminus S'$ which accepts the priority $n-k$; this means that the REFUSE condition described in definition is false.
   (a) Choose a random complete assignment (except for the tasks which already temporary accept a low priority)
   (b) Apply an exact schedulability test.

   i. If the test succeeds, the assignment is correct and a solution has been found.
   ii. If one task $\tau_k \in S' \cup \{\tau_j\}$ with a temporary low priority exceeds its deadline, the algorithm backtracks to the step where $\tau_k$ temporarily accepts the lowest priority and chooses a next task.
   iii. Otherwise, the algorithm continues the search. It goes in step 1, assumes the priorities $n, n-1, \ldots, n-k$ have been assigned and tries to find a task in $S \setminus (S' \cup \{\tau_j\})$ to tentatively accept the lowest priority.

The search does not visit every solution so the result can be either a solution or no conclusion.

DEFINITION 1 (REFUSE condition). Let $S = \{\tau_1, \ldots, \tau_n\}$ be a periodic task set. $\tau_1 = (T_1, D_1, C_1, O_1)$ refuses the lowest priority if:

$$
\exists t \leq D_1 + O_1 \ s.t. \ \exists i, \ t - O_i = 0 \mod(T_i), \ \exists t - O_i C_i \times \frac{t - O_i}{C_i} + C_i > t \times m
$$

EXAMPLE 2. Let us consider the following task set and a platform composed of 2 processors:

<table>
<thead>
<tr>
<th>Task</th>
<th>C</th>
<th>D</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_0$</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

The heuristic finds the solution shown on the right side. When applying the algorithm, $\tau_0$ temporarily accepts the lowest priority. Indeed: for $k = 0$ and $t = 6$, $\tau_0$ and $\tau_2$ execute exactly once. Thus, we have $(4 + 2) + 4 \leq 6 \times 2$. For $k = 1$ and $t = 4$, $(8 + 4) + 8 \leq (8 + 4) \times 2$. For $k = 2$ and $t = 7$, $(16 + 8) + 12 \leq (16 + 7) \times 2$.

As a second illustration, let us consider the following task set which is schedulable by a fixed priority policy:

<table>
<thead>
<tr>
<th>Task</th>
<th>C</th>
<th>D</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_0$</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_1$</td>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>6</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
The heuristic fails to find a solution. It is therefore suboptimal. \( \tau_j \) must have one of the 2 highest priority but none of the tasks \( \tau_0 \) and \( \tau_1 \) accept the lowest priority according to the REFUSE condition.

4. OFF-LINE SCHEDULE CONSTRUCTION

In this section we describe the exhaustive search among the behaviours for finding a valid schedule.

Optimal scheduling.

A task set is feasible for a given architecture if there exists a schedule that respects the temporal and the precedence constraints of every task (regardless of any specific policy). A scheduling algorithm is optimal, with respect to an architecture and a class of policies (e.g. preemptive/non-preemptive, static/dynamic priority, etc.), if it can schedule all the feasible task sets. Hong and Leung [15] have exhibited an example of asynchronous jobs to illustrate the need to be clairvoyant (knowledge of future events, in particular, release times) to be optimal on multiprocessor platforms.

Example 3. We exhibit a synchronous task set which cannot be scheduled by any existing policy (FP, gEDF, gLLF or LLREF) on 2 processors, while it is feasible:

| \( \tau_0 \) | 5 | 2 | 1 | 0 |
| \( \tau_1 \) | 5 | 2 | 1 | 0 |
| \( \tau_2 \) | 5 | 2 | 1 | 0 |
| \( \tau_3 \) | 8 | 5 | 1 | 0 |
| \( \tau_4 \) | 8 | 5 | 7 | 0 |
| \( \tau_5 \) | 20 | 5 | 19 | 0 |

Implementation of brute force exploration for asynchronous task sets.

We only detail the asynchronous case which is a new contribution. We have encoded the problem so that a schedule cannot be found if an automaton can reach a particular state. Thus, find a solution consists in providing a counter example to a reachability property.

\( \forall k \leq \min(m, \text{nbtaskawake}), \forall i_1, \ldots, i_k, C_{i_0} > 0 \wedge \text{schedulable} \)

\begin{align*}
\text{Update State}(i_1, \ldots, i_k), t++
\end{align*}

The automaton works as follows:

- the condition on the transition \text{SAME\_PATTERN} states that the task set must be schedulable and the previous backup is exactly equal to the current configuration, meaning that we have found a repetitive schedule.

The question asked to the model checker is whether there is a path reaching the state sched (\( E<>\text{sched} \)).

5. PERFORMANCE EVALUATION

In order to make the approach user friendly, we have developed a converter that transforms a simple task set description into equivalent UPPAAL and C models.

Task set generation.

For evaluating the performance of the proposed methods, we need to apply them on “interesting” task sets. The generation of task sets is a real challenge in the scheduling area. Bini and Buttazo [4] explain that the generation method may produce biased result. They also proposed an utilisation based generation scheme with uniform distribution.

We borrowed their ideas with several modifications due to the fact that:

- we work with integer values instead of real,
- we need to bound the hyper-period in order to avoid combinatorial explosion. This is however quite realistic since multi-periodic real-time systems do not usually have many different and/or coprime periods,
- the previous modifications generate non-uniform distribution but prevent from producing trivially not feasible task sets.

The parameters are: the number of tasks \( k \), the geometric progression of periods \( g \), the period limit \( l \) and the global utilisation \( U \). They are generated by the rules:

\begin{align*}
T_i &= t * (1 + \text{rand}(0, g)) \\
C_i &= f(U, U_{<j<i}) \\
U_i &= C_i / T_i \\
O_i &= \text{rand}(0, T_i) \\
D_i &= \text{rand}(C_i, T_i)
\end{align*}

where \text{rand}(a, b) returns a random integer value within the interval \([a, b]\); the random generator is based on Linux /dev/urandom and \( f(U, U_{<j<i}) \) is implemented by:

\begin{align*}
U_a &= \sum_{j=1}^{i-1} C_j / T_j \\
U_u &= \sum_{j=i+1}^{k} C_j / T_j \\
C_{\max} &= \min(T_i, \max(1, \text{floor}(U - U_a - U_u))) \\
I(U, U_{<j<i}) &= \text{rand}(1, C_{\max})
\end{align*}

Acceptance ratio.

For concluding on the efficiency, benchmarks rely on the acceptance ratio. Unfortunately, for multiprocessor platforms, we cannot evaluate acceptance ratio because we do not have an efficient manner to determine whether a taskset is schedulable by a fixed priority assignment (except if the brute force method concludes) or is simply feasible (unless the UPPAAL model concludes). We therefore propose to compare the results with the schedulability by other policies (gEDF and gLLF).
6. CONCLUSION

We have studied an automatic manner for providing an off-line schedule for asynchronous dependent periodic task sets using a model checker. The sub optimal heuristic for finding fixed priority assignment is very efficient and manages to schedule as many task sets as gEDF. Off-line schedule construction is not yet efficient and we will pursue our work to find a better way to construct cyclic sequences. Finally, we will also improve the task set generation to explore more feasible but not on-line schedulable task sets.

7. REFERENCES


