Hard real-time static scheduling of synchronous data flow graphs with simulated annealing

Benoît Miramond and Denis Dupont
Université d'Evry Val d'Essonne
LaMI - UMR 8042 CNRS
Tour Evry 2, 523 place des terrasses de l'agora,
91000 Evry, France
{miramond, dupont}@lami.univ-evry.fr

Keywords—**hard real-time process, static scheduling, cyclic tables, synchronous data flow graphs, simulated annealing.**

Abstract— **In this paper, we describe a scheduling optimization technique which minimizes size of the resulting static scheduling tables in embedded memory. To reduce the size of embedded tables of classical scheduling techniques, we propose to use an extended synchronous data flow (*SDF*) representation. This model permits a compact representation of periodic applications and then affords the flexibility of scheduling the tasks on a little part of the hyper-period.**
The paper presents the model and the algorithms used to find small regular structures in real-time schedules. The technique reduces 70% of tables size in embedded memory.

I. Introduction

A key feature of embedded software is the size of embedded memory which is a substantial portion of system cost.
This paper describes a novel static scheduling algorithm of embedded applications under hard real-time constraints. The main contribution of this approach is to use a compact description format of the process structure that can be transformed in order to define reduced scheduling tables.
Classical static scheduling algorithms generate big scheduling tables, it is the reason why our objective is to optimize the size of embedded memory taken by this tables as shown in figure II.
The inputs of this approach are a standard application specification (capacities, periods, deadlines and data dependencies of tasks) and a description of the computing architecture. Considered applications are multimedia and signal processing applications executing on heterogeneous multiprocessor architectures. This is a challenging software synthesis domain since it requires high performance hardware/software systems at low implementation costs. Memory size becomes an other critical resource of this kind of systems with the growing complexity of embedded applications.
When building off-line schedules, a straightforward way to implement the scheduler is to store the precomputed schedule as a table. Each entry in this table refers to a basic cycle in the system clock and contains the identification number of the process to wake-up at this instant.
In this paper, we describe the optimization technique of scheduling tables size. When it is possible, the approach significantly reduces table size (70% on average) while meeting periodic deadlines, it gives the initial table otherwise.
The next section discusses interest of static schedules in real-time context. The problem of size of scheduling tables generated by classical related approaches is illustrated. In the way to minimize tables, section 3 presents a new description model of applications with periodic constraints inspired by synchronous data flow computation model (SDF)[3]. This description model is transformed in order to find a repetitive scheduling structure (a pattern) with minimal size and that can be infinitely repeated while satisfying periods and deadlines of applications.
The heuristic is iterative, its first step analytically defines the minimal size of this pattern, this step is described in section 4.
Activation times of tasks inside the pattern are then determined after having been mapped onto the architecture (section 5). Section 6 outlines some actual results and section 7 draws conclusions and future works.

II. Motivations

Several works treat real-time scheduling on multiprocessors in literature and propose efficient (sometimes optimal) solutions in either dynamic or static scheduling contexts. In dynamic scheduling, approaches are flexible but need high run-time cost to take on-line decisions. In this context we find earliest deadline first algorithm (EDF) that find optimal solutions, or RMS algorithm [2] that reduces on-line decisions by assigning fixed (static) priorities. On an other hand static approaches are dedicated to a single

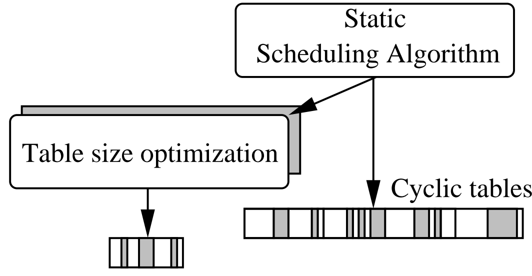


Fig. 1. Our technique takes an input schedule and generates reduced embedded table.

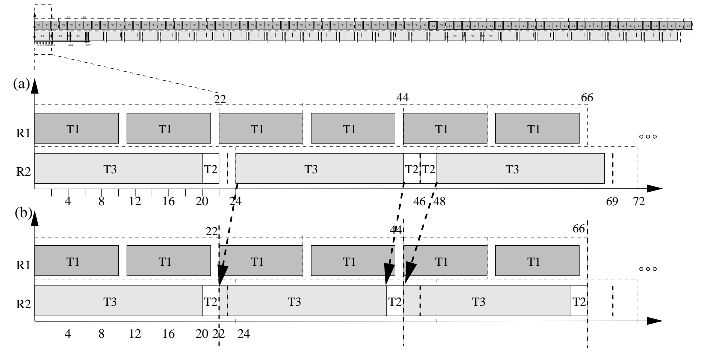


Fig. 2. Embedded scheduling tables for $T_1(10, 11)$ on R_1, and $T_2(2, 23), T_3(20, 24)$ on R_2.

application but allow deterministic behavior and because the schedule is computed off-line, we can afford to use complex, sophisticated algorithms.

However, for satisfying timing constraints in hard real-time systems, predictability of the system's behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system [7].

But most works in static scheduling context compute schedules for the entire set of periodic processes occurring within a time equal to the least common multiple (lcm) of the periods of the given set of processes. Then, at run-time, static (or pre-run-time) scheduling consists in executing the periodic processes in accordance with the previously computed schedule.

Consider the application example composed by 3 independent periodic tasks executing on a 2 homogeneous processor architecture. This tasks have the following characteristics:

- the first one (noted T_1) takes 10 processor time units to execute and must compute with a period of 11 time units. In the rest of the paper, we will use the following notation: a task T_i is such that $T_i = (c_i, p_i)$ where c_i is the capacity of the task and p_i its period. We assume that its deadline d_i is such that $d_i = p_i$. Here $T_1 = (10, 11)$,
- the next task (T_2) takes 2 time units when executing and its period is 23, $T_2 = (2, 23)$,
- and the third task is described by $T_3 = (20, 24)$.

When assigning T_1 to R_1 and T_2, T_3 to R_2, the usual utilization computation $U = \sum_{i=1}^{3} c_i/p_i$ gives $U > 90\%$ on each processor. However a valid scheduling can be found easily. We call a **valid** solution, a schedule that satisfies the deadlines of each task. The Gantt diagram of a simple solution is drawn in figure II(a). For this example, the hyper-period H of the given tasks T_i is equal to 6070 time units. The corresponding embedded table can be compared to an array of integers (here 2 tasks means 2 bits/entry), with size equal to 1518 bytes. This size is an upper bound since table can be compressed. For example, when only memorizing starting time stamps, the size is reduced to 268 bytes Now look at the figure II(b), by shifting T_2 at time 22 a regular structure appears.

This pattern increases the processors's utilization, but when repeated infinitely, sufficient time slots (CPU time) are allocated to each task invocation to be executed during its period and completed before its deadline. The size of this cyclic pattern is 9 bytes, which represents a reduction greater than 90% of memory size.

Once the table statically generated one has to choose at run-time an under set of time slots between the ones stored in the table. That choices can be done at the beginning of each pattern repetition, allowing to take minimal on-line decisions while preserving predictability. This point will be discussed in section 5.3.

The main objective of our approach is then to find a repetitive structure (pattern) that minimize the size of static scheduling tables usually computed on the hyper-period H.

The paper presentation follows the steps of our global approach (Algorithm 1). First, we build an application description graph called a Period Graph (PG) and described using our modelisation format (step 1 in algorithm). This

---

Algorithm 1 Global approach.

---

minSchedule(){
if (isSchedulable())
  (1) PG = constructPeriodGraph()
  (2) q = minimumTable() //compute q minimum
  (3) unfoldPG(q)
  (4) mappingAndScheduling(q)
  (5) computeReservationTable()
endif
}

---

representation is used by the heuristic $minimumTable()$ (step 2) to find the size of the minimal cyclic pattern. In this step are also defined the number of invocations of each task in the pattern depending on its period. By making that the structure has the desired characteristics, we can ensure that the cyclic schedule respect all real-time constraints.
Then (step 3) the initial graph PG (which represents only one invocation of each task) is unfolded on the length of the pattern.
Finally a mapping of each task invocation inside the pattern is determined (step 4) and the schedule table is generated (step 5).

III. Application modelisation

In hard real-time systems there are periodic processes and synchronous processes. A periodic process consist of a computation that is executed repeatedly, once in each fixed period of time. An asynchronous process consists of a computation that responds to internal or external events. Digital signal processing (DSP) applications represent computations on an indefinitely long data sequence. The synchronous data flow model or SDF [3] efficiently describes continuous computations and is often used in asynchronous contexts with self-timed scheduling [4]. No explicit period and deadline constraints are considered (several executions of a task can occur simultaneously if datas are available). In our context of periodic processes, iterations can't occur before the beginning of a new period interval (datas are available at fixed times) and must complete before their deadline.
In this section, we aim to build a description model based on SDF properties that explicitly express period and deadline constraints of real-time applications. This constraints are imposed so that we may identify regular structures in schedules that allow to guarantee deterministic behaviors.

A. Notations

Our model [6] represents each task $T_i = (c_i, p_i),\ i = 1, ...n$ of an application by a connected directed graph $G_i = \ < T, \Delta >$ where each actor $\tau \in T$ has the following attributes:

- a capacity $c(\tau)$, which is a worst case estimate of actor execution time,
- a start date $s(\tau, k)$ of the kth execution of actor τ,
- an end date $e(\tau, k)$ of the kth execution of τ.

Contrary to SDF, in our model all actors (or processes) of the task T_i share:

- the same period $p(\tau) = p_i$,
- and the same deadline $d(\tau) = d_i$. (We have supposed in this presentation that $d(\tau) = p(\tau)$).

Moreover, for each edge $\delta \in \Delta$, we note:

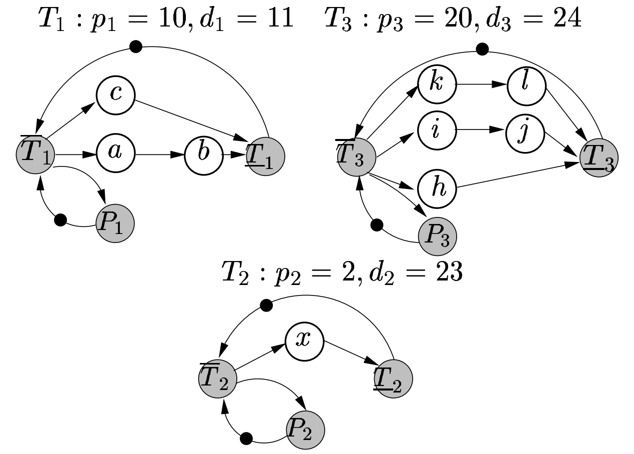


Fig. 3. Task graph representation.

- $d(\delta)$ the delay associated to the buffer δ. Edges with delays (represented using bullets on edges) can be interpreted as data dependencies across iterations of the graph.

In our representation each task graph contains 3 particular actors:

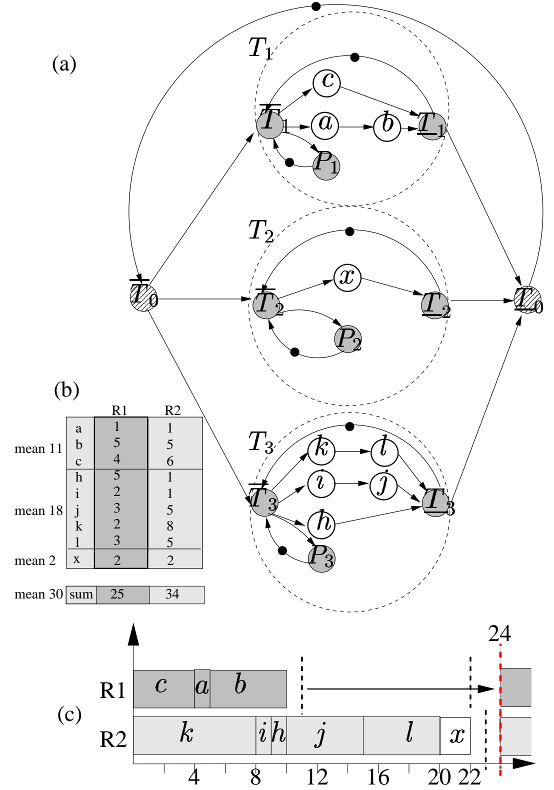
- an **activation actor** $\overline{T}_i$ with zero capacity and such that T_i is connected to each process without input data dependencies.
- a **termination actor** $\underline{T}_i$ with zero capacity and such that $\underline{T}_i$ is connected to each process without output data dependencies. $\overline{T}_i$ and $\underline{T}_i$ are linked by a one delay edge.
- an **explicit period actor** P_i with capacity equal to p_i. Its presence imposes the periodicity of T_i's processes. P_i is added in the graph such that a cycle is made with $\overline{T}_i$.

Figure 3 illustrates this model with task graphs corresponding to 3 processes structures of tasks T_1, T_2 and T_3. In the graph, the delay on edge $\delta(\underline{T}_1, \overline{T}_1)$ impose the constraint $s(\overline{T}_1, k) \geq e(\underline{T}_1, k-1)$, or $s(\overline{T}_1, k) \geq max(e(c, k-1), e(b, k-1), s(\overline{T}_1, k-1) + p_1)$. So this representation is efficient at expressing period constraints in periodic processes.

B. Application graph

In order to represent the all application, task graphs are then collected in a particular format called Period Graph PG near from the period graph of Bhattacharyya et. al [1] used in self-timed context. All tasks graphs G_i are linked in a same representation from which it will be possible to estimate throughput once mapped onto an architecture [1] (see section 5.2).
The way to build the period graph is hierarchic. Each task graph is considered as a macro-task. An activation actor

and a termination actor are added in the same way than previously. This actors will represent the synchronization of tasks at end and beginning of pattern repetitions and are consequently noted $\overline{T}_0$ and $\underline{T}_0$.
Determining the period of the entire graph is the main goal of our approach and is then not currently defined.
The period graph of tasks T_1, T_2 and T_3 in figure 4(a) illustrates this common representation. It well defines a cyclic pattern since activations of tasks are synchronized by edge $(\underline{T}_0, \overline{T}_0)$. However this first structure only represents one activation of each task. Mapping and scheduling the corresponding pattern from table 4(b) brings to the Gantt diagram in figure 4(c) which is not valid. The period (and the length) of this pattern is equal to the sum of execution times on R_2: 22 time units which is 2 times the value of p_1.
In order to determine a valid cyclic pattern we propose to modify the structure of the period graph PG by using the unfolding property of SDF graphs. The aim is to repeat each graph task G_i a number of times noted q_i inside the period graph.
In the example of figure 4(c), one would include 2 invocations of T_1 in the period graph in order to satisfy the deadline d_1 as it is shown in figure 5(b).
We describe in the next section the procedure that systematically find a vector $q = (q_1, q_2, ..., q_n)$.



		R1	R2
mean 11	a	1	1
	b	5	5
	c	4	6
mean 18	h	5	1
	i	2	1
	j	3	5
	k	2	8
	l	3	5
mean 2	x	2	2
mean 30	sum	25	34

Fig. 4. (a) Homogeneous Period Graph (HPG) of tasks of figure 3. (b) Execution times of processes on resources R_1 and R_2. (c) Gantt diagram corresponding to map T_2 and T_3 on R_2 and T_1 on R_1.

IV. Pattern recognition

A. Processors utilization bounds

Mapping and scheduling a set of tasks T_i, $i = 1, ..., n$, on a N processor architecture, without considering periodic constraints, can lead to several qualities of solution depending on whether solutions take advantage of parallelism. More precisely the lower bound of a scheduling period corresponds to the mapping of the entire set of tasks on the same processor, this bound can be written $P_{low} = \sum_{i=1}^{n} c_i$. Conversely, the theoretical upper bound of period corresponds to a uniform mapping of tasks on processors, with $P_{up} = \lceil \frac{1}{N} \sum_{i=1}^{n} c_i \rceil = \lceil \frac{1}{N} P_{low} \rceil$.
When now considering real-time applications, few scheduling satisfy period and deadline constraints. The usually computed period of such schedules is the hyper-period H whose value is equal to the lcm of tasks's periods p_i, with $P_{up} \leq H \leq P_{low}$.
If $H = P_{up}$, minimization is impossible and the input schedule is unchanged, else we will take advantage of the free time slots (difference between H and P_{up}) to find a regular repetitive scheme (pattern) whose period is P.
In order to repeat pattern all over the hyper-period, P is needed to be a divisor of H: $P_{up} \leq P \leq H$ such that $H \ mod \ P = 0$. In our context we will choose P equal to a factor of the smallest period p_1:

$$P = k \cdot p_1. \tag{1}$$

If $P \neq H$ then some of the tasks T_j with $j \neq 1$ have their period $p_j > P$. Then the needed condition to find a pattern satisfying real-time constraints is that the pattern contain one entire execution of each task T_j, which implies

$$P > max_{j \neq 1}(c_j). \tag{2}$$

Others tasks will have a period $p_j < P$. This tasks will be executed more than once inside the pattern in order to have $q_j \cdot p_j > P$. Returning to the last case, formula (2) is updated:

$$P > max_{j \neq 1}(q_j c_j). \tag{3}$$

Then a pattern will contain a number of execution of each task T_i that we note q_i. The set of repetition values of each task is stored in a vector q called repetition vector as in SDF representation, but in this vector q_i values are stored in ascending order: $q_1 \leq q_2 \leq ...q_n$. When q_1 is

fixed (to k), the number of repetitions of each tasks can be calculated from (1) and (3):

$$q_j = \lfloor \frac{q_1 \cdot p_1}{p_j} \rfloor,\ for\ j \neq 1. \quad (4)$$

B. Pattern determination

As introduced in section 2, the second step of our method determine the repetition vector q of tasks in the cyclic pattern. Since the period P depends on the number of repetitions q_1 of task T_1 (with period p_1), we must determine the good value of q_1.

The algorithm begins with an homogeneous vector (filled

Algorithm 2 Repetition vector determination.

```
(1) vector p, c, q
    minimumTable(){
     H = lcm(p_i)
     k_max = H/T_i
(5) For i from 1 to k_max do
        T = ⌈(1/N) Σ_{i=1}^{n} q_i c_i⌉;
        if (T ¡ min_{i=1}^{n}(q_i p_i))
           return q
        endif
(10)    q_1++
        updateQ()
     endfor
     return q
    }
```

with 1 values). The value of q_1 is iteratively increased and the rest of the vector values are updated with formula (4). At each iteration of the algorithm (2) we then test if the upper period bound (the maximum utilization of system's resources) is consistent with period constraints:

$$\lceil \frac{1}{N} \sum_{i=1}^{n} (q_i \cdot c_i) \rceil \leq min_{i=1}^{n}(q_i \cdot p_i). \quad (5)$$

Take figure 4(c) as an example, the initial homogeneous vector $q = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ doesn't respect constraint (5): $30/2$ is greater than $(1 \cdot p_1) = 11$. (When heterogeneous processors are used, mean values of c_i are used). Then only q_1 is increased (line 11 and 12 in algorithm 2) since $(2 \cdot q_1) < p_2$. At second iteration $q = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$. This vector respect constraint (5) since $41/2 \leq 2 * 11$.

A possible mapping of the corresponding pattern is proposed in figure 5(b).

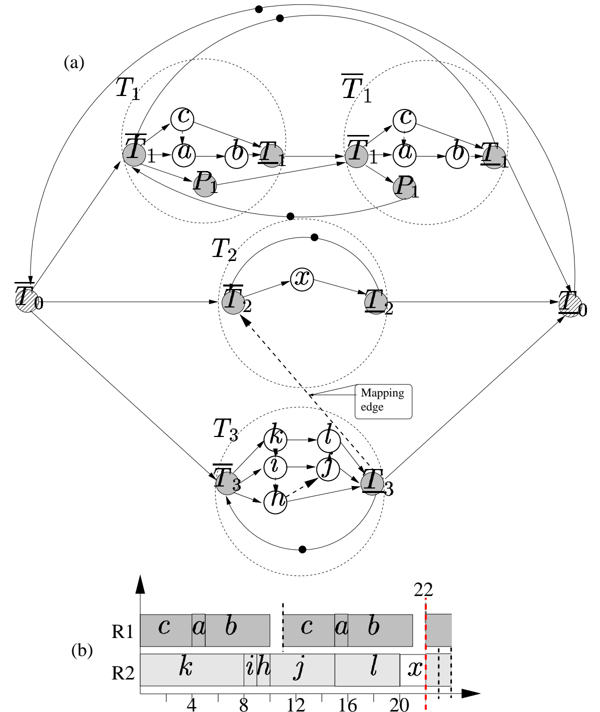


Fig. 5. (a) Unfolded Period Graph (UPG) with $q^T = (2\ 1\ 1)$. (b) A possible mapping solution if T_2 and T_3 execute on R_2 and T_1 on R_1.

This formulation ensure that there is a way to map tasks and subtasks to processors inside the pattern leading to a valid scheduling solution.

We will now use the resulting repetition vector to find this mapping.

V. Pattern mapping

Finding a valid pattern schedule amounts to find the schedule with minimum length ($P \approx P_{up}$). Also, we formulate pattern mapping as an optimization problem resolved with a local search method. Optimization criterion of this problem is the period P of the pattern computed with a maximum cycle mean calculus in a connected graph.

Consequently, the next step of our method (step 3 in algorithm 1) consists in transforming the period graph so that it represents all repetitions of tasks inside the pattern.

A. Graph unfolding

In this third step each task graph in the homogeneous period graph is repeated by the (unfolding) factor q_i stored in the previously determined repetition vector q. The unfolded period graph corresponding to the homogeneous period graph from figure 4(a) is represented in figure 5(a).

As shown in this figure, for tasks T_j, $j \neq 1$ only explicit period actors P_j of iterations $k < q_i$ are conserved from

Homogeneous Period Graph. Deleting the last repetition of period actors allows to find an integer multiple of q_1 (formula 1).
The resulting Unfolded Period Graph (UPG) can be mapped and scheduled on a given architecture. But assigning each task to a processing element adds virtual data dependencies between tasks and processes assigned to a same resource since computation is supposed to be sequential on processors. Each mapping and scheduling in the solution space is then described in our model by adding virtual edges (dashes in figures) between two consecutive tasks on processors.
For example, mapping T_1 to R_1 and T_2, T_3 to R_2 changes the graph structure in the manner depicted in figure 5(a).

B. Mapping exploration with simulated annealing

Mapping an application graph onto a multiprocessor architecture implies assigning actors to processors, ordering the actors assigned to processors, and determining activation dates. In order to establish the best (valid) task partition and edges configuration, we use the mapping method developed in [5] based on simulated annealing (SA) algorithm. This version of simulated annealing uses an adaptive cooling schedule to rapidly explore solution space.
Simulated annealing is a probabilistic algorithm based on local search strategies where optimization is made by successive partial changes called **moves**.
In our formulation a solution is defined by a mapping and a total order on each processor. For this reason, 2 types of move are used to explore solution space:

- a first move consists in changing the total order on a processor. This kind of move modifies the configuration of edges between actors assigned to a same processing element in the UPG.
- a second move consists in moving a node from one resource to another changing its execution time and the configuration of edges in the source processor and in the destination processor.

Exploration starts from an initial randomly generated mapping and ordering. Then a move is randomly selected and proposed at each algorithm iteration (for example move first execution of actor c from R_1 to R_2). If a proposed move is accepted the state of the current solution is updated. When the system is considered frozen the process is terminated. At this point, the current state of the system is the solution of our optimization problem.
The resulting solution minimizes the period P of the pattern and respect the period and deadlines constraints imposed by the period graph for a single repetition of the pattern.
To be sure that periods and deadlines will be satisfied when patterns and periods $p_j, j \neq 1$ will be shifted, only final solutions that respect the following constraints are accepted:

$$\begin{gathered} s(T_j,k) \leq s(T_j,k-1) + p_j, \ for\ j \neq 1,\ k > 1 \\ P - s(T_j,q_j) + s(T_j,1) \leq p_j \end{gathered} \tag{6}$$

This constraints impose that the interval between two consecutive task activations is no longer then a period p_i (but activations belongs to dissociated periodic intervals: $s(\overline{T}_j,k) \geq s(\overline{T}_j,k-1)+p_j$). In other words, for a task T_i there is no intervals with length greater than p_i that doesn't contain at least c_i time slots reserved to T_i.
The constraints are inserted in the optimization problem and resolved by adaptativly restricting solution space to the acceptable region [5].

C. Performance estimation

Since the method optimizes pattern length, we must estimate the period from the modified (mapped) period graph. It has been showed that the maximum cycle mean (MCM) of the period graph can be used as an efficient estimate of period [1]. Using MCM avoids performing expensive simulation sessions.
This maximum cycle mean is the maximum over all directed cycles C of the sum of the task execution times in C divided by the sum of the edge delays in C.

D. Embedded table generation

The resulting scheduling solution is then stored in a table. The size of the table is equal to the pattern period P and each entry contains the process Id to activate at the corresponding processor time unit. The table is read in a cyclic manner at execution time bringing to possible split task executions (as in figure II) that can be considered as statically defined preemptions.
In order to avoid launching extra tasks computations, we can count the time slots used in the table for each task at execution time. Then for each task T_i only the c_i first time slots are used. This run-time "scheduler" only needs $2n$ additional entries in memory: n counters and n needed time slots (c_i).

VI. Results

We ran experiments on 100 randomly generated application graphs (structure, periods, capacities and number of homogeneous processors). For each example we applied our method, found a pattern and collected informations about the size P of the pattern and the ratio between P and the length of the Hyperperiod H. Moreover we compute for each example the maximum potential uniform utilization of processors: $U_{max} = \frac{1}{N}\sum_{i=1}^{n}\frac{c_i}{p_i}$. We plot in figure

6 the distribution of solutions, first we plot the pattern size according to U_{max} and the we plot the ratio ($r = 1 - P/H$) according to U_{max}. This plots show that the method allows to reduce the table size from 10% to more than 90%, the average reduction value being around 70%.
The size of the resulting patterns naturally grows with the maximum utilization of processors.

VII. Conclusion

In this paper, we have described a static scheduling method that generate reduced scheduling tables. The gain in embedded memory size can reach more than 90% in some cases. Moreover, this tables ensure to always satisfy real-time constraints by making sure the structure has the desired characteristics. The pattern determination procedure is structured in the following steps: period graph construction, pattern size determination, period graph unfolding, and unfolded period graph mapping. Our method is then build around a new real-time description model which inherits SDF properties (delayed cycles, unfolding, MCM calculus).
Real-time constraints are explicitly expressed in this representation from which it is possible to extract regular scheduling structures that minimizes the embedded scheduling informations.
This scheduling technique can be used in real-time systems where worst case estimates are reliable, allowing deterministic analysis of system behavior.

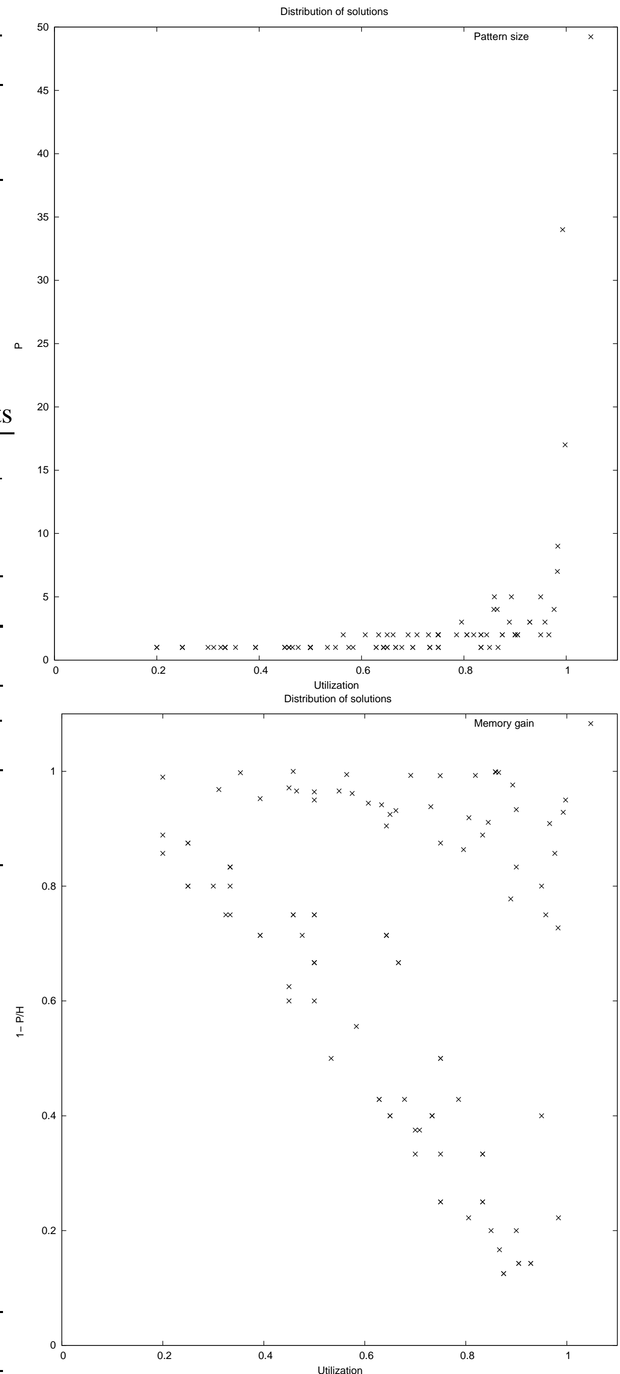


Fig. 6. Results on 100 randomly generated examples. (up) Pattern size function of the maximum potential utilization U_{max}. (down) Gain in memory size function of U_{max}.

References

[1] S. S. Bhattacharyya, N. Bambha, M. Khandelia, and V. Kianzad. Mapping DSP applications onto self-timed multiprocessors. In *IEEE Conference on Signals, Systems and Computers*, 2001.

[2] J. Layland and C. Liu. Scheduling algorithms for multiprogramming in hard real-time environment. *of the association for computing machinery*, 20(1), Jan. 1973.

[3] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, pages 1235–1245, 1987.

[4] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *Proc. Globecom*, November 1989.

[5] B. Miramond. *Global optimization method for hardware/software partitionning of applications described with hetrogeneous models (in French)*. PhD thesis, Universite d'Evry, Dec. 2003.

[6] B. Miramond and L. Pontani. Synchronous data flow representation of applications with hard real-time constraints. Technical report, Universite d'Evry, March 2004.

[7] J. Xu and D. L. Parnas. On satisfying timing constrints in hard-real-time systems. *IEEE transactions on sofware engineering*, 19(1):70–84, Jan. 1993.