# Global Multiprocessor Real-Time Scheduling as a Constraint Satisfaction Problem

Liliana Cucu, Olivier Buffet

# Global Multiprocessor Real-Time Scheduling as a Constraint Satisfaction Problem

Liliana Cucu-Grosjean & Olivier Buffet
*INRIA Nancy Grand-Est*
*615 rue du Jardin Botanique*
*54600 Villers-lès-Nancy, France*
firstname.lastname@loria.fr

***Abstract*—In this paper we address the problem of global real-time periodic scheduling on heterogeneous multiprocessor platforms. We give a solution based on a *constraint satisfaction problem* that we prove equivalent to the multiprocessor problem. A solution has to *satisfy* a set of constraints and there is no performance criterion to optimize. We propose two different CSP formulations. The first one is a basic encoding allowing to use state of the art CSP solvers. The second one is a more complex encoding designed to obtain solutions faster. With these encodings, we then study the resolution of the scheduling problem using *systematic search algorithms* based on backtracking.**

## I. Introduction

Real-time systems are systems for which the logical results of the computation as well as the time at which these results are produced are important. Nowadays real-time systems have become the focus of much study. These systems typically involve the sharing of one or more resources among various processes and such systems are modelled as finite collections of repetitive tasks, each of which generates jobs. If the jobs are generated in a predictable manner, then we deal with *periodic task systems*. Each periodic task $\tau_i$ generates jobs at each integer multiple of its period $T_i$ with the restriction that the first job is released at time $O_i$ (the task offset).

The *scheduling algorithm* determines which job[s] should be executed at each time instant and on what processor. When there is at least one schedule satisfying all constraints of the system, the system is said to be *feasible*.

*Uniprocessor* real-time systems are well studied since the seminal paper of Liu and Layland [1] which introduces a model of periodic task systems. The literature considering scheduling algorithms and feasibility tests for uniprocessor scheduling is tremendous. In contrast for *multiprocessor* platforms the problem of meeting timing constraints is a relatively new research area.

In the design of scheduling algorithms for multiprocessor environments, one can distinguish between at least two distinct approaches. In *partitioned scheduling*, all jobs generated by a task are required to execute on the *same* processor. *Global scheduling*, by contrast, permits *task migration* (i.e., different jobs of an individual task may execute upon different processors) as well as *job migration* (an individual job that is preempted may resume execution upon a processor different from the one upon which it had been executing prior to preemption). In this work we consider global scheduling.

From a theoretical and practical point of view we can distinguish between at least three kinds of multiprocessor platforms (from less general to more general). We deal with *identical processors* if all processors are identical, in the sense that they have the same computing power. By contrast, in the case of *uniform* processors, each processor $P_j$ is characterized by its own computing capacity: a job that executes on processor $P_j$ of computing capacity $s_j$ for $t$ time units completes $s_j \times t$ units of execution. Finally in the case of *heterogeneous processors* there is an execution rate $s_{i,j}$ associated with each job-processor pair: a job $J_i$ that executes on processor $P_j$ for $t$ time units completes $s_{i,j} \times t$ units of execution. The heterogeneous processors model dedicated processors (e.g., $s_{i,j} = 0$ means that $P_j$ cannot serve job $J_i$).

**Related research.** The problem of scheduling periodic task systems on several processors was originally studied in [2]. Recent studies provide a better understanding of that scheduling problem and provide first solutions. E.g., [3] presents a categorization of real-time multiprocessor scheduling problems.

The difficulty when one studies multiprocessor scheduling comes from the fact that uniprocessor feasibility results do not always hold for multiprocessor scheduling. For instance the synchronous case (i.e., considering that all tasks start their execution synchronously) is not a worst case for all asynchronous situations upon multiprocessors. Initial results indicate that real-time multiprocessor scheduling problems are typically not solved by applying straightforward extensions of techniques used for solving similar uniprocessor problems because of *scheduling anomalies* [4].

### *Contribution of this paper*

In this paper we propose a solution based on constraint satisfaction problems. A similar approach has already been used for multiprocessor real-time scheduling but in the partitioned case [5]. To our best knowledge this is the first paper using constraint satisfaction techniques to solve the global multiprocessor real-time scheduling problem.

*Organization of the paper*

The paper is organized as follows. Section II provides the model of tasks and its associated notations. In Section III we present to the unfamiliar reader notions on constraints satisfaction problem and we define the constraints of a multiprocessor global real-time scheduling problem. The first CSP formulation is provided in Section IV and the second one in Section V. Possible extensions are formulated in Section VI. The experiments associated to the two CSP formulations are presented in Section VII. Discussion on obtained results as well as on future work are provided in the last section of the paper.

## II. Model and associated notations

We consider the global preemptive[1] scheduling of periodic tasks on heterogeneous processors. A task system $\tau$ is composed by $n$ tasks and each task is characterized by a 4-tuple $(O_i, C_i, D_i, T_i)$ where:

- $O_i$ is the *offset* of task $\tau_i$;
- $C_i$ is the *worst-case execution time* (WCET) of task $\tau_i$;
- $T_i$ is the *period* of task $\tau_i$. Let $T_{max} = \max\{T_1, T_2, \cdots, T_n\}$ ;
- $D_i$ is the *deadline* of task $\tau_i$, i.e., each job $k$ generated at time $O_i + (k-1)T_i$ must finish its execution before $D_i + O_i + (k-1)T_i$.

In this paper we consider the case of constrained deadline task systems, i.e., $D_i \leq T_i, \forall i \leq n$ and the case of arbitrary deadline task systems. The time being discrete, all these parameters take integer values.

A solution of the *Multiprocessor Global Real-Time Scheduling* (MGRTS) problem for any task system $\tau = \{\tau_1, \ldots, \tau_n\}$ and any set of $m$ processors $\{P_1, \ldots, P_m\}$ is defined by a *schedule* $\sigma : \mathbb{N} \rightarrow \{0, 1, \ldots, n\}^m$ where $\sigma(t) \stackrel{\text{def}}{=} (\sigma_1(t), \sigma_2(t), \ldots, \sigma_m(t))$ with

$$\sigma_j(t) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if there is no task scheduled on } P_j \\ & \text{at instant } t; \\ i, & \text{if } \tau_i \text{ is scheduled on } P_j \text{ at instant } t; \end{cases}$$

$\forall 1 \leq j \leq m$.

We consider that the parallelism is forbidden, i.e., a task is scheduled at any time instant on at most one processor. Moreover, any processor can execute at any time instant at most one task.

A feasible schedule is a schedule with all deadlines met for all jobs of all tasks. Therefore in a feasible schedule and from the definition of the deadline, a job $k$ of a task $\tau_i$ must be scheduled within its *availability interval* $[O_i + (k-1)T_i, D_i + O_i + (k-1)T_i)$.

**Example 1.** *In the remainder of the paper, we use the following running example:* $m = 2$*,* $n = 3$*, and the tasks are defined by:*

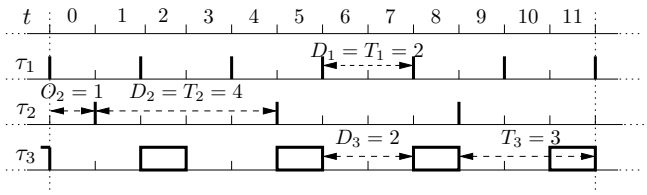


Figure 1. Representation of the availability intervals of example 1's tasks during one hyperperiod (with $O_1 = O_3 = 0$)

| $\tau_i$ | $O$ | $C$ | $D$ | $T$ |
|---|---|---|---|---|
| $\tau_1$ | 0 | 1 | 2 | 2 |
| $\tau_2$ | 1 | 3 | 4 | 4 |
| $\tau_3$ | 0 | 2 | 2 | 3 |

Let $T$ be the least common multiple of all periods: $T = lcm(T_1, \cdots, T_n)$. Since the same pattern of availability intervals repeats every $T$ time instants for any task set, $T$ is the *hyperperiod* of the availability intervals. In Example 1, the hyperperiod is $T = 12$ and the pattern of availability intervals is the one shown on Figure 1.

We denote:

- $I_{i,k}$ the $k^{\text{th}}$ availability interval for task $\tau_i$:

$$I_{i,k} = [O_i + (k-1)T_i, O_i + (k-1)T_i + D_i - 1];$$

- $U = \sum_i \frac{C_i}{T_i}$ the *utilization factor* for a given problem (a necessary condition for a problem to be feasible is that $U \leq m$); and
- $r = \frac{U}{m}$ the *utilization ratio* for a given problem (so that the above condition can be rewritten $r \leq 1$).

## III. Finding a feasible schedule

Since we deal with constrained deadline systems and because of the existence of an availability intervals pattern, there exists a feasible schedule for all jobs of all tasks if and only if there exists a feasible finite schedule within an interval of length $T$. This schedule is obtained by repeating the finite schedule infinitely. A more formal discussion is given in the proof of Theorem 1.

By focusing our effort on periodic solutions, we ensure that a finite number of unknown variables is sufficient, which is a prerequisite to translate the MGRTS problem to a constraint satisfaction problem.[2]

Before proposing formulations of the MGRTS problem as a CSP, this section gives some background on CSPs: how they are defined and how they can be solved exactly.

[1] A task is preemptive if it can be interrupted at any time.

[2] With aperiodic sets of 2D tiles, there exist non-periodic tilings of the plane but no periodic tilings.

### A. Constraint Satisfaction Problems

As our focus is on real-time scheduling, we deal with *Constraint Satisfaction Problems* (CSPs) [6; 7]. A CSP is defined by a set of *variables* $X_1, \cdots, X_p$ — each defined on a *domain* $D_i$ — and a set of *constraints* $C_1, \cdots, C_q$ involving one or several variables. A *state* is a partial or complete assignment of values to variables. Solving a CSP means either 1- finding a solution, i.e. a complete state satisfying all constraints, or 2- making sure that no valid/consistent solution exists.

A real-world problem can often be formalized as a CSP in various ways and various resolution strategies can be envisioned. The difficulty is then to find the best combination.

In our setting (MGRTS), the domains are finite sets: variables take positive (and bounded) integer values, and constraints take the form of (linear) equalities or inequalities.

### B. Exactly Solving a CSP

The search space being finite (because we have a finite number of variables each with a finite domain), a systematic generate-and-test procedure can be employed. This can be achieved for example with a depth-first search starting from the "empty-assignment" state and trying all possible value assignments for variable $X_i$ when at depth $i$ (the algorithm backtracks when a constraint is violated). This search algorithm can be improved in various ways, such as:

- *propagating constraints*, i.e. using constraints to reduce the variables' domains; this process can be iterated repeatedly until this "chain reaction" has no more effects; one issue is that searching for applicable constraints and applying them can be time consuming and hinder the benefit of this process;[3]
- *ordering the variables* to prune the search space more efficiently; this can be achieved for example by considering the most constrained variables first and by grouping variables linked by some constraints;
- *ordering the values* in each variable's domain so as to find a consistent solution earlier;
- *adding constraints* to reduce the size of the search space if this does not make the problem unsolvable.

A given real-world problem may be formulated in various ways. This leads to a choice that can change the complexity of the scheduling task.

### C. Definition of the multiprocessor global real-time scheduling problem (MGRTS)

The following two sections propose two different formulations based on constraint satisfaction problem for the considered multiprocessor global real-time scheduling problem. We denote by MGRTS-ID the problem of finding a feasible schedule for a periodic task system $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$ upon $m$ identical processors $P_1, P_2, \cdots, P_m$. Each task $\tau_i$ is given by $(O_i, C_i, D_i, T_i)$ and we have $D_i \leq T_i, \forall i \leq n$. Finding a feasible schedule implies that the following conditions are met:

- C1 Each job of task $\tau_i$ is scheduled within its corresponding availability interval.
- C2 At time $t$, on processor $P_j$, at most one task is running.
- C3 At time $t$, task $\tau_i$ runs on at most one processor.
- C4 Task $\tau_i$ should last exactly $C_i$ during each availability interval.

[3]Note: Sudoku games are often solvable with constraint propagation and no backtracking.

## IV. CSP ENCODING #1

In this first proposition our objective is to find a generic model that could be tested on various state-of-the-art CSP solvers. This leads to focusing on boolean variables so that even boolean satisfiability (SAT) solvers could be used.

### A. Variables

This CSP uses one binary variable $x_{i,j}(t)$ per task $\tau_i$, processor $j$ and time step $t$ — where $t \in 1..T$ — indicating whether task $\tau_i$ runs on processor $j$ at time $t$:

$$x_{i,j}(t) = \begin{cases} 1 & \text{if } \tau_i \text{ on } P_j \text{ at } t; \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

We therefore have $n * m * T$ variables, each taking one of two possible values ($D_{i,j}(t) = \{0, 1\}$, where 0=false and 1=true).

### B. Constraints

With these variables, we define the following constraint satisfaction problem that we denote by CSP1:

$$x_{i,j}(t) = 0, \ \forall t \notin I_{i,1} \cup \cdots \cup I_{i,\frac{T}{T_i}}; \quad (2)$$

$$\sum_i x_{i,j}(t) \leq 1; \quad (3)$$

$$\sum_j x_{i,j}(t) \leq 1; \quad (4)$$

$$\sum_{t \in I_{i,k}} \sum_j x_{i,j}(t) = C_i, \ \forall k \in 1..\frac{T}{T_i}. \quad (5)$$

As it can be observed, constraint propagation can here be very efficient:

- with constraint (2), the values of out-of-interval variables can be set before backtracking; then, the number of "real" variables goes from $\sum_i m * (\frac{T}{T_i} * T_i)$ down to $\sum_i m * (\frac{T}{T_i} * D_i)$;
- with constraints (3) and (4), if task $\tau_i$ is set to run on processor $j$ at time $t$ ($x_{i,j}(t)$), this sets the value of at least:
  - the $n-1$ variables $x_{i',j}(t)$ for $i' \neq i$, and
  - the $m-1$ variables $x_{i,j'}(t)$ for $j' \neq j$.

We prove in Theorem 1 that solving CSP1 is equivalent to solving MGRTS-ID.

**Theorem 1.** *$x^0$ is a solution of CSP1 if and only if $\sigma : \mathbb{N} \to \{0, 1, \ldots, n\}^m$ with $\sigma_j(t) = \sigma_j(t + kT) = \begin{cases} i_0, & \text{if } x^0_{i_0,j}(t) = 1; \\ 0, & \text{otherwise.} \end{cases}$ $\forall k \geq 1$ and $\forall t \geq 0$ is a solution of MGRTS-ID.*

*Proof:* Since $x^0$ is a solution of CSP1 then $x^0$ satisfies all constraints (2) - (5) given in Section IV-B. Based on the definition of $\sigma$ and because of (2) and (5) each job of task $\tau_i$ is scheduled within its availability interval and it lasts for exactly $C_i$. Moreover constraint (3) implies that at time $t$ on processor $P_j$ at most one task is running. Constraint (4) implies that at time $t$ task $\tau_i$ runs on at most one processor.

We may observe that by construction of schedule $\sigma$, this schedule is periodic and it is build from 0 that we define as the time instant when the schedule repeats. The existence of this time instant is ensured by the periodicity property of any feasible schedule of constrained deadline task systems [8]. ■

We may observe that the solution proposed by Theorem 1 considers the worst-case execution of tasks. If any job of a task does not need the entire amount of time, then the processor is considered idled in order to avoid scheduling anomalies.

### C. Search Strategy

We do not discuss possible search strategies related to this model. We see in Section VII that we have decided to leave this problem in the hands of a generic CSP solver.

## V. CSP ENCODING #2

In this second proposition we look for both a CSP formulation and a search strategy that would make the search more effective. This leads to reducing the number of variables and using symmetries of the problem to restrict the search space.

### A. Variables

To reduce the number of variables, one has to replace binary (true/false) variables with multi-valued variables telling for example on which processor a task is running at a given time instant, or conversely which task is running on some processor at a given time instant.

In this CSP, we use one n-ary variable $x_j(t)$ per processor $j$ and time step $t$ — where $t \in 1..T$ — indicating which task $\tau_i$ ($-1$ if no task) runs on processor $j$ at time $t$:

$$x_j(t) = \begin{cases} i & \text{if } \tau_i \text{ on } P_j \text{ at } t; \\ -1 & \text{otherwise.} \end{cases} \quad (6)$$

We therefore have $m * T$ variables, each taking one of $n+1$ possible values ($D_j(t) = \{-1, 1, 2, \cdots, n\}$).

By reducing the number of variables and because some constraints are induced by the choice of the variables and their domains, the problem to solve is defined only by conditions (C1), (C3) and (C4) (see Section III-C).

### B. Constraints

With these variables, we define the following constraint satisfaction problem that we denote by CSP2:

$$x_j(t) \neq i, \ \forall t \notin I_{i,1} \cup \cdots \cup I_{i,\frac{T}{T_i}}; \quad (7)$$

$$x_j(t) = x_{j'}(t) \Leftrightarrow x_j(t) = -1; \quad (8)$$

$$\sum_{t \in I_{i,k}} \sum_j \delta(i, x_j(t)) = C_i, \ \forall k \in 1..\frac{T}{T_i}; \quad (9)$$

where $\delta(a, b) = \begin{cases} 1 & \text{if } a = b, \\ 0 & \text{if } a \neq b. \end{cases}$

We prove that the new formulation is equivalent to CSP1 and therefore equivalent to MGRTS-ID.

**Theorem 2.** *CSP1 has a solution if and only if CSP2 has a solution.*

*Proof:* Let be $x^0$ a solution of CSP1. We build one variable $x^1$ with $x^1_j(t) = \begin{cases} i & \text{if } x^0_{i,j}(t) = 1 \\ -1 & \text{otherwise.} \end{cases}$. Note that this construction is a bijection. The constraints of CSP1 are directly obtained from the conditions of CSP2 and the variables' domains and vice versa. ■

### C. Search Strategy

As explained in Section III-B, efficiently solving CSPs with a backtracking algorithm requires carefully controlling the search. We now explain the choices made in our implementation.

*1) Ordering the Variables:* Considering the variables, the main decision is to order them first in chronological order. Even if time is cyclic (periodic modulo $T$) in our problem, this ordering should be beneficial by ensuring that new decisions are taken given the knowledge of most past events.

As only identical processors are considered, all processor orderings are equivalent. So we just order them according to their id number.

*2) Ordering the Values:* The values correspond to the possible tasks ($1..n$) or to the absence of any task ($-1$). Various heuristics can be envisioned to order the tasks, such as:

- RM: tasks with smallest period first (Rate Monotonic);
- DM: tasks with smallest deadline first (Deadline Monotonic);
- T-C: tasks with smallest $T - C$ value first;
- D-C: tasks with smallest $D - C$ value first.

We do not mention the "no task" value ($-1$) now, as it is considered as a special case in the next section.

*3) Adding Constraints:* We are here looking for constraints to reduce the search space. But, instead of adding them explicitly to the CSP, we are able to encode them directly in the search strategy.

Considering the "no task" value, it does not make sense to leave a processor idle at time $t$ if some task can run on it. Thus, a first rule is: the "no task" value should be used only when no tasks are available for running.

One can also observe that, at any time step $t$, all permutations of tasks on processors are equivalent. The number of value assignments for time step $t$ can therefore be divided by up to $m!$ by applying a second rule: tasks and processors should be considered in ascending order only, i.e.:

$$(j < j') \Leftrightarrow (x_j(t) \leq x_{j'}(t)). \tag{10}$$

## VI. Extensions

The following extensions have not been implemented for now, but we propose the corresponding formulations.

### A. What if Processors are Heterogeneous ?

Let us now add the constraint that processors are heterogeneous. How should we adapt our two CSPs to take this heterogeneity into account?

In both CSP encodings, moving from homogeneous to heterogeneous processors first means rewriting constraint (C4). But if some processor $P_j$ cannot execute a task $\tau_i$, we should also modify some variables' domains.

*1) CSP1:* In CSP1, the heterogeneity leads to redefining the domains for each variable $x_{i,j}(t)$:

$$D_{i,j}(t) = \begin{cases} \{0,1\} & \text{if } s_{i,j} > 0, \\ \{0\} & \text{if } s_{i,j} = 0; \end{cases}$$

and rewriting constraint (5) as

$$\sum_{t \in I_{i,k}} \sum_{j} s_{i,j} x_{i,j}(t) = C_i, \ \forall k \in 1..\frac{T}{T_i}. \tag{11}$$

We do not mention search strategies here since it was decided to let a generic solver do the work.

*2) CSP2:* Similarly, in CSP2, this heterogeneity translates into a similar change in the domain definitions:

$$\forall t, \ s_{i,j} > 0 \Leftrightarrow i \in D_j(t);$$

and into rewriting constraint (9) as

$$\sum_{t \in I_{i,k}} \sum_{j} s_{i,j} \delta(i, x_j(t)) = C_i, \ \forall k \in 1..\frac{T}{T_i}. \tag{12}$$

Moreover, the search strategies defined when designing CSP2 need to be revised because of the processor heterogeneity:

- The ordering of variables should now take care of the processors' heterogeneity. One suggestion is to put less capable processors first in order to prune the search tree as early as possible. Processor $P_j$'s quality $Q(P_j)$ could be measured on the basis of its performance for each task in the problem:
$$Q(P_j) = \sum_i s_{i,j} \frac{C_i}{T_i}.$$
- The ordering of values could now put a higher priority on tasks that can run on few processors.
- Avoiding "no tasks" is the same problem as before, except that, for processor $P_j$, only the tasks that can run should be considered.
- Permuting tasks on processors is not as easy as it was. Constraint (10) can be adapted by restraining it to pairs of identical processors $P_j$ and $P_{j'}$ (noted $P_j \sim P_{j'}$). This is appropriate since the variable ordering groups identical processors together.
$$(j < j' \,\&\, P_j \sim P_{j'}) \Leftrightarrow (x_j(t) \leq x_{j'}(t)). \tag{13}$$

### B. What if task systems are arbitrary deadline ?

Up to now we have only considered settings where, for each task $\tau_i$, the duration of the availability interval is less than or equal to the period ($\forall i \in [1..n] \ D_i \leq T_i$), case commonly known as constrained deadline task systems. Let us now consider that this assumption is not met, i.e., we deal with arbitrary deadline systems. In this new setting, two instances of some task $\tau_i$ could run simultaneously on two separate processors. The proposed constrained satisfaction problems cannot model this situation.

To solve this problem, the CSPs need to distinguish multiple instances of the same task $\tau_i$. We call these $k_i$ instances *clones* of $\tau_i$ and note them $\tau_{i,i'}$, where $i'$ is their identification number as a clone. To know how many clones of $\tau_i$ are required, we need to count how many instances of $\tau_i$ could be allowed to run at the same time, i.e. how many availability intervals of $\tau_i$ can intersect simultaneously. The answer is simply $k_i = \lceil D_i/T_i \rceil$.[4]

A second question is how to define each clone $\tau_{i,i'}$ ($i' = 1..k_i$), i.e. what values to choose for the 4-tuple $(O_{i,i'}, C_{i,i'}, D_{i,i'}, T_{i,i'})$ ?

- $O_{i,i'}$: The availability intervals should be started one after another with a delay corresponding to the period $T_i$, hence $O_{i,i'} = O_i + (i' - 1) * T_i$.
- $C_{i,i'}$: For each instance, the worst-case execution time remains the same: $C_{i,i'} = C_i$.
- $D_{i,i'}$: For each instance, the duration of the availability interval remains the same: $D_{i,i'} = D_i$.
- $T_{i,i'}$: The period of a clone $\tau_{i,i'}$ should be a multiple of the original period $T_i$ and be greater than $D_{i,i'}$. The smallest such integer is $T_{i,i'} = k_i * T_i$.

To summarize, for each task $\tau_i$ (even if $D_i \leq T_i$), $k_i = \lceil D_i/T_i \rceil$ clones $\tau_{i,i'}$ have to be created, each specified by:

$$\begin{aligned} O_{i,i'} &= O_i + (i' - 1) * T_i, \\ C_{i,i'} &= C_i, \\ D_{i,i'} &= D_i, \\ T_{i,i'} &= k_i * T_i. \end{aligned}$$

[4]Where $\lceil x \rceil$ denotes the smallest integer greater than or equal to $x$.

With this, we can directly find a feasible schedule by building and solving a CSP as explained in Sections IV and V. No additional changes are required. The resulting problems have an increased complexity because of 1- the larger number of tasks to schedule and 2- the often longer hyperperiod. The schedule obtained from a CSP formulation is feasible for all jobs because of the periodicity property of any feasible schedule of arbitrary deadline task systems [9].

## VII. Experiments

For these experiments, we have decided to implement:

- CSP1 with a generic CSP solver, `Choco` [10],[5] using its default search strategy; and
- CSP2 in C++, using our own search strategy.

This choice makes it possible to compare our hand-made search strategies with generic search strategies from a state-of-the-art CSP solver. Plus, it turns out that the first implementation (CSP1, almost bug-free because it relies on an existing solver) has helped debugging the second implementation (CSP2) by comparing their respective results: some bugs are rare and hardly noticeable.

Each experiment was run on one core of a Core2Quad CPU at 2.4GHz.

### A. Randomly Generating Problems

These experiments are conducted on random problems. Generating these problems requires ensuring that i) the constraint $0 \leq C_i \leq D_i \leq T_i$ is satisfied; and ii) $1 < m < n$ (otherwise the problem is too easy to solve).

When randomly generating problems, we make the choice of specifying:

- $n > 2$;
- $m \in 1..(n-1)$ — although it could be uniformly sampled — as this provides a better control over the problem's difficulty;
- $T_{max} > 1$ the maximum period:[6] $\forall i \in 1..n\ 1 \leq T_i \leq T_{max}$.

Now that we have these global parameters, one choice remains to be done: for each task, in which order should its parameters be selected. For a task $\tau_i$, $O_i$ is independent of other parameters, but $C_i$, $D_i$ and $T_i$ are dependent on each other. Any of the 3! orderings over these three parameters is possible, but results in a different distribution over problems, e.g.:

- $C_i \rightarrow D_i \rightarrow T_i$ favors large periods since $C_i \sim U(1..T_{max})$, $D_i \sim U(C_i..T_{max})$ and $T_i \sim U(D_i..T_{max})$;
- $T_i \rightarrow D_i \rightarrow C_i$ favors short WCETs since $T_i \sim U(1..T_{max})$, $D_i \sim U(1..T_i)$ and $C_i \sim U(1..D_i)$;

where $X \sim U(min..max)$ means that $X$ is sampled uniformly over $min..max$. Our choice is the intermediate solution of sampling $D_i$ first, then $C_i$ and $T_i$ in any order (they are independent given $D_i$).

One could find a solution so that all problems have equal chances of being sampled, but this does not give any guarantee that the result is representative of a real-world distribution.

### B. Randomness & Reproducibility

A first observation is that our CSP2 solver is completely deterministic — restarting the same algorithm twice on the same problem returns the same outcome — whereas `Choco` appears to use a randomized search algorithm, so that multiple executions of the CSP1 solver on a given problem may return different outcomes.

In our setting, obtaining one feasible scheduling or another feasible schedule does not matter much. A real issue is that, for a given problem, some executions of the CSP1 solver may be very quick while others are very slow. We do not provide statistics about this phenomenon, but it has been clearly observed, some problems being more likely to be solved quickly.

### C. Comparing CSP1 and CSP2

The objective of the first experiment is to figure out which of our various CSP solvers is the most efficient. We therefore experiment with CSP1, CSP2 and four variants of CSP2 (each with a different task ordering). The experiments are conducted on 500 random problems with $m = 5$, $n = 10$, $T_{max} = 7$ and a maximum resolution time of 30 seconds (we are talking here about the solver's *resolution time*, not a schedule's *execution time*). We do not filter out problems which, obviously, cannot be solved because there are not enough processors for all the jobs (i.e. whose utilization ratio $r$ is greater than 1).

Table I gives, for each solver, the number of runs that reached the 30 seconds time limit. We distinguish instances *solved* by at least one solver from *unsolved* instances[7]. It is clear that CSP2+(D-C) is here the best approach: only 12 out of 295 instances are solved by another algorithm and not by CSP2+(D-C). Interestingly, all CSP2 approaches are equally bad when it comes to unsolved problems.

| # overruns | CSP1 | CSP2 | +RM | +DM | +(T-C) | +(D-C) | Total |
|---|---|---|---|---|---|---|---|
| solved | 202 | 133 | 115 | 111 | 34 | 12 | 295 |
| unsolved | 205 | 189 | 189 | 189 | 189 | 189 | 205 |

Table I
Number of runs reaching the 30s time limit

Table II details the results for unsolved instances by distinguishing those instances that could have been pruned out from other instances because $r > 1$. A positive point

[5]This solver is implemented in Java.

[6]The resulting hyperperiod $T$ is likely to be greater than $T_{max}$.

[7]The fact that no solver found a solution in 30 seconds does not mean necessarily that no solution exist.

is that a large proportion of unsolvable instances (183 out of 205) can be easily detected. A negative point is that, out of the 22 unfiltered — and unsolved — instances, only 3 are provably unsolvable (the algorithm stopped before the 30 seconds for all other instances).

| # overruns | CSP1 | CSP2 | +RM | +DM | +(T-C) | +(D-C) | Total |
|---|---|---|---|---|---|---|---|
| filtered | 183 | 170 | 170 | 170 | 170 | 170 | 183 |
| unfiltered | 22 | 19 | 19 | 19 | 19 | 19 | 22 |

Table II
NUMBER OF *unsolved* RUNS REACHING THE 30S TIME LIMIT

An interesting observation is that most executions not reaching the 30 seconds limit last less than 1 second. In particular, a number of instances lead to an overrun with some solvers but are treated (solved or proved unsolvable) in less than a second by other solvers.

### D. Where are Difficult Problems

Table III shows how the utilization ratio $r$ is distributed over the 500 generated problems and how the average resolution time (over all solvers) evolves with $r$.

| $r_{min}$–$r_{max}$ | $\#_{instances}$ | $\overline{t_{res.}}$ |
|---|---|---|
| 0.0–0.4 | 0 | – |
| 0.4–0.5 | 2 | 5.0 |
| 0.5–0.6 | 4 | 2.1 |
| 0.6–0.7 | 29 | 6.5 |
| 0.7–0.8 | 79 | 7.7 |
| 0.8–0.9 | 98 | 10.7 |
| 0.9–1.0 | 105 | 18.7 |
| 1.0–1.1 | 87 | 28.5 |
| 1.1–1.2 | 51 | 29.1 |
| 1.2–1.3 | 35 | 28.1 |
| 1.3–1.4 | 7 | 30.0 |
| 1.4–1.5 | 1 | 30.0 |
| 1.5–1.6 | 1 | 30.0 |
| 1.6–1.7 | 1 | 30.0 |
| 1.7–2.0 | 0 | – |

Table III
DISTRIBUTION OF THE 500 INSTANCES ACCORDING TO THEIR UTILIZATION RATIO AND AVERAGE RESOLUTION TIME

The distribution of problem instances is clearly centered around the 0.9–1.0 interval. This could be changed by modifying the random generation (e.g. by increasing or decreasing $m$). More related to the solvers is the fact that the average resolution time increases with $r$. This reflects the fact that the systematic search i) is less and less likely to find a feasible solution early and ii) has more and more difficulties to quickly prove that a problem is unsolvable.

### E. Increasing the Number of Tasks

In this second set of experiments, we consider problems with a larger maximum period $T_{max} = 15$ and a varying number of tasks ($n \in \{4, 8, 16, 32, 64, 128, 256\}$). For each problem, the number of processors is chosen as the minimum value such that the problem is not pruned out:

$$m_{min} = \left\lceil \sum_i \frac{C_i}{T_i} \right\rceil.$$

We limit our study to CSP1 and CSP2+(D-C), running them on 100 instances in each setting.

Table IV presents the numerical results. The first columns give, for each $n$, the average utilization ratio $\overline{r}$, the average number of processors $\overline{m}$ and the average hyperperiod $\overline{T}$ of the generated problems. One can observe that $\overline{r}$ converges to 1, the average number of processors growing linearly with the number of tasks. The average hyperperiod also converges to the maximum value $T_{max} = 2^3 \times 3^2 \times 5^1 \times 7^1 \times 11^1 \times 13^1 = 360360$.

| $n$ | $\overline{r}$ | $\overline{m}$ | $\overline{T}$ (×1000) | CSP1 solved | CSP1 $\overline{t_{res.}}$ | CSP2+(D-C) solved | CSP2+(D-C) $\overline{t_{res.}}$ |
|---|---|---|---|---|---|---|---|
| 4 | 0.74 | 2.15 | 2.60 | 29% | 19.52 | 81% | 0.01 |
| 8 | 0.84 | 3.56 | 2.79 | 1% | 29.58 | 66% | 0.05 |
| 16 | 0.93 | 6.87 | 111.21 | 0% | 30.00 | 10% | 0.02 |
| 32 | 0.96 | 13.02 | 285.29 | – | – | 0% | 0.00 |
| 64 | 0.98 | 25.82 | 345.95 | – | – | 0% | 0.00 |
| 128 | 0.99 | 51.07 | 360.36 | – | – | 0% | 0.00 |
| 256 | 0.99 | 101.28 | 360.36 | – | – | 0% | 0.00 |

Table IV
EXPERIMENTS WITH A GROWING NUMBER OF TASKS

The CSP1 solver quickly shows its limits: it suffers from many overruns and runs out of memory on "large" instances. CSP2+(D-C) scales much better: it does not suffer from any overrun. The decreasing success reflects the fact that less instances are solvable when $n$ increases (probably due to $\overline{r}$ converging to 1). It would be interesting to use an algorithm which incrementally searches for the smallest number of processors $m$ required to schedule a given set of tasks.

Note that increasing the largest period should not be done without taking special care as the hyperperiod increases at a very fast rate.

## VIII. DISCUSSION AND FUTURE WORK

This work is a first step towards efficiently applying classical search techniques to MGRTS problems. They can be restated as Constraint Satisfaction Problems in various ways, leaving us with the task of finding the most efficient encoding and search strategy. The experimental results show that this approach already gives satisfying results on many instances. But experiments on larger problems and/or real-world problems would, of course, be a plus.

The search algorithms employed here are systematic searches, meaning that they stop if and only if a solution is found or the problem is proved infeasible (not mentioning time limits). No formal study of the space and time *complexity* of both encodings and of there respective

search algorithms has been conducted. But one can expect to observe a combinatorial explosion with the number of tasks, the number of processors and the hyperperiod. So, it is unlikely that such approaches are reasonable for hard instances, so that future work should consider alternatives such as:

- using the same CSP formalizations with *local search algorithms*, although they won't be able to prove that a given instance is infeasible;
- considering the problem from a different viewpoint, e.g. searching for a *feasible priority assignment* among the $n!$ possible orderings of $n$ tasks. The experiments presented here indicate that CSP2+(D-C) is the best approach. This implies that an optimal priority assignment algorithm could be built starting from a first ordering based on a (D-C) criterion.
- looking at partitioning or mixed approaches.

In a longer term, one of our objectives is to move from the usual deterministic setting — where worst-case execution times are considered — to probabilistic settings — e.g. where a probability distribution over execution times is known for each task $\tau_i$. This is a very different problem that requires different approaches, but it seems to be a natural prerequisite to first study the deterministic case.

## REFERENCES


[1] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] C. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment," *JPL Space Programs Summary 37-60(II)*, pp. 28–31, 1969.

[3] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," *Handbook of Scheduling*, 2005.

[4] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, pp. 127–140, 1978.

[5] A. D. P.E. Hladik, H. Cambazard and N. Jussien, "Solving a real-time allocation problem with constraint programming," *Journal of Systems and Software*, vol. 81, no. 1, pp. 132–149, 2008.

[6] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: prentice Hall, 1995.

[7] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI magazine*, vol. 13, no. 1, 1992.

[8] L. Cucu and J. Goossens, "Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors," *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'07)*, 2007.

[9] ——, "Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems," *Proceedings of the 10th Design, Automation and Test in Europe (DATE'07)*, 2007.

[10] The Choco Team, "Choco: An open source java constraint programming library," Ecoles des Mines de Nantes, Tech. Rep., August 2008, http://choco.emn.fr/.