INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fine Tuning the Scheduling of Tasks on Posix1003.1b Compliant Systems

Nicolas Navet , Jörn Migge

No 3730

July 12, 1999

THÈME 1

ISSN 0249-6399

Fine Tuning the Scheduling of Tasks on Posix1003.1b Compliant Systems

Nicolas Navet * , Jörn Migge †

Thème 1 — Réseaux et systèmes
Projet TRIO

Rapport de recherche n ° 3730 — July 12, 1999 — 37 pages

Abstract: Posix1003.1b compliant operating systems provide two real-time scheduling policies, namely *sched_fifo* and *sched_rr*, which under some limited hypotheses are respectively the equivalent of Fixed Priority Pre-emptive (FPP) and Round-Robin (RR). In the field of processor scheduling, schedulability analysis has been extensively studied and the problem of assessing the schedulability of multi-policy systems has been recently addressed. A schedulability analysis provides valuable help for the application designer but it simply asserts whether a given configuration is *feasible* or not, in general it does not propose any feasible configurations (1) and, as stated by Gerber and Hong *"it can rarely help to tune the system (2), which is the inevitable next step"*. To address problems *(1)* and *(2)*, we propose in this study an approach using a Genetic Algorithm (GA) to best set task priorities and scheduling policies, according to a chosen criterion, on Posix 1003.1b uniprocessor systems. Moreover, it will be shown that the use of RR, in conjunction with FPP, may improve the schedulability as well as the satisfaction of additional application-dependant criteria.

Key-words: Real-Time Systems, Scheduling, Genetic Algorithm, Operating Systems, Posix1003.1b

(Résumé : tsvp)

* Loria-Inpl, Ensem, 2 avenue de la Forêt de la Haye, 54516 Vandoeuvre, France. Email: Nicolas.Navet@loria.fr

† Loria-Inpl, Ensem, 2 avenue de la Forêt de la Haye, 54516 Vandoeuvre, France. Email: Jorn.Migge@loria.fr

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : 03 87 20 35 00 - International: +33 3 87 20 35 00
Télécopie : 03 87 76 39 77 - International : +33 3 87 76 39 77

Optimisation de l'ordonnancement de tâches dans les systèmes Posix1003.1b

Résumé : Le standard Posix1003.1b définit les deux politiques d'ordonnancement temps réel *sched_fifo* et *sched_rr* qui, sous certaines hypothèses limitées, sont respectivement équivalentes à la politique FPP (ordonnancement préemptif à priorités fixes) et à Round-Robin (politique dite de temps partagé). Les analyses d'ordonnançabilité dans un contexte mono-processeur ont fait l'objet de très nombreuses études et récemment une solution pour les systèmes dans lesquels plusieurs politiques peuvent coexister, a été proposée. Une analyse d'ordonnançabilité fournit une aide appréciable au concepteur de l'application mais, en général, elle permet uniquement d'affirmer qu'un ordonnancement est *faisable* ou non, sans proposer de solutions faisables. D'autre part, comme cela a été souligné par Gerber et Hong, une analyse d'ordonnançabilité n'aide pas à l'optimisation du système qui est pourtant souvent une étape incontournable. Nous proposons dans ce rapport une approche basée sur un algorithme génétique pour fixer au mieux politiques d'ordonnancement et priorités dans les systèmes Posix1003.1b . D'autre part, nous montrons que l'utilisation conjointe de Round-Robin et de FPP augmente l'ordonnançabilité du système ainsi que la satisfaction de critères additionnels choisis en fonction de l'application.

Mots-clé : Systèmes temps réel, Ordonnancement, Algorithme génétique, Systèmes d'exploitation, Posix1003.1b

1 Introduction

Context of the study. In [43], Stankovic et al. stated that *"due to economic and portability considerations, the tendency towards the use of off-the-shelf hardware and software components to build real-time systems is increasing"*. In the field of RTOSs (Real-Time Operating Systems), numerous Posix1003.1b compliant kernels, for instance LynxOS and QNX, are commercially available with real-time features such as very fast context-switch latencies, semaphores, programmable timers and multi-level priorities [18]. Those Operating Systems (OSs) offer cost and performance-effective solutions to application designers for building real-time systems using Commercial Off-The-Shelf (COTS) OSs. The answer to the "Make-or-Buy" [3] decision is typically "buy" when short development time, portability and tool availability are matters of concern (TCP/IP networking, flash file system, HTTP server, PCMCIA drivers ...). On the other hand, the use of COTS RTOS often leads to certification problems and their use in fault-tolerant systems is problematic because their behaviour in the presence of faults is often weak [41, 30] and not well defined.
Posix (Portable Operating System Interface) is a standard which defines features and interfaces that a Unix-like OS must have. It aims to improve portability at the source code level and ease of programming. Posix 1003.1b standard [27], formerly Posix.4, defines real-time extensions to Posix mainly concerning signals, inter-process communication, memory mapped files, memory locking, synchronous and asynchronous I/O, timers and scheduling policies. Most of today's real-time operating systems conform, at least partially, to this standard although it has been criticised [37] for not possessing features appropriate to small embedded systems.
Posix 1003.1b specifies 3 scheduling policies : *sched_rr*, *sched_fifo* and *sched_other*. These policies apply on a process-by-process basis : each process runs with a particular policy and a given priority. Each process inherits its scheduling parameters from its father but may also change them at run-time.

- *sched_fifo* : fixed pre-emptive priority with fifo ordering among same-priority processes. In the rest of the paper, it will be assumed that all *sched_fifo* tasks of an application have different priorities. With this assumption and without priority change during run-time, *sched_fifo* is equivalent to the Fixed Pre-emptive Priority policy, FPP for short.

- *sched_rr* : Round-Robin policy (RR for short) which allows processes of the same priority to share the cpu. Note that a process will not get the cpu until all higher priority ready-to-run processes are executed. The quantum value may

be a system-wide constant (e.g. QNX OS), process specific (e.g. VxWorks OS) or fixed for a given priority level.

- *sched_ other* is an implementation-defined scheduler. It could map onto *sched_ fifo* or *sched_ rr*, or also implement a classical Unix time-sharing policy. The standard merely mandates its presence and its documentation. Because we cannot rely on the same behaviour of *sched_ other* under all Posix compliant operating systems, it is strongly suggested not to use it if portability is a matter of concern and we won't consider it in our analysis.

Associated with each policy is a priority range. Depending on the implementation, these priority ranges may or may not overlap but most implementations allow overlapping. Note that these previously explained scheduling mechanisms similarly apply to Posix threads with the *system contention scope* as standardised by Posix 1003.1c standard [27].
Real-time scheduling theory is based on the concept of recurrent tasks. In the context of RTOS, we define a task as a recurrent activity which is either performed by repetitively launching a process (or a thread) or by a unique process that runs in cycles.

Problem definition. A schedulability analysis for Posix 1003.1b compliant systems, such as proposed in [36] (see section 2 for a recap), simply says whether a given allocation strategy for policies and priorities is feasible or not. An optimal[1] allocation strategy, such as Audsley's algorithm [2] for FPP, has not yet been found when RR and FPP are used in conjunction. In fact, the problem does not only consist in setting the priorities as in the FPP case, but also in choosing, for each task, the policy (either RR or FPP) and the time quantum under RR. Furthermore, when several feasible solutions exist, such an algorithm would not help to choose the best solution for a particular application when additional criteria besides feasibility are taken into account.
In this paper, we investigate a method for fixing scheduling policies and priorities such as to obtain (1) a feasible solution and (2) a solution that performs well according to some additional criteria. Depending on the application, the objective can, for instance, be the minimising of end-of-execution jitter for a given task if its work has to be produced as periodically as possible or can be to maximise the freshness or the consistency of a set of input data (see 3.3.2).

[1] Optimal in the sense that if a feasible priority assignment exists, it will be found by the algorithm.

Limits of schedulability analysis The possibility of finding a usable solution on Posix1003.1b systems with the use of schedulability analysis alone has some limits that we discuss in the following.
A problem encountered with schedulability analysis on Posix1003.1b systems is that it is not always easy to find even one feasible solution and this is particularly true on heavily loaded systems with tight timing constraints and many tasks. With such systems, a great number of unsuccessful attempts can be necessary to find one feasible solution. Of course, one solution is to limit ourselves to the use of the FPP policy and in this case the Audsley algorithm enables the designer to determine an optimal priority assignment in an efficient way. However, this solution is clearly unsatisfactory because not using RR reduces the schedulability as is shown in Section 5.1.
Furthermore, a schedulability analysis is of limited help for fine tuning a system because of its necessarly pessimistic assumptions on the worst-case execution times (as pointed out in [18]) and also because its results are worst-case oriented (the only considered trajectory of the system is the most pessimistic one starting at the "critical instant"). Given a set of feasible configurations, how should we choose the best one for a particular application just by considering schedulability analysis results ? In our opinion, probabilistic properties on the behaviour of the system may add useful information when tuning an application. In this study, such information is obtained through simulation.

Overview of the approach Due to the combinatorial nature of the problem of best setting priorities and scheduling policies for a set of tasks (see 3.1 for the exact complexity of the problem) on a Posix 1003.1b compliant system, the use of an optimisation technique is required. To address this problem, we propose in this study a GA based on a hybrid deterministic/stochastic approach :

- schedulability analysis is used for distinguishing *feasible* and *non-feasible* solutions, using *worst-case execution times*.
- the quality of each feasible solution is evaluated through simulation using *stochastic execution times* that can be derived from measures taken from a prototype or from analyses [32]. The use of simulation is required because of the nature of the fitness criteria (see 3.3.2).

Our approach allows the designer to improve the quality of the solution whilst preserving the feasibility which is the basic requirement for real-time computing.

Related work. Tindell et al. proposed the use of simulated annealing for allocating tasks to processors in a real-time distributed application networked with the token protocol in [45]. In this study, the problem addressed is not the setting of priorities and policies : priorities are assigned using the deadline monotonic scheme with FPP policy, but the problem is where to locate the tasks in such a way that the solution meets physical (memory, cpu utilisation) and timing constraints (end-to-end delays). The aim of the paper was not to tackle the problem of tuning the system according to a chosen criterion but it points out that the use of an optimisation technique can be an effective approach for solving scheduling problems in the field of real-time computing. Simulated annealing was also used in [12] for finding feasible schedules with minimised jitter in the distributed case. The authors assume that the assignment of the tasks to the processors is fixed and they calculate feasible non-preemptive schedules of tasks on the local CPUs using simulated annealing with regard to access to shared resources such as the communication network. The optimisation is only based on worst-case assumptions, for example, all instances of the tasks need their worst-case execution time to finish.
In this study, we decided to use the technique of GAs because it has already been extensively used for solving scheduling problems such as Job-Shop [10, 38, 16, 13, 40], the Travelling Salesman problem [39, 22], instruction scheduling [6, 5], or scheduling on multiprocessor environments [28, 42, 14, 31, 9, 15, 46] and has proven to be successful. GA performances have been compared with other optimisation techniques such as Tabu Search [4] or Simulated Annealing [44] and the results show that GAs are never bad on average for a large variety of problems : GA is said to be a robust approach [21]. For instance, previous studies [24] (quoted in [9]) have shown that GAs are far less sensitive to parameter values than Simulated Annealing whose performances are largely dependant on the fine tuning of the parameters. Furthermore, because there is a single solution that is modified over time, Simulated Annealing is in essence a serial algorithm which is difficult to parallelize.

Working Assumptions Due to the complexity of the general problem, the following restrictions to the problem of fixing priorities and policies of a set of periodic/sporadic tasks on Posix1003.1b uniprocessor systems are placed :

1. Tasks have no jitter in their availability date.
2. Context switch latencies are neglected.
3. Tasks do not have critical sections during which they cannot be pre-empted.

4. Since a priority level without any task has no effect on the scheduling, we impose that the priority range be contiguous.

5. Two tasks obeying different policies must not share the same priority.

6. Tasks scheduled according to the *sched_fifo* policy must all be assigned different contiguous priorities and their priority must not change at run-time. As previously mentioned, with these assumptions the *sched_fifo* policy becomes FPP.

7. Tasks obeying the *sched_rr* policy are not allowed to change their priority at run-time and the quantum value is a system-wide constant.

Jitter in task availability dates (assumption 1) and context switch latencies (assumption 2) can be taken into account in the schedulability analysis developed in [36] by extending it as in [7]. Critical sections (assumption 3) can be handled by the schedulability analysis but they have not been considered in this paper for the sake of simplicity. Assumptions 5 can be partially relaxed by considering a process specific time quantum but because very few Posix1003.1b compliant OSs possess this feature, we will restrain ourselves to the system-wide quantum case.

Organisation of the paper. The paper is organised as follows : in section 2, the principles of the schedulability analysis of Posix1003.1b systems are recapitulated, section 3 is devoted to the description of the GA. In section 4, the performances of the proposed algorithm are assessed. Finally, in section 5, we will show that the RR policy can be useful in real-time applications.

2 Schedulability Analysis of Posix1003.1b systems : A Recap

Scheduling according to Posix can be described as a superposition of layers of priorities [36]. Let the set of recurrent tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ be partitioned into separate layers $\lambda_l = \{\tau_k \mid m_{l-1} < k \leqslant m_l\}$, where $1 \leqslant m_{l-1} < m_l \leqslant m$. Tasks are scheduled according to the global rule: an instance $\tau_{k,n}$ of a task τ_k in a layer λ_l is executed as soon and as long as no instance in the (higher priority) layers $\lambda_1, \ldots, \lambda_{l-1}$ is pending. Inside each layer, tasks are scheduled either according to FPP or RR.

The FPP policy is realized if an instance $\tau_{k,n} \in \lambda_l$ of task τ_k is executed as soon as and as long as no instance of (higher priority) tasks τ_j, $m_{l-1} < j < k$ in the same

layer λ_l (and higher priority layers) is pending. Under RR, a task gets repeatedly the opportunity to execute during a time slot of maximal length Ψ_k. If the task has no pending instance or less pending work than the slot is long, then the rest of the slot is lost and the task has to await the next cycle to resume. The time between two consecutive slots may vary, depending on the actual demands of other tasks, but it is bounded by $\Psi^l = \sum_{\tau_k \in \lambda_l} \Psi_k$ (plus possibly the preemption from higher priority layers) in any interval where the considered task has pending instances at any moment. If on some subinterval no instance of the task is pending, the time between two used slots can eventually be longer because the task could not use each of its slots. If the demand of other tasks is lower, this time can be shorter or even zero.

In [36], response time bounds for layered priorities have been derived in a general way, independently of a specific type of task. These bounds are based on the concept of majorizing work arrival functions, which measure the processor demand, for each task, over an interval starting with a "generalized critical instant". With $c_{k,n}$ and $a_{k,n}$ being the execution and release time of the n^{th} instance of the task τ_k after the critical instant, the majorizing work arrival function is

$$s_k(t) = \sum_n c_{k,n} \cdot \mathbb{1}_{[a_{k,n}<t]}.$$

If τ_k is a sporadic task with maximal execution time C_k and minimal interarrival time T_k, then $c_{k,n} = C_k$ and $a_{k,n} = n \cdot T_k$ $(n = 0, 1, \dots)$ and

$$s_k(t) = \sum_n C_k \cdot \mathbb{1}_{[n \cdot T_k < t]} = C_k \cdot \left\lceil \frac{t}{T_k} \right\rceil.$$

The response time bouds can be expressed as

$$\max_{n<n^*}(e_{k,n} - a_{k,n}), \tag{1}$$

where $n^* = \min\{i \mid e_{k,i} \leqslant a_{k,i+1}\}$, with $a_{k,n}$ being the n^{th} release time of τ_k after the critical instant. If τ_k is in an FPP layer, then

$$e_{k,n} = \min\{t > 0 \mid \tilde{s}_{k-1}(t) + \tilde{s}_k^n = t\}, \tag{2}$$

where $\tilde{s}_{k-1}(t) = \sum_{i=0}^{k-1} s_i(t)$ is the demand from higher priority tasks and $\tilde{s}_k^n = \sum_{i=0}^{n} c_{k,i}$ the demand from previous instances and the current instance. If τ_k is in an RR layer, then

$$e_{k,n} = \min\{t > 0 \mid \overline{\psi}_k(t) + \tilde{s}_k^n = t\}, \tag{3}$$

where

$$\overline{\psi}_k(x) = \min\left(\left\lceil \frac{\tilde{s}_k^n}{\Psi_k} \right\rceil \cdot \overline{\Psi}_k + \tilde{s}_{m_{l-1}}(x),\ s_k^*(x)\right), \tag{4}$$

with $\overline{\psi}_k = \psi^l - \psi_k$ being the sum of the quantums of all other tasks of the RR layer and

$$s_k^*(x) = \max_{u \geqslant 0}\big(s_k(u) + \widetilde{s}_{m_l}(u+x) + \overline{s}_k(u+x) - u\big), \tag{5}$$

where $\overline{s}_k(u+x) = \sum_{\tau_i \in \lambda_l, \tau_i \neq \tau_k} s_i(u+x)$. The algorithms for computing the response time bounds can be found in [36].

3 Setting priorities and scheduling policies of tasks

When setting priorities and policies, there is a constraint that must be taken into account : there may exist tasks for which the priority and/or the scheduling policy are a priori imposed. In effect, in the development process of a real application, one will likely be working with libraries from third-party suppliers or from the OS vendor for functions like network communication, resource management or graphic display. Such functions, such as kernel I/O drivers, may sometimes require to be run at a particular priority, usually the highest, and/or with a given policy, usually *sched_fifo*, for correctly performing work on which the application relies.

Definition 1
A priority and policy allocation strategy that respects the a priori constraints on tasks priorities and policies, is a possible solution.

Definition 2
A possible solution *which has successfully passed the schedulability analysis, and thus ensures the application's timing constraints to be met, is a* feasible solution.

3.1 Complexity of the problem

If several tasks are assigned to the same priority level, then *sched_rr* must be chosen, according to our working assumptions 5 and 6. On the other hand, *sched_rr* for a level with a unique task is strictly equivalent to *sched_fifo*.
Assigning n tasks to priority level(s) is like subdividing a set of n elements into non-empty subsets, recall assumption 4. Suppose the partitioning leads to k subsets. For

k subsets the number of possibilities is by definition equal to the Stirling number of the second kind (see [1], page 824)

$$\frac{1}{k!}\sum_{i=0}^{k}(-1)^{(k-i)}\binom{k}{i}i^n.$$

There are $k!$ different possible priority orderings of the k subsets of tasks. Furthermore, n tasks can be subdivided into $k = 1, 2, \ldots, n$ many subsets. Thus, there are

$$\sum_{k=1}^{n}\sum_{i=0}^{k}(-1)^{(k-i)}\binom{k}{i}i^n$$

different possibilities of assigning tasks to priority level(s), the policies being induced by the number of tasks for each level.
As can be seen in Figure 1, the size of the solution space increases faster than exponentially, but slower than n^n. For instance, for $n = 10$, an exhaustive search through the entire solution space would require more than three years and two months, assuming $1s$ of computation time per possibility.

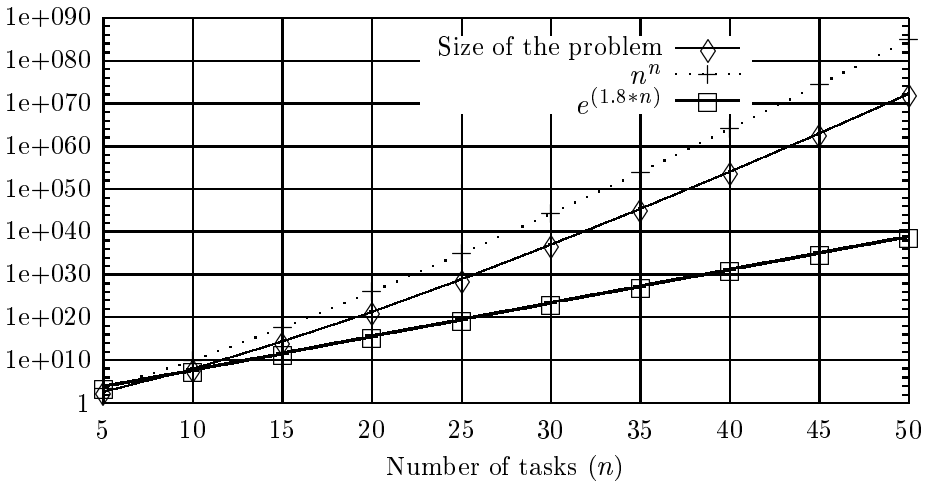


Figure 1: Complexity of the problem for a number of tasks varying from 5 to 50.

3.2 Principle of the algorithm

Usually a computing project starts with an informal description of the functionalities and requirements. In a real-time system, the focus is set on the timing constraints. For instance, in the chemical process described in Section 5.2, one of the constraints is that *"the temperature of the vessel must be updated every 12ms"*. Starting from such informal requirements, one will derive a set of tasks with given characteristics and timing constraints that actually performs the work needed by the application.

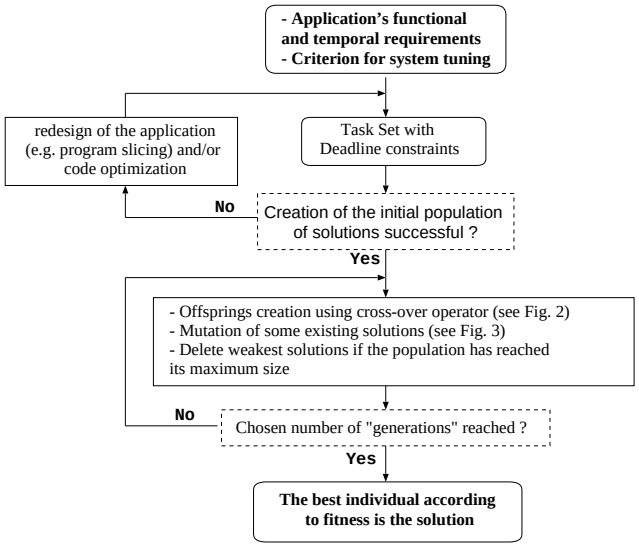


Figure 2: Overview of the approach.

The set of tasks with their constraints and a fitness criteria are the inputs of the GA which will iteratively search through the solution space to find a satisfactory solution until the chosen number of generations is reached. If the GA failed to find 2 feasible solutions[2] in the initial population, then the application has to be either rewritten, at least partially, to diminish the task execution time or redesigned for instance using task slicing techniques such as proposed in [19] by Gerber and Hong. The authors propose the tasks to be decomposed into two fragments, one that is *time-critical* (e.g. inputs from sensors, production of actuator commands and outputs to actuators) and the other that is *unobservable* (e.g. state update). The unobservable

[2]Two solutions are at least needed for crossover.

part of the task without tight timing constraints is then moved to the end of the task which has the benefit of enhancing overall schedulability.

3.3 Genetic Algorithm

Genetic algorithms, GA for short, are biological phenomenon imitations, they were developed by J. Holland and his colleagues at the University of Michigan [23]. GA are search algorithms that explore a solution space and try to mimic the processes observed in the natural evolution of a living population, that is the survival and reproduction of the individuals that are best-suited to the environment and the disappearance of the weakest. Each individual of the population is a solution to the problem and is characterised by its genetic footprint which is, from an implementation point of view, a more-or-less complex data structure. The initial step of a GA is the creation of the initial population, then each step of the algorithm consists of generating new individuals by crossover (also called recombination) and mutations and then choosing the best individuals, according to a "fitness" function, for survival. The result is a population that is globally better at each step with possible improvement for the best individual which will be, at the end of the execution, the actual (sub-optimal) solution of the algorithm. There are in fact a great variety of GA but generally the key points are the choice of the data that will compose the chromosome(s) and the design of efficient crossover and mutation operators. In this study, we adopt the following functioning scheme with overlapping populations inspired from a scheme proposed in [11] :

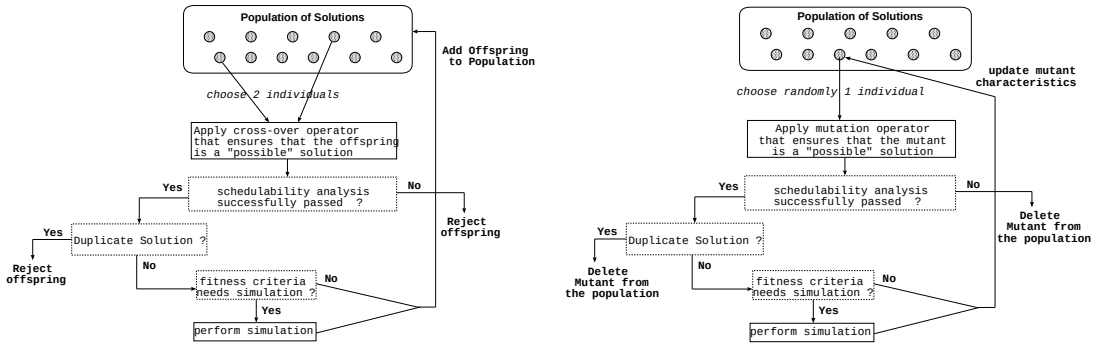
1. Creation of n_{ini} possible solutions for the initial population using Rate Monotonic heuristic (1/4 n_{ini}), Deadline Monotonic heuristic (1/4 n_{ini}) and pure hazard (1/2 n_{ini}), see 3.3.5. Deletion of non-feasible or duplicate solutions and evaluation of the fitness of each feasible individual. Let p_{size} be the size of the population which is currently $\leq n_{ini}$.

2. Creation through crossover of m_{co} new possible solutions which are termed the offspring, see Fig. 3(a). If an offspring is non-feasible or if an identical solution already exists in the population, it is deleted. Otherwise, the evaluation of its fitness value precedes its incorporation in the population of solutions.

3. Mutation of m_{mut} individuals, see Fig. 3(b). As for crossover, the non-feasible or duplicate ones are deleted while the others are integrated in the population after their fitness has been evaluated.

4. If the population is larger than the desired maximum size ($p_{size} > p_{max}$), suppression of the $p_{size} - p_{max}$ weakest members of the population.

5. Go to step 2 if the maximum number of generation is not reached.



(a) Creation of an offspring through crossover.

(b) Mutation of an existing solution.

Figure 3: The genetic operators.

3.3.1 Chromosome coding

In GAs, one or several *chromosomes* describe the solution to the considered problem. Each chromosome is composed of *genes* whose values are called *alleles*. At the beginning of GA, chromosomes were essentially binary strings, but when representing solutions to complex problems, binary representations[3] tend to be less convenient than symbolic representations [40]. For our specific scheduling problem, we propose to code the total genetic package (or *genotype*) of a solution on a single chromosome with each task being composed of 4 genes :

priority	δ_{prio}	scheduling policy	δ_{pol}

Figure 4: The 4 genes coding a task.

[3] In the literature e.g. [35], a GA where the chromosome coding is not fully binary is sometimes termed *"evolutionary algorithm"*.

where the gene :

- priority is the task's priority $\in \{1..card(\mathcal{T})\}$.
- δ_{prio} is binary coded : | fixed | if the task's priority is an a priori constraint | not_fixed | otherwise
- scheduling policy is binary coded : | FPP | if the task is *sched_fifo* | RR | if the task is *sched_rr*
- δ_{pol} is binary coded : | fixed | if the task's policy is an a priori constraint | not_fixed | otherwise

A chromosome solution to a n tasks problem is an array that is the concatenation of the tasks' genes :

Figure 5: A chromosome C, solution to a n tasks problem.

We define the following functions that apply to chromosome C for extracting the properties of tasks :

- $\text{prio}(C,i) \in \{1..card(\mathcal{T})\}$: priority of task τ_i in chromosome C.
- $\delta_{prio}(C,i) \in \{\text{fixed}, \text{not\_fixed}\}$: δ_{prio} value of task τ_i in chromosome C.
- $\text{pol}(C,i) \in \{\text{FPP}, \text{RR}\}$: scheduling policy of task τ_i in chromosome C.
- $\delta_{pol}(C,i) \in \{\text{fixed}, \text{not\_fixed}\}$: δ_{pol} value of task τ_i in chromosome C.

For a chromosome to be a *possible* solution, the following invariants must always be verified :

1. $\forall \tau_i, \tau_j \in \mathcal{T}$ with $i \neq j$, if $(\text{pol}(C,i) = \text{pol}(C,j) = \text{FPP})$ then $\text{prio}(C,i) \neq \text{prio}(C,j)$: two *sched_fifo* tasks can not possess the same priority (assumption 6).
2. $\forall \tau_i, \tau_j \in \mathcal{T}$ if $(\text{pol}(C,i) \neq \text{pol}(C,j))$ then $\text{prio}(C,i) \neq \text{prio}(C,j)$: two tasks having different policies must not share the same priority (assumption 5).

3. $\forall \tau_i \in \mathcal{T}$, $\text{prio}(C,i) \leq card(T)$: for reducing the solution space, we impose that all tasks have a priority within $[1, card(T)]$ (assumption 4).

From an implementation point of view, a chromosome is an array of genes, each gene $C[i]$ being a record made of the priority ($C[i].prio$), the δ_{prio} value ($C[i].\delta_{prio}$), the policy ($C[i].pol$) and the δ_{pol} value ($C[i].\delta_{pol}$). For each individual of the population, in addition to the chromosome, a data of type Real is needed to store the fitness once evaluated. We define the function fitness(), which takes a chromosome as argument and returns its fitness.

3.3.2 Fitness function

The fitness function gives the quality of an individual according to a given criterion. Before evaluating fitness it is mandatory for the individual to be a feasible solution that is, it has successfully passed the schedulability analysis and thus satisfies the application's timing constraints. As represented in figure 3, the evaluation of some criteria may require the performance of simulation. Although an optimisation criterion is by its nature application-dependant, we identify the following general quality criteria that may apply to the whole task set or to a subset of tasks :

- Minimise end-of-execution jitter [requires simulation] : After the schedulability analysis, we know that the end of two successive instances of the same periodic task τ_i is separated by an amount of time varying between $T_i - R_i + C_i$ and $T_i + R_i - C_i$. If the work of task τ_i has to be delivered as regularly as possible, an objective to achieve is to minimise this jitter. As the criterion to be minimised, denoted by $\mathcal{C}^-$, we chose the standard deviation of the response times

$$\mathcal{C}^- = \sum_{i \in \mathcal{T}} \sigma(\widetilde{R}_i) \cdot \Phi_i \tag{6}$$

  with $\widetilde{R}_i = \{R_{i,n} \mid n \vdash E_{i,n} \leq t_{sim}\}$, the sample made of the response times of the instances of task τ_i collected during the simulation of length t_{sim}, with $\sigma()$ the function computing the standard deviation of a set of data and Φ_i, the weight given to task τ_i in the criterion (e.g. $\Phi_i = 0$ means that task τ_i is not considered in the calculus).

- Maximise freshness of input data [requires simulation] : let a task τ_i need input data generated by a subset $\mathcal{T}_i$ of tasks. The data produced by an instance $\tau_{h,a}$ of a task $\tau_h \in \mathcal{T}_i$ is available after its execution end, $E_{h,a}$. To be usable by an instance $\tau_{i,n}$ it must be available before the beginning of its execution, $B_{i,n}$.

The shorter the time between $E_{h,a}$ and $B_{i,n}$ the greater the freshness of the data. The criterion $\mathcal{C}_i^-$ to minimise is therefore:

$$\mathcal{C}_i^- = \sum_{n \vdash E_{i,n} \leq t_{sim}} \sum_{\tau_h \in \mathcal{T}_i} \sum_{a \vdash E_{h,a} \leq B_{i,n} < E_{h,a+1}} (B_{i,n} - E_{h,a}) \tag{7}$$

This criterion can be extended to take account of several tasks with different weights :

$$\mathcal{C}^- = \sum_{\tau_i \in \mathcal{T}} \mathcal{C}_i^- \cdot \Phi_i. \tag{8}$$

- Maximise data consistency [requires simulation] : when a task τ_i needs several input data $X_1, .., X_n$, the production date of these input data should be as close in time as possible to ensure data consistency. It is assumed that the data $X_1, .., X_n$ is available at the end of the execution of task $\tau_1, .., \tau_m$ respectively. We note $\mathcal{I}_i$, the set of tasks whose results will be used by task τ_i as input data. For $\tau_k \in \mathcal{I}_i$, we define :

$$j_{k,i,n} = \max\{j \mid E_{k,j} \leqslant B_{i,n}\}$$

which is the index of the last instance of task τ_k which ends before the start of execution of the n^{th} instance of task τ_i. The criterion which has to be minimised can be expressed as :

$$\mathcal{C}_i^- = \sum_{n \vdash B_{k,n} \leq t_{sim}} \sigma(\widetilde{E}_{i,n}) \tag{9}$$

where $\sigma(\widetilde{E}_{i,n})$ is the standard deviation of $\widetilde{E}_{i,n} = \{E_{k,j_{k,i,n}} | \tau_k \in \mathcal{I}_i\}$, the sample made of the end-of-execution times of all tasks whose results are used by the n^{th} instance of task τ_i. If more than one task has to maximise the consistency of its input data possibly with different weights for each task, the criterion becomes :

$$\mathcal{C}^- = \sum_{\tau_i \in \mathcal{T}} \mathcal{C}_i^- \cdot \Phi_i. \tag{10}$$

Other possible criteria are the minimising of the average/worst-case response time or the maximising of the average/worst-case laxity. Note that one can easily build

an optimisation criterion which is a tradeoff between several criteria. Because the criterion can be evaluated through simulation, it is also possible to consider much more "fine grained" criteria, such as to "maximise the freshness of input data X_1 produced by task A after t_1 units of execution and consumed by task B after t_2 units of execution".

3.3.3 Crossover Operator

Crossover applies to couples of mates that are selected with a probability function of their fitness. This point is crucial since in natural selection, it is assumed that best genitors will create better young. The technique for choosing the couple of mates, denoted by C_1 and C_2, whose principle is described in [40], uses a roulette wheel with slots sized according to fitness :

1. Consider the nb chromosomes composing the population in any order $C_1, C_2, \ldots C_{nb}$.
2. Compute $S = \sum_{j=1}^{nb} \text{fitness}(C_j)$, the sum of chromosomes' fitness for the whole population.
3. Generate randomly a real number $\alpha \in [0, 1]$.
4. Select one chromosome as follows :
   - If the fitness criterion has to be maximised, take C_i such that :
     $$\frac{\sum_{j=1}^{i-1} \text{fitness}(C_j)}{S} \leqslant \alpha < \frac{\sum_{j=1}^{i} \text{fitness}(C_j)}{S}$$
     With $i = nb$, the inequality on the right becomes $\leq$.
   - If the fitness criterion has to be minimised, find the maximum fitness of the population $max_{fit} = \max\{1 \leqslant i \leqslant nb \mid \text{fitness}(C_i)\}$, and take C_i such that :
     $$\frac{\sum_{j=1}^{i-1}(max - \text{fitness}(C_j))}{nb \cdot max_{fit} - S} < \alpha \leqslant \frac{\sum_{j=1}^{i}(max - \text{fitness}(C_j))}{nb \cdot max_{fit} - S}$$
     The inequality on the left becomes $\leq$ with $i = 1$.
5. Go to step 3 if the two mates have not been selected yet.

Once two mates C_0 and C_1 have been selected, we propose to apply a 2 point crossover operator inspired from the Partially Mapped X (PMX) operator proposed by Goldberg and Lingle for the travelling salesman problem [22]. The crossover of individuals C_0 and C_1 creates one offspring C_2 as follows :

1. Choose randomly two integers α_1, α_2 such that $1 \leqslant \alpha_1 \leqslant \alpha_2 \leqslant n$.

2. crossover applies between genes α_1 and α_2 as follows :

   for $i := \alpha_1$ **to** α_2 **do**
   begin
   if $\delta_{prio}(C_0, i) = \text{not\_fixed}$
   then
   $C_2[i].prio := \text{random}(C_0[i].prio, C_1[i].prio, 0.5)$;
   $C_2[i].\delta_{prio} := \text{not\_fixed}$;
   else
   $C_2[i].prio := C_0[i].prio$;
   $C_2[i].\delta_{prio} := \text{fixed}$;
   fi
   if $\delta_{pol}(C_0, i) = \text{not\_fixed}$
   then
   $C_2[i].pol := \text{random}(C_0[i].pol, C_1[i].pol, 0.5)$;
   $C_2[i].\delta_{pol} := \text{not\_fixed}$;
   else
   $C_2[i].pol := C_0[i].pol$;
   $C_2[i].\delta_{pol} := \text{fixed}$;
   fi
   od

   Where $\text{random}(a, b, \alpha)$ is a function returning value a with probability α and value b with a probability $1 - \alpha$.

3. Genes $C_2[1]..C_2[\alpha_1 - 1]$ are all taken uniquely from parent C_0 or uniquely from parent C_1 with a probability 0.5 .

4. Genes $C_2[\alpha_2 + 1]..C_2[n]$ are all taken uniquely from parent C_0 or uniquely from parent C_1 with a probability 0.5 .

5. "Repair" chromosome C_2 if invariants 1,2 or 3 are not verified. The predominance is given to the newly created genes, located between α_1 and α_2 in chromosome C_2, over existing ones. This step ensures that offspring C_2 is a *possible* solution.

3.3.4 Mutation Operator

To avoid the problem of degeneracy that occurs when always mating individuals of a same population, a mutation step takes place after the crossovers. The individuals to which the mutation operator applies are chosen randomly in the population of solutions, only the best solution is excluded from mutation. Once a chromosome C is selected, the mutation operator makes one gene evolve as follows :

1. **repeat**
   $i :=$ random$(1, n)$;
   until $(\delta_{prio}(C, i) =$ not_fixed$) \vee (\delta_{pol}(C, i) =$ not_fixed$)$;
   if $\delta_{prio}(C, i) =$ not_fixed
   then $C[i].prio :=$ uniform$(1, n)$;
   /* the new priority is set randomly between 1 and n */
   fi
   if $\delta_{pol}(C, i) =$ not_fixed
   then $C[i].pol :=$ random$(0, 1, 0.5)$; /* either sched_rr or sched_fifo */
   fi

   Where uniform(a, b) is a function returning a uniform random number between a and b.

2. Repair chromosome C if invariants 1,2 or 3 are not verified. In the same way as in the crossover operator, the predominance is given to the modified gene over existing ones.

3.3.5 Creation of the initial population

This is a crucial step of the algorithm : in effect, if after a maximum number of successive attempts (n_{ini} value), less than the 2 solutions needed for crossover were found, the algorithm has failed. The application has then to be redesigned, for instance using program slicing techniques, such as proposed by Gerber and Hong in [19], or, at least partially rewritten to shorten the code execution time. The initial population is created using *Rate Monotonic* and *Deadline Monotonic* heuristics for setting the priorities as well as using pure chance :

- Rate Monotonic (RM) heuristic : priorities are set according to the scheme, "the smaller the period, the higher the priority", scheduling policies are then fixed randomly.

- Deadline Monotonic (DM) heuristic : priorities are set according to the scheme, "the smaller the deadline, the higher the priority", scheduling policies are then fixed randomly.
- Randomised creation : priorities as well as policies are set randomly.

4 Implementation and experiments

The approach developed in this paper was implemented through 3 distinct executable programs written in C++ : one for the genetic part, one for schedulability analysis[4] and one for the simulation. The program implementing the GA calls the schedulability analysis program to determine whether a possible solution is a feasible one or not (see figure 3). If the solution is feasible, a simulation is then performed to determine its fitness. Much care was taken for the implementation of the simulation program because it is the one that requires the most computation time : it has been written from scratch without the use of any simulation library and, if enough memory is available, all events of the simulation are pre-computed and stored in the scheduler. Since tasks are periodic, interarrival times are constant and the flow of arrivals is non-ergodic. For this reason it is difficult to choose a sufficient length and a sufficient number of trajectories (because in the general case it can not be assumed that the first instances of the tasks are released simultaneously) in order to get a satisfactory accuracy in the results of the simulations. A tradeoff between computation time and accuracy has to be found. We consider a result sufficiently accurate if other simulations performed in the same condition give close results. In our experiments, we found that simulating 10 trajectories, each having the length of 10 LCMs of task periods, produces a satisfactory accuracy and all experiments of this section were performed using these parameters.

Evaluating the performance of an optimisation algorithm is difficult because the optimal result is generally not known. To assess the performance of the proposed GA, that will be denoted by F-GA for "full" GA, its results are compared on the same test cases with the results produced by a "weaker" version of the algorithm, termed W-GA, that does not possess mutation and crossover operators. This will help us (1) to determine whether the proposed GA is efficient compared to a simple randomised search through the solution space and (2) to quantify the improvement. In the weak

[4]The program which implements the schedulability analysis is freely available and can be downloaded at *http://www.inria.fr/mistral/personnel/Jorn.Migge/rts.html*.

version of the algorithm, except for the initial population for which heuristics are used, new individuals are solely created randomly, its functioning scheme being :

1. Creation of n_{ini} possible solutions for the initial population using RM heuristic (1/4 n_{ini}), DM heuristic (1/4 n_{ini}) and pure chance (1/2 n_{ini}). Deletion of non-feasible or duplicate solutions and evaluation of the fitness of each feasible individual. Let p_{size} be the size of the population which is currently $\leq n_{ini}$.

2. Randomised creation of $m_{co} + m_{mut}$ new possible solutions. Deletion of the non-feasible solutions or duplicate ones and evaluation of the fitness value of the rest of them precedes their incorporation in the population of solutions.

3. If the population is larger than the desired maximum size ($p_{size} > p_{max}$), suppression of the $p_{size} - p_{max}$ weakest members of the population.

4. Go to step 2 if the maximum number of generation is not reached.

All the tests were performed with parameter $n_{ini} = 50$, $m_{co} = 40$, $m_{mut} = 20$, and $p_{max} = 100$, the number of successive generations being fixed at 100. The simulation program considers the task execution time to obey the uniform distribution between $[C/2, C]$ where C is the worst-case execution time taken for the schedulability analysis.

4.1 Performance of the GA on a 20-task problem

The 20-task example problem and the best schedule found are described in Appendix A. The total cpu utilisation is 85.7% and the objective is to minimise the end-of-execution jitter of tasks τ_9, τ_{10}, τ_{11}, τ_{12}, τ_{13}, τ_{14}, τ_{15}, τ_{16}, τ_{17}, τ_{18}, τ_{19} and τ_{20}, each task possessing the same weight in the criterion (i.e. in equation 6 $\forall i \in \{9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$ $\Phi_i = 1$, otherwise $\Phi_i = 0$). Some a priori constraints are enforced : tasks τ_1 and τ_2 must obey FPP with priority 1 and 2 respectively, tasks τ_{15} and τ_{20} have to be run under the RR policy at the lowest priority level and task τ_6 has to be scheduled with priority level 3. The quantum for RR is set to 2 units of time.
In figure 6, we observe an important improvement in the mean fitness of the population up to the 30^{th} generation using the GA with crossover and mutation operators. After this point, the population converges, this means that all individuals are very similar in term of fitness and further improvements of the best solution are likely to occur in the event of a favourable mutation. The convergence of a population at the i^{th} generation can be measured by the standard deviation of the sample made of the

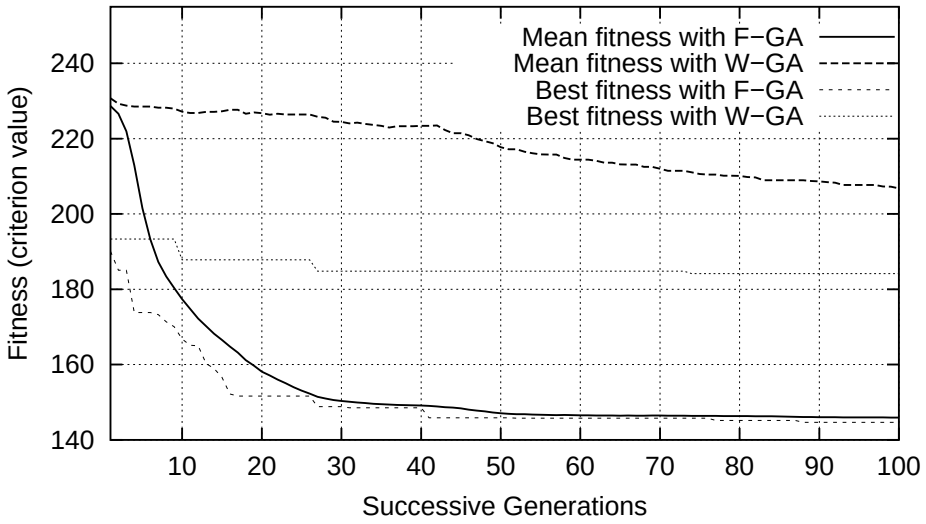


Figure 6: Mean and best fitness evolution through 100 generations on a 20-task problem for F-GA and W-GA.

fitness values denoted σ_i : after the first generation, $\sigma_0 = 9.75$ when $\sigma_{31} = 0.6$ and $\sigma_{100} = 0.31$.
On the other hand, the mean fitness with W-GA does improve much more slowly, only 9% improvement after 100 generations compared to 36.1% with the full GA. This clearly indicates that the genetic operators are far more efficient than a simple randomised search through the solution space. Finally, the best solution with the full GA, which is described in Appendix A, is 21.44% better than the one found with W-GA.

4.2 Performance of the GA on a 30-task Problem

The description of the 30-task test problem is given in Appendix B. The 30 tasks induce a total cpu utilisation of 82.8% and the objective is to minimise the end-of-execution jitter of all tasks of the applications with the same weight in the criterion. Tasks τ_{20} and τ_{21} must be scheduled under RR at the lowest priority level, τ_{13} and τ_{14} have to obey FPP while τ_1, τ_2 and τ_3 must be run under FPP with priority 1, 2, 3 respectively. The quantum for RR is set to 2 units of time.

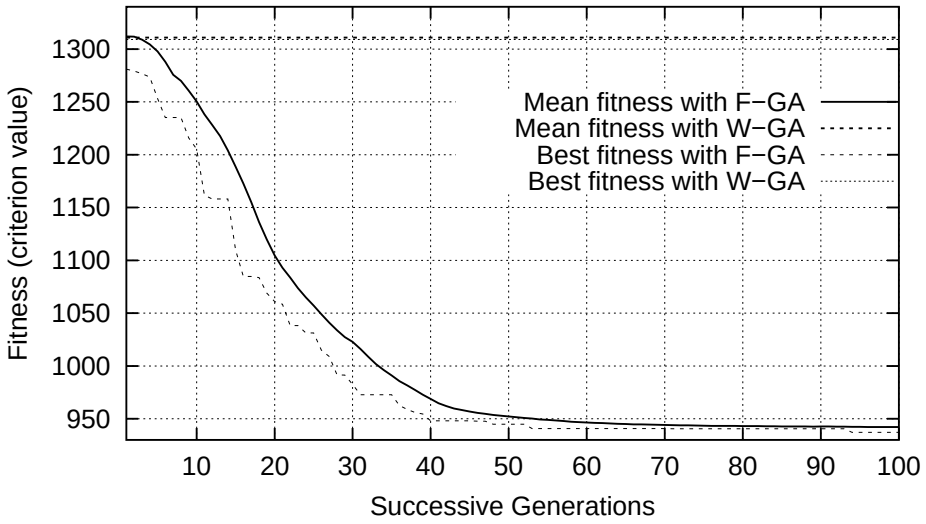


Figure 7: Mean and best fitness evolution through 100 generations on a 30-task problem for F-GA and W-GA.

From figure 7, we observe that W-GA gives very bad results on the 30-task problem : in fact, all the solutions of W-GA were found with the heuristics used for the creation of the initial population. This suggests to us that (1) the proposed heuristics are efficient, and thus are good starting points for the GA, and (2) that a randomised search is completely inappropriate when the size of the problem is large. Compared to the 20-task problem, the speed of population convergence with F-GA becomes logically slower with the increase of the size of the problem which, in our particular case, means that the bigger the problem, the more generations needed. On this 30-task problem, F-GA performs 28.1% better than W-GA in terms of mean fitness and 28.4% better in terms of best fitness.

4.3 Coarse grain parallelisation

The approach developed in this study is computationally intensive, for instance the 30-task example with the chosen parameters requires about 2 days of computation for 100 generations on a *SUN ultra 5* workstation. On the one hand, it is not mandatory to run 100 generations because the population converges sooner and thus

it is very likely that a satisfactory solution will have been found sooner. On the other hand, it is possible to shorten the computation time by reducing the simulation time but the results then lose in accuracy and thus in validity. We point out that such an algorithm has to be executed at the design phase of the system; it is clearly inadequate for on-line reconfigurations. Since the computing time can nevertheless raise a problem, we investigated how to parallelize the GA.

GAs are abstraction of the evolutionary process which is in essence a highly parallel process. They are thus easily parallelized and much work has been done in this field, the reader might refer to [8] for a good survey of research on Parallel Genetic Algorithms (PGAs). Basically, there are 3 distinct approaches in the parallelizing of GAs :

- Global parallelization : in this model, a single population exists but the evaluation of the individuals and possibly the application of the genetic operators are done in parallel. Communication between processors occurs at the end and at the start of the distributed functions.

- Coarse grained[5] parallelization : the total population is split into a few subpopulations, each of which evolves on a distinct processor. The different subpopulations exchange a few individuals from time to time (e.g. every n generations, each t units of time). This model introduces a migration operator that is used to send some individuals from one subpopulation to another.

- Fine grained parallelization : this third approach is similar to the coarse grained one in the sense that multiple subpopulations exist. The difference lies is in the very large number of small subpopulations making this approach well suited to massively parallel computers.

We chose the fine grained approach because it possesses several key advantages : first its ease of implementation, second because it does not require any particular machine such as massively parallel computers and finally because the expected gain is not only a speedup but also better results. In effect, the different subpopulations are likely to explore different regions of the search space and the migrations between the populations create a source of genetic diversity.
The migration is controlled by several parameters; the topology that defines the connection between the subpopulations, the rate of migration that specifies the number of individuals that have to migrate and the migration interval that affects how often

[5]The "grain size" refers to the ratio of time spent in computation and time spent in communication, coarse grained corresponds to a high ratio.

migrations occur. In our parallelized GA, each sub-population makes available to all others 1/10th of its population at the end of each generation. The individuals are chosen randomly for a half and the other half are the best individuals. The parameters n_{ini}, m_{co}, m_{mut} and p_{max} stay unchanged.
The question that ought to be answered is whether the migration process between sub-populations improves the overall performance of the algorithm. The parallelized and the non-parallelized GA have been compared over the same number of generations on the 30-task example problem : 2 populations evolving over 50 generations on 2 distinct processors for the parallelized GA versus 1 population that evolves over 100 generations for the non-parallelized version.

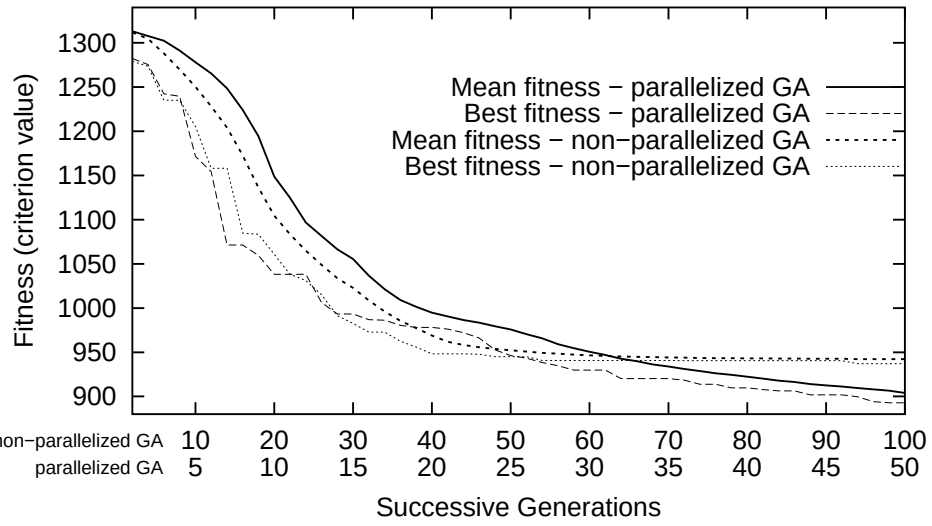


Figure 8: Comparison of the mean and best fitness evolution between parallelized F-GA (2 populations) and non-parallelized F-GA through 100 generations on a 30-task problem.

In figure 8, we observe that the non-parallelized version of the algorithm performs better in terms of best fitness up to the 50^{th} generation and up to the 62^{nd} generation for the mean fitness. Afterwards, both populations of the parallelized version begin to converge and better results are found. Finally, at the end of the 100 generations, the parallelized GA provides an improvement of 4.72% for the best fitness and 4.06% for the mean fitness over the non-parallezised one. This suggests to us

that it is worth considering the use of the parallelized GA, even on a uniprocessor computer, on condition that it executes for a sufficiently long time to let the different sub-populations converge.
Other tests performed with 3 and 4 and 8 sub-populations confirm that the parallelized algorithm performs better than the non-parallelized one once each of the sub-populations has converged. For instance, on this 30-task problem, 4 sub-populations running for 20 generations will not perform better than a single one evolving over 80 generations because 20 generations are not enough to ensure convergence, but 4 sub-populations over 40 generations are likely to give better results than one over 160 generations.

5 Interest of the RR policy for real-time computing

In our opinion, in literature the RR policy has not been seriously considered for use in the field of real-time computing. Traditionally the RR policy is considered useful for low priority processes performing some background computation tasks "when nothing more important is running" (see [17] pp163). In this section, we argue that RR should not be ruled out a priori because there are cases where RR is an efficient choice in terms of schedulability as well as for fine tuning the system.

5.1 RR and schedulability

The possibility of using RR improves the overall schedulability of the system because some tasks set are not schedulable under FPP alone. Imagine a basic application consisting of two tasks A and B with periods (denoted T) 15ms and 50ms respectively, with worst-case execution times (denoted C) 7ms and 10ms and with deadlines (denoted D) 15ms and 20ms, the quantum for RR being set to 1ms. It has been shown in [33] that the maximal response time of a task occurs after a *critical instant*, defined as a time where all tasks are released simultaneously. Only this response time needs to be analysed to decide upon feasibility. Figure 9 represents the scheduling of the two tasks obeying RR at the same priority level starting from a "critical instant". Usually an RR scheduler is so implemented that the first task queued will gain the CPU during the first time quantum. Whichever the first task queued, tasks A and B are schedulable according to RR policy with a time slot being equal to 1ms.
On the other hand, scheduling according to FPP does not lead to any feasible solution, whichever the priority of both tasks as shown in figure 10.
This small example illustrates the usefulness of the RR policy. Note that in general

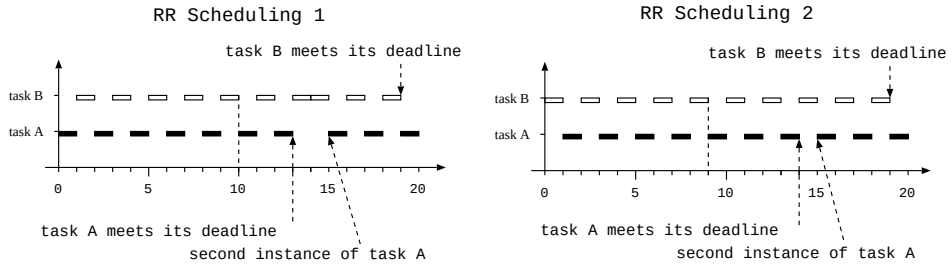


Figure 9: Task A and B scheduled according to RR.

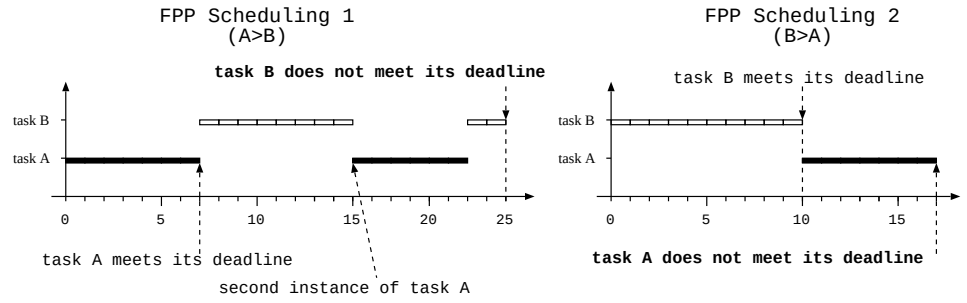


Figure 10: Task A and B scheduled according to FPP.

the schedulability is improved through the combined use of RR and FPP. To gain an insight into the efficiency of RR plus FPP, we have generated 2000 task sets of 10 periodic tasks with deadlines equal to periods which are not feasible under the Rate-Monotonic assignment. This latter is known [33] to be optimal when deadlines are equal to periods. The periods of all tasks are randomly taken between 50 and 1000 time units while their execution times are chosen so that the total workload lies between 75% and 90%. The system-wide time quantum for RR is fixed to 2 units of times. We generated a population of 2000 individuals using the heuristics that are employed for the creation of the initial population of the GA (see Section 3.3.5), note that in this particular case, because deadlines are equal to periods, the RM heuristic is equivalent to the DM heuristic. One half of the 2000 individuals was created using the Rate Monotonic heuristic while the other half was created randomly.
Under these conditions, we found for 15.3% (306 task sets) of the task sets at least one feasible allocation with both RR and FPP and this is probably an underesti-

mation because some feasible allocations may have not been found. Preliminary experiments show that this percentage can be increased if a different quantum may be chosen for each task under RR because quanta under RR play, to a certain extent, a similar role to priority under FPP.

5.2 RR for the fine tuning of the system

We now aim to show that RR plus FPP assignments are better than just straight FPP for the fine tuning of the system using the criteria defined in section 3.3.2. For instance, the use of the RR policy can in some cases be an efficient way of reducing time-skew between observations of the controlled system (e.g. coming from sensors) that are performed by different tasks without recourse to a task slicing technique such as described in [29]. Similarly, RR could also reduce time-skew between task outputs (e.g. commands to actuators).

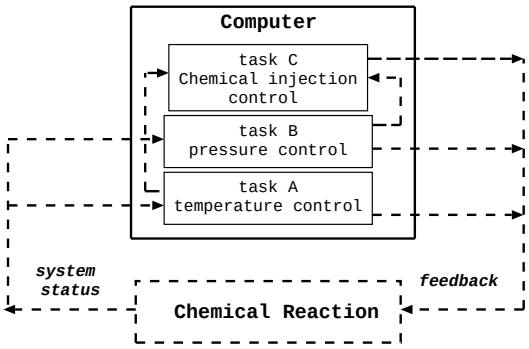


Figure 11: Application structure.

To illustrate this point, imagine a chemical process controlled by one computer (see figure 11). On the computer, two cyclical tasks A (C=4ms, T=12ms, D=12ms) and B (C=4ms, T=12ms, D=12ms) monitor and control the temperature and the pressure respectively. The reading of the current values is done at the beginning of both tasks and takes less than 1ms. Temperature and pressure have to evolve over time and the objective of tasks A and B is to keep their values as near as possible to the optimum in order to accomplish the chemical reaction. At the end of their execution, after having sent commands to actuators, tasks A and B place the current temperature and the current pressure value at the disposal of task C (C=2, T=12ms, D=12ms), scheduled under FPP with a lower priority than tasks A and B and which

runs on the same computer. Task C is responsible for the injection of the right amount of chemicals in the vessel with regard to the current state (temperature, pressure) of the chemical reaction provided by tasks A and B.

As illustrated in figure 12, tasks are schedulable both, with A and B obeying RR at the same priority level (with the slot time being 1ms) or, with FPP. The use of the RR policy has the advantage that input data of task C is more "consistent" from a temporal point of view because it was collected at nearer instants in time than with the FPP scheduling strategy. Task C will thus have a more accurate overview of the current state of the system which will lead to a better control of the physical process and in this particular case, the right amount of chemicals will be injected into the vessel possibly gaining a reduction in costs. The need for temporal consistency between input data is not universal, but, depending on the physical process and also depending on the control algorithm, it can be an objective worth achieving.

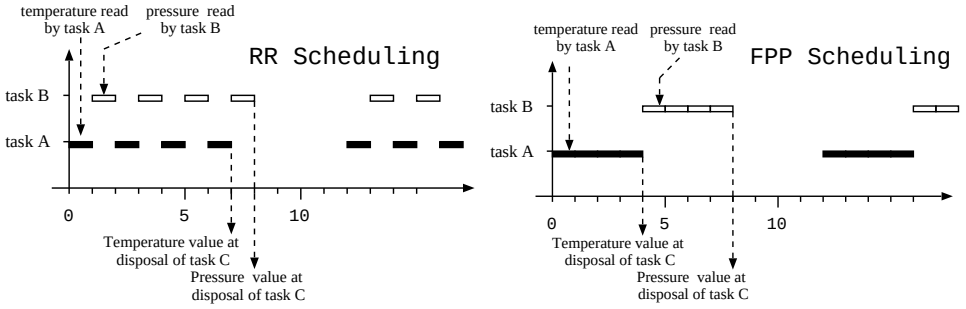


Figure 12: Task A and B scheduled according to RR and FPP.

To quantify the improvement brought about by the use of RR for the tuning of the system, a series of tests was conducted on random task sets composed of 10 to 20 tasks with deadlines equal to periods and a load varying from 30% to 70%. For a given task set, we ran the GA with the parameters given in Section 4, once with the restriction of not chosing RR and once without this restriction. We have restrained ourselves to tasks with constant execution times in order to avoid interference from the execution time distribution in the value of the fitness. For the end-of-execution jitter criterion (see Equation 6), $card(\mathcal{T})/2$ tasks of the task set $\mathcal{T}$ were randomly chosen to be taken into account in the criterion. For the freshness and consistency of input data (see respectively Equation 8 and 10), $card(\mathcal{T})/2$ tasks consuming input data are randomly chosen where each of these tasks requires between 1 and $card(\mathcal{T})/2$ number of input data which is in turn produced by randomly selected tasks. Whatever the criterion,

each selected task i has been given the same weight (i.e. $\Phi_i = 1$) in the calculation of the criterion. For each criterion, 100 random task sets were considered and as can

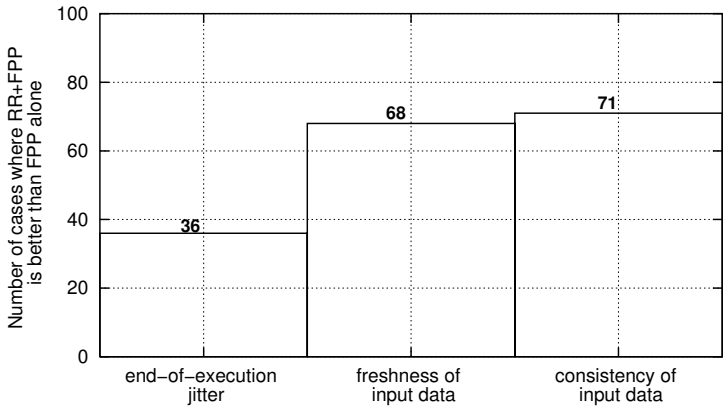


Figure 13: Improvement of fitness through the use of RR.

been seen in Figure 13, allowing the use of RR has improved the fitness of the best individual of the population in 36% of the test cases for the end-of-execution jitter, 68% for the freshness of input data and 71% for the consistency of input data.

6 Concluding remarks and future works

In the field of real-time computing, schedulability analysis has been extensively studied. However, the question of choosing the best among several feasible solutions for a particular application has not been thoroughly addressed. If several feasible solutions exist, then it is possible to consider additional criteria. In this study, we propose a genetic algorithm to best set the scheduling of tasks according to a chosen criterion such as the minimising of end-of-execution jitter or the maximising of the freshness or the consistency of a set of input data. The proposed GA, which has proven to be efficient in the experiments of section 4, makes use of the schedulability analysis, as well as simulation because of the stochastic measurements required by the fitness criteria.

We chose to focus our analysis on Posix 1003.1b compliant OSs because of the wide availability of OSs conforming to this standard. We argue for the usefulness of the RR policy that has been, in our opinion, not sufficiently considered for use in

the field of real-time. Our experiments have shown that the use of RR can improve the schedulability, the consistency and the freshness of input data as well as the end-of-execution jitter.

We are investigating the possibility to allow non-feasible schedules in the population of solutions. It would require a measure of the feasibility finer than just yes or no (for instance, with the taking into account of laxities), which could be integrated into the calculus of fitness by weighting the criteria with the degree of feasibility. This extension might be efficient, in particular, for finding feasible schedules at the very start of the GA. Other possible extensions of this study include the taking into account of shared resources and the extension to the distributed case. It would be of particular interest to consider Controller Area Network (CAN, see [26, 25]) as the communication protocol because Posix1003.1b nodes distributed over CAN represent, in our opinion, a very effective solution for distributed real-time systems made of off-the-shelf components. Finally, we think it is interesting to investigate the possibility that the GA gives some feedback when the algorithm fails, and this in order to help the designer to identify the bottlenecks of the system. In the medium term, it should be possible to conceive a CASE tool dedicated to real-time that includes worst-case execution time analysis, for instance using the work undertaken on *gcc* in [34], as well as an automatic task priority and scheduling policy allocation strategy such as that proposed in this study.

References

[1] M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions*. Dover Publications (ISBN 0-486-61272-4), 1970.

[2] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164, University of York, November 1991.

[3] T. Barret. How to write an rtos - a guide for anyone pondering the "make-or-buy" decision. *Real-Time Magazine*, 97-3:43–45, 1997.

[4] S.J. Beaty. Genetic algorithm versus tabu search for instruction scheduling. In *International Conference on Neural Networks and Genetic Algorithms*, April 1993.

[5] S.J. Beaty, S. Colcord, and P. Sweany. Using genetic algorithms to fine-tune instruction-scheduling heuristics. In *Second International Conference on Massively Parallel Computing Systems (MPCS'96)*, May 1996.

[6] S.J. Beaty, J. Whitley, and G. Johnson. Motivation and framework for using genetic algorithms for microcode compaction. In *Proceedings of the 23rd Annual Workshop in Microprogramming and Microarchitecture (MICRO-23)*, December 1990.

[7] A. Burns, K. Tindell, and A.J. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[8] E. Cantù-Paz. A summary of research on parallel genetic algorithms. Technical Report IlliGAL Report No. 95007, University of Illinois at Urbana-Champain, 1995.

[9] H. Chen, N.S. Flann, and D. Watson. Parallel genetic simulated annealing : A massively parallel SIMD algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):126–136, February 1998.

[10] L. Davis. Job-shop scheduling with genetic algorithms. In *Proceedings of the First Int. Conf. on Genetic Algorithms*, pages 136–140, 1985.

[11] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New-York, 1991.

[12] M. DiNatale and J.A. Stankovic. Applicability of simulated annealing methods to real-time scheduling and jitter control. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, December 1995. also available at http://retis.sssup.it/papers/abstracts.html.

[13] L. Djerid, M.-C. Portmann, and P. Villon. Performance analysis of previous and new proposed cross-over genetic operators designed for permutation scheduling problems. In *Proceedings International Conference on Industrial Engineering and Production Management*, pages 487–497, Marrakech (Maroc), April 1995.

[14] K. Dussa-Zieger. Task scheduling on configurable parallel systems by genetic algorithms. In *Proceedings of Telecommunication, Distribution, Parallelism (TDP'96)*, pages 485–494, July 1996.

[15] K. Dussa-Zieger and M. Schwehm. Scheduling of parallel programs on configurable multiprocessors by genetic algorithms. *International Jounal of Approximate Reasoning*, 19(1-2):23–38, July 1998.

[16] E. Falkenauer and S. Bouffoix. A genetic algorithm for job shop. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 824–829, 1991.

[17] B.O. Gallmeister. *Programming for the Real World - Posix 4*. O'Reilly & Associates, 1995. ISBN 1-56592-074-0.

[18] R. Gerber. Languages and tools for real-time systems: Problems, solutions and opportunities. Technical Report UMD Technical Report CS-TR-3362, UMIACS-TR-94-117, University of Maryland, October 1994.

[19] R. Gerber and S. Hong. Slicing real-time programs for enhanced schedulability. Technical Report UMD Technical Report CS-TR-3477, UMIACS TR 95-62, University of Maryland, May 1995.

[20] R. Gerber and S. Hong. Slicing real-time programs for enhanced schedulability. *ACM Transactions on Programming Languages and Systems*, 19(3), May 1997.

[21] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Welsey, 1989.

[22] D.E. Goldberg and R. Lingle. Alleles, loci, and the travelling salesman problem. In *Proceedings of the First Int. Conf. on Genetic Algorithms*, pages 154–159, 1985.

[23] J.A. Holland. *Adaptation in Natural and Artificial Systems.* Mit Press, Cambridge, Mass., 1975.

[24] L. Ingber and B. Rosen. Genetic algorithm and very fast simulated reannealing : A comparison. *Mathematical Computer Modeling*, 16(11):87–100, 1992.

[25] International Standard Organization ISO. *Road Vehicles - Interchange of Digital Information - Controller Area Network for high-speed Communication.* ISO, 1994. ISO 11898.

[26] International Standard Organization ISO. *Road Vehicles - Low Speed serial data communication - Part 2: Low Speed Controller Area Network.* ISO, 1994. ISO 11519-2.

[27] (ISO/IEC). *9945-1:1996 (ISO/IEC)[IEEE/ANSI Std 1003.1 1996 Edition] Information Technology - Portable Operating System Interface (POSIX) - Part 1 : System Application : Program Interface*. IEEE Standards Press, 1996. ISBN 1-55937-573-6.

[28] M.D. Kidwell. Using genetic algorithms to schedule distributed tasks on a bus-based system. In *Proceedings of Fifth International Conference on Genetic Algorithms*, pages 368–374, 1993.

[29] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and Harbour M.G. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.

[30] P. Koopman, J. Sung, C. Dingman, and D. Siewiorek. Comparing operating systems using robustness benchmarks. In *Symposium on Reliable Distributed Systems*, pages 72–79, October 1997.

[31] Y.K. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing - Special Issue on Parallel Evolutionary Computing*, 47(1):58–77, November 1997.

[32] K. Lin, K. Kenny, S. Natarayan, and J. Liu. Flex: A language for real-time systems programming. In A. Tilborg and G. Koob, editors, *Foundations of Real-Time Computing : Formal Specifications and Methods*, pages 251–290. Kluwer Academic Publishers, 1990.

[33] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of ACM*, 20(1):40–61, February 73.

[34] D. Macos and F. Mueller. Integrating gnat/gcc into a timing analysis environment. In *10th EUROMICRO Workshop on Real Time Systems*, 1998. WIP Session.

[35] J.A. Michalewicz. *Genetic Algorithm + Data Structure = Evolution Program*. Springer Verlag, 1992.

[36] J.M. Migge and A Jean-Marie. Real-time scheduling: Non-preemption, critical sections and round robin. Research Report 3678, INRIA, April 1999. Available at ftp://ftp.inria.fr/INRIA/publication/RR/RR-3678.ps.gz.

[37] J. Moses. Is posix appropriate for embedded systems. *Embedded Systems Programming*, July 1995.

[38] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop problems. In *Proceedings of the Fourth Int. Conf. on Genetic Algorithms*, pages 474–479, 1991.

[39] I.M. Oliver, G.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the travelling salmesman problem. In *Proceedings of the Second Int. Conf. on Genetic Algorithms*, pages 224–230, 1985.

[40] M.-C. Portmann. Scheduling methodology: Optimization and compu-search approaches i. In A. Artiba and S. E. Elmahgraby, editors, *Production and Scheduling of Manufacturing System*, pages 271–300. Chapman & Hall. 1997, 1996.

[41] F. Salles, J. Arlat, and J.-C. Fabre. Can we rely on COTS microkernels for building fault-tolerant systems. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 189–194, Octobre 1997.

[42] M. Schwehm and T. Walter. Mapping and scheduling by genetic algorithms. In *Third Joint International Conference on Vector and Parallel Processing (CONPAR94), Linz (Austria)*, number 854 in Lecture Notes in Computer Science, pages 833–841. Springer Verlag, 1994.

[43] J.A. Stankovic, A. Burns, K. Jeffay, M. Jones, G. Koob, I. Lee, J. Lehoczky, J. Liu, A. Mok, K. Ramamritham, J. Ready, L. Sha, and A. Van Tilborg. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4), December 1996.

[44] D. Thiel. Un algorithme génétique pour la résolution de problèmes d'affectation quadratique comparé aux performances du recuit simulé. *RAIRO - APII - JESA*, 31(9):1541–1564, 1998.

[45] K. Tindell, A. Burns, and A.J. Wellings. Allocating hard real time tasks (an np problem made easy). *Real-Time Systems*, 4(2):145–165, June 1992.

[46] T. Tsuchiya, T. Osada, and T. Kikuno. Genetics-based multiprocessor scheduling using task duplication. *Microprocessors and Microsystems*, 22(3-4):197–207, 1998.

Appendix A: 20-task example problem

The 20-task example problem is described in figure 14(a) as well as the best schedule found with F-GA in figure 14(b). $\rho_k = C_k/T_k$ is the cpu utilisation of task τ_k and $\rho_{1..m}$ is the cumulated cpu utilisation from the highest priority task up to the considered task. The laxity of task τ_k is defined as $D_k - R_k$ where R_k is the bound on the response time of task τ_k given by the schedulability analysis. In figure 14(a), one of the a-priori constraints is that tasks τ_{20} and τ_{15} obey RR at the lowest priority level which initially, because the schedule of the other tasks is unknown, has to be set to 20. The best solution preserves the lowest priority level but because we want the priorities to be contiguous, priority 20 becomes 18 which in terms of scheduling does not change anything.

task	C	T	D	prio.	pol.
τ_{20}	10	1000	1000	20	RR
τ_{19}	13	1000	1000		
τ_{18}	8	800	800		
τ_{17}	12	800	400		
τ_{16}	5	600	600		
τ_{15}	10	500	750	20	RR
τ_{14}	10	500	500		
τ_{13}	15	300	300		
τ_{12}	5	300	300		
τ_{11}	4	250	250		
τ_{10}	7	225	225		
τ_9	15	200	200		
τ_8	5	175	350		
τ_7	6	150	500		
τ_6	6	150	150	3	
τ_5	8	120	120		
τ_4	9	100	100		
τ_3	5	75	120		
τ_2	6	60	60	2	FPP
τ_1	7	50	50	1	FPP

(a) 20-task problem with the a priori constraints.

layer	task	prio.	R	laxity	ρ_k	$\rho_{1..m}$
Round Robin	τ_{20}	18	444	556	1.0	85.7
	τ_{15}	18	444	306	2.0	84.7
FPP	τ_{13}	17	297	3	5.0	82.7
	τ_{19}	16	282	718	1.3	77.7
	τ_{17}	15	269	131	1.5	76.4
	τ_9	14	189	11	7.5	74.9
	τ_3	13	120	0	6.7	67.4
	τ_4	12	99	1	9.0	60.7
	τ_5	11	90	30	6.7	51.7
	τ_{14}	10	82	418	2.0	45.1
	τ_{16}	9	72	528	0.8	43.1
	τ_{12}	8	67	233	1.7	42.2
	τ_7	7	49	451	4.0	40.6
	τ_{10}	6	43	182	3.1	36.6
	τ_{11}	5	36	214	1.6	33.5
Round Robin	τ_8	4	30	320	2.9	31.9
	τ_{18}	4	32	768	1.0	29.0
FPP	τ_6	3	19	131	4.0	28.0
	τ_2	2	13	47	10.0	24.0
	τ_1	1	7	43	14.0	14.0

(b) Best schedule with fitness equal to 144.66 found at the 88^{th} generation with F-GA.

Figure 14: 20-task example problem with the best solution found.

Appendix B: 30-task example problem

task	C	T	D	prio.	pol.
τ_{30}	50	5000	5000		
τ_{29}	40	2500	2500		
τ_{28}	15	2000	2000		
τ_{27}	60	1500	1000		
τ_{26}	19	1500	1500		
τ_{25}	10	1500	1000		
τ_{24}	10	1200	1200		
τ_{23}	12	1200	1200		
τ_{22}	15	1100	550		
τ_{21}	10	1000	1000	30	RR
τ_{20}	13	1000	1200	30	RR
τ_{19}	14	1000	1000		
τ_{18}	40	1000	500		
τ_{17}	16	1000	600		
τ_{16}	11	750	400		
τ_{15}	8	750	800		
τ_{14}	12	750	150		FPP
τ_{13}	14	750	200		FPP
τ_{12}	15	500	500		
τ_{11}	5	500	500		
τ_{10}	10	500	500		
τ_9	15	300	300		
τ_8	5	300	300		
τ_7	4	250	250		
τ_6	7	250	250		
τ_5	15	200	200		
τ_4	5	200	175		
τ_3	6	150	150	3	FPP
τ_2	5	50	50	2	FPP
τ_1	7	50	50	1	FPP

(a) 30-task problem with the a priori constraints.

layer	task	prio.	R	laxity	ρ_k	$\rho_{1..m}$
Round Robin	τ_{21}	26	977	23	1.0	82.8
	τ_{20}	26	980	220	1.3	81.8
FPP	τ_{27}	25	945	55	4.0	80.5
	τ_{30}	34	729	4271	1.0	76.5
	τ_{29}	23	597	1903	1.6	75.5
	τ_{18}	22	492	8	4.0	73.9
	τ_{26}	21	434	1066	1.3	69.9
	τ_{28}	20	383	1617	0.8	68.7
	τ_{19}	19	368	632	1.4	67.9
	τ_{25}	18	342	658	0.7	66.5
	τ_9	17	294	6	5.0	65.9
	τ_{17}	16	279	321	1.6	60.9
	τ_{12}	15	240	260	3.0	59.3
	τ_5	14	193	7	7.5	56.3
	τ_{13}	13	178	22	1.9	48.8
	τ_{14}	12	146	4	1.6	46.9
	τ_{16}	11	134	266	1.5	45.3
	τ_{10}	10	123	377	2.0	43.8
	τ_{23}	9	113	1087	1.0	41.8
	τ_{15}	8	89	711	1.1	40.8
Round Robin	τ_{11}	7	72	428	1.0	39.8
	τ_{22}	7	81	469	1.4	38.8
Round Robin	τ_6	6	47	203	2.8	37.4
	τ_{24}	6	49	1151	0.8	34.6
FPP	τ_8	5	32	268	1.7	33.8
Round Robin	τ_7	4	26	224	1.6	32.1
	τ_4	4	27	148	2.5	30.5
FPP	τ_3	3	18	132	4.0	28.0
	τ_2	2	12	38	10.0	24.0
	τ_1	1	7	43	14.0	14.0

(b) Best schedule with fitness equal to 937.12 found at the 94^{th} generation with F-GA.

Figure 15: 30-task example problem with the best solution found.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399