Genetic Algorithms in Real-Time Imprecise Computing

Leo Budin, Domagoj Jakobović, Marin Golub
Faculty of Electrical Engineering and Computing
Unska 3, HR-10000 Zagreb, Croatia
phone: +385 1 61 29 935, fax: +385 1 61 29 653
e-mail: {leo.budin, domagoj.jakobovic, marin.golub}@fer.hr

***Abstract* - This article describes the use of genetic algorithms in real-time systems that employ the imprecise computation paradigm. In real-time systems, the focus is on ensuring that a set of tasks each complete within their deadlines. Faults may occur in the computation or the environment that can cause missed deadlines. That is why the idea of using partial results when exact ones cannot be produced within the deadline has been introduced. This idea has been formalized using the concepts of anytime algorithms and imprecise computation and specific techniques have been developed for designing programs which can produce partial results and for developing systems that can support imprecise computation techniques. Genetic algorithms are methods that can be, without any adaptation, used in an imprecise computation system. They produce a solution that bears a certain measure of reliability. During the process of their execution, this solution is constantly improving. They can be used as a part of a real-time system, especially for optimizing tasks where the classical algorithms are not applicable or its computational time proves to be too expensive.**

I. INTRODUCTION

Real-time systems are now used in a wide variety of applications, including space and defense systems, process control and signal processing. Conventionally, real-time systems are designed to perform a given set of tasks where each task is bounded with its time constraints. A task set in a classical real-time system can also be interdependent; a successor-predecessor relation is defined between some of the tasks. In order for all of these constraints to be satisfied a feasible schedule must be produced prior to or even during the system run. This, with the general problem of scheduling resources being optimally NP-hard, calls for fast and effective methods for resolving the scheduling problem.

While the scheduling is in most cases fixed at design time, a static well-planned real-time system will never miss a deadline. In practice, however, several dynamic situations may arise which affect the scheduling. Tasks may overrun their expected computation time due to larger amount of input data or because an iterative algorithm takes a longer time to converge. The concept of imprecise and approximate computations has emerged as the basis of a new approach of dealing with these issues. When time and resources are not sufficient for computations to complete within the deadline, there may still be enough resources to produce approximate results of acceptable, if not desired, quality.

The nature of many of the algorithms is such that they can adapt to the imprecise computation concept; that is, an algorithm can produce an approximate result before its regular execution time is finished. Genetic algorithms, an example of heuristic directed random search methods, fit perfectly with the idea of imprecise computing. They are iterative algorithms which refine their output with time and can handle a vast majority of computational and optimization tasks in everyday practice.

A genetic algorithm may be viewed as an evolutionary process wherein a population of solutions evolves over a sequence of generations. The algorithm maintains a set of solutions which are evaluated by fitness function in each generation. After evaluation, they are selected for reproduction based on their fitness. Selection embodies the principle of *survival of the fittest*: *good* solutions are selected for reproduction and *bad* ones are eliminated. The selected solutions then undergo recombination under the action of genetic operators *crossover* and *mutation*. Crossover causes exchange of genetic material between solutions. Crossed solutions can produce ones with better (or worse) fitness value. The role of mutation is in restoring lost or unexplored genetic material. After performing genetic operators, a generation cycle is concluded and a test is performed in order to determine whether a termination condition is reached or not.

In this work the use of genetic algorithms in imprecise real-time systems is analyzed, the features of such systems are described, and a few guidelines are stated for their efficient design. A scheduling algorithm is designed which can be used in systems that include genetic algorithms, as well as in other real-time systems with imprecise computation.

II. IMPRECISE COMPUTATION TECHNIQUE

Meeting timing or deadline constraints is one of the most important concerns in real-time systems. Unfortunately, due to nonpredictive elements in dynamic real-time implementations, such as variations in processing times of algorithms and constantly changing environment demands, it is sometimes impossible to schedule all of the tasks so that their deadlines are met at all times. This situation occurs quite often when the system is in peak load. The *imprecise computation technique* represents an approach that trades off the quality of the results produced by the tasks with the amounts of processing time required to produce the results. This technique assures that an approximate result of an acceptable quality is available to the user whenever the exact result of the desired quality cannot be obtained in time.

A. Task requirements

In order for a real-time system to support imprecise computation, every time-critical task in the system has to be structured in a way that it can be logically decomposed into two subtasks: a *mandatory* subtask and an *optional* subtask. The mandatory subtask is the portion of computation that has to be done for a task to produce a meaningful result and it has to be completed before the deadline. The optional subtask is the portion of the computation that refines the result. It can be left unfinished at the expense of the quality of the overall result produced by the task.

There exist several methods to adapt the task execution so it can be used in imprecise computation. If a task generates the result in some form of iterative refinement, we can record the intermediate results at appropriate instances of the task execution. The mandatory part of a task executes first, producing a result with the minimum acceptable level of reliability. This result is then refined by the optional part which stores the current output value in predefined time intervals. Upon request, the latest recorded value of intermediate result is available to the user. This method for returning imprecise results is called the *milestone method*.

If the milestone method is not applicable, then it can be possible to compose a task where mandatory parts are interleaved with computational steps that can be skipped in producing a minimally acceptable solution. These parts are called *sieve functions*. If a sieve function is not completed, then its inputs, rather then outputs, are used by later mandatory computation. Iterative computation can also be viewed as a series of sieve functions.

When neither the milestone method nor the sieve method can be used, we can almost always use the *multiple versions* of the tasks. In this approach we need to provide two (or more) versions of each task: the primary and the alternate version. The primary version produces a precise result but uses more computation time. We may want to schedule the alternate version, which has a shorter processing time and generates an imprecise but acceptable result, when it is not possible to complete the primary version of a task by its deadline.

We have the maximum flexibility in scheduling when all the time-critical computations are designed to be monotone. The quality of the intermediate result produced by monotone algorithm is non-decreasing as it executes longer. The longer a monotone task executes before its termination, the smaller is the error of its imprecise result. Monotone algorithms exist in many problem domains such as numerical computation, statistical estimation and prediction, heuristic search or sorting. There are also efforts to develop monotone algorithms in application domains where such algorithms are needed. When tasks are monotone, the scheduling can be done dynamically and on-line, or nearly on-line, because the scheduler can terminate a task at any time after it has produced an acceptable result.

B. *Scheduling for a purpose*

Given a set of tasks in a real-time system, we have to schedule them so that deadline constraint is met for every task. Apart from satisfying timing constraint, we may also want to achieve a certain performance regarding some other criteria. In imprecise real-time systems there are several different performance metrics. If our goal is to minimize the *total* or *maximum error*, we will try to schedule the tasks in such a fashion that every task returns as good result as possible. The problem becomes more demanding if every task has a certain weight factor that determines exactly how important its result is to the system.

If we are given a certain total error threshold, we may want to minimize the *number of late* or *tardy tasks*, that is, the ones whose mandatory subtask cannot even meet its deadline. Given the same threshold value, we may choose to minimize the *average response time* or *mean flow time*, i.e. the average amount of time a task spends in the system until it completes, which includes possible waiting as well as running.

The goal may also sometimes be to minimize the number of *discarded optional tasks*, the ones whose optional part is not computed. If the real-time system is realized by using the multiple versions of the tasks, it is then often called the imprecise computation with 0/1 constraint. Scheduling such a system to minimize total error has proved itself to be very demanding.

III. SCHEDULING GENETIC ALGORITHMS

Genetic algorithms have not been excessively utilized in hard real-time systems so far. In such a system the user needs the output promptly and accurately, which genetic algorithms are not designed for. Of course, an algorithm always has to exist that will produce the result. In cases where there is no algorithm which will yield the correct result or its execution might be too slow or the problem is NP-hard we might use a genetic algorithm. If a real-time system is also based on an imprecise computing model, a genetic algorithm as a task may be the solution.

Genetic algorithms, as well as other heuristic random search methods (simulated annealing, evolutionary strategies etc.), are monotone algorithms that are suitable for implementation in an imprecise computation system. The system can record the current solution in every generational cycle of the algorithm. The computational time of the mandatory subtask of genetic algorithm is virtually non-existent, because the algorithm provides the initial solution in the first iteration. The optional part, on the other hand, has an undetermined execution time, because there is no way to know whether the algorithm has reached the correct solution. Formally, we can either denote the computational time of mandatory subtask as zero and of the optional subtask as infinite or vice versa, depending on the nature of the real-time system. Genetic algorithms can also very easily be designed to execute in

parallel. That approach can produce better results in the same amount of time on multiprocessor systems.

The accuracy of the solution provided by genetic algorithm cannot be measured, which is a significant drawback; in general, we can only hope that the solution is *good enough* as there is no error estimate. What we can tell is that the solution can only be better if the algorithm executes longer. If we want to calculate the quality dependence of the results the genetic algorithm will produce on the execution time of the algorithm, we can perform a series of optimizations of the same class of the problem. One should keep track of the intermediate results and times when they were recorded, and determine the quality of the best result the algorithm can give. That way, for a given execution time, we could estimate the quality of the solution produced so far or schedule the amount of computation time needed for the desired result quality. It is important to note that this method also bears a certain measure of unreliability - there is no way we could predict the exact quality of the results, given only the execution time of the algorithm.

There are two ways of incorporating genetic algorithms into real-time systems. They can either replace a task or several of them or the system may consist entirely of genetic algorithms. In the first approach the genetic algorithm can perform a job for which there is no effective algorithm or the job is too time-consuming for a classical algorithm. If that is the case, the computation time of the mandatory part of genetic algorithm should be defined as infinite so it can run as long as possible. That way it will always be late, but only formally, as it can never complete in traditional sense anyway. The computational error of the algorithm is unknown and therefore can only be ignored.

What criteria should we consider when making a schedule for a real-time system including or consisting entirely of genetic algorithms? Minimizing total or maximum error is not possible because we have no mean to evaluate it. In classical imprecise computations the error is usually defined as a function of the time portion of the optional subtask that was discarded in the schedule, which cannot be measured if a task is a genetic algorithm. The user can, in that case, assign higher weight values to more important parts of the computation. The number of late tasks is also without meaning in such system, as well as the problems with 0/1 constraints. As for the average response time, there is no sense in terminating a genetic algorithm if it is not absolutely necessary; it can only produce a better result over time. Thus, when scheduling genetic algorithms, we should allow each one to execute as long as possible in order to get more accurate results. We can assign weight factors to each algorithm so that more important tasks get more execution time.

The scheduling algorithm should also take advance of the property of the genetic algorithms which allows them to be easily configured to execute on more than one processor simultaneously. Such a parallelization can be achieved by dividing an algorithm into a number of *threads* where each thread operates on a single set of solutions. The number of threads is not limited which in turn allows a genetic algorithm to run on as many processors as we can provide, whereas some other types of algorithms can appropriately run only on a certain maximum number of processors. That number is called the maximum degree of concurrency and is not defined for genetic algorithms. In the same time, a genetic algorithm divided in threads and executing on a number of processors doesn't have any multiprocessing overhead since no interprocessor communication or synchronization is needed. In fact, that parameter cannot be defined because of the undetermined computation time of the algorithm.

The migration problem, i.e. transferring the algorithm from one processor to another, is not an issue when genetic algorithms are in question. Since there exists only one set of solutions, available to all processors, on which the threads operate, any processor can continue the work of every other. More details on implementing parallel genetic algorithms can be found in [1].

On the other hand, when constructing a feasible schedule, we should also try to avoid the preempting of a genetic algorithm. Every algorithm operates on a population of solutions that can occupy a significant amount of memory, depending of the number and the size of the population members. If we are to interrupt the algorithm, we have to preserve or store the entire population. Not only that it is not recommended but it can also block some valuable resources and demand more computation time for context switching.

IV. SCHEDULING ALGORITHM

Having in mind the goals stated for scheduling imprecise real-time systems with genetic algorithms, a scheduling algorithm can be designed which will take in account those priorities. An imprecise real-time system is defined in a following manner.

We are given a set of n preemptable tasks, indicated as $\mathbf{T}=\{T_1, T_2, ..., T_n\}$. Each task is characterized by the following parameters, which are rational numbers:

- *ready time* r_i' at which T_i becomes ready for execution,
- *deadline* d_i' by which T_i must be completed,
- *mandatory processing time* m_i that is required to execute the mandatory part of T_i (not defined for genetic algorithms)
- *optional processing time* o_i for the optional subtask (not defined for genetic algorithms)
- *weight* w_i that is a positive number greater or equal to one and measures the relative importance of the task.

The weights of the tasks are determined prior to the scheduling or executing and are considered to be constant. Should the importance of a task change during the system run, the schedule can be redesigned regarding new weight

values and the current time instance as a new begin time of the system.

The dependencies between the tasks in **T**, if any, are specified by their precedence constraints; they are given by a partial order relation < defined over **T**. The relation $T_i < T_j$ holds if the execution of T_j cannot begin until the task T_i is completed and terminated. In order for a schedule to be valid, all the precedence constraints must be satisfied. It is possible that the given deadline of a task is later than that of its successors, or the given ready time may be earlier than that of his predecessors. Instead of working with given ready times and deadlines, modified values are used that are consistent with the precedence constraints. Those values are computed as follows. If a task has no successors, the modified deadline is equal to its given deadline. If there exist successors of a task T_i, let A_i be the set of deadline times of all successors of T_i. The modified deadline d_i of T_i is

$$d_i = \min\left\{d_i', \min\left\{d_j\right\}\right\}, \quad \forall d_j \in A_i . \tag{1}$$

Similarly, the modified ready time of a task that has no predecessors is equal to its given ready time. Let B_i be the set of ready times of all predecessors of T_i. The modified ready time r_i of T_i is

$$r_i = \max\left\{r_i', \max\left\{r_j\right\}\right\}, \quad \forall r_j \in B_i . \tag{2}$$

Working with the modified ready times and deadlines allows the precedence constraints to be ignored temporarily. If an algorithm finds an invalid schedule in which T_i is assigned a time interval later then some intervals assigned to T_j and $T_i < T_j$, then a valid schedule can be constructed by exchanging the time intervals assigned to T_i and T_j to satisfy their precedence constraint without violating their timing constraints. Hereafter by ready times and deadlines we mean modified ready times and deadlines.

The scheduling algorithm devised in this work is oriented towards scheduling genetic algorithms as tasks, but it can also be used in any imprecise real-time system. We only have to keep in mind the algorithm's assumptions and priorities, which are:

- every task is supposed to be parallelizable; the algorithm schedules them as on the one-processor system, but a task is supposed to occupy all the available processors in its time interval;
- the algorithm tries to give more time to more important tasks, but only as long as less important tasks keep a certain minimum quantity of time (defined later in the text), which can be set by the user;
- if a task is scheduled more than one distinct time interval, the scheduling algorithm tries to rearrange the intervals to merge them into one.

Rather then stating the algorithm and than presenting a scheduling example, we will describe the algorithm along with resolving an imprecise real-time system schedule. The real-time system which is going to be scheduled is defined in Table 1.

TABLE 1
THE EXAMPLE REAL-TIME SYSTEM

	r[i]	*d[i]*	*w[i]*
T_1	0	6	3
T_2	4	12	2
T_3	0	14	1

The given parameters are presented in the algorithm in the following data structures (their values for the specified system are also given):

- *n*, the number of tasks in the system;
- *r[i], i* = 1..*n*, array of (modified) ready times of every task;
- *d[i], i* = 1..*n*, array of (modified) deadlines of every task;
- *a[]*, array of distinct time values which is obtained by sorting the lists of ready times and deadlines of all the tasks in **T** and deleting duplicate entries in the list, without the last one (i.e. all distinct values of *r[i]* and *d[i]*); here *a[]*=(0,4,6,12);
- *w[i], i* = 1..*n*, array of task weights;
- *h[]*, array of distinct weight values, sorted in descending order and without the smallest value; here *h[]*=(3,2);
  mT[i], i = 1..*n*, array of minimum time quantities for each task;
- MT[i], i = 1..n, array of the maximum allowed time for each task, MT[i] = d[i]-r[i].

The minimum time for a task is defined as

$$mT[i] = \frac{w[i]}{\sum_{j=1}^{n} w[j]}\left(d[i] - r[i]\right). \tag{3}$$

The user can redefine the minimum time a task is assigned, but in that case there may not always exist a feasible schedule.

Let the tasks' indexes be arranged in such a fashion that d[1] ≤ d[2] ≤ ... ≤ d[n], i.e. a task with a greater index has a later deadline. The output of the scheduling algorithm is a list of time intervals along with an index of the task which is executed in that time. It consists of three phases: in the first phase the initial feasible arrangement is made. In the second phase the intervals are modified, if possible, so that the tasks with higher weight values get even more computational time, keeping the other tasks with their defined minimum. Finally, the algorithm finds the preempted entries in the third phase and tries to merge them if possible.

Additional data structures that are used in the process are also:

- *L[i][2]*, list of time intervals; the first component of the list element denotes the task index *j* = *L(i, 1)* and a second one the allotted duration *t* = *L(i, 2)*;
- *p*, total number of entries in the list;

- *F[i], i = 1..n*, array which denotes the last entry of the task *i* in list *L*;
- *D[i], i = 1..n*, array of total time duration given to the task *i*;
- *M[i][2], i = 1..l*, array of amounts of time an interval can be 'shifted' in the schedule in both directions, the distance from the task's ready time in one direction and its deadline in the other (defined in the algorithm).

In our example system, the defined minimum time quantities for tasks T_1, T_2 and T_3 are 3, 8/3 and 7/3, respectively. We will, however, redefine those values into, for example, 3, 3 and 2.5.

The total execution time of the system is divided by values in array *a[]* in distinct time intervals. For each interval the algorithm detects *active* tasks – the ones whose ready time comes earlier and deadline time later than the interval boundaries, i.e. the task is considered active if *r[task]<=a[i]* and *d[task]>=a[i+1]*.

The initial arrangement allocates to each of the tasks the amount of time proportional to its weight value. Time quantities are calculated for each distinct interval and new entries are added to the schedule if the task has not yet been given one; otherwise, the calculated time quantity is added to the existing entry of a task already in the list. New entries for existing tasks in the schedule can be given if a task's scheduled time has reached its deadline – in that case a new set of entries for all active tasks must be introduced.

The first phase of the scheduling algorithm is presented in Fig 1.

```
for every i in a[] {
   W = weight sum of all active tasks;
   if(p=0 or M[L(p,1)][2]=0)
      for every active task j {
         p++;
         add entry L(p,1)=j; F[j]=p;
         L(p,2)=w[j]/W*(a[i+1]-a[i]);
         D[j] += L(p,2); }
   else
      for every active task j {
         if(a[i]=r[j]) {
            p++;
            add entry L(p,1)=j; F[j]=p;
            L(p,2)=w[j]/W*(a[i+1]-a[i]);
            D[j] += L(p,2); }
         else {
            L(F[j],2) += w[j]/W*(a[i+1]-a[i]);
            D[j] += L(F[j],2); } }
   if d[L(p,1)]=a[i+1]
      M[L(p,1)][2]=0;
c=0;
for every i in L {
   M[i][1]=c-r[L(i,1)];
   c += L(i,2);
   M[i][2]=d[L(i,1)]-c; }
```

Figure 1. The first section of the scheduling algorithm

After the first phase we have the following entries in *L* as (task, duration) and *M* as (left shift, right shift):

L(task, duration)	*M*
(T_1, 4)	(0, 2)
(T_3, 3.333)	(4, 6.666)
(T_2, 4.666)	(3.333, 0)
(T_3, 2)	(12, 0)

The values in array *M* denote how much we can shift the specified time interval to the left (earlier in time) or right (later in time) before the boundaries reach the ready time or the deadline of the task.

In the next section the algorithm rearranges the schedule so the tasks with higher weight values get more computational time. A task, in fact an entry in the list, with higher weight *borrows* time from other entries with smaller weight values. Time reallocation is undertaken with preserving the minimum time a task must have and not violating the determined ready times and deadlines. An entry from the list can gain more time from any other entry as long as there is minimum time left (to the task in the other entry) and no time constraint is violated for any list member in between. The reallocation is done in steps according to values in array *h[]*. If all the tasks bear the same weight value, there is no change in the schedule. The second phase of the algorithm is listed in Fig 2.

```
for every i in h[]
   for every entry j : w[L(j,1)]=h[i] AND
   D[L(j,1)]<MT[L(j,1)] {
      k=j; end=0; taker=L(j,1)
      while M[k][2]>0 AND end=0 AND
       D[taker]<MT[taker] {
         k++; giver=L(k,1);
         if(w[giver]<h[i] AND [giver]>mT[giver]){
            qty=min( D[giver]-mT[giver],
             MT[taker]-D[taker], L(k,2) )
            qty=min( M[m][2] ), m=j..k-1
            L(j,2) += qty;
            D[taker] += qty; M[j][2] -= qty;
            L(k,2) -= qty;
            D[giver] -= qty; M[k][1] += qty;
            if qty=L(k,2)
               remove entry k from L
            for every entry m from j+1 to k-1 {
               M[m][1] += qty;
               M[m][2] -= qty;
               if M[m][2]=0
                  end=1; } }
      (endwhile) }
      k=j; end=0;
      while M[k][1]>0 AND end=0 AND
       MT[taker]>D[taker] {
         k--; giver=L(k,1);
         if w[giver]<h[i] AND D[giver]>mT[giver]{
            qty=min( D[giver]-mT[giver],
             MT[taker]-D[taker], L(k,2) )
            qty=min( M[m][1] ), m=k+1..j
            L(j,2) += qty;
            D[taker] += qty; M[j][1] -= qty;
            L(k,2) -= qty;
            D[giver] -= qty; M[k][2] += qty;
            if qty=L(k,2)
               remove entry k from L
            for every entry m from k+1 to j-1 {
               M[m][1] -= qty;
               M[m][2] += qty;
               if M[m][1]=0
                  end=1; } }
      (endwhile) } }
```

Figure 2. The second section of the scheduling algorithm

After the second section the schedule for our example is as follows:

L(task, duration)	*M*
(T1, 6)	(0, 0)
(T3, 0.5)	(2, 7.5)
(T2, 5.5)	(2.5, 0)
(T3, 2)	(12, 0)

The tasks T_1 and T_2 have been given additional amounts of computation time taken from the task T_3. The latter one is, on the other hand, preempted with T_2. In the last section the scheduling algorithm finds such duplicate entries and tries to resolve them, but only if such rearrangement would not interfere with the timing constraints of the entries that are being shifted in time (which are all the ones between the merging two). The third phase of the algorithm is shown in Fig. 3. Finally, the schedule for the example real-time system takes the following form :

L(task, duration)	*M*
(T1, 6)	(0, 0)
(T2, 5.5)	(2, 0)
(T3, 2.5)	(11.5, 0)

```
for every entry i : F[L(i,1)]>i
   repeat {
      k=i+1; z=L(i,1);
      max_up=max_dn=MT[z];
      while L(k,1)!=z {
         max_dn=min( max_dn, M[k][1] );
         max_up=min( max_up, M[k][2] ); }
      if max_dn>=L(i,2) {
         L(k,2) += L(i,2);
         for every entry m from i+1 to k-1 {
            M[m][1] -= L(i,2);
            M[m][2] += L(i,2); }
         remove entry i from L; }
      else if max_up>=L(k,2) {
         L(i,2) += L(k,2);
         for every entry m from i+1 to k-1 {
            M[m][1] += L(k,2);
            M[m][2] -= L(k,2); }
         remove entry k from L; } }
   while F[z]>k AND L(i,2)>0;
```

Figure 3. The third section of the scheduling algorithm

We now have a schedule in which the tasks with higher priorities have an optimal arrangement of computing time. In addition, there is no preempting which was one of the priorities during the design of the algorithm. If we are running on a multiprocessor system, a genetic algorithm or any other parallelizable algorithm can occupy all the available processors in its time interval.

As it was mentioned before, the scheduling algorithm was devised primarily for scheduling genetic algorithms as tasks in an imprecise real-time system, but it can also be used for scheduling of 'ordinary' imprecise systems' tasks. If that is the case, the minimum task running time should be set to the computational time of the mandatory part m_i, and the maximum task running time to the sum of the mandatory and the optional time o_i. Finally, the user may choose to schedule the system with some of the existing algorithms, in which case the mandatory and optional part of a genetic algorithm should only formally be defined as it was described in section III.

The schedule itself is static and fully determined prior to system execution. Further development of the algorithm would include dynamical scheduling of the tasks, which would allow certain parameters, as deadline times or task weights, to change during the process.

V. ACKNOWLEDGMENT

This work was carried out within the research project "Problem-solving Environments in Engineering", supported by the Ministry of Science and Technology of the Republic of Croatia.

REFERENCES

[1] Budin, L., Jakobović, D., Golub, M. (1998), "Parallel Adaptive Genetic Algorithm", *Proc. Int. Conf. Neural Computing NC'98*, Wienna, October 1998.

[2] Davis, L. (1991) *Handbook of Genetic Algorithms,* Van Nostrand Reinhold, New York.

[3] Garvey, A., Lesser, V. (1995), "Representing and scheduling satisfying tasks", *Imprecise and approximate computation*, Kluwer Academic Publishers, pp. 23-34

[4] Leung, Joseph Y-T. (1995), "A survey of scheduling results for imprecise computation tasks", *Imprecise and approximate computation*, Kluwer Academic Publishers, pp. 35-42

[5] Liu, J. W-S., Lin, K-J., Shih, W-K., Yu, A. C., Chung, J-Y., Zhao, W. (1991), "Algorithms for Scheduling Imprecise Computations", *IEEE Computer*, 24, pp. 58-68

[6] Michalewicz, Z. (1992), *Genetic Algorithms + Data Structures = Evolutionary Programs*, Springer-Verlag, Berlin.