Evolution-based scheduling of fault-tolerant programs on multiple processors

Piotr Jedrzejowicz[1], Ireneusz Czarnowski[1], Henryk Szreder[1], and Aleksander Skakowski[1]

[1]Chair of Computer Science, Gdynia Maritime Academy, ul.Morska 83
81-225 Gdynia, Poland
{pj,irek,hsz,askakow}@wsm.gdynia.pl

Abstract. The paper introduces a family of scheduling problems called fault-tolerant programs scheduling (FTPS). Since FTPS problems are, in general, computationally difficult, a challenge is to find effective scheduling procedures. Three evolution-based algorithms solving three basic kinds of FTPS problems have been proposed. The problems involve scheduling multiple variant tasks on multiple identical processors under time constraints. To validate the algorithms computational experiment has been carried. Experiment results show that evolution based algorithms produce satisfactory to good solutions in reasonable time.

1 Introduction

Complex computer systems are often facing the conflicting requirements. One of the frequently encountered conflicts in the fault-tolerant computing involves high dependability and safety standards required versus system performance and cost.

A unit of software is fault-tolerant (f-t) if it can continue delivering the required service after dormant imperfections called software faults have become active by producing errors. To make simplex software units fault-tolerant, the corresponding solution is to add one, two or more program variants to form a set of N$\geq$2 units. The redundant units are intended to compensate for, or mask a failed software unit. The evolution of techniques for building f-t software out of simplex units has taken several directions. Basic models of f-t software are N-version programming (NVP) [1], recovery block (RB) [15], [12], and N-version self-checking programming (NSCP) [18], [13]. A hybrid solution integrating NVP and recovery block is known as the consensus recovery block [16]. In all the above listed techniques, the required fault tolerance is achieved by increasing the number of independently developed program variants, which in turn leads to a higher dependability at a cost of the resources used. Self-configuring optimal programming [5] is an attempt to combine some techniques used in RB and NVP in order to exploit a trade-off between software dependability and efficiency. In fact, the approach does attempt to dynamically optimize a number of program variants run but it offers local optimization capabilities rather then global ones.

Relation between use of resources and reliability is not straightforward, and often difficult to identify [16], [2]. Yet, besides the already mentioned self - configuring optimal programming, some direct optimization models of software with redundancy have been proposed [3]. Rationale for optimizing a single f-t program structure is clearly limited. Hence, it is suggested to manage complexity and costs issues of the fault-tolerant programs not at a single program level, and thus, to introduce a family of scheduling problems called fault-tolerant programs scheduling (FTPS). The paper focuses on multiple-variant tasks. Concept of the multiple-variant task (or program) serves here as a general model of the fault-tolerant program structure. Each task is characterized by: ready time, due date or deadline, number of variants and variant processing time. FTPS problem is how to distribute (or schedule) the f-t components among processing elements deciding also how to choose their structure or execution mode to achieve some overall performance goals. FTPS problems differ from the traditional scheduling problems. Their decision space is extended to include additional variables describing f-t components structure or execution mode (i.e. number of variants and adjudication algorithm used).

Considering important application areas for the f-t programs (real-time systems, highly dependable systems, safety systems), some useful optimality criteria for FTPS problems could include (among others):

- total number of the executed program variants scheduled without violating deadlines V ($V = \sum V_j$, where V_j is the number of variants of task $j, j = 1, 2, \ldots, n$, used to construct a schedule);
- sum of penalties for violating time and dependability constraints - PF ($PF = \sum PF_j$, where PF_j is the penalty associated with scheduling task j).

It should be noted that scheduling fault-tolerant, multiple variant programs requires a different approach than scheduling programs to achieve system fault tolerance in presence of hardware faults (the latter has been investigated in, for example, [14],[7]).

2 FTPS Problem Formulation

To identify and classify scheduling problems Graham notation [8] is used. In this notation, three fields represent, respectively: processor environment, task with resource characteristics, and optimality criterion. The paper considers three basic FTPS problems. The first one, denoted as $P|m-v, r_j|V$, is characterized by multiple, identical processors (P) and a set of multiple-variant ($m-v$) tasks. Each task is a set of independent, non-preemptable variants with ready times -(r_j) and deadlines differing per task. Variants have arbitrary processing times. Optimization criterion is the number of the program variants scheduled without violating deadlines. Decision variables include assignment of tasks to processors and number of scheduled variants for each task. Tasks can not be delayed. There might be a constraint imposed on minimal number of variants of each task to be scheduled.

As it was pointed out in [4], there exist scheduling problems where tasks have to be processed on more than one processor at a time. During an execution of these multiple-processor tasks communication among processors working on the same task is implicitly hidden in a "black box" denoting an assignment of this task to a subset of processors during some time interval. To represent multiprocessor problems a $size_j$ parameter denoting processor requirement is used. Thus, the second of the considered problems denoted as $P|r_j, size_j|V$ is characterized by multiple, identical processors and a set of multiple-processor tasks. Each task is a set of independent, non-preemptable variants with ready times and deadlines differing per task which have to be processed in parallel. Tasks have arbitrary processing times, which may differ per size chosen. Optimization criterion remains the number of the program variants scheduled. Decision variables include assignment of tasks to processors and size of each task. Tasks can not be delayed. There might be a constraint imposed on minimal size of each task to be scheduled.

The third FTPS problem denoted as $P|b-t-b, r_j|PF$ is characterized by multiple, identical processors and a set of two-variant tasks. Each task is a set of two independent, non-preemptable variants with ready times and deadlines differing per task, which have to be processed in one of the three possible modes. In the first mode only the first variant is executed. The second mode involves execution of only the second variant. Third mode requires parallel, that is back-to-back ($b-t-b$) processing of both variants. Tasks have arbitrary processing times, which may differ per execution mode chosen. Optimization criterion is the expected sum of penalties associated with task delays and failure probabilities. Decision variables include assignment of tasks to processors and execution mode for each task. Tasks can be delayed. $P|b-t-b, r_j|PF$ assumes that the following additional information with respect to each task is available: version reliabilities, penalty for delay, penalty for the safe failure and penalty for the unsafe failure.

The problem $P|m-v, r_j|V$ can be considered as a general model of scheduling N-version programs under hard real-time constraints. The problem $P|r_j, size_j|V$ can be considered as a model of scheduling N-version programs or recovery block schemes, or consensus recovery block schemes, or mixture of these under hard real-time constraints. Finally, $P|b-t-b, r_j|PF$ can be considered as a model of scheduling fail-safe tasks under soft real-time constraints (see for example [17]).

3 Evolution-based Approach to Solving FTPS Problems

In general, scheduling of independent, non-preemptable tasks with arbitrary arrival and processing times is NP-hard [4]. FTPS problems are not computationally easier. It is easily shown by polynomial transformation from the result of [6] that FTPS problems of maximizing number of variants executed even on one processor ($1|r_j|V$) is NP-hard. Similar conclusion can be inferred with respect to complexity of the $P|r_j, size_j|V$ problem. $P|b-t-b, r_j|PF$ problems with penalty function aggregating dependability/reliability properties with schedule length, maximum lateness, number of tardy tasks, etc., remain NP-hard. Since the dis-

cussed FTPS problems are, in general, NP-hard it is not likely that polynomial-time algorithms solving them can be found. The following subsections include a description of the three evolution-based algorithms developed to deal with the $P|m-v, r_j|V$, $P|r_j, size_j|V$, and $P|b-t-b, r_j|PF$ problems, respectively.

3.1 Algorithm EVOLUTION $P|m-v, r_j|V$

Algorithm EVOLUTION $P|m-v, r_j|V$ is based on the following assumptions:

- an individual within a population is represented by vector $x_1, \ldots, x_n$ where x_j, $(1 \leq x_j \leq n_j)$ is the number of variants of the task j that will be executed;
- initial population is composed from random individuals;
- each individual can be transformed into a solution by applying *APPROX* $P|m-v, r_j|V$, which is a specially designed scheduling heuristic;
- each solution produced by the *APPROX* $P|m-v, r_j|V$ can be directly evaluated in terms of its fitness, that is number of variants scheduled;
- new population is formed by applying three evolution operators: crossover, offspring random generation, and transfer of some more "fit" individuals. Some members of the new population undergo further transformation by means of mutation.

EVOLUTION $P|m-v, r_j|V$ involves executing steps shown in the following pseudo-code:

set population size PS, generate randomly an initial population I_0;
set evolution parameters x, y, z $(0 < x, y, z < 1,\ x+y+z=1)$;
set $i := 0$; (iteration counter);
While no stoping criterion is met **do**
  set i:=i+1;
  calculate, using *APPROX* $P|r_j|V$, fitness factor for each individual in I_{i-1};
  form new population I_i by: selecting randomly x/PS individuals from I_{i-1} (probability of selection depends on fitness of an individual); producing y/PS individuals by applying crossover operator to x/PS previously selected individuals; randomly generating z/PS individuals; applying mutation operators to a random number of just created individuals;
End While

APPROX $P|m-v, r_j|V$ heuristics used within *EVOLUTION* $P|m-v, r_j|V$ algorithm is carried in three steps:
Step 1. Set of variants that belong to an individual is ordered using earliest ready time as the first, and earliest deadline as the second criterion.
Step 2. List-schedule variants, allocating them one-by-one to one of the processors, minimizing at each stage processors idle-time, and not violating time constraints. If there are more than one feasible allocation with identical, minimal, idle time, choose a solution where scheduled variant finishes **later**. Repeat list scheduling with the opposite rule for breaking times (i.e. choosing a solution

where scheduled variant finishes **earlier**).
Step 3. If variants can not be allocated to processors without violating deadlines than fitness of the solution is set to 0. Otherwise evaluation function generates fitness factor equal to max { $\sum x_i$ (**later**), $\sum x_i$ (**earlier**) }.

To summarize main features of the *EVOLUTION* $P|m-v, r_j|V$, a sheme proposed by [9] is used: *individuals* - part of solution; *evolution process* - steady state evolution with variable fixed part; *neighborhood* - unstructured; *information source* - two parents; *unfeasibility* - penalized; *intensification* - none; *diversification* - noise procedure (mutation).

3.2 Algorithm EVOLUTION $P|r_j, size_j|V$

Algorithm EVOLUTION $P|r_j, size_j|V$ is based on the following assumptions:

- an individual is represented by n-element vector $y_{\sigma(1)}, \ldots, y_{\sigma(n)}$ where σ is a permutation, y_j $(1 \leq y_j \leq max\{size_j\})$ is a size of the task j to be executed;
- the place of an element within $y_{\sigma(1)}, \ldots, y_{\sigma(n)}$ tells in which order the algorithm $APPROX$ $P|r_j, size_j|V$ will allocate it to processors;
- an initial population is composed in part from individuals with random size and order of tasks, and in part from individuals with random size and order determined by non-decreasing ready time as the first criterion and non-decreasing deadlines as the second;
- each individual can be transformed into a solution by applying $APPROX$ $P|r_j, size_j|V$, which is a specially designed algorithm for scheduling multiple-variant and multiple-processor tasks;
- each solution produced by the $APPROX$ $P|r_j, size_j|V$ can be directly evaluated in terms of its fitness, that is total of task sizes scheduled without violating deadlines;
- new population is formed as in $EVOLUTION$ $P|m-v, r_j|V$

$EVOLUTION$ $P|r_j, size_j|V$ involves executing steps shown in the following pseudo-code:

set population size PS, generate an initial population I_0 (part of the individuals with random size and order, and part with random size and fixed order);
set $i := 0$; (iteration counter);
While no stoping criterion is met **do**
set i:=i+1;
calculate, using $APPROX$ $P|r_j, size_j|V$, fitness factor for each individual in I_{i-1};
form new population I_i by: selecting randomly a fixed number of individuals from I_{i-1} (probability of selection depends on fitness of an individual); producing one part of individuals by applying crossover operator to previously selected individuals from I_{i-1}; generating one part of individuals with random size, and order; generating another part with

random size, and fixed order; applying mutation operators to a random number of just created individuals;

EndWhile

APPROX $P|r_j, size_j|V$ algorithm used within *EVOLUTION* $P|r_j, size_j|V$ is carried in three steps:
Step 1. Set loop over tasks.
Step 2. Within the loop, allocate current task to multiple processors minimizing beginning time of its processing. Continue with tasks until all have been allocated.
Step 3. If the resulting schedule has task delays, fitness factor of the individual from which the schedule is generated is set to 0. Otherwise, fitness factor is set to $\sum size_i$.

Main features of the *EVOLUTION* $P|r_j, size_j|V$ include: *individuals* - part of solution; *evolution process* - steady state evolution, population of constant size; *neighborhood* - unstructured; *information source* - two parents; *unfeasibility* - penalized for a solution, repaired for an individual; *intensification* - none; *diversification* - noise procedure (mutation).

3.3 Algorithm *EVOLUTION* $P|b-t-b, r_j|PF$

Algorithm *EVOLUTION* $P|b-t-b, r_j|PF$ is based on the following assumptions:

- an individual within a population is the solution represented by matrix **S** of four rows and n columns where element $s_{1,j}$ - represents number of task, $s_{2,j}$ - its execution mode, $s_{3,j}$ - number of processor on which task is to be executed, and $s_{4,j}$ - number of another processor allocated to execute a task in fail-safe mode, otherwise $s_{4,j} = 0, j = 1, 2, \ldots, n$;
- an initial population is composed, in part, from random individuals, and in part, from individuals with values in the first row, that is numbers of task, ordered by their non-decreasing ready time as the first criterion and non-decreasing deadlines as the second. Values of the remaining rows are random;
- each solution can be directly evaluated in terms of its fitness using *EVALUATION* $P|b-t-b, r_j|PF$ algorithm;
- new population is formed by applying the some principles as in *EVOLUTION* $P|m-v, r_j|V$.

EVOLUTION $P|b-t-b, r_j|PF$ involves executing steps shown in the following pseudo-code:

set population size PS, generate an initial population I_0 (part of the individuals with random order of tasks, and part with the fixed one);
set $i := 0$; (iteration counter);
While no stopping criterion is met **do**
set $i := i + 1$;
calculate, using *EVALUATION* $P|r_j|PF$ algorithm, fitness factor for each individual in I_{i-1};

form new population I_i by: selecting randomly fixed number of individuals from I_{i-1} (probability of selection depends on fitness of an individual); producing part individuals by applying crossover operator to previously selected individuals from I_{i-1}; generating another part with random size and order; generating further part with random size and fixed order; applying mutation operators to a random number of just created individuals;

End While

EVALUATION $P|b-t-b,r_j|PF$ algorithm is carried in three steps:
Step 1. Set loop over tasks.
Step 2. Within the loop, allocate current task to a processor (or processors) specified in the second row of matrix representing an individual, assuming processing begins as early as possible. Calculate penalty for delay and expected penalties for failing safe and failing unsafe. Update total incurred penalties. Continue with tasks until all have been allocated.
Step 3. Set fitness factor of the individual from which the schedule is generated to total incurred penalties.

Main features of the *EVOLUTION* $P|b-t-b,r_j|PF$ include: *individuals* - feasible solutions of the problem; *evolution process* - steady state evolution, population of constant size; *neighborhood* - unstructured; *information source* - two parents; *unfeasibility* - repaired; *intensification* - none; *diversification* - noise procedure (mutation).

4 Computational Experiment Results

To verify and evaluate the presented evolution-based algorithms several computational experiments have been carried. The first experiment included 50 randomly generated scheduling problems belonging to the $P|m-v,r_j|V$ class. Problems involved scheduling 10 - 29 multiple (3 5) variant tasks on 2 - 5 processors. Problems have been solved using *EVOLUTION* $P|m-v,r_j|V$ algorithm and simple PJ-heuristics proposed in [10]. Solutions have been compared with an upper bound on optimal solution, which can be calculated in polynomial time. In Table 1 relative distances from upper bound on optimal solutions are shown.

To evaluate the performance of *EVOLUTION* $P|m-v,r_j|V$ against optimal algorithm a subset of 20 problems has been selected from the previously used 50 cases. For these problems exact solutions have been obtained using a VISUAL C++ application, based on CPLEX callable library. Mean, smallest and largest relative errors for the *EVOLUTION* $P|m-v,r_j|V$ algorithm and PJ-heuristics are, respectively, 1.34% and 4.05% (mre), 0.00% and 4.76% (sre) and 0.00% and 14.63% (lre).

Next experiment included 50 randomly generated scheduling problems belonging to the $P|r_j,size_j|V$ class. Problems involved scheduling 10 - 15 tasks, each task with maximum size of 4, on 3 - 10 processors. All problems have been solved using *EVOLUTION* $P|r_j,size_j|V$ algorithm, and APPROX heuristics proposed in [11]. Solutions have been compared with the optimal ones obtained

Table 1. Mean, smallest and largest relative distances (mrd, srd, lrd) from an upper bound on optimum solution for the *EVOLUTION* $P|m-v,r_j|V$ algorithm and *PJ*-heuristics

No of proc.	mrd *EVOL.*	mrd *PJ-HEUR.*	srd *EVOL.*	lrd *EVOL.*	srd *PJ-HEUR.*	lrd *PJ-HEUR.*
2	13.80%	20.17%	5.00%	22.45%	7.50%	38.78%
3	15.68%	18.17%	10.53%	24.14%	14.29%	27.59%
4	18.39%	21.50%	10.81%	25.53%	15.52%	31.91%
5	18.52%	19.88%	11.43%	22.86%	13.16%	26.47%
Overall	16.60%	19.93%				

using branch-and-bound algorithm. In Table 2 the respective mean relative errors are shown.

Table 2. Mean, smallest and largest relative errors (mre, sre, lre) from an optimum solution for the *EVOLUTION* $P|r_j,size_j|V$ algorithm, and APPROX heuristics.

No of proc.	mre *EVOL.*	mre *APPROX*	sre *EVOL.*	lre *EVOL.*	sre *APPROX*	lre *APPROX*
3	0.00%	4.55%	0.00%	0.00%	0.00%	4.55%
4	0.00%	4.84%	0.00%	0.00%	0.00%	9.68%
5	1.04%	1.84%	0.00%	4.17%	0.00%	5.56%
6	2.07%	2.32%	0.00%	5.00%	0.00%	8.00%
7	2.77%	7.66%	0.00%	8.33%	0.00%	24.14%
8	1.13%	4.36%	0.00%	3.33%	0.00%	14.81%
9	0.61%	5.01%	0.00%	3.03%	0.00%	12.12%
10	0.00%	8.26%	0.00%	0.00%	2.44%	15.63%
Overall	0.95%	4.85%				

The last experiment included 50 randomly generated scheduling problems belonging to the $P|b-t-b,r_j|PF$ class. Problems involved scheduling 10 tasks on 2 - 4 processors. All problems have been solved using *EVALUATION* $P|b-t-b,r_j|PF$ algorithm, and *GREEDY* heuristics. In *GREEDY* heuristics tasks are ordered in accordance with non-decreasing ready-times as the first criterion and non-decreasing due dates as the second one. Then, all possible combinations of execution modes are considered. Solutions have been compared with the optimal ones obtained using branch-and-bound algorithm. In Table 3 the respective relative errors are shown.

Table 3. Mean, smallest and largest relative errors (mre, sre, lre) from an optimum solution for the *EVALUATION* $P|r_j|PF$ algorithm, and GREEDY heuristics.

No of proc.	mre *EVOL.*	mre *GREEDY*	sre *EVOL.*	lre *EVOL.*	sre *GREEDY*	lre *GREEDY*
2	24.02%	11.92%	0.00%	46.09%	0.00%	36.84%
3	28.30%	11.16%	14.59%	53.59%	0.00%	39.46%
4	31.92%	12.54%	7.51%	69.69%	0.00%	34.46%
Overall	28.08%	11.87%				

5 Conclusions

In this paper fault tolerant programs scheduling problems are considered. Since, as it was shown, FTPS problems are, in general, computationally difficult, a challenge is to find an effective scheduling procedure. Three evolution-based algorithms solving, respectively, $P|m-v, r_j|V$, $P|r_j, size_j|V$ and $P|b-t-b, r_j|PF$ problems have been proposed. The approach seems promising and the algorithms produce excellent to satisfactory solutions. In case of $P|m-v, r_j|V$ experiment results proved that the evolution-based approach generates considerably better results then a good polynomial-time heuristic algorithm. It also produces results not too far from an upper bound on optimal solution. In case of $P|r_j, size_j|V$ evolution based algorithm produced, on average, solutions not worse more than 1% from optimum. For $P|b-t-b, r_j|PF$ problems evolution based algorithm generates only satisfactory solutions (average error margin is about 28% and this performance is worse than greedy heuristics). For all three evolution-based algorithms there is still room for improvements which can be expected from introducing some better solution-improvement strategies at a self-adaptation phase. Another promising direction is searching for effective hybrid evolutionary-heuristics algorithms.

References

1. Avizienis A., L.Chen: On the implementation of the N-version programming for software fault tolerance during execution, *Proc. IEEE COMPSAC 77*, 1977, pp. 149 - 155.
2. Belli F., P.Jedrzejowicz: Fault Tolerant Programs and Their Reliability, *IEEE Trans. on Reliability*, 39(2), 1990, pp. 184 - 192.
3. Belli F., P.Jedrzejowicz: An Approach to the Reliability Optimization of Software with Redundancy, *IEEE Trans. on Software Engineering*, 17(3), 1991, pp. 310 - 312.
4. Blazewicz J., K.H.Ecker, E.Pesch, G.Schmidt, J.Weglarz: Scheduling Computer and Manufacturing Processes, *Springer*, Berlin, 1996.
5. Bondavalli A., F.Di Giandomenico, J.Xu: Cost-effective and flexible scheme for software fault tolerance, *Computer System Science & Engineering*, 4, 1993, pp. 234 - 244.

6. Garey M.R., D.S.Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness, *W.H.Freeman*, New York, 1979.
7. Ghosh S., R.Melham, D.Mosse: Fault-Tolerance through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems, *IEEE Trans. on Parallel and Distributed Systems*, 8(13),1997, pp. 272-284.
8. Graham R.L., E.L.Lawler, J.K.Lenstra, A.H.G.Rinnooy Kan: Optimization and approximation in deterministic sequencing and scheduling: a survey, *Annals Discrete Math.*, 5, 1979, pp. 287 - 326.
9. Hertz A., D.Kobler: A Framework for the Description of Population Based Methods, *16th European Conference on Operational Research - Tutorials and Research Reviews*, Brussels, 1998, pp. 1 - 23
10. Jedrzejowicz P.: Scheduling multiple-variant programs under hard real-time constraints, *Proc. International Workshop Project Management and Scheduling*, Istambul, 1998, pp.
11. Jedrzejowicz P.: Maximizing number of program variants run under hard time constarints, *Proc. International Symposium Software Reliability*, Paderborn, 1998, pp.
12. Kim K.H.: Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults, *Proc. 4th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1984, pp. 526 - 532.
13. Laprie J.C., J.Arlat, C.Beounes, K.Kanoun: Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures, *IEEE Computer*, 23(7), 1990, pp. 39-51.
14. Liestman A.L., R.H.Campbell: A Fault-tolerant Scheduling Problem, *IEEE Trans. on Software Engineering*, SE-12(11), 1988, pp. 1089-1095.
15. Melliar-Smith P.M., B.Randell: Software reliability: the role of programmed exception handling, *SIGPLAN Notices* 12(3), 1977, pp. 95 - 100.
16. Scott R.K., J.W.Gault, D.F. Mc Allister: Fault tolerant software reliability modelling, *IEEE Trans. on Software Engineering*, 13(5), 1987, pp. 582 - 592.
17. Vouk, M.A.: Back-to-back testing, *Information and software technology*, vol.32, no 1, 1990, pp. 34-35.
18. Yau S.S., R.C.Cheung: Design of Self-Checking Software, *Proc. Int. Conf. on Reliable Software*, 1975, pp. 450 - 457.

Acknowledgement: This research has been supported by the KBN grant NR 301/T11/97/12