Tool System and Algorithms for Scheduling of Computations in Integrated Modular Onboard Embedded Systems

Vasily V. Balashov*, Valery A. Kostenko*, Vadim A. Balakhanov*, Sergei A. Tutelian*

** Department of Computational Mathematics and Cybernetics,*
Lomonosov Moscow State University,
Leninskie Gory, MSU, 1, Bldg. 52, Room 764, Moscow, Russia,
(e-mail: {hbd, kost, baldis}@lvk.cs.msu.su, sergei.tutelian@yandex.ru)

Abstract: Scheduling of computations is an essential step in the process of real-time systems design. In this paper, the scheduling problem is addressed for integrated modular onboard embedded systems (IMOES). This class of systems uses a mix of static and dynamic scheduling. A family of algorithms for workload distribution and schedule construction for IMOES is presented, along with the results of their experimental evaluation on synthetic and real-world data. The algorithms are implemented in a tool system, which is accepted for operation by one of the leading Russian aircraft design companies.

Keywords: Real-Time Multiprocessor Systems; Microprocessor Based Control Systems; Scheduling Algorithms; Design Tools and Application Software.

1. INTRODUCTION

Onboard embedded systems for aircraft and naval purposes are responsible for control of vehicle subsystems and for processing of control tasks such as navigation, collision avoidance, etc. Instead of having federated architecture with dedicated hardware and software for each logical subsystem, the current trend for onboard systems is to run the software of multiple subsystems on a unified platform with standard API and modular hardware, which constitute integrated modular architecture (Wind River / IEEE, 2008). Such integrated modular onboard embedded systems (IMOES) include from several to dozens of modules, each usually containing a multicore CPU. Modules are connected by a network, typically a switched one with support for virtual channels (Schaadt, 2007). Workload for such system consists of a set of periodic tasks with data dependencies. A dependency between tasks corresponds to a message to be transferred between the sender task and the receiver task. In this paper we consider partitioned task sets in which tasks are grouped into subsets (partitions), each of which represents an application.

As IMOES are inherently multiprocessor systems with unified interface for application tasks, a problem of workload scheduling arises, which includes distribution of partitions to CPU cores and construction of partitions execution schedules. IMOES of a modern aircraft runs several hundred tasks with complex data dependencies, so the scheduling problem needs tool support.

In this paper we present a tool system for scheduling of computations in IMOES and describe the scheduling algorithms implemented in this system. Results of tool and algorithms evaluation are also presented, both for synthetic tasks sets and a task set from a real-world IMOES.

This work is partially supported by the Russian Foundation for Basic Research under grant №16-07-01237.

2. STRUCTURE OF THE SCHEDULE

A two-level scheduling scheme is used in IMOES. On the top level, partitions execution is organized via static schedule. For each partition, there is a set of execution windows, i.e. time intervals in which tasks from the partition are executed. We consider systems in which every partition is statically bound to a specific CPU core, thus all execution windows for the partition belong to that core. Partitions binding to CPU cores, as well as the set of execution windows, are to be constructed in advance, prior to the target system startup.

Within an execution window, tasks from the corresponding partition are scheduled dynamically according to their fixed priorities. This constitutes the bottom level of the scheduling scheme. A task which depends on data from another task with the same frequency (from the same or from another partition) can start only after data arrival. This is a synchronous dependency between tasks. A data dependency between tasks with different frequencies does not force the receiver task to wait for input data (asynchronous dependency).

There is a set of constraints on the partitions execution windows defined by the target system (IMOES) specifics. For instance, there can be lower and upper limits on the execution window duration.

3. SCHEDULING PROBLEM

The scheduling problem for a given IMOES and workload (set of periodic tasks grouped into partitions) breaks down into following subproblems:

1) distribute the workload, i.e. bind the partitions to CPU cores;

2) construct the schedule of partitions execution windows.

There are following constraints on partitions binding:

- binding for some partitions can be restricted to a subset of CPU cores, e.g. cores of a specific module;
- total load for a core must not exceed a given limit, which can be defined individually for each core;
- each partition must be bound do a single CPU core.

In course of IMOES evolution, the set of partitions can be extended. So the algorithms for workload distribution must support incremental mode in which previously constructed binding for partitions (all or some of them) cannot be altered.

CPU core load by a partition is calculated as a sum $\sum_i f_i \cdot c_i$, where f_i is the task frequency and c_i is its worst case execution time (WCET). For every task, its WCET is a part of input data for the scheduling problem; for different types of CPU cores used in the target system, WCET for the same task may be different.

We consider workload distribution as an optimization problem with network load as the objective function to be minimized. The network load is calculated as the total size of messages transferred between modules during a single iteration of the schedule, the duration of which is estimated as the least common multiple of the tasks' periods. Messages between tasks running on the same module are transferred through the module's local memory and do not contribute to the network load.

Constraints on the schedule of partitions execution windows are as follows. For each core, the execution windows must not overlap; exactly one partition can be assigned to an execution window; for each core, only partitions bound to this core can be assigned to execution windows for this core; there are lower and upper limits on the execution window duration, common for all cores; the set of execution windows for all cores of the same module must be the same.

The exact set of constraints on workload distribution and on the schedule of partitions execution windows is determined by the specifics of the target IMOES and may vary from system to system. The constraints described above correspond to an IMOES of a modern Russian aircraft and can be considered typical for an onboard computer system.

As the tasks within execution windows are scheduled dynamically, the schedule of windows must guarantee that all of the tasks' executions are performed within deadlines under control of a given dynamic scheduler. Single execution of a periodic task is called a job. If T is the task period (reciprocal of its frequency), i is the number of task iteration (numbering starts from 1), then the deadline interval for the job is $\left[(i-1)\cdot T; i\cdot T\right]$.

In terms of jobs, the schedule of windows must guarantee that all jobs of all tasks are executed within corresponding deadline intervals. This can be checked via construction of a job execution sequence taking into account tasks' priorities and data dependencies, and assuming that each job's execution duration (not counting preemption and waiting for input data) equals to task's WCET.

4. OVERVIEW OF EXISTING SOLUTIONS

Since the scheduling problem for IMOES is divided into two sufficiently different subproblems (workload distribution and execution windows schedule construction), we will consider existing solutions for these subproblems separately.

The workload distribution problem for IMOES resembles the multiple container packing problem (MCPP). The latter is the problem of choosing several disjoint subsets of n items to be packed into distinct containers, such that the total value of the selected items is maximized, without exceeding the capacity of each of the containers (Raidl, 1999). In our case partitions correspond to items, CPU cores correspond to containers. Item's volume is the partition's contribution to the CPU core utilization; item's cost is the part of traffic which becomes "internal" for the module when the item is placed in the container (i.e. the partition is assigned to a CPU core). However the problem statement differs from traditional MCPP (with fixed volumes and costs of objects). First, the volume of an item depends on the choice of the container, as a task can have different WCETs for different types of CPU cores. Second, the value of an item depends on the container's contents, i.e. the set of other items in the container.

Due to this difference in the problem statement, existing algorithms for solving of MCPP cannot be used "as is". Following approaches applicable to MCPP were taken as the base for our workload distribution algorithms:

- greedy heuristic search – a common way to quickly find an acceptable solution when the constraints are not too strict; see (Crainic et al, 2012) for example of its application to MCPP;
- branch-and-bound method – looks for the exact optimal solution, working time and scalability greatly depends on the quality of search space pruning; applied to MCPP in (Fukunaga et al, 2007);
- genetic algorithms – chosen for presumably good scalability (opposite to branch-and-bound) and ability to avoid dead ends during the search (opposite to greedy heuristics); applied to MCPP in (Raidl, 1999).

These basic approaches were modified to match the workload distribution problem stated above; see Section 5 for description of the resulting algorithms.

As IMOES, including ARINC 653-based avionics systems, are becoming more widely adopted, there is an increasing amount of research on the techniques for schedule construction for such systems. However in most of the published approaches either the workload is analysed in insufficient level of detail, or some important constraints are not taken in account.

In (Goltz, H.-J., 2009) the workload is represented as a set of partitions, and individual tasks within partitions are not considered. Each partition has a period (however in real systems, some partitions include tasks with different periods); data dependencies between partitions are not taken in account. The scheduling problem in this paper is stated as a constraint satisfaction problem, and solved using techniques of constraint logic programming.

The level of detail for workload representation in (Al Sheikh, 2010) is similar, with no individual tasks taken in account, but the data dependencies between partitions are accounted for. This makes the proposed approach more realistic, yet it remains too coarse-grain for our problem, as in fact a specific task, not a whole partition, has to wait for input data arrival. So the mixed integer linear programming approach proposed by the authors is hardly applicable in our case.

In (Easwaran et al, 2009) the workload is analysed in more detail. Individual tasks are described in terms of priority, period, execution time, deadline interval and release jitter. The authors propose to express data dependencies in terms of deadline intervals of dependent tasks. This makes the proposed approach incomplete, as there is a distinct problem of deriving the deadline intervals from the dependency graph without making the results too constraining for the main scheduling problem. In the paper, only partitions with aliquant periods are considered; this is not the case for complex IMOES with partitions corresponding to different logical subsystems. More recent papers by the same research group (see http://repository.upenn.edu/) do not solve this issue, focusing on scheduling of mixed-criticality task sets on systems with multiple modes of operation.

The author of the thesis (Craveiro, 2013) proposes a complicated approach to multi-level scheduling of workload in partitioned systems. The two-level scheme used in IMOES is a particular case of such problem. The proposed scheduling framework describes the set of target system’s resources; scheduling is performed with respect to tasks’ demand for the resources. The scheduling algorithms are flexible enough to support concurrent execution of tasks from the same partition on several CPU cores. The approach proposed in the thesis does not take in account data dependencies between tasks, and such dependencies cannot be expressed via shared resources within the described framework. The reason is that this framework does not allow specification of resource access sequence, e.g. “the receiver can access the resource *message* only after the sender”.

The MASIW toolset (Buzdalov et al, 2014) developed by the Institute for System Programming of the Russian Academy of Sciences includes a scheduling subsystem. The problems of workload distribution and scheduling of partitions are both solved by the algorithms implemented in MASIW. However the problem of partitions scheduling is stated in a very specific way, requiring any task to be started exactly in the beginning of its period (maybe with some constant offset), and to use at least one CPU tick before being (possibly) preempted by a different task. For non-aliquant period of tasks from different partitions, this requirement is too constraining, leading to local peaks of switching between very small windows. The requirement to have the same grid of execution windows for all cores of the same module makes the situation even worse. Also, the requirement to start the tasks in precisely defined time instants is not well compatible with priority-based dynamic scheduling used in many real IMOES, including those based on ARINC 653 standard. Finally, data dependencies between tasks are not taken in account by MASIW partitions scheduler.

Several commercial tools for IMOES development also include scheduling subsystems, with no detailed description of the implemented algorithms.

The conclusion from the overview is that to the date there is no published approach to scheduling of IMOES workload which solves the problem stated in Section 3 and takes in account the real-world constraints listed in that section.

5. PROPOSED SCHEDULING ALGORITHMS

5.1 Algorithms for workload distribution

As mentioned in the previous section, three approaches were selected as a base for solving the workload distribution subproblem: greedy heuristics, branch-and-bound, and genetic algorithms. All algorithms presented in this subsection minimize a common objective function, which is the total size of messages transferred between modules through the duration of the scheduling interval. Intensity of communication between a pair of partitions is estimated in the same way.

The greedy algorithm attempts to minimize the objective function by assigning most intensely interacting (communicating) partitions to CPU cores of same modules, while it is still possible without exceeding the limits on core load. The algorithm operates according to the following scheme:

```
1) choose an unassigned partition $P$ which has most intense
interaction with already assigned partitions (if no partitions
are assigned, choose the partition which most intense total
interaction with all other partitions);

2) find the module $M$, assigning $P$ to which maximizes
interaction between $P$ and partitions previously assigned to
cores of the same module;

3) choose a CPU core in $M$, to which $P$ can be assigned
without exceeding the limit on core load;
if such core exists in $M$, assign $P$ to this core and go to step 6;

// in case there is no such core in $M$

4) try to redistribute partitions previously assigned to $M$
between cores of $M$ in order to offload one of the cores so
that $P$ could be assigned to it;
if redistribution is successful, assign $P$ to this core and go to
step 6;
```

// in case *M* is not suitable for *P*

5) exclude *M* from the list of modules to which *P* can be bound; if the list is empty, **stop** (unsuccessful completion); else go to step 6;

6) if all partitions are assigned to CPU cores, then **stop**, else go to step 1.

Computational complexity of this algorithm mainly depends on the numbers of partitions (N_P), modules (N_M), and cores (N_C). The order of the algorithm's complexity is $O(N_P \cdot (N_P \cdot N_M + N_C))$.

This greedy algorithm (with limited enumeration on step 4) can quickly find a solution of the workload distribution problem. Its drawback is that for highly loaded systems and/or systems with high number of partitions and cores, it often produces significantly suboptimal results, or even fails to find correct solution.

To resolve this issue, we apply the branch-and-bound method using the greedy approach for pruning of the search space. The branch-and-bound based algorithm incrementally constructs the workload distribution by implicitly traversing a tree of partial solutions. Root of the tree is the empty partial solution. An arc from a vertex to another vertex corresponds to assignment of a partition to a CPU core. Arcs from a vertex of Nth level correspond to assignment of Nth partition (in some fixed ordering) to different (as a total, all) CPU cores. Kth arc from a vertex corresponds to assignment of a partition to the Kth core (in a continuous numbering of cores).

The tree is traversed depth-first. Search space pruning is performed as follows. If going through some arc leads to an incorrect partial solution (i.e. one that violates the constraints on CPU core load or contains non-allowed binding), the subtree starting from vertex for that partial solution is not traversed. If going through some arc leads to a partial solution with value of objective function greater than for best of the previously found correct and complete solutions (leaves of the tree), the subtree starting from vertex for that partial solution is not traversed.

Efficiency of this scheme substantially depends on the quality of search space pruning, which in turn depends on the order of tree traversal. In worst case, the branch-and-bound scheme is almost as poorly efficient as complete enumeration.

To improve the efficiency of search space pruning, we sort the partitions according to the criterion similar to the one used in the greedy algorithm. The list of partitions is composed so that the Nth partition in the list is one which has most intense interaction with (N-1) partitions already put in the list. The original greedy criterion assumes that assignment of those (N-1) partitions to cores is known, so this criterion cannot be used "as is".

The improved branch-and-bound scheme also uses a modified criterion of choosing an arc from the current vertex (the level of which defines the partition): the arcs (i.e. cores) are ordered by decrease of the amount of current partition's interaction with partitions bound to a corresponding core. This criterion is also inherited from the greedy algorithm described above.

Rough estimation of complexity for this algorithm is $O({N_c}^{N_p})$, which is expectable for an exact algorithm.

The weak point of the branch-and-bound based algorithm is its scalability (see Section 7 for details). With total number of partitions and cores over several dozens, its working time for a modern CPU exceeds 10^5 seconds (more than one day), so in current state it is applicable to moderate scale systems only. For really complex IMOES, with 50 to 100 total number of partitions and cores, better scalability is needed.

To cope with the scalability issue, we use evolutionary algorithms (EA). In order to apply the evolutionary scheme to a specific problem, following properties must be defined (Sivanandam, 2010): fitness function; solution encoding; selection, crossover and mutation operators; stopping condition.

The fitness function in our EA is the same as the objective function of the workload distribution problem (see Section 3). Solution is encoded as an array in which the position of an element corresponds to the number of partition, and the value of an element corresponds to the core to which the partition is bound. This encoding was selected to simplify the crossover and mutation operators. Selection is performed according to tournament scheme, as it has lesser tendency to prematurely converge than the proportional and the roulette-wheel schemes, which intensely prefer solutions with good value of objective function and discard other solutions. Uniform crossover is used, as it allows the population to quickly expand into previously unexplored areas of the search space, in comparison to single point crossover. To avoid frequent generation of invalid solutions (with overloaded CPU cores), we use a combination of simple mutation (some random partition migrates to another core) and exchange mutation (if simple mutation overloads the target core, then some other partition migrates in opposite direction). The algorithm stops after a specified number of iterations. General scheme of an evolutionary algorithm, as well as description of particular types of operators (uniform crossover, etc) selected for our workload distribution algorithm, can also be found in (Sivanandam, 2010).

For better performance on multicore computers, the island model was implemented (Whitley, 1999) which splits the population into several groups, performs a given number of "conventional" EA iterations within each group as a separate population, then migrates some solutions between groups; these two steps are performed in a loop, until the total number of "conventional" iterations reaches the given limit.

Order of complexity for the evolutionary algorithm is $O(N_P^2 \cdot N_{pop}^2 \cdot N_{iter}^2)$, where N_{pop} is the population size, N_{iter} is the number of iterations.

All workload distribution algorithms described above can be used in incremental mode, in which there is a set of partitions initially assigned to CPU cores, and their reassignment is prohibited.

5.2 Algorithm for scheduling the execution of partitions

This algorithm constructs a static schedule of partitions execution windows for each CPU core of the target IMOES. It takes the task set description and workload distribution as input.

The general idea of the algorithm is to construct a temporary multi-processor static schedule of jobs execution (*jobs schedule*) for the duration of the scheduling interval and, in parallel, determine the bounds of the execution windows. The jobs schedule is constructed concurrently for all processor cores of the target system. For each *i*-th core (in a continuous numbering of all processor cores), a “current time” counter t_i is maintained. Each t_i starts from *0* and can only grow through the algorithm’s execution. Jobs for the *i*-th core can only be scheduled at t_i or later.

A job is considered *ready* for scheduling on the *i*-th core, if its partition is assigned to this core, t_i belongs to the job’s deadline interval, and all synchronous input messages for this job have arrived. On each iteration the algorithm processes the core with minimum t_i. For this core the algorithm chooses a ready job with maximum priority (among ready jobs for this core); if the current execution window is shorter than minimum duration, only jobs from the current partition are considered. The chosen job is scheduled on the current core. If the job belongs to a different partition than the previous job scheduled on the same core, a new window is opened (the current window is closed), and the job starts in it. On all other cores of the same module, new windows are also opened (and current ones are closed). After a job is scheduled, arrival times for all its output messages are calculated, taking in account the delays for data transfer between modules (start deadlines of the dependent jobs are corrected accordingly); t_i is shifted to the job’s finish time, and the algorithm starts a new iteration. If no job is chosen and scheduled on an iteration (this means that no jobs are ready for the *i*-th core), the algorithm shifts t_i to the minimum time at which a ready job will be available at any core, *plus* one minimum time increment if that job is from another core. The jobs that failed to be scheduled within their deadline intervals are moved to the set of unscheduled jobs. If the current window reaches the maximum allowed duration, a new window is opened on every core of the module, possibly for the same partition as the previous window.

This fixed priority scheme of dynamic task scheduling inside the windows is identical to one used in the target class of ARINC 653 based IMOES. So we assume that if the jobs schedule constructed by the algorithm is complete (includes all jobs, with durations equal to WCET) and correct (all jobs meet their deadlines), then the scheduler on the target system will also schedule the jobs within their deadlines. In the tool system (see Section 6) this assumption is verified by simulation of the dynamic scheduler operation.

6. SCHEDULING TOOL SYSTEM

The scheduling algorithms presented in this paper were implemented in a tool system for scheduling of computations in IMOES. The target platform for the tool system is ARINC 653 based operating system developed in Russia for use in modular onboard avionics systems for airplanes and helicopters.

The tool system has following essential features: import of input data (task set and target system description) from the project database; automatic distribution of workload to modules and CPU cores, with support for manual correction; automatic construction of partition execution schedules (sets of windows), with support for manual correction; visualization of the task set structure as a graph; hierarchical visualization of workload distribution to modules and CPU cores; visualization of the schedule as a time diagram; generation of customizable reports on input data and on results of tool application; export of output data as a part of target OS configuration. The tool system supports iterative workflow, in which the next version of the target system configuration (task set, workload distribution, etc) is based on the previous version, and the scheduling algorithms are used in incremental mode. The tool system is written in C++ and uses QT 5.x GUI library. The system runs both on Windows (XP and later) and Linux.

The tool system is accepted for operation by one of the leading Russian aircraft design companies and integrated into the toolchain for onboard systems development.

7. EVALUATION OF ALGORITHMS AND THE TOOL

7.1 Scalability of the workload distribution algorithms

As mentioned in Section 5, the major drawback of both the greedy algorithm and the branch-and-bound based algorithm is their poor scalability to large systems (with over several dozens of partitions and CPU cores), while they are applicable to systems of lesser scale. Experiments were performed on synthetic input data to explore scalability of all three algorithms (including the evolutionary one). Data sets were randomly generated, with expected CPU core load of approximately 50%, synchronous dependencies between tasks from different partitions, and an average of 2 partitions per core. The number of iterations for the evolutionary algorithm was set to 1000 (100 iterations between migrations; 10 migrations). Population size was set to 1000. The number of cores on PC for EA execution (and thus, islands) was 8.

Figure 1 shows the relative deviation of the objective function on the solution from the estimated optimum, which is the function’s value on the *best* solution found by 100 sequential runs of the evolutionary algorithm on the same data. Diamond-shaped markers corresponds to the solution found by the greedy algorithm; round markers corresponds to the *worst* solution found by the EA during those 100 runs. If a point is present on the graph for the EA but missing on the graph for the greedy algorithm, then for this data set the greedy algorithm failed to find a solution.

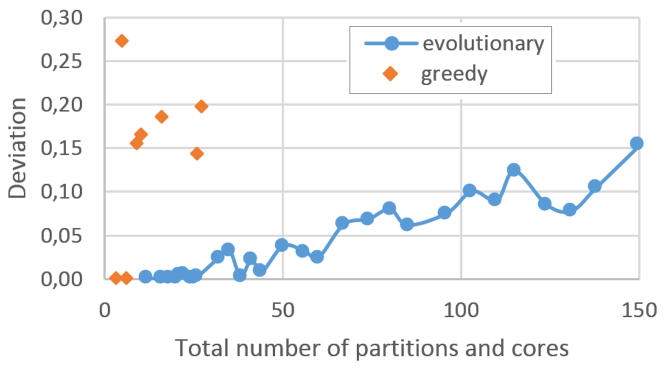

Figure 1

Figure 2 shows the growth of algorithms execution time with the scale of target system. Execution time for the greedy algorithm is not shown, as it finishes within one or several seconds.

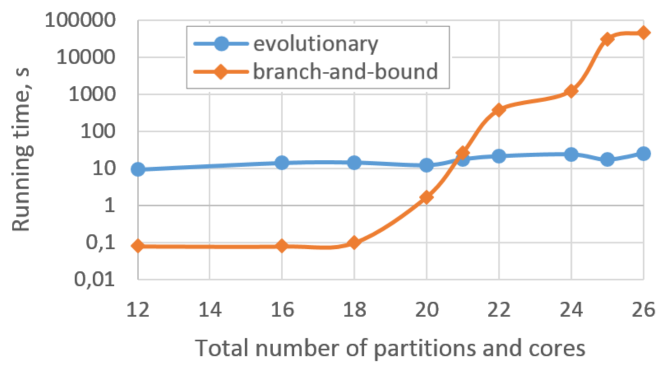

Figure 2

The two figures demonstrate that the EA is significantly more scalable than two other algorithms, which are applicable only to moderate scale systems (e.g. the system from subsection 7.2).

7.2 Evaluation on data for a real target system

The tool system was tested on several data sets for real onboard systems. The task set for one of the target systems included over 10 partitions with a total of approximately 200 periodic tasks with synchronous dependencies. Frequencies (and thus periods) of the tasks were generally not aliquant, e.g. there were tasks with frequencies of 12.5 Hz and 10 Hz. Highest frequency of a task was 100 Hz, and the lowest frequency was 1 Hz. The target system contained 3 modules with several processor cores on each module.

All three workload scheduling algorithms found good solutions; greedy and evolutionary algorithms produced nearly optimal solutions in approximately one second, and branch-and-bound found the optimum for reasonable time (less than one minute on a Core i5 CPU). The partitions scheduling algorithm, running for several seconds, successfully constructed a correct schedule of partition execution windows.

8. CONCLUSIONS AND FUTURE WORK

Future work on the scheduling algorithms and the tool system includes: improving scalability of the branch-and-bound based algorithm for workload distribution, to enable finding optimum solutions for larger scale systems; development of formal methods to prove that the constructed schedule of partitions execution guarantees correct (i.e. within deadlines) operation of the target system with task execution times less than WCET; integration of the tool system with an external IMOES simulation tool (which is under development) to provide independent verification of constructed schedules.

REFERENCES

Al Sheikh, A., Brun, O., and Hladik, P.E. (2010). Partition scheduling on an IMA platform with strict periodicity and communication delays. *Proc. 18th International Conference on Real-Time and Network Systems*, 179–188.

Buzdalov, D.V., Zelenov, S.V., Kornykhin, E.V., Petrenko, A.K., Strakh, A.V., Ugnenko, A.A., and Khoroshilov, A.V. (2014). Tools for System Design of Integrated Modular Avionics. *Proc. Institute for System Programming*, 26 (1), 201–230.

Crainic, T.G., Perboli, G., and Tadei, R. (2012). Recent advances in multi-dimensional packing problems. In *New Technologies – Trends, Innovations and Research*, 91–110. Intech, Croatia.

Craveiro, J.P. (2013). Real-Time Scheduling in Multicore Time- and Space-Partitioned Architectures. *Ph.D. thesis, University of Lisbon.*

Easwaran, A., Lee, I., Sokolsky, O., and Vestal, S. (2009). A Compositional Framework for Avionics (ARINC 653) Systems. *Proc. IEEE Real-Time Computing Systems and Applications*, 371–380.

Fukunaga, A.S., Korf, R.E. (2007). Bin Completion Algorithms for Multicontainer Packing, Knapsack, and Covering Problems. *Journal of Artificial Intelligence Research*, 28, 393–429.

Goltz, H.-J., Pieth, N. (2009). A Tool for Generating Partition Schedules of Multiprocessor Systems. *Proc. 23rd Workshop on (Constraint) Logic Programming*, 167–176.

Raidl, G.R. (1999). The multiple container packing problem: a genetic algorithm approach with weighted codings. *ACM SIGAPP Applied Computing Review*, 7 (2), 22–31.

Schaadt, D. (2007). AFDX/ARINC 664 Concept, Design, Implementation and Beyond. *SYSGO AG White Paper*.

Sivanandam, S.N., Deepa, S.N. (2010). *Introduction to Genetic Algorithms*. Springer Berlin Heidelberg, Germany.

Whitley, D., Rana, S., Heckendorn, R.B. (1999). The Island Model Genetic Algorithm: On Separability, Population Size and Convergence. *Journal of Computing and Information Technology*, 7, 33–47.

Wind River / IEEE (2008). ARINC 653 – An Avionics Standard for Safe, Partitioned Systems. *Wind River Systems / IEEE Seminar*. [http://www.computersociety.it/wp-content/uploads/2008/08/ieee-cc-arinc653_final.pdf]