Flow algorithm for scheduling in multiprocessing heterogeneous systems with integrated modular architecture

1nd Smirnov Aleksandr
Moscow State University
Moscow, Russia
alsmirnovser@gmail.com

2st Kostenko Valery
Moscow State University
Moscow, Russia
kost@cs.msu.su

***Abstract*—The article propose algorithm for scheduling tasks in heterogeneous real-time systems with an architecture of integrated modular avionics based on finding maximum flow in transport network. The results of an experimental study for periodic programs are performed.**

***Index Terms*—real-time system, IMA, scheduling, flow technique**

I. Introduction

Onboard real-time computing systems of previous generations of aircraft had a federal architecture. Programs of each onboard subsystem (for example, location, navigation, engine control) were executed on their own computers, very often specialized. The same computer was not allowed to run programs for different onboard subsystems. This made it possible to isolate the subsystem's programs from each other, but resulted in high hardware costs.

Real-time computing systems for new-generation aircraft have an architecture of integrated modular avionics (IMA) [1]. A single onboard computer is built from a set of standardized computing modules [2]. Hardware resources of a single computing module can share application programs of different onboard systems. It allows to reduce the hardware cost. However, this requires isolation of programs from different systems. Isolation extends to all resources, including register memory, processor caches, and input/output buses.

The well-developed structure of integrated modular avionics (IMA) has found application far beyond the areas of aviation alone. This architecture proved to be convenient and reliable enough for use in other integrated systems and automated process control systems.

An important point in the construction of real-time systems with the IMA architecture is the construction of a static-dynamical schedule for the execution of application programs [3]. The schedule must guarantee that programs run in the specified time interval for each program. To construct such a schedule in a real-time computing system with an integrated modular avionics architecture, it is necessary to solve the problem of constructing static-dynamic schedules. For each program, you can set the time of its execution, that can vary depending on the processor type, the directive interval, and assigning the program to subsystems that we will call partition. A static-dynamic schedule is constructed if the following parameters are defined: attaching partitions to processors; a set of windows for each partition; and opening and closing times for each window. Each partition can have its own scheduler, which is responsible for scheduling tasks. Windows are allocated to partition according to a prebuilt schedule. Programs inside the window run dynamically. You can interrupt the program and then execute it in this window or in one of the following windows in the section. The scheduling algorithms used by partitions may be different.

In this paper, we consider an algorithm for constructing multiprocessor static-dynamic schedules in heterogeneous systems based on finding the maximum flow in the transport network. The closest problems for which algorithms based on finding the maximum flow in the transport network are known are the problems of constructing schedules with program interrupts.

The main challenges of applying known algorithms of scheduling with interrupts, which are based on finding maximum flow in a network to build static-dynamic schedules are: the problem of the attaching partitions for processors and the problem of determining the correct set of windows.

II. Motivation

Building real-time computing systems with an IMA architecture containing a large number of subsystems and application programs is problematic without computer-aided design systems. The development of such systems requires the development of algorithms for solving problems that arise during the construction of the system. One of these tasks is to determine the minimum required amount of hardware resources to run programs in real time and build static-dynamic schedules for their execution. In some real-time operating systems [4], [5] based on ARINC-653 specification [6] for their correct and effective operation, it is necessary to solve the problem of scheduling processor time, taking into account the specifics of these mechanisms and the features of programs in

The work was carried out with the financial support of the RFBR № 19-07-00614.

the field of real-time systems. Also there need to be systems to prove the correctness and perform configuration of the system behaviour [7]–[9].

III. State of the Art / Related Work

In [10], ant algorithms were proposed for constructing static-dynamic schedules. These algorithms are only applicable for a variant of the problem where program interruption is not allowed. This means that the program can only run in one window of its section, which significantly limits the practical application of algorithms.

In [11], [12], algorithms based on the decomposition of the problem into two subproblems were considered: binding partitions to processors and building a set of windows for each processor. The disadvantage of these algorithms is that we can't change first step of algorithm, when we can use additional information from second step.

The closest problems for which algorithms based on finding the maximum flow in the transport network are known are the problems of building schedules with interruptions of work. The method of using algorithms for finding the maximum flow in the network to build schedules with interruptions was proposed in [13]. In [14], an algorithm for building schedules for a heterogeneous multiprocessor system (processors have different performance) was considered. Interrupts and switching operations do not require time, but in our system we need time to switch from program of one partition to another partition. In [15], [16], an algorithm was described for the problem, which took into account the limited amount of processor memory.

The main problems that prevent the direct application of the known algorithms based on finding the maximum flow in the transport network for building static-dynamic schedules are: accounting for work belonging to partitions and constructing the set of windows that need to consider switching time.

In [17] a flow algorithm was proposed for constructing static-dynamic schedules for single processor system and in [18] for multiprocessor systems that consider switching time between partitions, but those works only with homogeneous systems.

IV. Problem Statement

We assume that the following data are known:

- n is the number of programs.
- m is the number of processors.
- q is the number of partitions.
- $M = \left\{p_j = \langle s_j, R_j\rangle \mid j = \overline{1,m}\right\}$ is the set of processors, where s_j is the performance of the processor j which is measured in units of work per unit of time, R_j is the set of functional capabilities of the processor j.
- $A = \left\{a_k = \left\langle b_k^A, f_k^A, c_k^A, d_k^A, r_k^A\right\rangle \mid k = \overline{1,n}\right\}$ is the set of programs, where b_k^A is the beginning of the directive interval, f_k^A is the end of the directive interval, c_k^A is the complexity of the program execution which is measured in units of work, d_k^A is the partition number to which the program is attached, r_k^a is the set of processor's functionality capabilities requirements for program execution.
- w is the complexity of the switching window, which is measured in units of work, we assume that this parameter is common for all processor.

A static-dynamic schedule is constructed if the set of windows (time intervals) is defined for each processor $l = \overline{1,m}$:

- m_l is the number of windows on the processor;
- $W_l = \left\{w_i^l = \left\langle b_i^{W_l}, f_i^{W_l}, d_i^{W_l}, A_i^{W_l}\right\rangle \mid i = \overline{1,m_l}\right\}$ is the set of windows where $b_i^{W_l}$ is the time of window opening, $f_i^{W_l}$ is the time of window closing . $d_i^{W_l}$ is the partition number to which the window is attached.
- $A_i^{W_l} = \{(a_j, c_j) \mid a_j \in A, j \subset \overline{1,n}, c_j \leq c_j^A\}$ is the set of programs that are executed in the window (c_j indicates the time given for executing of the program j in the window).

Conditions for the static-dynamic schedule's correctness for each window sets W_l:

1) Windows do not overlap one another and the switch time is considered:

$$b_{i+1}^{W_l} - f_i^{W_l} \geq w \ \forall i \in \overline{1,m-1}. \tag{1}$$

2) The sum of the durations of the parts of the programs in the same window does not exceed the duration of the window:

$$\sum_{a_j \in A_i^{W_l}} \frac{c_{ji}}{p_l} \leq f_i^{W_l} - b_i^{W_l} \ \forall i \in \overline{1,m}. \tag{2}$$

3) In a window, only programs or their parts that have the same partition number with the window can be executed:

$$\forall a_j, a_k \in A_i^{W_l} \rightarrow d_j^{W_l} = d_k^{W_l} \ \forall i \in \overline{1,m}. \tag{3}$$

4) A program can only be executed on a processor if the processor meets program's functional capabilities requirements:

$$\forall a_k \in A_i^{W_l} \rightarrow r_k^A \subseteq R_l \tag{4}$$

5) The program is placed if it is fully executed entirely on a single processor:

$$\forall a_k \in A_i^{W_l} \sum_{A_j^{W_l} | a_k \in A_j^{W_l}} c_{kj}^l = c_k^A. \tag{5}$$

The c_{ij}^l indicates the number of units of work performed on the processor l for the program i in the window j.

Our goal is to construct correct static-dynamic schedule, that places as many programs as it's possible.

A. Computing system

.

There are several types of operating systems running on the system. Partition operating systems and the main (modular) operating system [4]. The modular operating system is responsible for scheduling time windows for each operating system

in the partition. It also passes information to the operating system of the partition about the quantum of time that should be allocated for the execution of each program inside this window. And the operating system scheduler of the partition inside the window puts programs on execution in accordance with the priority for the nearest completion time and removes programs from execution after the expiration of the quantum of time given to this program for execution in this window. We assume that if programs meets their requirements then all messages that should be performed in the system meets their requirements too. For messages are usually used special networks [19].

B. Periodic programs

.

Also, instead of the set of programs, a set of periodic programs can be known, which can be reduced to a set of programs with individual directive intervals:

$AP = \left\{ a_k^P = \left\langle T_k^{AP}, b_k^{AP}, f_k^{AP} c_k^{AP}, d_k^{AP}, r_k^{AP} \right\rangle \mid k = \overline{1, n^P} \right\}$

is the set of periodic tasks, where T_k^{AP} is the period of task execution, b_k^{AP} is the jitter to start executing the program within the period, f_k^{AP} is the jitter for the end of program execution within the period, for these parameters, the following relations must be met: $0 \leq b_k^{AP} \leq f_k^{AP} {}_k^{AP}$ c_k^{AP} is the complexity of executing the program instance. d_k^{AP} is the partition number partition number to which the periodic program is attached. r_k^{Ap} is the set of processor's functionality capabilities requirements for program execution.

In this case, to reduce this task to set of programs with individual directive intervals, we should compute the duration of planning cycle as least common multiple of all periodic programs periods. Then we create program instances for each periodic program with individual directive intervals, which are calculated as follows: the periodic task is divided into N programs, where N is the duration of planning cycle divided by the periodic program period. Program instances are numbered starting from 0. And for each program instance with the number $i = 0, ..., N-1$ the following directive interval $(T_k^{AP} \cdot i + b_k^{AP},\ T_k^{AP} \cdot i + f_k^{AP}]$ is assigned. The program instance partition is the periodic program partition. The complexity of the program instance is the complexity of periodic program.

V. Algorithm for constructing a static-dynamic schedule based on the algorithm for finding the maximum flow in the network

The algorithm consists of three main stages:

1) Constructing a transport network (oriented graph).
2) Finding the flow in the transport network.
3) Restoring the schedule from the received flow.

A. Construction of a transport network

It's similar to transport network from [18], but now we need to take into account that our system is heterogeneous and the duration of execution of the same program will vary depending on the processor.

In accordance with the programs directive intervals (start time and end time of the directive interval), a set of disjoint time intervals is constructed in time order $I = I_i = \tau_{i-1}, \tau_i$, where $\tau_0 \leq \cdots \leq \tau_s$ is all different values of b_k^A and f_k^a. The program k can only be executed in intervals that intersect with its directive interval: $I_i \in (b_k^A, f_k^A]$.

The network consists of a bipartite graph with two additional vertices: source and drain. The first part of the graph consists of vertices that corresponds to a program (vertices-programs) and the second part consists of vertices corresponding to a pair of interval and processor (vertices-intervals-processors).

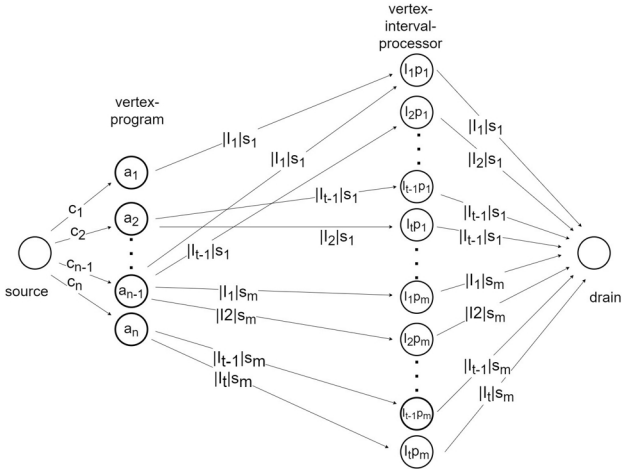

Fig. 1. Transport network

The source vertex is connected to the program vertices by edges of capacity equal to the complexity of executing this program c_k^A. Vertex programs are joined with the vertex-intervals-processors, on which the corresponding program is available for execution, i.e., $I_i \in (b_k^A, f_k^A]$ and the functional capabilities requirements are met $r_k^A \in R_j$, edges capacity is equal to complexity handled by the processor for the duration of the given interval $s_j|I_i|$. Finally, all vertices-intervals-processors are connected to the drain by edges with capacity equal to $s_j|I_i|$. The result network is on figure 1.

For network vertices, the following symbols will be used: u, v. The parameter $f(u, v)$ and $c(u, v)$ denotes the flow and capacity from the vertex u to the vertex v.

Also, for the convenience of further discussion, we will assume that each vertex has the following characteristics:

- e_u is the excess vertex flow,
- h_u is the height of the vertex.

that each vertex-program has the following characteristics:

- d_u is the partition number, that program attaches to,
- c_u is the complexity of the program.

and each vertex-interval-processor:

- ni_v is the next vertex-interval-processor,
- pi_v is the previous vertex-interval-processor,
- cw_v is the number of window switches inside the vertex,

- ir_v is the indicator for the switching on the right border of the interval,
- il_v is the indicator for the switching on the left border of the interval,
- PT_v^i is the structure for storing total flows from all partitions $i = \overline{1,q}$,
- PS_v is the set of partitions with a non-zero incoming flow,
- fp_v is the first section in the interval,
- lp_v is the last section in the interval,
- $|I_v|$ is the duration of the interval,
- $proc_v$ is processor number.

All vertices are grouped by the partition they belong to, and all vertices-intervals-processors by processors they belong to. The K parameter specifies the maximum number of processor reallocation attempts. The PR_d structure specifies the processor to execute for all partitions $d = \overline{1,q}$. There is a list of all excess vertices.

B. Find the flow in the transport network

The search for a flow is similar to algorithm from [18] and use it based operations with some modifications. All modifications are described here. Algorithm defines the main operation - vertex discharge, based on two operations: lifting the vertex, pushing from vertex to vertex that check for correctness of the schedule, which are clearly described in [18]. The main idea of pushing with checking the correctness is consist of reserving flow for switch window operation when one vertex-interval-processor receive flow from vertex-programs that are owned by different partition. This reserving consist of reducing capacity of the edge from that vertex-interval-processor to the drain.

Since the transport network may change the capacity of edges due to the need to take into account the switching time between windows the result of the algorithm may be non-optimal.

The algorithm has been modified to work with heterogeneous systems, the main additional operations to maintain the correctness of the schedule under construction are detach partition from processor and decide partition attachment that are applying during the algorithm work and consider current situation in the network.

Modified operations:

Decide partition attachment

For finding partition attachment we will use greed criterion that is based on maximum of free space after partition attachment to the processor. Each partition i has it's complexity $C_i = \sum\limits_{a_k in A | d_k^A = i} c_k^A$. And each processor j has it's free space, that as the beginning of the algorithm is equal to $main_loop * s_j$. After attaching partition i to processor j free space on processor j is reduced by C_i. Also all edges, that lead to another vertices-processors-intervals that are owned by other processors not equal j from vertices-programs that owned by i partition should reduce their capacity to 0.

Detach partition from processor

When partition detached from processor we need to return all flows, that go from program vertices that owned by i partition to processors intervals vertices that are owned by processor j. It can be done with push operations and window corrects operation that are described in [18] by applying it in back order. By that way the correctness of schedule will remain.

Network Initialization.

1) Natively, the preflow is equal to the capacity for all edges coming out of the source (this is a flow only to the u program vertices), and the opposite for reverse vertex pairs:
   $f(source, u) = c(source, u)$,
   $f(u, s) = -c(source, u)$,
   $e_u = c(source, u)$.
2) For the other pairs of vertices, the preflow is zero.
3) The initial height is equal to H (algorithm parameter) in the network for the source, 1 – for program vertices (since you will have to immediately push the flow to the processor-interval vertices), 0-for processor-interval vertices and the drain.
4) For each partition in order of ascending complexity apply decide attachment operation.
5) All program vertices are added to the list of excess vertices.

Pushing from u to v.

1) Flow $f(u, v)$ increases by the value $deltaf(u, v) = \min(e_u, c(u, v) - f(u, v))$.
2) $deltaf(u, v)$ increases the excess flow ofe_v.
3) the Reverse flow $f(v, u)$ and the excess flow e_u are reduced by $deltaf(u, v)$.
4) If u is a program vertex and v is a vertex-interval-processor, perform the partition accounting operation (check how many different partitions flows to v, see [18]) when adding a flow of the size $\delta f(u, v)$ of the d_u section to the vertex-interval-processor v and perform the window correction operation in v (count number of switching in the vertex, see [18]).
5) If u is a vertex-interval-processor, v is a vertex-program, perform the partition accounting operation when deleting a flow of the size $\delta f(u, v)$ of the d_v section to the vertex-interval-processor and perform the window correction operation in u.
6) If $e_u = 0$, the vertex u is removed from the list of excess vertices in the corresponding section.
7) If $e_v > 0$, the vertex is added to the list of excess vertices.

Discharge of vertex u

While $e_u 0$: in order, we consider all available vertices for pushing $v : f(u, v) < c(u, v)$ and $h_u = h_v + 1$.

1) If v is encountered for the first time and it is a vertex-interval-processor, then we perform the push from u to v operation to the vertex-interval-processor v, taking into

account the excess flow in drain (it is like bipush to the drain, see [18]).
2) If v occurs repeatedly or is not a vertex-interval-processor, perform the push operation from u to v.
3) If there are no v vertices left, perform the operation lifting u vertex.

General scheme of the algorithm.

1) Perform initialization operation.
2) While there are excess vertices:
   a) Selects a partition which list of overloaded vertices is not empty (if there is no such section, go to step 3), then the following operations are performed for these vertices u: discharge of the vertex u.
   b) As long as the lists of overflow partition vertices and processor interval vertices are not empty, the ordered discharge of these vertices is performed: if there is an excess vertex from the number of processor interval vertices, it is discharged first.
   c) If discharging leads to flow into source that means that the partition can be executed on that processor and we should detach it from current processor and decide new attachment for this as it was described in two operation upper. Increase K by one.
   d) Go to step 2.
3) If there are programs that are not fully placed, the first such program is deleted from the network (remove all flow from this vertex-program, see [18]) and go to step 1.

C. Restore schedule by flow

The following operations are performed sequentially (in the order of time intervals) for all processor vertex-interval-processor that belong to the same processor for each processors.

1) If there is a window switch on the left, the current window is closed, the switching time is taken into account, the window of the first section opens in this interval, and one is subtracted from the number of window switches in this interval.
2) If there is a window switch in the middle ,then (as long as there are switches in the middle) the window closes after a time equal to the flow of this section to this vertex, divided by processor performance. Sections are sorted, starting from the first and ending with the last (no matter what order is in the center), switching is taken into account, and the next section window opens.
3) If there is a window switching on the right, the current window closes, the switching time is taken into account, and the window of the first section of the next interval opens.
4) to specify the programs that are running in the window, as well as their execution time allocated in each window, it is sufficient for each vertex-program to take the flow in the corresponding window interval and divide it by processor performance. If there is no flow, its value is zero and the program is not running in this window.

As a result, a set of windows is formed for each processor with an indication of how much processor time each program needs to allocate within the window.

D. Correctness of the constructed schedule

The result of the algorithm meets the conditions for correct scheduling. In the algorithm, the main operation is the vertex discharge operation, which consists of operations of pushing from vertex to vertex and operations of lifting the vertex. The lift operation only affects the order of pushes and does not affect the correctness of the schedule. The push operation is designed in such a way that if there is no excess flow in the network, all the conditions for the correct schedule will be met. Design of the network not allowed flow, that exceed the ability of processors to perform tasks and the push operation associated with the interval vertex performs window adjustment operations that add window switching between partitions to the corresponding vertex-reserving time for switching between partitions so that other operations cannot use it. Windows are built based on the duration of one section of work that follows in a row. In duration of algorithm all not fully placed programs will be removed from schedule.

Since placement occurs immediately after the attaching of the partition to the one of the processors, further placement of programs of this partition is possible only on this processor (since the capacity to the other processors is equal to zero). Until the programs wants to make a push to the source, which is equivalent to not being able to place the entire partition on this processor. Then the detached operation is performed. The flow is removed from all vertices from all programs vertices of this partition (the schedule remains correct, since all window switches are taken into account). And capacity to the rest of the processors is restored. Therefore, the condition cannot be violated: the programs of a same partition is performed on a same processor.

E. Computational complexity and completeness

The network is built for $O(mn^2)$ as there are only $2n$ possible intervals and if we assume that we have fully connected bipartite graph it means that we will have $2n+n$ vertices and $2n^2$ edges. And for every edge we need to perform a check and assign capacity. The schedule is restored for $O(mn)$ as it is linear operation to check all vertecies-interval-processor in time order.

As algorithm was based on preflow algorithm [20] it has the same complexity $O(mn^3)$. As we have checks about pushes than the height of the source is bound number of attemption to reallocate the flow from one partition layer to processor layer. And there are only K attempts to reallocate partition attachment. That leads that in finite number of steps and the algorithm will end it work.

Therefore the total complexity of algorithm is $O(mn^3)$

VI. Implementation

The program for construction a static-dynamic schedule is implemented in C++. It has a modular structure based on the Web class that aggregate Vertex class, that has all required fields for implementing algorithm. Network architecture and operations of the proposed algorithm. The transport network consists of layers that consist of vertices. Each layer have type: partition layer or processor layer. All vertices of partition layers associated with vertex-programs and all vertices of processor layers associated with vertex-intervals-processors. To link vertex-programs with source there is a special field in Vertex class for partition layers and to link vertex-intervals-processors with drain there is a special field in Vertex class for processor layers. All required operations implement as methods of Web class.

To construct Web class there is a special class, that can read an a csv file with system configuraton and programs to create a instance of Web class. The format of system file is as in table I:

TABLE I
FORMAT OF SYSTEM FILE

Number	Performance	Capabilities
0	4	f1
1	1	f1;f2
2	5	f2

The format of programs csv file is as in table II:

TABLE II
FORMAT OF PROGRAMS FILE

Num.	Part.	Complexity	Period	Left	Right	Req.
0	1	20	100	10	50	f1
1	2	40	50	0	50	f1;f2
2	3	10	200	20	200	f1

TABLE III
FORMAT OF STATIC-DYNAMIC SCHEDULE FILE

Processor	Partition	Window	Programs
0	1	0-10	1-1.5
0	2	11-20	2-2.5
1	3	0-20	3 - 5

There is also a separate module for reading data and building a network based on it, as well as a module for building a final schedule from the final flow after the algorithm is running. The result of the program is an csv file that contain static-dynamic schedule in next format as in table III.

For testing, a synthetic data generator was written in python, since it is difficult to find open data on this topic. The language was chosen based on the convenience of working with data. Such libraries in python are pandas and numpy allow to create source data with different distribution of parameters.

VII. Evaluation

The experimental study was conducted in a system with Windows 10 64bit, Intel Core i3-2370M 2.39 GHz processor, 8GB RAM.

The purpose of the experimental study is to determine the sets the known data on which the algorithm works effectively.

To do this, we consider three type of heterogeneous systems with two, four and eight processors.

After this for those three systems by generator we create a lot of set of periodic programs with different parameters: number of partitions, total load of the system. The relationship between duration of the programs was taken random. The periodic programs were chosen as more compatible for real-time onboard systems.

We consider programs with completion period is selected randomly from the range 100, 500, 1000, 1500 and 3000. Number of tasks in each partition was picked up between 2 and 5. All programs have different percentage of available computing system load (that is, the performance of only those processors on which this task can be placed is considered) that picked up randomly. For each set of parameters we take more then 100 experiments. Than we compute average time and average percentage of placed programs.

We consider areas of source data for loading the computer system these are low-load systems (up to 60%), medium-loaded systems (up to 70%) and high-load systems(up to 80%).

The summary results of the experimental study are shown in the table IV:

TABLE IV
EXPERIMENTAL STUDY

Procs	load	Parts	Time(ms)	Placed(%)
2	0.6	3	5.6	100
2	0.7	5	6.3	100
2	0.8	7	9.7	100
4	0.6	5	8.4	100
4	0.7	7	12.7	100
4	0.8	9	16.8	100
8	0.6	9	20.9	100
8	0.7	11	27.2	100
8	0.8	13	36.4	100

Based on the average values of these two parameters, it is concluded that the algorithm is effective in this area of source data when the load of the system is under 80% and allow feasible schedule.

VIII. Conclusion

The problem of constructing static-dynamic schedules arises when designing real-time information and control systems with integrated modular avionics architecture. Algorithms based on finding the maximum flow in the transport network have shown high efficiency in terms of accuracy and computational complexity at constructing schedules with interruptions. The main problems that prevent the use of known algorithm based on finding the maximum flow in the transport network for building static-dynamic schedules are: the problem of

uniform processor. So that algorithm was modified to work with heterogeneous systems. Experimental research of the algorithm properties has shown its high efficiency in terms of accuracy and computational complexity on wide class of source data.

REFERENCES

[1] Fedosov E., Kos'janchuk V., Sel'vesjuk N. Integrirovannaja modul'naja avionika. Radiojelektronnye tehnologii, 2015, vol. 1, pp. 66-71 (in Russian).

[2] Kostenko V.A. Arhitektura programmno-apparatnyh kompleksov bortovogo oborudovanija. Izv. vuzov. Priborostroenie, 2017, vol. 60, no. 3, pp. 229—233 (in Russian).

[3] Smeljanskij R.L., Kostenko V.A. Problemy postroenija bortovyh kompleksov s arhitekturoj integrirovannoj modul'noj avioniki. Radiopromyshlennost', 2016, no. 3, pp. 63–70 (in Russian).

[4] VxWorks 653 multi-core edition [PDF] (https://www.windriver.com/-products/product-overviews/vxworks-653-product-overview-multi-core/vxworks-653-product-overview-multi-core.pdf)

[5] Solodelov Ju. A., Gorelic N.K. Sertificiruemaja bortovaja operacionnaja sistema real'nogo vremeni JetOS dlja rossijskih proektov vozdushnyh sudov, Trudy ISP RAN, 2017, vol. 29, no. 3, pp. 171–178 (in Russian).

[6] Arinc Specification 653. Airlines Electronic Engineering Committee. (http://www.arinc.com).

[7] Balashov V.V. Semejstvo sistem avtomatizacii proektirovanija bortovyh vychislitel'nyh sistem real'nogo vremeni. Programmnye produkty, sistemy i algoritmy, 2017, no. 4, pp. 1–19 (in Russian).

[8] Balashov V.V., Kostenko V.A. Sredstva konfigurirovanija bortovyh sistem s arhitekturoj integrirovannoj modul'noj avioniki. Trudy FGUP NPCAP. Sistemy i pribory upravlenija, 2018, vol. 3, no. 45, pp. 56–58 (in Russian).

[9] Balashov V., Balakhanov V., Kostenko V., and Tutelian S. Tool system and algorithms for scheduling of computations in integrated modular onboard embedded systems. IFAC Proceedings Volumes (IFAC-PapersOnline), 2016, vol. 49, no. 25, pp. 505–510.

[10] Balahanov V.A., Kostenko V.A. Sposoby svedenija zadachi postroenija statiko-dinamicheskogo odnoprocessornogo raspisanija dlja sistem real'nogo vremeni k zadache nahozhdenija na grafe marshruta. Programmnye sistemy i instrumenty, 2007, no. 8, pp. 148-156 (in Russian).

[11] Balashov V. V., Balakhanov V. A., Kostenko V. A. Scheduling of Computational Tasks in Switched Network-based IMA Systems. Proc. Intern. Conf. on Engineering and Applied Sciences Optimization. National Technical University of Athens (NTUA), Athens, Greece, 2014, pp. 1001–1014.

[12] Gonzales T., Sanhi S. Preemptive Scheduling of Uniform Processor Systems. J. Association for Computing Machinery, 1978, vol. 25, no. 1.

[13] Federgruen A., Groenevelt H. Preemptive Scheduling of Uniform Machines by Ordinar Network Flow Technique. Management Science, 1986, vol. 32, no. 3.

[14] Furugjan M.G. Planirovanie vychislenij v mnogoprocessornyh ASU real'nogo vremeni s dopolnitel'nym resursom. AiT, 2015, no. 3, pp. 144–150 (in Russian).

[15] Furugjan M.G. Planirovanie vychislenij v mnogoprocessornyh sistemah neskol'kimi tipami dopolnitel'nyh resursov i proizvol'nymi processorami. Vestn. MGU. Ser. 15. Vychislitel'naja matematika i kibernetika, 2017, no. 3, pp. 38–45 (in Russian).

[16] Furugjan M.G. Sostavlenie raspisanij v mnogoprocessornyh sistemah s dopolnitel'nymi ogranichenijami. Izv. RAN. TiSU, 2018, no. 2, pp. 52–59 (in Russian).

[17] Kostenko V.A., Smirnov A.S. An Algorithm for Constructing Single Processor Static–Dynamic Schedules. Moscow University Computational Mathematics and Cybernetics, 2018, vol. 42, no. 1, pp. 44–50.

[18] Kostenko V.A., Smirnov A.S. Flow Algorithms for Scheduling Computations in Integrated Modular Avionics. J. Computer and Systems Sciences International, 2019, vol. 58, no. 3, pp. 404–414.

[19] Vdovin P. M., Kostenko V. A. Organizacija peredachi soobshhenij v setjah AFDX. Programmirovanie, 2017, no. 1, pp. 3–18 (in Russian).

[20] Ahuja R.K., Orlin J.B., Stein C., Tarjan R.E. Improved Algorithms for Bipartite Network Flow. SIAM J. Comput, 1994, vol. 23, no. 5, pp. 906-933.