
SYSTEMS ANALYSIS AND OPERATIONS RESEARCH

Flow Algorithms for Scheduling Computations in Integrated Modular Avionics

V. A. Kostenko^{a,*} and A. S. Smirnov^a

^a Moscow State University, Moscow, Russia

*e-mail: kost@cs.msu.su

Received December 26, 2018; revised January 25, 2019; accepted January 28, 2019

Abstract—Algorithms for scheduling tasks in real-time systems with integrated modular architecture that are based on finding the maximum flow in a transportation network are proposed. Results of the experimental evaluation of these algorithms for a single processor and multiprocessor version of the computation scheduling problem are discussed.

DOI: 10.1134/S1064230719030110

INTRODUCTION

To satisfy the requirements of isolation and real time operation of subsystems of real-time information and control systems (RICSs) with integrated modular architecture [1], the standard ARINC 653 [2] was developed. The most widely used approach to designing RICS with integrated modular architecture is known as integrated modular avionics (IMA). The Russian-made real-time operating system (RTOS) meeting the ARINC 653 standard is the RTOS Baget 3.0 [3, 4]. In Baget 3.0, ARINC 653 is used as the basic standard. All mandatory functions of ARINC 653 are implemented. The standard POSIX is used to the extent in which it does not contradict ARINC 653.

The isolation of programs of different subsystems is ensured by introducing partitions and windows. For programs of each subsystem, a partition and the set of time windows (nonoverlapping time intervals) are allocated. Programs assigned to a partition may be executed only within the allocated time windows; each partition is assigned a chunk of memory that cannot be accessed by programs from other partitions. Programs assigned to a partition are started within time windows dynamically, e.g., when the data is ready and according to priority. A program may be interrupted and then resumed in the same window or in a later window of the same partition. Programs assigned to different partitions can interact only by passing messages; i.e., application programs are executed according to a static–dynamic schedule. The static–dynamic schedule is ready if partitions are assigned to processors of the computer system, a set of windows for each partition is defined, and the beginning and end of each time window is determined.

Various versions of the problem of constructing static–dynamic schedules are defined by dynamic task schedulers in partitions, by the way of specifying input data, and by a set of technological constraints. Examples of technological constraints are the maximally acceptable window size and the minimally acceptable gap between windows in different partitions for context switching. The concepts of partition and window are common for all operating systems meeting the standard ARINC 653.

In this paper, we propose to construct static–dynamic schedules using algorithms based on finding the maximum flow in a transportation network.

1. SIMILAR PROBLEMS AND KNOWN ALGORITHMS

In [5, 6] ant colony algorithms for constructing static–dynamic schedules were proposed. These algorithms are applicable when programs cannot be interrupted. This implies that a program can run only in one window of its partition, which significantly narrows the scope of the practical application of these algorithms.

In [7, 8], algorithms based on the decomposition of the problem into two subproblems—assigning partitions to processors and constructing the set of windows for each processor were considered. A drawback of these algorithms is that they should take into account the validity constraints of static–dynamic sched-

ules that are specific for a specific RICS and that their accuracy depends on the class of the input data (more precisely, on the time complexity of the tasks to be executed).

The closest problems to the ones studied in the current paper for which algorithms based on finding the maximum flow in a network are known are the problems constructing preemptive schedules. A procedure for using algorithms for finding the maximum flow in a network for constructing preemptive schedules was proposed in [9, 10].

In [11], an algorithm for constructing schedules for a heterogeneous multiprocessor system (in which the performance of processors is different) was considered. It was assumed that preemption and switching between tasks do not take additional time. In [12], an algorithm in which the limited cash memory of processors is taken into account was described. An algorithm for the case when the task execution times linearly depend on the amount of the allocated resource was proposed in [13].

The main difficulties preventing the direct use of the known algorithms based on finding the maximum flow in a transportation network for designing static–dynamic schedules are the fact that the membership of tasks in partitions must be taken into account and the set of windows must be constructed.

In [14], an algorithm for the single processor version of designing a static–dynamic schedule based on finding the maximum flow in a transportation network was described. However, this algorithm can be used for the multiprocessor version of the problem only in combination with an algorithm that assigns partitions to processors. To achieve a level of high accuracy, such a combination of algorithms requires significant modification of the criteria for assigning a partition to a processor used in the algorithm for assigning partitions to processors in such a way that each processor is assigned a set of partitions (the class of input data) for which the flow algorithm gives highly accurate results.

2. THE PROBLEM OF CONSTRUCTING A STATIC–DYNAMIC SCHEDULE

For constructing static–dynamic schedules for real time systems with the integrated modular avionics architecture, the following input data are specified: n is the number of the task, p is the number of processors, q is the number of partitions, c is the window switching time, and $A = \{a_k \mid k = \overline{1, n}\}$ is the set of tasks.

The requirements for the execution of tasks can be specified in two ways.

Method 1: $a_k = \langle s_k^A, f_k^A, t_k^A, d_k^A \rangle$, where s_k^A is the due time at which task k can start executing (the task execution cannot begin earlier than at the due time), f_k^A is the release time of task k (the task execution must be completed before this time), t_k^A is the time needed to execute the task, and d_k^A is the partition to which the task is assigned.

Method 2: $a_k = \langle F_k^A, t_k^A, d_k^A \rangle$, where F_k^A is the frequency ($1/F_k^A$ is the period) of the task execution, t_k^A is the time needed to execute the task, and d_k^A is the partition to which the task is assigned.

The second method of specifying the requirements for executing tasks in real time can be reduced to the first method. The large cycle is computed as the least common multiple of the task execution periods. The number of running the task in the large cycle is the number of its periods in the large cycle. The due times of each task execution are determined by the beginning and end of the corresponding period.

It is required to construct a static–dynamic schedule containing the maximum number of tasks from the given set of tasks. The schedule is ready if the set of windows for each processor l is defined: $W_l = \{w_i = \langle s_i^{W_l}, f_i^{W_l}, d_i^{W_l}, A_i^{W_l} \rangle \mid i = \overline{1, m_l}\}$, where m_l is the number of windows, $s_i^{W_l}$ is the window's opening time, $f_i^{W_l}$ is the window's closing time, $d_i^{W_l}$ is the index of partition that includes this window, and $A_i^{W_l} = \{(a_j, t_j) \mid a_j \in A, j \in \overline{1, n}, t_j \leq t_j^A\}$ is the set of tasks executed within the window (with the indication of the time allocated for the task execution in this window).

The schedule must satisfy the following correctness conditions for each set of windows W_l .

1. The windows do not overlap, and the minimum switching time is taken into account:

$$c \leq s_{i+1}^{W_l} - f_i^{W_l} \quad \forall i \in \overline{1, m_l - 1}. \quad (2.1)$$

2. The sum of the durations of the tasks executed within each window does not exceed the window's duration:

$$\sum_{a_j \in A_i^{W_l}} t_{ji} \leq f_i^{W_l} - s_i^{W_l} \quad \forall i \in \overline{1, m_l}. \quad (2.2)$$

3. Only parts of the tasks or entire tasks that belong to the partition to which the window is assigned can be executed within this window:

$$\forall a_j, a_k \in A_i^{W_l} \rightarrow d_j^{W_l} = d_k^{W_l} \quad \forall i \in \overline{1, m_l}. \quad (2.3)$$

4. The task is allocated if it is completely executed:

$$\sum_{a_k \in A_j^{W_l}} t_{kj} = t_k^A \quad \forall a_k \in \bigcup_{i=1}^{m_l} A_i^{W_l}. \quad (2.4)$$

5. All tasks of each partition are executed on the same processor.

3. THE ALGORITHM FOR CONSTRUCTING A STATIC–DYNAMIC SCHEDULE BASED ON FINDING THE MAXIMUM FLOW IN A NETWORK

The algorithm consists of three main stages:

- (1) Constructing a transportation network (directed graph);
- (2) Finding the maximum flow in this network;
- (3) Recovering the schedule from the flow.

According to the due and release times (execution intervals) of the tasks, an ordered set of nonoverlapping time intervals is constructed. Every task may be executed only in the intervals intersecting with its due interval. The transportation network is constructed by defining the task vertices, the vertices corresponding to processor intervals related to the processor index (they are called interval vertices), and adding a source and sink vertices. The source vertex is connected to the task vertices by edges with the capacities equal to the task execution duration. The task vertices are connected to the vertices corresponding to processor intervals within which the task may be executed; the capacities of these edges are equal to the duration of this interval. Each interval vertex is connected to the sink vertex by an edge with the capacity equal to the interval duration.

The maximum flow is sought using the lift-to-front algorithm because it has the lowest complexity and is simple from the viewpoint of verifying the schedule correctness. This algorithm involves three basic operations:

- (1) Lifting a vertex;
- (2) Pushing the flow from vertex-to-vertex;
- (3) Discharging the vertex.

The lift-to-front algorithm is significantly modified since the capacities of edges in the network can change because we must take into account the time of switching between windows. The push operation takes into account the partition from which the flow was directed to the interval vertex or was removed from it and the flow value. Based on these data, it is determined whether the time for switching between windows in this vertex must be allocated; if this is the case, then the capacity of the edge connecting the interval vertex with the sink is modified. The lift operation finds the vertex of the minimum height such that there exists a preflow into this vertex with a value lower than the capacity of this edge, and it increases the height of the initial vertex by one. The vertex height determines the order of applying the push operation. To obtain a correct static–dynamic schedule, the vertex discharge operations are ordered; they have two modes of operation. The first mode can check if the resulting flow can be pushed, and the second mode cannot perform such a check. The first push from the task vertex to the interval vertex corresponds to assigning a partition to this processor, and the capacities of all edges leading to other processors become equal to zero. If a task of the partition cannot be allocated, then the flow from all tasks of this partition must be reversed, and the capacities of all edges leading from these task vertices to the interval vertices of this processor must be made equal to zero.

All vertices are divided into groups—task vertices of each partition and interval vertices—and there are $q + 1$ groups in total. While there are overwhelmed vertices, the algorithm performs the vertex discharge operation. The order of discharging is as follows: the first partition is selected, and all its vertices are discharged while checking the possibility of pushing the flow into the sink; next the interval vertices are dis-

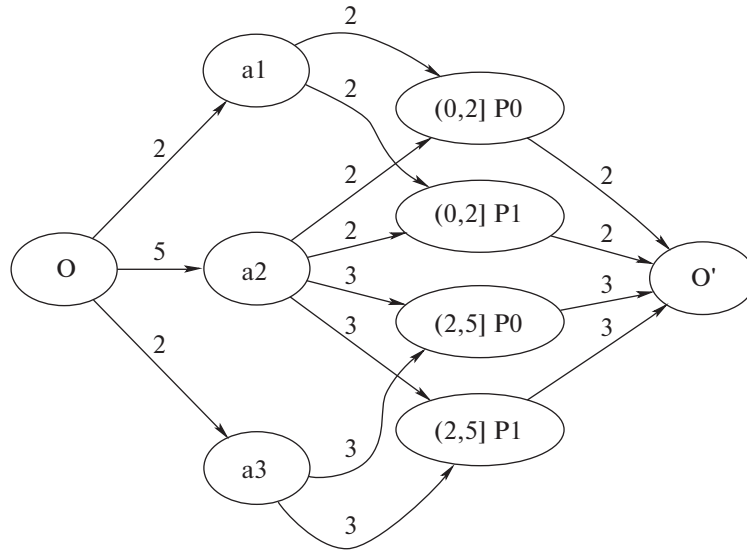


Fig. 1. Transportation network.

charged. The discharge operation continues until there are no overwhelmed vertices of the group of this partition and overwhelmed interval vertices. This operation is performed as follows: a task vertex is discharged; if overwhelmed interval vertices emerged as a result of this operation, then they are discharged, and then task vertices of the partition are discharged again.

The schedule recovery is to consider the interval vertices in their natural time order and to analyze the incoming flows from tasks of different partitions and the number of window switchings in the interval vertices.

Let us consider each stage of the algorithm in more detail.

3.1. Construction of the Transportation Network

Let $\tau_0 < \dots < \tau_s$ be all different values s_k^A and f_k^A ($k = 1, \dots, n$) and $I_i = (\tau_{i-1}; \tau_i]$, $i = \overline{1, s}$. Task k may be executed at the time t_0 if $s_k^A < t_0 < f_k^A$.

The network is a bipartite graph with two additional vertices—the source O and the sink O' (see Fig. 1). The left part of this graph consists of vertices that are in one-to-one correspondence with the tasks (task vertices). The right part of this graph consists of vertices corresponding to the interval-processor pairs (interval vertices) (a one-to-one correspondence). The source is connected with all task vertices and the capacity of the edge leading to the k th task vertex is t_k^A ($k = \overline{1, n}$). Each task vertex is connected with all interval vertices on which this task may be executed. The capacities of these edges are t_k^A . All interval vertices are connected with the sink O' by the edges with the capacities equal to the interval width.

3.2. Finding the Maximum Flow in the Network

The following data are required to construct a static–dynamic schedule: $dist$ is the sink vertex, s is the source vertex, $c(u, v)$ are the capacities of the edges, and $f(u, v)$ is the preflow function for network edges at a step of the algorithm. Each vertex has integer parameters: excess flow e_u , height h_u , and the list of task vertices arranged by partitions; the task vertices also have the partition index pt_u to which the task is assigned. The interval vertices know the preceding and succeeding intervals ni_u and pi_u ; they also have the integer parameter cw_u indicating the number of switchings within the interval, and the parameters ir_u and il_u show if there is a window switching at the right and left endpoints of the interval, respectively. The interval vertices include a structure for storing the incoming flows arranged by partitions PT_u^i , where $i = \overline{1, q}$ (i.e., we are able to find out how much time is allocated to this partition in the interval), the set of parti-

tions PS_u the flows from which enter this vertex, the first fp_u and the last lp_u partitions associated with the window, and the interval duration dur_u . All interval vertices v are also assigned the index of the processor $proc_v$ associated with this interval and the maximum number of attempts to redistribute the processors K . The assignment of each partition to a processor is denoted by PR_{part} .

Consider the algorithms designed for the execution of the basic operations involved in finding the maximum flow in the network.

Network Initialization

1. Initially, the preflow equals the capacity of all edges outgoing the source, and it is opposite for the reversed pairs of vertices:

$$\begin{aligned} f(s, u) &= c(s, u), \\ f(u, s) &= -c(s, u), \\ e_u &= f(s, u). \end{aligned}$$

2. For the other pairs of vertices, the preflow is zero.

3. The initial height of the source is 10, the initial height of the task vertices is 1, and the height of the interval vertices and sink is 0.

Lifting Vertex (u)

From the set of accessible vertices u (the vertices such that there exists an edge (u, v) with $f(u, v) < c(u, v)$), vertex v (where v is distinct from $dist$) with the minimal height is selected, and the height of u is set to $h_v + 1$.

Adding a Window Switching to (v)

In this operation, v is an interval vertex.

1. The capacity $c(v, dist)$ is decreased by c .
2. The number of window switchings cw_v is increased by 1.
3. If $f(v, dist) > c(v, dist)$, then the excess flow is modified.
4. The overwhelmed flow changes by the formula $e_v = e_v + f(v, dist) - c(v, dist)$ and the flow $f(v, dist)$ becomes equal to $c(v, dist)$.
5. If $e_v > 0$, then vertex v is added to the list of overwhelmed vertices.

Removing a Window Switching from (v).

In this operation, v is an interval vertex.

1. The capacity $c(v, dist)$ is increased by c .
2. If $e_v \geq c$, then

$$\begin{aligned} f(v, dist) &= f(v, dist) + c, \\ f(dist, v) &= -f(v, dist), \\ e_v &= e_v - c, \\ e_{dist} &= e_{dist} + c. \end{aligned}$$

3. If $e_v = 0$, then remove the vertex from the list of overwhelmed vertices.
4. The number of window switchings cw_v is decreased by 1.

Taking into Account the Partition When Adding the Flow of Size Value of the Partition Part Emerging from the Interval Vertex v

1. The flow is added to the structure of incoming flows of the vertex:

$$PT_{part} = PT_{part} + value.$$

2. If the flow of this partition enters the interval vertex for the first time, then it is added to the set PS_v .
 - 2.1. If this is the first partition, then set $fp_v = part$ and $lp_v = part$.
 - 2.2. If this is the second partition, then it is compared to the first partition of the next interval:
 - 2.2.1. If this is the same partition, then set $lp_v = part$;
 - 2.2.2. If this is a different partition, then set $fp_v = part$.
 - 2.3. If this is the third or greater partition, then it is compared to the first partition of the next interval:
 - 2.3.1. If this is the same partition, then set $lp_v = part$;
 - 2.3.2. Otherwise, it is compared to the first partition of the next interval and if they are equal, then set $fp_v = part$.

Taking into Account the Partition When Removing the Flow of Size Value of the Partition Part from the Interval Vertex v

1. The flow is removed from the structure of incoming flows of the vertex:

$$PT_{part} = PT_{part} - value.$$

2. If the flow of this partition is zero (or, equivalently, if this partition is no longer in this interval), then it is removed from the set PS_v .
3. If there are no more partitions, then set $fp_v = 0$ and $lp_v = 0$.
4. If only one partition pt remains, then set $fp_v = pt$ and $lp_v = pt$.
5. If more than one partition remains, then the following holds:
 - 5.1. If the removed partition coincides with the last partition in the interval (pt_{nf} is the partition different from the first partition in the interval), then set $lp_v = pt_{nf}$;
 - 5.2. If the removed partition coincides with the first partition in the interval (pt_{nl} is the partition different from the last partition in the interval), then $fp_v = pt_{nl}$.

Modifying Windows in the Interval Vertex (v)

1. Modify the internal switchings. The number of internal switchings $cw_{in} = cw_v - ir_v - il_v$ is calculated. The operations of switching removal in (in) or switching addition in (in) are performed the required number of times to make the number of switchings taken into account equal to cw_{in} .
2. Modify the right switching.
 - 2.1. If there is a nonempty interval vertex ni on the right, then the following holds:
 - 2.1.1. If $fp_{ni} \neq lp_v$ and there are no switchings on the right in v ($ir_v = false$) and there are no switchings on the left in ni ($ir_{ni} = false$), then the following holds:
 - 2.1.1.1. If $c(ni, dest) - f(ni, dest) \geq cw$ (there is a place for switching in the next interval) and $e_v + f(v, dest) > c(v, dest) - cw$ (there is no place for switching in the current interval), then a window switching is added in (ni), and $il_{ni} = true$ is set;
 - 2.1.1.2. Otherwise, a window switching is added in (v), and $ir_v = true$ is set.
 - 2.1.2. If $fp_{ni} = lp_v$ and there is a switching in (v) on the right and ($ir_v = true$), then remove the window switching in (v) and set $ir_v = false$; if there are switchings in ni on the left ($il_{ni} = true$), then remove the window switching in (ni) and set $il_{ni} = false$.
 - 2.2. If all vertices on the right are empty and there is a switching on the right in v , then remove the window switching in (v) and set $ir_v = false$.

3. Modify the left switching (similarly to how it was done for the right switching).

3.1. If there is a nonempty interval vertex pi on the left, then the following holds:

3.1.1. If $lp_{pi} \neq fp_v$ and there are no switchings on the left in v ($il_v = false$) and there are no switchings on the right in pi ($ir_{pi} = false$), then the following holds:

3.1.1.1. If $c(pi, dest) - f(pi, dest) \geq cw$ (there is a place for switching in the preceding interval) and $e_v + f(v, dest) > c(v, dest) - cw$ (there is no place for switching in the current interval), then a window switching is added in (pi) $ir_{pi} = true$;

3.1.1.2. Otherwise, a window switching is added in (v) , and $il_v = true$ is set.

3.1.2. If $lp_{pi} = fp_v$ and there is a switching on the left in v ($il_v = true$), then remove the window switching in (v) and set $il_v = false$; if there are switchings in on the right in pi ($ir_{pi} = true$), then remove the window switching in (pi) and set $ir_{pi} = false$.

3.2. If all vertices on the right are empty and there is a switching on the right in v , then remove the window switching in (v) and set $il_v = false$.

Removing Task u from the Network

For all pairs task vertex—interval vertex (u, v) , where the task vertex u corresponds to a partially allocated task, the following operations are performed.

1. The flow from the task vertex to the interval vertex is removed, and the same flow is subtracted from the flow directed from the interval vertex to the sink. The flow from the source to the task vertex is removed:

$$value = f(u, v),$$

$$e_{dist} = e_{dist} - f(u, v),$$

$$f(v, dist) = f(v, dist) - f(u, v),$$

$$f(dist, v) = -f(v, dist),$$

$$f(u, v) = 0, \quad f(v, u) = 0, \quad f(0, u) = 0, \quad f(u, 0) = 0.$$

2. Perform the operation of taking into account the partition when the flow of size $value$ of the partition pt_u is removed from the interval vertex v .

3. Modify windows in v .

4. The capacity of the edge leading from the source to the task vertex becomes zero: $c(0, u) = 0$.

Allocating the Partition Part to the Processor Proc

For all task vertices u such that $pt_u = part$ and all edges (u, v) such that $proc_v \neq proc$, set the capacity $c(u, v) = 0$ and $PR_{pt_u} = proc$.

Removal of the Partition part from the Processor proc

For all task vertices u such that $pt_u = part$ do the following:

1. For all edges (u, v) such that $proc_v \neq proc$, set $c(u, v) = dur_v$.

2. For all edges (u, v) such that $proc_v = proc$, do the following:

2.1. Perform the operation of taking into account the partition when the flow of size $value$ of the partition pt_u is removed from the interval vertex v ;

2.2. Modify the windows in v ;

2.3. Set the capacity of the edge leading from the task vertex to the interval vertex to zero: $c(u, v) = 0$.

Pushing (u, v)

1. Increase the flow $f(u, v)$ by $\delta f(u, v) = \min(e_u, c(u, v) - f(u, v))$.
2. Increase the excess flow e_v by $\delta f(u, v)$.
3. Decrease the reverse flow $f(v, u)$ and the excess flow e_u by $\delta f(u, v)$.
4. If u is a task vertex, v is an interval vertex, and the partition has not yet been allocated to a processor, then allocate the partition pt_u to the processor $proc_v$, perform the operation of taking into account the partition when the flow of value $\delta f(u, v)$ of the partition pt_u is added to the interval vertex v , and modify the windows in (v) .
5. If u is an interval vertex and v is a task vertex, then perform the operation of taking into account the partition when the flow of value $\delta f(u, v)$ of the partition pt_v entering the interval vertex is removed and modify the windows in (u) .
6. If $e_v = 0$, then remove the vertex from the list of overwhelmed vertices of the corresponding partition.
7. If $e_v > 0$, then add the vertex to the list of overwhelmed vertices of the corresponding partition.

Pushing (u, v) Into the Interval Vertex Taking into Account the Residual Flow (v, dist)

In this operation, v is the interval vertex.

1. Increase the flow $f(u, v)$ by

$$\delta f(u, v) = \max(0, \min(e_u, c(u, v) - f(u, v), c(v, dist) - f(v, dist) - ccw - e_v)). \quad (3.1)$$
2. Increase the excess flow e_v by $\delta f(u, v)$. Decrease the reverse flow $f(v, u)$ and the excess flow e_u by $\delta f(u, v)$.
3. If u is a task vertex, v is an interval vertex, and the partition has not yet been allocated to a processor, then allocate the partition pt_u to the processor $proc_v$, perform the operation of taking into account the partition when the flow of value $\delta f(u, v)$ of the partition pt_u is added to the interval vertex v , and modify the windows in (v) .
4. If $e_v = 0$, then remove the vertex from the list of overwhelmed vertices of the corresponding partition.
5. If $e_v > 0$, then add the vertex to the list of overwhelmed vertices of the corresponding partition.

Discharging the Vertex u

While $e_u > 0$, do the following.

1. Consider all available vertices for u one-by-one.
 2. For the first examination of the vertex:
 - 2.1. If v is an interval vertex, then cw is the number of window switchings that happen while the flow is pushed in the interval vertex v ;
 - 2.2. The value of the flow being pushed is calculated under the condition $h_u = h_v + 1$ by formula (3.1); if this value is distinct from zero, then push (u, v) into the interval vertex taking into account the residual flow $(v, dist)$.
 3. For the next examinations of the same vertices:
 - 3.1. If $h_u = h_v + 1$, v is the source, and $K > 0$, then set $K = K - 1$ and remove the partition pt_u from the processor PR_{pt_u} ; otherwise, push (u, v) .
 4. Lift u .
 5. Return to Step 1.
- Possible discharge of vertex u .
In this operation, u is a task vertex.
For all edges (u, v) do the following.

1. If v is the source, then if $h_u = h_v + 1$ and $K = 0$, then set $K = K - 1$ and remove the partition pt_u from processor PR_{pt_u} ; otherwise, push (u, v) .
2. If v is an interval vertex, then if $h_u = h_v + 1$, calculate the value of the flow being pushed by formula (3.1); if this value is distinct from zero, then push (u, v) into the interval vertex taking into account the residual flow $(v, dist)$.

Recovering the Schedule

For all processors, the following operations are sequentially executed.

1. If there is a window switching on the left, then the current window is closed, the switching time is taken into account, the window of the first partition in this interval is opened, and the number of window switchings in this interval is decreased by one.
2. If there is a window switching in the middle, then (while there are switchings in the middle) the window is closed after the time equal to the flow of this partition into this vertex. The partitions are iterated through beginning from the first one and to the last one (irrespective of the order in the center), the switching time is taken into account, and the window of the next partition is opened.
3. If there is a window switching on the right, then the current window is closed, the switching time is taken into account, and the window of the first partition in the next interval is opened.

To specify the tasks executed within a window and the execution duration allocated in each window, it is sufficient to find the flow entering the corresponding window interval. The value of this flow is exactly the execution duration of the task in this window. If there is no flow (its value is zero), then the task is not executed in this window.

The Basic Scheme of the Algorithm

1. Initialize the network.
2. Construct the maximum flow.
 - 2.1. Select a partition with a nonempty list of overwhelmed vertices (if there is no such partition, then go to Step 3); for these overwhelmed vertices u , perform the operation of possible discharge of vertex u .
 - 2.2. While the lists of overwhelmed vertices of the partition and of interval vertices are not empty, do the following:
 - 2.2.1. For all overwhelmed interval vertices v , discharge vertex v ;
 - 2.2.2. For one overwhelmed task vertex u of the partition, discharge vertex u .
 - 2.3. Go to Step 2.
3. If there are partially allocated tasks, then remove the first such task from the network and go to Step 2.
4. Recover the schedule.

4. PROPERTIES OF THE ALGORITHM OF CONSTRUCTING THE STATIC–DYNAMIC SCHEDULE BASED ON FINDING THE MAXIMUM FLOW IN A NETWORK

The result produced by the algorithm satisfies the schedule validity conditions. The main operation of the algorithm is the vertex discharge operation, which, in turn, consists of pushing operations from vertex-to-vertex and vertex lift operations. The lift operation affects only the order of pushes and does not affect the schedule's validity.

The push operation is designed in such a way that the schedule's validity conditions are satisfied if there is no excess flow in the network.

The push operation related to an interval vertex performs windows correction operations; the windows correction operation adds window switchings between partitions to the corresponding vertex—it reserves the time for switching between partitions so that other tasks would not be able use it. The windows are determined based on the durations of tasks consecutively following one another. The repeated search of the not completely allocated tasks removes such tasks from the network.

Since the allocation is performed immediately after the first push of the task to an interval vertex of the processor, the subsequent allocation of tasks of this partition is possible only on the same processor (because the capacities of the edges leading to the other processors are zero) until the time the task wants to push the flow into the sink, which implies that the entire partition cannot be allocated to this processor.

Then, the second operation is performed. The flow from all interval vertices connected with the task vertices of this partition is withdrawn from (the schedule remains valid because all window switchings are taken into account). The capacities of edges leading to other processors are restored. Therefore, the condition that all tasks of one partition are executed on the same processor cannot be violated. The network construction takes a time $O(pn^2)$. The schedule recovery takes $O(pn)$.

For one processor, the following conditions are satisfied: if n is the number of tasks, then $V \leq 3n + 2$ and $E \leq 3n + 2n^2$. The first step of the algorithm is to discharge all vertices of one partition, if possible; in the worst case, this step requires $2n^2$ push operations. The next step is to discharge all interval vertices, which requires $2n(n + 1)$ push operations in the worst case. These steps are performed for each partition; therefore, $n(2n^2 + 2n(n + 1))$ push operations are required. Let us prove that not more than ten push operations can be made for each pair task vertex—interval vertex. Consider the height function for task vertices; it is always greater than zero. If it equals eleven, then the push is into the sink, and this flow never appears in the network again. The sink height equal to ten gives five push attempts from one task vertex into an interval vertex. Since the heights of interval vertices do not decrease (when the push operation returns the flow into the interval vertex), the height becomes greater than the height of the task vertex from which the push operation was made. Therefore, for all pairs of task vertices and interval vertices, not more than ten push operations are possible. If a task is removed, the algorithm should be executed again; at maximum, n tasks can be removed, which corresponds to n iterations of the algorithm; thus, the total computational complexity of the algorithm is $n(2n^3 + 2n(n + 1) + 20n^2)$. Thus, the total complexity of the algorithm for finding the maximum flow for one processor is $O(n^4)$ if some tasks should be removed and $O(n^3)$ if all tasks can be scheduled. For the multiprocessor schedule, the complexity for all processors is $O(pn^3)$.

Since there are only K attempts of redistributing the partition, the complexity is $O(Kpn^3)$ in the case of complete scheduling and $O(Kpn^4)$ if some tasks are removed.

Experiments were performed on a 2.39 GHz Intel Core i3-2370M computer with 8 Gb of memory in Windows 7 64 bit.

For all generated input data, the algorithm of constructing the single processor static—dynamic schedule allocated all tasks. It turned out that the time needed to obtain the solution only weakly depends on the processor workload and does not exceed 1.2 s for 1000 tasks. The number of partitions barely affects the accuracy and time of solution either.

For all the generated input data, the algorithm scheduled all tasks in the case of two processors with the workload not exceeding 90%. The increase of the number of processors negatively affects the number of scheduled tasks; e.g., for three processors all tasks can be scheduled if the processor workload does not exceed 70%; if the workload is greater, 99% of all tasks can be scheduled. If the number of processors is increased to eight, 99% of tasks can be scheduled if the workload does not exceed 70%, and 90% of tasks can be scheduled if the workload does not exceed 90%. The time taken by the construction of a multiprocessor static—dynamic schedule grows with the increase of the processor workload (for eight processors). This is due to the fact that the number of scheduled tasks is less than 100%, and additional time is needed for removing unscheduled tasks. The time of constructing such a schedule can be as long as 20 min. For 2–4 processors, the time of constructing a static—dynamic schedule does not exceed 4 min and the accuracy is 99–100% for the processor workload up to 90%.

CONCLUSIONS

The problem of constructing static—dynamic schedules arises in designing real-time information and control systems with the integrated modular avionics architecture.

In some cases of scheduling preemptive tasks, algorithms based on finding the maximum flow in a transportation network proved to be efficient in terms of accuracy and computational complexity. The main difficulties preventing the use of known algorithms based on finding the maximum flow in a transportation network for constructing static—dynamic schedules are the need to take into account the membership of tasks in partitions and to construct the set of windows.

In the proposed algorithm, these difficulties are resolved due to a modification of the algorithm finding the maximum flow in a network.

The experimental investigation of the algorithm illustrated its efficiency in terms of accuracy and computational complexity on many classes of initial data.

FUNDING

This work was supported by the Russian Foundation for Basic Research, project no. 17-07-01566.

REFERENCES

1. V. A. Kostenko, "Architecture of software and hardware complexes of on-board equipment," *Izv. Vyssh. Uchebn. Zaved., Priborostr.* **60**, 229–233 (2017).
2. Arinc Specification 653. Airlines Electronic Engineering Committee. <http://www.arinc.com>.
3. A. N. Godunov, "Real-time operating systems baguette 3.0," *Program. Produkty Sist.*, No. 4, 15–19 (2010).
4. A. N. Godunov and V. A. Soldatov, "Operating systems of the baguette family (likeness, differences and perspectives)," *Programmirovaniye*, No. 5, 69–76 (2014).
5. V. A. Balakhanov and V. A. Kostenko, "Ways of reducing the building task of a static-dynamic uniprocess schedule for real-time systems to the problem of finding the route graph," *Program. Sist. Instrum.*, No. 8, 148–156 (2007).
6. V. A. Balakhanov, V. A. Kokarev, and V. A. Kostenko, "The possibility of using ant algorithms to solve the problem of constructing static-dynamic schedules," in *Proceedings of the 5th Moscow International Conference on Operation Study ORM2007* (MAKS Press, Moscow, 2007), pp. 238–240.
7. V. V. Balashov, V. A. Balakhanov, and V. A. Kostenko, "Scheduling of computational tasks in switched network-based IMA systems," in *Proceedings of the International Conference on Engineering and Applied Science's Optimization, Athens, Greece, 2014*, pp. 1001–1014.
8. V. V. Balashov, "Family of design automation systems for real-time onboard computing systems," *Program. Produkty, Sist. Algoritmy*, No. 4, 1–19 (2017).
9. A. Federgruen and H. Groenevelt, "Preemptive scheduling of uniform machines by ordinary network flow technique," *Manage. Sci.* **32** (3) (1986).
10. T. Gonzales and S. Sanhi, "Preemptive scheduling of uniform processor systems," *J. Assoc. Comput. Mach.* **25** (1) (1978).
11. M. G. Furugyan, "Computation planning in multiprocessor real time automated control systems with an additional resource," *Autom. Remote Control* **76**, 487 (2015).
12. M. G. Furugyan, "Computation scheduling in multiprocessor systems with several types of additional resources and arbitrary processors," *Mosc. Univ. Comput. Math. Cybern.* **41**, 145–151 (2017).
13. M. G. Furugyan, "Scheduling in multiprocessor systems with additional restrictions," *J. Comput. Syst. Sci. Int.* **480**, 222 (2018).
14. V. A. Kostenko and A. S. Smirnov, "Algorithm for static-dynamic scheduling of uniprocessor systems," *Vestn. Mosk. Univ., Ser. Vychisl. Mat. Kibernet.*, No. 1, 45–52 (2018).

Translated by A. Klimontovich

SPELL: OK