The Impact of Blocking Factor on Real-Time Applications Feasibility

Sergey Baranov, Victor Nikiforov
SPIIRAS, ITMO University
St. Petersburg, Russia
snbaranov@gmail.com, nik@iias.spb.su

Abstract—**An approach to estimating the blocking factor impact on feasibility of software applications in real-time systems on multi-core processors is described. An option for compound blocking to occur is demonstrated for the priority inheritance protocol as well as for other known protocols for access to shared resources. A method to estimate the impact of chained blocking on feasibility of particular application tasks and the application as a whole is described. A method to estimate the blocking factor value for systems with compound and chained blocking on multi-core processors is presented.**

I. Introduction

The criterion of feasibility of tasks constituting a software application for a real-time system (RTS) is guaranteed on-time realization of RTS functions assigned to its tasks τ_1, τ_2, ..., τ_n. Each task τ_i is characterized by at least two parameters: its period T_i and weight C_i (the processor time, that it may consume). Each task τ_i of the applications should comply with the constraint D_i imposed on its response time R_i (which is the maximal time interval between activation of the task τ_i and its termination). With this notation the criterion of task τ_i feasibility is meeting the requirement $R_i \le D_i$. The value of the response time R_i depends on the adopted scheduling mode and protocol of access to shared informational resources.

The scheduling mode determines the order of allocating the executive resource (processor time) to tasks which request it. Studies aimed at developing efficient scheduling modes for RTS started over 40 years age [1] and were successfully pursued for RTS with a single executive resource ([2], [3]) as well as for multi-processor systems and multi-core processors ([4], [5], [6]). Studies in this area are going on up to now, especially in the area of further improving the methods of estimating feasibility of applications for multi-core processors ([7], [8]). In particular new methods for increasing the utility load of multi-processor systems and for estimating feasibility of applications with non-trivial structures of composing tasks were suggested quite recently ([9], [10]).

Any scheduling mode may be specified by the way how integral priorities are assigned to tasks.

Protocols of access to shared informational resources determine the order of entering and exiting for those code segments (critical intervals) of programs which access those resources and consist of: a) a precondition for entering a critical interval; b) modification of system attributes at actual entering the critical interval; and c)modification of system attributes at exiting the critical interval [11].

Let's consider that scheduling mode with statically assigned basic priorities is used and that the basic priority of the task τ_i is determined by the value of its index i (the task τ_i priority is higher than that of task τ_j if $i<j$). Feasibility of an RTS application as a whole, is understood as feasibility of each of its constituting tasks.

Two approaches to estimating feasibility of software applications are being developed. The first one is based on estimating the integral utility $U=\sum_{1<i<n} u_i$, where $u_i=C_i/T_i$. With this approach the value U is compared against some boundary value UB (*utility bound*) of the integral utility. If the inequality $U \le UB$ holds, then feasibility of all application tasks is guaranteed. Checking application feasibility within this approach is called a *UB-test*.

The second approach is based on estimating the response time R_i for each task τ_i in the application. Checking application feasibility within this approach is called an *RespT-test* (Response Time test).

UB-tests are quick, but they give a rough (pessimistic) estimation of feasibility in the sense that if a UB-test passes, then feasibility of each application task is guaranteed, but a test failure does not imply that some task may violate its deadline.

The first UB-test was described over 40 years ago – it was demonstrated in [1] that in applications consisting of n independent tasks executed on classical single core processors with the RM scheduling, provided $T_i=D_i$, each task is feasible if the inequality $U \le UB=n\times(2^n\text{-}1)$ holds. This UB-test was been developed on a constructive manner: the worst configuration of application was found, that becomes infeasible, if this UB-test is failed.

Later this UB-test was extended to systems on multi-core processors ([12], [13]) but in non-constructive manner. The known UB value for applications, implemented on multi-core processors, has some reserve – if the UB-test is satisfied, application is feasible, but no configuration of application is known, for which UB-test failing certainly implies the unfeasibility of application.

RespT-tests are based on calculations which take more processor time (compared to UB-tests) but provide higher accuracy – they are much less pessimistic. In case of classical single-core systems, algorithms for response time estimating provide exact results because they involve analysis of system functioning in the worst (critical) scenarios of the system events. In this case substituting the calculated response time in the inequality $R_i \leq D_i$ provides the necessary and sufficient condition for feasibility of the task τ_i.

For independent tasks the response time R_i equals to the sum of two components: $R_i = C_i + I_i$ where C_i is the task τ_i weight factor and I_i is the priority factor – maximal duration of its existence in the "ready" state.

For classical single-core processors contribution of each task τ_j of higher priority than that of τ_i in the value of I_i equals to $\lceil R_i/T_j \rceil$, where $\lceil x \rceil$ is the integral ceiling of x. In this case the priority factor equals to the sum $I_i = \sum_{j<i} C_j \times \lceil R_i/T_j \rceil$. Substituting this sum in the expression for R_i a recurrence equation is obtained. A solution for this equation is obtained by the method of successive approximations [11].

Generalizations of feasibility estimation methods with RespT-tests for systems with multi-core processors (e.g., suggested in [14] and [12]) may produce over-estimations of the response time – i.e., may have a pessimistic bias. This is due to the fact that no universal means for constructing a critical scenario for system events is known for the systems with multi-core processors.

In case of systems with interdependent tasks the expression for response time R_i should be complemented with the blocking factor B_i which reflects increase of the existence interval of jobs of the type τ_i on behalf of possible existence in the "waiting" state:.

$$R_i = C_i + I_i + B_i\ . \qquad (1)$$

In this paper we present the method of estimating the B_i value for application on multi-core processors.

II. Estimating the Response Time

When estimating R_i with the proposed method, we'll use the index h for high-priority tasks (task τ_h with the basic priority higher than that of the task τ_i) and the index l for relatively low-priority tasks.

In case of multi-core processors, the existence interval of the next job of the task τ_i type (the interval of the next execution of the task τ_i) consists of segments of two kinds:

a) segments where the job of the τ_i type either owns the processor resource, or is waiting for a low-priority task to release the required informational resource);
b) segments where the job of the τ_i type is waiting for one of processor cores to be allocated to it.

When running a software application on a multi-core processor, existence intervals of jobs of the type τ_i do not contain segments of the b) kind for $i \leq m$. Really, according to priority-based scheduling modes at any given time some processor core is ready to execute either the code of τ_i, or the code of a critical interval of a job which inherited the priority of the task τ_i for a period of this blocking critical interval.

The maximal possible total duration of segments of the kind a) for instances of the task τ_i is equal to $C_i + B_i$ (the sum of the values of the weight factor and the blocking factor of this task). The maximal possible total duration of segments of the kind b) corresponds to the value I_i of the task τ_i priority factor.

As mentioned above, the priority factor equals to zero for the tasks $\tau_1, \tau_2, ..., \tau_m$.

The key principle determining the proposed approach to estimating the task response time for systems on multi-core processors is based on the following assertion: for instances of the task τ_i with $i>m$ at segments of the kind a), the processor cores are loaded neither with code of high-priority tasks τ_h, nor with execution of those critical intervals of low-priority tasks τ_l which block high-priority tasks τ_h. At these segments the processor cores either serve low-priority tasks (along with τ_i), or stay idle.

Each task τ_h contributes to the overall amount of computation performed within the existence interval of an instance of the task τ_i. The size of this contribution is determined by the expression $(C_h + BI_h(\tau_i)) \times N(i,h)$, $BI_h(\tau_i)$ being the indirect blocking factor of τ_h (the maximal possible duration of blocking the high-priority task τ_h by tasks with priorities lower than that if the task τ_i), and $N(i,h)$ being the maximal possible number of task τ_h activations within the time period of R_i.

For $i>m$, the maximal possible amount of computations performed at segments of the kind b) of the existence interval of a task τ_i instance is determined by the expression $\sum_{i<h}(C_h + BI_h(\tau_i)) \times \lceil R_i/T_h \rceil$, $\lceil x \rceil$ being the integral ceiling of the number x. This leads to the conclusion that in accordance with the key principle formulated above (these computations are performed by all m cores of the processor), the value I_i of the priority factor is determined by the expression:

$$I_i = (1/m) \sum_{i<h} (C_h + BI_h(\tau_i)) \times \lceil R_i/T_h \rceil. \qquad (2)$$

A particular case when all B_i in expression (1) and all $BI_h(\tau_i)$ in expression (2) are equal to zero, corresponds to systems without critical intervals for access to shared resources. A method of estimating the response time for applications on multi-core processors with no sharing of informational resources is derived through skipping the components B_i and $BI_h(\tau_i)$ and by substituting (2) for (1).

For systems with informational resource sharing, estimating methods of factors B_i and $BI_h(\tau_i)$ are very much needed. These estimates are performed differently for different protocols of access to shared resources [5].

III. Protocols of Access to Shared Resources

An access protocol determines specific features of executing the operation of requesting a resource:

- the required pre-conditions for a permit for the task to occupy the requested resource to be issued;
- system side effects occurring when a critical interval on access to the allocated resource is entered.

Priority inheritance protocol. With the simplest protocol (SP), there's only one pre-condition for allocating a requested resource: the resource should be unoccupied, and no side effects occur at entering the critical interval (except for the mandatory one – the resource state becomes "occupied"). Using the SP may result in priority inversion, when high-priority task waiting for a resource is prolonged because of actions of tasks with intermediate priorities. Eliminating the possibility for priority inversion to occur is ensured with the priority inheritance protocols (PIP). The pre-condition for a requested resource to be allocated is the same as with the SP (the resource should be unoccupied); however, a specific system side effect occurs when a task τ_i requests a resource occupied by task τ_l: the priority of task τ_l which owns the occupied resource temporarily, until its exit from the critical interval, is raised to that of the task τ_i (τ_l inherits the priority of τ_i).

Priority Ceiling Protocols. Using PIP does not prevent compound blocking of high-priority tasks and a occurrences of situations with mutual task blocking (deadlines and clinches) [14]. To eliminate the possibility of mutual or compound blocking, access protocols use *priority ceilings* of resources – statically defined parameters of each shared resource: the resource priority ceiling equals to the highest base priority of a task which may occupy this resource. Two breeds of such protocols are known – the priority ceiling protocol (PCP) and the preventive priority inheritance protocol (PPIP) [11].

PCP is characterized by one more pre-condition for allocating a requested resource, in addition to that of PIP: the requested resource is allocated to the requesting task only when its base priority exceeds the maximum of priority ceilings of all resources currently occupied by other tasks.

PPIP is characterized by increasing its system side effect against that of PIP at entering a critical interval: when the requested resource is allocated to task τ_i, the priority of τ_i is immediately raised up to the priority ceiling of this resource; the task enjoys this raised priority until it releases the allocated resource.

When running applications with shared resources on single-core processors, both PCP and PPIP ensure that mutual task blocking for any application configurations is impossible. When running on multi-core processors, PCP still preserves this useful feature, while PPIP does not. For single-core processors both PCP and PPIP ensure the impossibility of compound blocking, while running on multi-core processors compound blocking may occur for both PCP and PPIP [15].

IV. Compound blocking

Software applications considered in this section should meet the following constraint.

Constraint 1. Application tasks do not contain intersecting critical intervals for accessing resources.

Including the indirect blocking factor $BI_h(\tau_i)$ in formula (2) for calculating the priority factor I_i allows to take into account the possibility of task τ_h priority inheritance by instances of tasks with priorities lower than that of task τ_i. In case such priority inheritance occurs, the critical interval of a low-priority task is executed with the priority level of task τ_h, which allows to consider execution of this critical interval as part of computation for task τ_h in segments of the kind *b*) of execution of the next instance of task τ_i.

When an instance of high-priority task τ_h is blocked by τ_i, execution of the blocking critical interval refers to kind *a*) segments of the existence interval of the task τ_i instance.

Let's estimate the maximal duration of the blocking interval of a high-priority task τ_h by a low-priority instance of τ_l under the following constraint.

Constraint 2. Any blocked task contains only one critical interval.

As one can see, constraint 2 strengthens the constraint 1.

Let there be in the code of a number of low-priority tasks τ_l critical intervals capable to block τ_h when requesting a resource g, $C_l(\tau_l, g)$ being the duration of a critical interval of access to the resource g in the code of task τ_l. $C_l(\tau_l, g)=0$ when there's no access to g in the code of τ_l.

The nomenclature of critical intervals determining the value $BI_h(\tau_i)$, depends on the used protocol of access to resources. In case of PIP, when a resource g is requested by a high-priority job of the type τ_h, this job may be blocked only by critical intervals of low-priority tasks performing access to this very resource g requested by a high-priority task τ_h. Therefore, in case of PIP, the maximal possible value of indirect blocking factor $BI_h(\tau_i)$ is:

$$BI_h(\tau_i)=\max(C_l(\tau_l, g)\,|\,l>i\} \,, \qquad (3)$$

g being the resource used by the task τ_h.

With PCP, the nomenclature of blocking critical intervals for τ_h may be larger than that with PIP. In case of PIP, an instance of a high-priority task τ_h is blocked by not only critical intervals of access to resource g, but by critical intervals of tasks τ_l of access to any resource with the priority ceiling not lower than the base priority of τ_h. A formal estimate of the indirect blocking factor when PCP is used, is given by the expression:

$$BI_h(\tau_i) = \max(C(\tau_l, g') \mid l>i, \pi(g')<h\} \qquad (4)$$

g' being any of resources used by low-priority tasks with the priority ceiling $\pi(g')$ not lower than that the base priority of task τ_h.

The maximal possible value of the blocking factor B_i is determined in a similar way:

- critical intervals in the code of low-priority tasks τ_l, capable to block τ_i are identified;
- the value of the blocking factor B_i is determined as the maximum of durations $C_l(g)$ of the identified critical intervals.

When PIP is used, the value of the blocking factor is given by the expression:

$$B_i = \max\{C(\tau_l, g)\,|\,l> i\}, \qquad (5)$$

g being the resource used by task τ_i.

To estimate the response time of tasks which meet the constraint 2, the value (3) is substituted in (2) and then the values (2) and (5) are substituted in (1).

In case of PCP, the nomenclature of critical intervals blocking τ_i may turn to be larger than that for PIP. The causes of this difference are similar to those highlighted for the indirect blocking factor $BI_h(\tau_i)$. The value of the blocking factor B_i with PCP is given by the expression:

$$B_i = \max(C(\tau_l, g')|l > i, \pi(g') \leq i\}, \qquad (6)$$

g' being any resource with the priority ceiling not lower than the base priority of the task τ_i. As for $BI_f(\tau_i)$, the value B_i for PCP is in general not lower and in a number of cases even higher than that for PIP.

To estimate the response time of tasks with PCP which meet the constraint 2, the value (4) is substituted in (2) and then the values (2) and (6) are substituted in (1).

Compound blocking. Let's eliminate the constraint 2: let the code of any task to contain a number of non-intersecting critical intervals. Then the possibility of compound blocking is invited – during task run the task may be blocked not once when requesting access to various resources.

In Fig. 1*a* an application consisting of 3 tasks is presented in the formalism of route networks [16]. The task τ_1 contains two non-intersecting critical intervals of access to resources g_1 and g_2. The weight of each interval equals to 2. The time of the first activation of each task is specified by its phase: $\varphi_1 = 4$, $\varphi_2 = 2$, and $\varphi_3 = 0$.

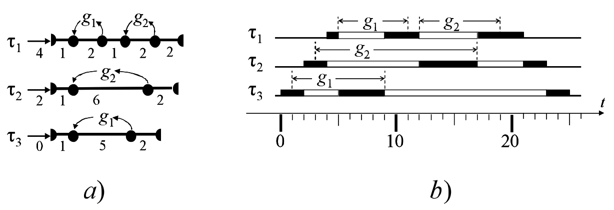

Fig. 1. Compound blocking with PIP on a single-core processor:
a) application configuration consisting of 3 tasks;
b) ordering of requested resource allocation

In Fig. 1*b* the starting piece of the run chart of the application $\{\tau_1, \tau_2, \tau_3\}$ is presented when running on a single-core processor and using PIP. At time $t = 1$ the task τ_3 occupies the resource g_1, at $t = 3$ the task τ_2 occupies the resource g_2. At $t = 5$ the task τ_1 becomes blocked because of its request for an occupied resource g_1; the priority of τ_1 is inherited by task τ_3.

At $t = 12$ the task τ_1 becomes blocked again because of its request for the resource g_2, occupied by task τ_2. Thus, Fig. 1 demonstrates a possibility for compound blocking to occur when running inter-dependent tasks with PIP on a single-core processor. Compound blocking is impossible with PCP or PPIP; however, this is no more true when applications run on a multi-core processor.

Fig. 2 represents the configuration and starting piece of a run chart of an application of 3 tasks with PCP running on a two-core processor.

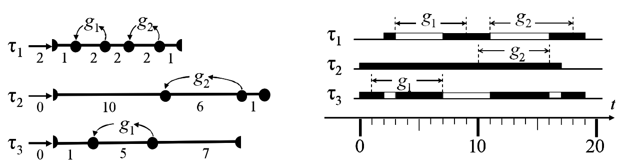

Fig. 2. Compound blocking with PCP on a two-core processor

At time $t = 1$ the task τ_3 occupies the resource g_1, at $t = 3$ the task τ_1 becomes blocked because of its request for an occupied resource g_1 and resumes its run after g_1 is released at $t = 7$. During all this time the other processor core executes the code of the task τ_2, which occupied the resource g_2 at $t = 10$. The task becomes blocked for the second time when requesting access to resource g_2 at $t = 11$. Thus, Fig. 2 demonstrates the possibility of compound blocking when inter-dependent tasks with PCP run on a two-core processor.

The possibility of compound blocking of a task τ_i, the code of which contains a series of critical intervals of access to resources $g_{i,1}$, $g_{i,2}$, ... $g_{i,k}$, ... being taken into account, to estimate the value of the blocking factor B_i with PIP, the following sum should be calculated:

$$B_i = \sum_{1 \leq k} \max\{C(\tau_l, g_{i,k}) \mid l > i\} \qquad (7)$$

$g_{i,k}$ being the resource used at the k^{th} critical interval of task τ_i, and $C(\tau_l, g_{i,k})$ – being the maximal duration of the critical interval of access to the resource $g_{i,k}$ in the code of a low-priority task τ_l. When PCP is used, instead of the sum (7) the sum:

$$B_i = \sum_{1 \leq k} \max\{\, C(\tau_l, g')|l > i, \pi(g') \leq i\,\} \qquad (8)$$

should be calculated, g' being any resource with the priority ceiling not lower than the base priority of task τ_i. Actually, this means that in order to obtain the value of B_i the expression in the scope of the summation sign should be multiplied by the number of critical intervals in the code of the task τ_i.

The estimating formula for indirect blocking factor $BI_f(\tau_i)$ with PIP and several critical intervals looks as:

$$BI_h(\tau_i) = \sum_{1 \leq k} \max\{C(\tau_l, g_{i,x}) \mid l > i\}, \qquad (9)$$

$g_{i,x}$ being the resource used at the x^{th} critical interval of the task τ_h. When PCP is used, an estimate of indirect blocking factor $BI_h(\tau_i)$ for tasks with several critical intervals is given by the expression:

$$BI_h(\tau_i) = \sum_{1 \leq x} \max\{\, C(\tau_l, g')|l > i, \pi(g') \leq i\,, \qquad (10)$$

g' being any of resources with priority ceiling $\pi(g')$ not lower than that the base priority of task τ_h.

Formulae (7-10) reflect the contribution of compound blocking in the overall estimate of the response time for systems which do not meet the constraint 2. To obtain the value for response time R_i in systems with compound blocking, expressions (7-10) should be substituted in formulae (1) and (2) in the same order as expressions (3-6) are substituted for systems without compound blocking.

The considered methods for estimating the blocking factor value are applicable to systems which meet the constraint 1. In such systems a task may wait for releasing of one occupied resource at the entry to each critical interval. For systems not meeting the constraint 1, in addition to compound blocking *chained blocking* becomes possible, when a task may wait for releasing of two or more occupied resources at the entry to a critical interval. The next section represents an approach to estimating the contribution of chained blocking in the overall task response time.

V. CHAINED BLOCKING

Let's eliminate the constraint 1: let the code of any task contain intersecting critical intervals as well. Their existence leads to the possibility of not only compound blocking, but to the possibility of blocking any single resource request from several active jobs threaded through a chain of their critical interval dependencies. The name of chained blocking suits this breed of blocking pretty well. Fig. 3 and Fig. 4 represent the configuration and initial run diagram respectively for an application consisting of 4 tasks with chained blocking.

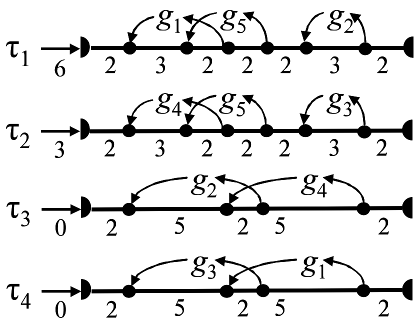

Fig. 3. Application with intersecting critical intervals

A characteristic feature of the 4-task application in Fig. 3 is that its each task contains intersecting critical intervals. Such applications are free of mutual blocking and therefore, PIP may be used within them.

The chart in Fig. 4 demonstrates that a request for the resource g_2 by task τ_1 results in blocking τ_1; first, directly by task τ_3 which owns the requested resource, and then indirectly (at t=19) by task τ_2 owning the resource g_4 needed not by task τ_1 but rather by task τ_3 which blocks τ_1. Thus, critical intervals of tasks τ_3 and τ_2 form two cohesions of a chain which blocks execution of task τ_1.

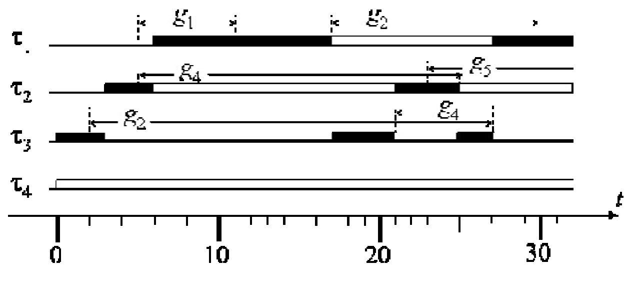

Fig. 4. Run diagram for the application in Fig. 3

How many active tasks may be involved in a blocking chain? Analysis of a special oriented graph – the graph of bundles and critical intervals – may answer this question.

Two critical intervals form a bundle if the end of the first one (the head of the bundle) is inside the second one (the tail of the bundle), while the beginning of the tail is inside the head. A graph of bundles and critical intervals is a multi-partite graph, its i^{th} partite consisting of vertices which correspond to:

- bundles of critical intervals of the task τ_i code;
- critical intervals within the bundles of the task τ_i code;
- free critical intervals (critical intervals within the task τ_i code which belong to no bundle).

The arcs of the bundle and critical interval graph are constructed according to the following rules.

Rule 1. Two vertices L_a and L_b (corresponding to the bundles L_a and L_b), are connected by an arc from L_a to L_b, if L_a and L_b belong to different graph partites and the head resource of the bundle L_b coincides with the additional resource of the bundle L_a.

Rule 2. An arc is drawn from vertex $G(g)$ to vertex L, if these vertices belong to different graph partites and the head resource of L coincides with the resource g.

Rule 3. An arc is drawn from vertex L to vertex $G(g)$, if the resource g coincides with the additional resource of bundle L.

Fig. 5. represents a graph of bundles and critical intervals for the application in FIG. 3 (arcs corresponding to Rule 3 are omitted).

Let's label each arc of the bundle graph with parameter W, which is called arc weight. The weight of an arc incoming to the vertex $G(g)$, equals to the amount of computation needed to execute the critical interval corresponding to this vertex. The weight of an arc incoming to the vertex L, equals to the amount of computation needed to execute the head critical interval of the bundle which corresponds to this vertex.

In order to find which variants of chained blocking are possible at the entry to the k^{th} critical interval of the task $\boldsymbol{\tau}_i$, one should build blocking paths – inter-partite paths belongs to the lower (w.r.t. to $\boldsymbol{\tau}_i$) graph partite (i.e., not going beyond the graph part which corresponds to tasks with priorities less than the base priority of $\boldsymbol{\tau}_i$) for the vertex G corresponding to this critical interval in the graph of bundles and critical intervals. The path weight equals to the sum of weights of its constituting arcs. Chained blocking is possible if some blocking path contains more than one bundle.

Fig. 5 depicts a bundle and critical interval graph for a software application configuration of Fig. 3.

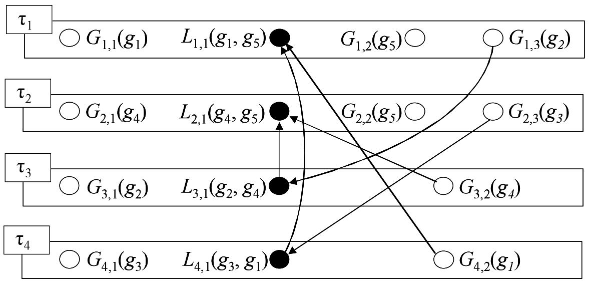

Fig. 5. Grah of bundles and critical intervals

There are two blocking paths for the vertex $G_{1,3}$ of the graph in Fig. 5 – paths ($G_{1,3}$, $L_{3,1}$) and ($G_{1,3}$, $L_{3,1}$, $L_{2,1}$). Path ($G_{1,3}$, $L_{3,1}$) contains an arc with the weight $W(G_{1,3}, L_{3,1}) = 6$. Path ($G_{1,3}$, $L_{3,1}$, $L_{2,1}$) contains two arcs with the total weight $W(G_{1,3}, L_{3,1}) + W(G_{1,3}, L_{3,1}) = 4+6=10$. Therefore, chained blocking is possible for a period of up to 10 timing units for the task τ_1 at the entry to its critical interval on the resource g_2. Note, that the path ($G_{2,3}$, $L_{4,1}$, $L_{1,1}$) does not present chained blocking because the vertex $L_{1,1}$ is beyond the graph part which corresponds to tasks with base priority less than that for task τ_2.

VI. Conclusion

The key requirement to a software application for real-time systems is feasibility of particular tasks constituting the application which means guaranteed on-time realization of the RTS functions performed by these tasks. The blocking factor – duration of intervals when the task is forced to wait for an access to informational resources temporarily occupied by other application tasks should be taken into account for feasibility analysis of tasks with shared common resources.

High-priority tasks may be delayed (blocked) at entries to code segments which realize access to shared resources (critical intervals of resource access). Such blocking lasts until the requested resource is released by a low-priority task which owned it. A real-time system should be built in such a way that delays caused by such blockings do not compromise the response time of high-priority tasks.

In systems on multi-core processors compound blocking becomes possible not only when the priority inheritance protocol is used, but with other known protocols of access to shared resources as well. The paper presents a method of estimating the response time in case of compound and chained blocking for software applications running on multi-core processors.

When implementing real-time multi-task software applications with the priority inheritance protocol, along with compound blocking another variant of blocking – chained blocking – becomes possible, when a high-priority task may be blocked by a number of tasks threaded through a chain of dependencies of their critical intervals. The proposed method of estimating the duration of chained blocking based on analysis of a multi-partite graph of bundles and critical intervals allows to estimate the impact of chained blocking on the overall response time of high-priority tasks.

Acknowledgment

This work was partially financially supported by the Government of the Russian Federation, Grant 074-U01.

References

[1] C. Liu, J. Layland, “Scheduling Algorithms for Multiprocessing in a Hard Real-Time Environment”, *Journal of the ACM*, vol. 20, n.1, 1973, pp. 46-61.

[2] A.D.Ferrari, “Real-Time Scheduling Algorithms”, *Dr.Dobb’s Journal*. 1994. no.12, pp. 60-66.

[3] V.V. Nikiforov, V.A. Pavlov, “Real-Time Operating Systems for Embedded Applications”, *Software and Systems*, n.4, 1999, pp. 24-30. (In Russian)

[4] A.I.Gruntal, “Scheduling for Systems with Asynchronous Start”, *Information Technologies and ComputerSystems*, n.1, 2012, pp. 32-51. (In Russian)

[5] S.K.Baruah, “Fairness in Periodic Real-Time Scheduling Algorithms”, *in Proc. of 16 IEEE Real-Time Systems Symposium*, 1995, pp.200-209.

[6] V.V. Nikiforov, M.V.Danilov, “Static Processing of Software System Specifications”, *Software and Systems*, n.4, 2000, pp. 13-19. (In Russian)

[7] Y. Sun, G. Lipari, N. Guan, W. Yi. “Improving the response time analysis of global fixed-priority multiprocessor scheduling”, *in Proc. of 20 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),* 2014, pp. 1-9.

[8] N. Guan, M. Han, Ch. Gu, Q. Deng, W. Yi. “Bounding Carry-in Interference to Improve Fixed-Priority Global Multiprocessor Scheduling Analysis”, *in Proc. of 21 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA),* 2015, pp. 11-20.

[9] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela. “The Global EDF Scheduling of Systems of Conditional Sporadic DAG Tasks”, *in Proc. of 27 Euromicro Conference on Real-Time Systems (ECRTS),* 2015, pp. 222 - 231.

[10] N. Guan. ; G. Chuancai, M. Stigge, Q. Deng. “Approximate Response Time Analysis of Real-Time Task Graphs”, *in Proc. of 35 IEEE Real-Time Systems Symposium (RTSS),* 2014, pp. 304 - 313.

[11] S.K. Dhall, C.L. Liu, “On a Real-Time Scheduling Problem”, *Operating Research*, vol. 26, n.1, 1978, pp. 127-140.

[12] T. Baker, “Multiprocessors EDF and Deadline Monotonic Schedulability Analysis”, *in Proc. of 24 IEEE Real-Time Systems Symposium*, 2003, pp. 120-129.

[13] V.V. Nikiforov, “Feasibility of Real-Time Applications on Multi-Core Processors”, *SPIIRAS Proceedings*, issue 8, 2009, pp. 255-284. (In Russian)

[14] V.V. Nikiforov, V.A. Pavlov, “Structured Models for Multi-Task Software System Analysis”, *Information-Measuring and Control Systems*, n.9, 2011, pp.19-29. (In Russian)

[15] V.V. Nikiforov, V.I. Shkirtil, “Compound Blocking of Dependent Tasks in Multicore Computers”, *Priborostroenie*, n.1, 2012, pp.25-31. (In Russian)

[16] V.V. Nikiforov, V.I. Shkirtil, “Route Networks – a Graphical Formalism for Representing the Structure of Real-Time Software Applications”, *SPIIRAS Proceedings*, issue 14, 2010, pp. 7-28. (In Russian)