Density of Multi-Task Real-Time Applications

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

Abstract—**An approach to estimate the efficiency of various combinations of scheduling modes and protocols for access to shared information resources in multi-task real-time software complexes is proposed. An efficiency criterion for such estimation is introduced. An architecture of a software tool to obtain concrete values of the introduced efficiency criterion for a given software application is described.**

I. Introduction

A software application for real-time systems (RTS) is usually built as a complex of cooperative tasks $\tau_1, \tau_2, \dots, \tau_n$ (i.e., tasks which jointly reach certain common goals for the given system) [1]. Each j-th activation of the task τ_i means respective generation of its j-th example – the job ${}_j\tau_i$. An RTS distinguishing feature is that its tasks are expected to be executed on time; this may be formally expressed as follows: duration $r({}_j\tau_i)$ of the existence interval of any example ${}_j\tau_i$ of the task τ_i shall never exceed some limiting value D_i. With the notion of the task *response time* $R_i = \max\{r({}_1\tau_i),\ r({}_2\tau_i),\ \dots\}$ the requirement of on time execution may be reformulated as $R_i \le D_i$. One distinguishes between hard and soft RTS [2]. For a hard RTS violation of the inequality $R_i \le D_i$ is not allowed. For a soft RTS violation of this inequality is acceptable for some small percentage of jobs or only the average value of durations $r({}_j\tau_i)$ of tasks τ_i is bounded.

Feasibility of a multi-task software application is its property which ensures that the inequality $R_i \le D_i$ holds for any task τ_i under any acceptable scenario of system events. To check feasibility of a software application some structured task models are built and analyzed, estimates of response time are performed for each task taking into account all factors which may impact the values R_i. These factors are:

• *external factors* (primarily variants of possible task activation scenarios for tasks which compose the system);

• *performance of execution resources* used for system implementation – the number of processors (processor cores) and their frequency;

• *the application structure* (includes details of the internal structure of each task and peculiar features of the inter-task interfaces nomenclature);

• methods used to access shared resources – *scheduling modes* which define the ordering of providing execution resources to active jobs and *access protocols* to shared information resources.

The most significant external factor for each task τ_i is the value T_i of its period which specifies the minimal duration of a time interval between arriving of similar jobs ${}_j\tau_i$ and ${}_{j+1}\tau_i$. The value D_i of the maximal acceptable response time of the task τ_i is also an external factor because it indicates the acceptable duration of external processes waiting for a control signal. In this paper a derivative parameter $H_i = T_i / D_i$ is considered which is the task period divided by the maximal duration of its execution. H_i characterizes the degree of hardness required for RTS tasks execution. The constraint $H_i \le 1$ means that existence intervals of similar jobs ${}_j\tau_i$ and ${}_{j+1}\tau_i$ do not intersect. The reverse condition $H_i > 1$ means that existence intervals of jobs ${}_j\tau_i$ and ${}_{j+1}\tau_i$ of the same type may intersect.

Performance P of the processor or its core may be represented as a number of some standard operations performed within one second. Floating point operations (flop) are often used as such standard ones; in this case P is characterized by the number of *flop/sec* (flops).

In order to verify feasibility of particular tasks and of the software application as a whole, structured task models are built – e.g., by means of route networks [3]. With this formalism the task code is represented as a series of segments separated by system operations of send/receive synchronizing information signals. Each segment is characterized by its weight w and the type of terminating operation. The segment weight represents the maximal possible amount of computational work performed within this segment and is measured with the number of standard operations. The total weight W_i of segments composing a task τ_i is the absolute task weight – the maximal possible number of standard operations performed within one activation of the task τ_i.

An alternative representation of the task weight takes into account the performance of a particular processor – the relative weight C_i of the task τ_i which is the absolute task weight W_i divided by the processor performance P ($C_i = W_i/P$) and is expressed in seconds (actually these are the number of seconds of processor work at the concrete performance).

A derivative parameter characterizing the task weight is the task utility $u_i = C_i/T_i$ which specifies the portion of the processor time which is needed for executing the task τ_i. Summing up all values u_i for all tasks produces the value of the overall utility $U = \Sigma_{1 \le i \le n} u_i$ (the portion of the processor time needed for the whole multi-task application). The value $1 - U$ specifies which portion of the processor time is not used by the application (the processor is either idle or is loaded with calculations unrelated to the RTS processing). Evidently, the overall utility depends on the processor performance. Increasing of the processor performance leads to an increase of the processor idle time (from the prospective of the given RTS software application).

The response time R_i of each task in the RTS software application depends substantially on the applied scheduling mode and the used protocol to access the shared system resources. Therefore, it is vital to know which combination of possible scheduling modes and access protocols is the most efficient for implementation of the given software application under the given circumstances. The maximal value of the overall utility which corresponds to the minimal processor performance with which the application is still feasible may serve as a criterion of such efficiency. Let's denote this extreme value of the overall utility as *Dens* and let's call it the application *density*. The value *Dens* is determined by external factors and structural features of the application, as well as by selection of the scheduling modes and protocols of access to shared informational resources. It may be used as a criterion of efficiency of the given combination the scheduling mode and access protocol. Use of a software tool that simulates the sequence of processor switching between the RTS tasks is a universal approach to determining the application density.

The following sections provide necessary information on scheduling modes and protocols for access to shared resources. Then an approach is described how to build a simulating software tool which estimates the task response time when executed under particular external conditions.

II. Scheduling modes

Any scheduling mode may be expressed through a method of assigning integral priorities prio$({}_j\tau_i, t)$. If this is defined, then the inequality prio$({}_k\tau_x, t) <$ prio$({}_l\tau_y, t)$ means that at the time moment t the task ${}_k\tau_x$ has a preference (with respect to ${}_l\tau_y$) for the processor resource. For some scheduling modes this way of defining the order in which the processor time is provided is possible, but it is not convenient; in this case scheduling modes are presented through a sorted list of active jobs and a way of modifying this list when particular system events occur.

The set of all possible scheduling modes is split into classes characterized by constraints on the allowed changes of task priorities. Class S_0 corresponds to modes with static task priorities (task priorities defined when the system is developed remain unchanged during the system work).

The class S_0 contains the most widely used Rate Monotonic (RM) mode: task priorities decrease as the value of T_i increases. For a set of independent tasks on a single core processor with the hardness value $H=1$ efficiency of the RM scheduling mode is characterized with the inequality $U<\ln 2$: the task feasibility is guaranteed if the overall utility is less than ln2 [4].

Weakening the constraints on task priority changes allows for selecting scheduling modes with higher efficiency of processor time usage.

With scheduling modes of the class S_1 priorities of jobs of the same type ${}_j\tau_i$ and ${}_k\tau_i$ $(j \neq k)$ may be different; i.e., different copies of the same task may have different priorities. A constraint on priority change for the class S_1 consists in that priority of a particular job ${}_j\tau_i$ stays unchanged within the whole interval of its existence.

Due to this weakening (with respect to the class S_0) of constraints on priority change, scheduling modes of the class S_1 allow to increase the efficiency of processor usage. Thus, the EDF mode (Earliest Deadline First – job priorities decrease as their deadlines increase on the time axe) ensures the maximal possible efficiency of processor time usage in a single processor RTS with a classical single core processor. When the EDF mode is used in such RTS, the sufficient condition of on time task feasibility is specified by the inequality $U \le 1$ [4].

Scheduling modes RM and EDF turn out to be inefficient [5] for RTS on multi-core processors. Therefore, modified versions of these modes have been developed. Scheduling mode RM_US of the class S_0 ensures the application feasibility if the inequality $U \le m/(3m-2)$ holds (m being the number of processor cores). Efficiency of the mode RM_US varies from 1/2 to 1/3 depending on the number of processor cores [6-8].

Elimination of constraints on job priority changing corresponds to the class S_2 – priorities of active jobs may change during their execution. This elimination makes possible to increase the efficiency of processor usage in RTS on multi-core processors. The Pfair scheduling mode of the class S_2 ensures feasibility of application if the inequality $U \le m$ holds (that means 100% efficiency of the processor resource usage) [8]; i.e., for systems implemented on multi-core processors efficiency of processor usage may be increased at 2-3 times with scheduling modes of the class S_2 compared to the ones of the class S_0.

III. Protocols of access to information resources

A piece of code with an implemented access to a global resource g is usually called a *critical interval* w.r.t. g.

Critical intervals are of the same type if they contain access to the same global resource. In order to ensure integrity of global resources, mechanisms should be used which prevent simultaneous access of the same resource by different jobs. To do this, a synchronizing element `mut_i` of the mutex type is formed in the software code for each shared resource g. Each piece of program code which implements access to the resource g (each critical interval with access to the resource g) is framed with operations on the mutex `mut_i`. A critical interval starts with the operator `lock(mut_i)` – lock the mutex `mut_i` and ends with the operator `unlock(mut_i)` – unlock the mutex `mut_i`. If the mutex `mut` is in the locked state when the operator `lock(mut)` is performed, then job execution is suspended until another job which owns the respective resource unlocks it by performing the operation `unlock(mut)`.

Operators `lock/unlock` split the task code into segments.

Fig.1 represents an interconnection structure of two tasks according to the approach proposed in [3] for representing inter-task interfaces with route networks.

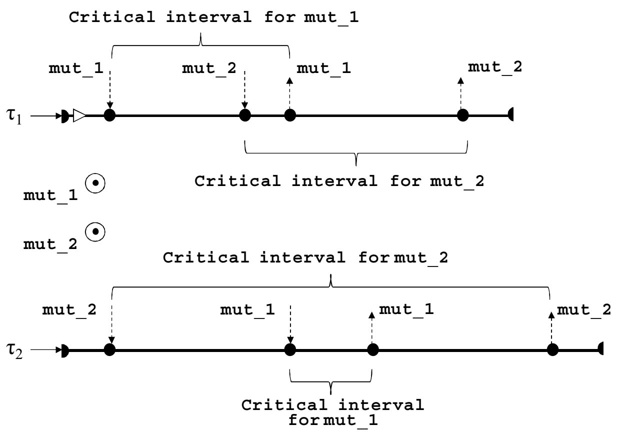

Fig. 1. Application Structure Represented as a Route Network

Each task in Fig.1 consists of 5 code segments and contains two critical intervals. Critical interval of the task τ_1 on the resource g_1 contains its code segments 2 and 3; critical interval on the resource g_2 contains its code segments 3 and 4. Critical interval of the task τ_2 on the resource g_1 consists of its code segment 3; critical interval on the resource g_2 contains its code segments 2, 3, and 4. It's essential that both tasks contain intersecting critical intervals:

• in τ_1 critical intervals are concatenated – segment 3, on one hand, terminates the critical interval on g_1 and, on the other hand, starts the critical interval on g_2;

• in τ_2 the critical interval on g_1 is nested into the critical interval on g_2.

When a software application contains tasks with intersecting critical intervals, mutual blocking of active jobs becomes possible. One can demonstrate that such situation may arise when running an application with the task code structure represented in Fig.1.

A standard approach to preventing mutual task blocking is based on providing the synchronization mechanisms of the mutex type with additional conditions and/or actions to be performed when executing operations on mutexes. The contents of such conditions-actions is called *protocols* for access to the shared informational resources.

When implementing software applications on single processor systems with classical single core processors, preventing mutual task blocking is performed through using the Protocol of Preventive Inheritance of Priorities (PPIP); for multi-processor systems and systems with multi-core processors the Protocol of Threshold Priorities (PTP) is used [9]. Both protocols assume scheduling modes of the class S_0 to be used.

Therefore, when building systems with intersecting critical intervals, developers prefer to use access protocols which do not allow more efficient scheduling modes of the classes S_1 or S_2. However, intersecting critical intervals do not assume a real threat of mutual task blocking. A method described in [10] checks whether mutual blocking is really possible.

The method is based on analysis of structural features of a special multi-partite graph – the graph of critical interval bundles. Based on this method, the developer can check whether the configuration of task interrelations requires standard access protocols to shared resources. If this check provides a negative answer (i.e., there's no option for mutual task blocking in spite of intersecting critical intervals), then there's no need to use PPIP or PTP. In this case the simplest protocol (SP) may be used for implementation of this RTS application, which does not require any additional conditions/actions when entering critical intervals for access to share resources; for SP it is sufficient that the respective mutex is unlocked.

In contrast to PPIP and PTP, using SP does not impose constraints on the scheduling mode – when SP is used any scheduling mode may be used instead of the modes from the class S_0. This option may turn out to be important because for multi-core processors scheduling modes of the class S_2 may provide 2-3 times increase of processor usage efficiency compared to modes of the class S_0.

The Protocol Preventing Mutual Blocking (PPMB) – is described in [11]. An advantage of PPMB is that, on one hand, it guarantees system protection from deadlocks and clinches along with PPIP and PTP; and on the other hand, it does not consider task priorities (and therefore, it may be combined with efficient scheduling modes of the classes S_1

and S_2). At the same time, as with other task priority ignorant protocols, the PPMB protocol allows for task chain blocking which results in increase of the response time and therefore in decrease of the software application density. One may ask: whether chain blocking compromises the advantages of scheduling modes of the classes S_1 or S_2? The answer may be yes and no depending on the structure and use conditions of a particular application. Instruments for simulating the sequence of processor switches between the RTS tasks are a universal tool for answering this question.

IV. An approach to modeling of execution a multi-task software application

The proposed approach to building a system for simulate execution of RTS is illustrated on a simplified software application with independent tasks (tasks without synchronizing inter-task interfaces). Moreover, only scheduling modes of S_0 class are supposed. However, the nomenclature of objects and procedures which ensure functioning of the simplified application may be modified and extended in a natural way to model the work of systems with arbitrary scheduling modes and with sharing information resources under arbitrary access protocols.

A. Input Data for the Modeling Process

The proposed architecture of a simulating software tool is based on using three chained lists:

`TaskList` – list of task descriptors;
`ReadyList` – list of active jobs;
`EventList` – list of scheduled system events.

The list `TaskList` of task descriptors, which compose the software application being modeled is the source data for the simulatng. Each task is characterized with three parameters:

`Period` – the length of the time interval between two adjacent activations;
`C_size` – the amount of processor time necessary for task execution;
`Piority` – an integer defining the task priority.

The nomenclature of tasks and the values of the enumerated parameters stay constant within the modeling session.

All objects, linked in the lists `TaskList`, `ReadyList`, and `EventList` belong to classes inherited from the class `ListNode`:

```
class ListNode {
  ListNode *Next;
  ListNode* Pop(){
    ListNode *ret_ptr = Next;
    if(ret_ptr != 0) Next = ret_ptr->Next;
    return(ret_ptr);
  }
  void Push(ListNode *chain){
    if(chain != 0) {
      chain->Next = Next;
      Next = chain;
    }
  }
}
```

The method `Push(ListNode *chain)` allows to insert a new element `chain` into the list; the method `Pop()` performs the reverse operation – deleting an element from the list. Joint usage of these methods allows to modify a list in the LIFO-buffering mode.

The class `OrderedNode` is the closest heir of the class `ListNode`; it is used to construct a list of objects sorted with respect to the special attribute `Ordering`:

```
class OrderedNode:ListNode {
  int Ordering;
  OrderedNode* Find(int key) {
    OrderedNode *suc_ptr;
    ListNode *pre_ptr = this;
    while((suc_ptr = (OrderedNode *)
                  pre_ptr->Next) != 0) {
      if(suc_ptr->Ordering >= key) break;
      pre_ptr = suc_ptr;
    }
    return(pre_ptr);
  }
}
```

The method `Find(int key)` ensures a search of the place in the list which corresponds to the value key. Sorted lists which contain objects of the class `OrderedNode` are intended to sort (according to priorities) objects used as task descriptors in the list `TaskList`.

Descriptor of each task is an object of the class `Task` which is a direct heir of the class `OrderedNode` (and therefore, an indirect heir of the class `ListNode`):

```
class Task:OrderedNode {
   int Period;
   int C_size;
   int R_time;
}
```

In addition to attributes of the class `OrderedNode` the task descriptor has attributes `Period` and C_size to represent the respective task parameters. The attribute R_time is used to represent the result of modeling: the maximal task response time.

Task descriptors are generated and stored in the list `TaskList` (sorted on priorities) during execution of the procedure `TaskBuild()`, which ensures construction of a task description either in a dialog with the user, or by a special generating program.

Task descriptors are linked into a chained list `TaskList` via the attribute `Next`, inherited from the class `ListNode`. Sorting task descriptors on their priorities is performed via the attribute `Ordering`, inherited from the class `OrderedNode`.

B. Dynamically Modified Lists

The task list `TaskList` is formed during preparation to modeling and remains unchanged during the modeling session. Lists `ReadyList` and `EventList` are formed at initialization of the modeling session and are modified during the session.

The list `ReadyList` of task descriptors contains objects of the class `Job`:

```
class Job:OrderedNode {
   int Start_time;
   int Rest_C_size;
   Task *Job_class;
}
```

Objects contained in the list `ReadyList` model jobs which are active at the current moment of the job modeling time – jobs which compete for the processor resource. The attribute `Start_time` of an object of the class `Job` specifies the moment of the modeling time when this object was generated. The attribute `Rest_C_size` specifies the portion of the processor time which has not yet been used by the job. The attribute `Job_class` links the job descriptor with the respective task descriptor.

Job descriptors are linked in a chained list via the attribute `Next`, inherited from the class `ListNode` and are ordered in this list with respect to the value of the attribute `Ordering`, inherited from the class `OrderedNode`. Special place is occupied by the job descriptor placed in the very beginning of the list `ReadyList` (the descriptor which heads the list `ReadyList`). This descriptor is called the descriptor of the current job.

The list `EventList` contains descriptors of system events – objects of the class `Event`:

```
class Event:OrderedNode {
  int Event_type;
  Task *Task_ptr;
  Job *Job_ptr;
}
```

The attribute `Event_type` may have one of two values – zero (if the event consists in activation of a task pointed to by the attribute `Task_ptr`) or one (if the event consists in terminating the job pointed to by the attribute `Job_ptr`).

The attribute `Ordering`, inherited from the class `OrderedNode` specifies the modeling time moment which corresponds to the given event (descriptors of system events are sorted with respect to this attribute in the list `EventList`). Binding descriptors of system events into a chained list is performed via the system attribute `Next`, inherited from the class `ListNode`.

C. Initializing the Modeling Process

Globally accessible objects `TaskList`, `ReadyList`, and `EventList` of the class `ListNode` are generated statically with the value `NULL` of the attribute `Next`. Besides that, global variable are generated:

```
int cur_time = 0;
context_switches = 0;
```

The variable `cur_time` is used to track the flow of the modeling time. Its maximal value is specified by the constant `TIME_LIMIT`. The variable `context_switches` is used to count the number of context switches.

Running the procedure `TaskBuild()` results in construction of the task list `TaskList`, the global variable `task_number` taking the respective value.

For each task an object of the class `Event` is generated with the attribute values:

```
Ordering = 0;
Event_type = 0;
Job_ptr = NULL;
```

The attribute `Task_ptr` is tuned to point to the descriptor of the respective task; attributes `Next` bind the generated objects of the class `Event` in a chained list in an arbitrary order; the list starts with the globally accessible object `EventList`.

With this initialization of the modeling process terminates, and a step-by-step processing of system event descriptors from the list `EventList` starts.

D. Performing a Step of Modeling

At each modeling step a descriptor of the system event allocated in the beginning of the list `EventList` (the list head descriptor) is processed. At this processing first specific actions of the current modeling step are performed: depending on the value of the attribute `EventList` these specific actions consist either in performing task activation, or in performing job termination.

1) Task Activation: If `Event_type = 0` for the head system event descriptor (the descriptor prescribes task activation) then specific actions consist in generation of the next job of the type `EventList.Next->Task_ptr`. To do this, a new job description is generated with the attribute values:

```
Start_time = cur_time;
Job_class = EventList.Next->Task_ptr;
```

```
Ordering =
   EventList.Next->Job_class->Ordering;
Rest_C_size =
   EventList.Next->Job_class->C_size;
```

The descriptor of the generated job is inserted in the list `ReadyList` of active jobs. Position of the job description in the `ReadyList` corresponds to the job priority. If jobs with equal values of the attribute `Ordering` are placed in the list of active jobs, then jobs with the lower value of the attribute `Start_time` are considered of higher priority.

A new object of the type `Event` is inserted into the list `EventList` with the following attribute values:

```
Ordering = cur_time +
    ReadyList.Next->Task_ptr->Period;
Event_type = 0;
Job_ptr = NULL;
Task_ptr = ReadyList.Next->Task_ptr;
```

The place of the newly generated system event descriptor in the list `EventList` is determined by the value of its attribute `Ordering`.

If the current job changed while task activation (the value of the attribute `ReadyList.Next` has changed) then the value of the global variable `context_switches` which tracks the number of context switches is incremented by 1.

2) Job Termination: If `Event_type=1` for the head job description (i.e., the system event consists in terminating the current job), then specific actions consist in possible modification of the response time for the respective task:

```
int job_resp = cur_time -
  EventList.Next->Job_ptr->Start_time;
if(ReadyList.Next->Task_ptr->Period <
                                  job_resp)
  ReadyList.Next->Task_ptr->Period =
                                  job_resp;
```

The value `ontext_switches` is incremented by 1.

E. Mandatory Actions of a Modeling Step

Upon completion of specific actions the following mandatory actions are performed for each step of operation modeling.

1) The assumed duration of the next step of the modeling process is calculated:

```
int step_length =
 EventList.Next->Ordering - cur_time;
```

2) If the list `ReadyList` of active jobs is not empty (`ReadyList.Next` is not equal to `NULL`) and the value of the attribute `Rest_C_size` does not exceed `step_length`, then the value of the variable `step_length` is decreased to the value `Rest_C_size` of the current job:

```
step_length = ReadyList.Next->Rest_C_size;
```

A new element – a descriptor of the system event consisting in termination of the current job – is inserted in the list of system events with the following attributes:

```
Ordering = cur_time + step_length;
Event_type = 1;
Job_ptr = ReadyList.Next;
```

3) For the current job the value `Rest_C_size` of not used amount of the processor time by the job is decreased:

```
ReadyList.Next->Rest_C_size -= step_length;
```

4) The value of the current moment of the modeling time is modified:

```
cur_time = EventList.Next->Ordering;
```

5) The head descriptor of the system event list is deleted from `EventList`:

```
EventList.Pop();
```

The new head descriptor in the list `EventList` of system events becomes the descriptor which followed the deleted one.

6) Conditions for terminating the modeling session are checked. The session terminates either if the counter `cur_time` exceeds the maximal acceptable value (abnormal situation), or if there are no more active jobs (i.e., if the list `ReadyList` becomes empty – normal termination of the session).

If conditions for terminating the modeling session are not satisfied, then upon completion of the listed mandatory actions transition to the next modeling step is performed.

F. Exit from the Modeling Session

By the moment of a normal exit from the modeling session the attribute `R_time` of each object of the type `Task` contains the response time of the respective task. The global variable `context_switches` contains the number of context switches within the modeling time interval of the length `cur_time`.

V. Comparative Efficiency of the EDF and RM Scheduling Modes

The presented approach was tried to estimate the dependency hardness/density for the EDF and RM scheduling modes (the approach to constructing an imitational model described in section IV was extended with a possibility to model modes of the class S_1). Modeling was performed for a system of ten independent tasks with parameters enlisted in Table I.

TABLE I. TASK PARAMETERS OF THE MODELED APPLICATION

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}
T_i	100	107	114	123	132	141	151	165	174	187
C_i	7.2	7.7	8.3	8.9	9.5	10.2	10.9	11.6	12.4	13.2

Modeling results are presenting in Fig.2. The axe X represents reverse values to hardness, the axe Y represents density values. Performed experiments with the values of $1/H$ not exceeding 0.46, the density values for the scheduling modes RM and EDF coincide. For the values of $1/H$ higher than 0.46 the EDF scheduling mode turns out to be more efficient than RM.

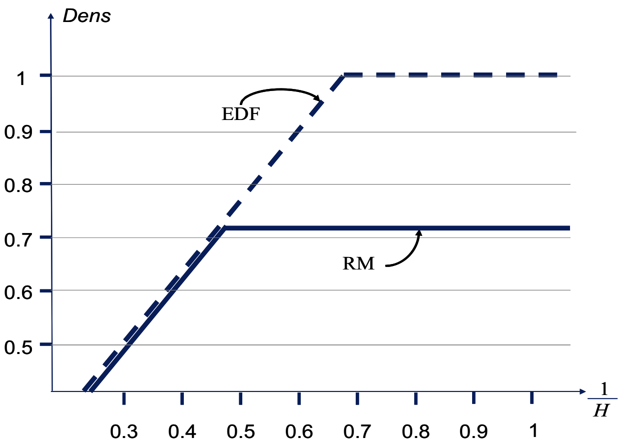

Fig. 2. Hardness/density dependences for EDF and RM scheduling modes

Experiments with the RM scheduling mode were performed for values of $1/H$ exceeding 1, when intersections of existence intervals of jobs of the same type occur. It was found that for values of $1/H$ not exceeding 1.3, the application density for RM is preserved at the level of 0.72, while for $1/H$ exceeding 3.5 the application density reaches 1.

VI. CONCLUSION

Using the presented simulation technique allows to perform comparative estimates for various combinations of scheduling modes and protocols of access to shares informational resources from the side of tasks which compose software real-time applications.

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

REFERENCES

[1] K.Ya. Davidenko, *Software Engineering for Automatic Control Systems of Technological Processes*. Moscow: Energoatomizdat, – 1985. (In Russian)

[2] J.W.S. Liu, *Real-Time Systems.* NJ: Prentice Hall, – 2000.

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

[4] 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.

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

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

[7] B. Andersson, S. Baruah J. Jonsson “Static-priority scheduling on multiprocessors”, *in Proc. of 22nd IEEE Real-Time Systems Symposium*, London, 2001, pp. 193-202.

[8] V.V. Nikiforov. “Feasibility of Real-Time Applications on Multi-Core Processors”, *SPIIRAS Proceedings*, Issue 8, – St.Petersburg: Nauka, 2009, pp. 255-284. (In Russian)

[9] L. Sha, R. Rajkumar, J.P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*. vol. 39(9), 1990, pp. 1175-1185.

[10] 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)

[11] V.V. Nikiforov. “Protocol for Preventing of Task Blocking in Real-Time Systems”, *Priborostroenie*, n.12, 2014, pp.19-29. (In Russian)