УДК 004.04

Application Density and Feasibility Checking in Real-Time Systems

Sergey Baranov (SPIIRAS, ITMO University),

Victor Nikiforov (SPIIRAS)

An approach to analyze the compatibility of real-time multi-task applications with various combinations of scheduling modes and protocols of access to shared resources when run on multi-core platforms is described. It is based on the recently introduced notion of application density derived from estimation of application feasibility for various values of the processor performance. The software architecture of a relatively simple simulation tool for estimation of the task response time (and therefore, application feasibility) is described, which provides more exact data compared to the known analytical methods when they are applicable. Results of running this tool on a number of benchmarks, including balanced Liu-Layland configurations, are presented along with their analysis and interpretation. The suggested approach allows to indentify an optimal combination of the scheduling mode and access protocol for the given application structure.

Keywords: *simulation, real-time, application density, application feasibility.*

1. Introduction

Software applications for real-time systems (RTS) are usually built as collections of prioritized tasks treated as sequential programs closed w.r.t. to control flow. During application runs the tasks may share common system resources: executive ones (processors and processor cores of multi-core processors) and informational ones – global data arrays, interface registers of peripheral devices, elements of human-machine interface, etc. Access to executive resources is governed by scheduling mode in use, while access to shared informational resources is controlled by access protocols.

Each task τ_i of an RTS application is characterized by its period T_i, its deadline D_i, and its weight W_i. The period and the deadline are specified in absolute time units (e.g., milliseconds) and are usually considered as "external constraints" for the application, while the weight is an "internal constraint". It determines the "amount of computational work" (and thus, the quantity of the executive resource) which is needed for the task to accomplish its function and is specified in the number of standard machine operations for the given task realization. Given a particular processor performance P, the task weight W_i may be converted into time units: $C_i = W_i/P$.

Application behavior is composed by consecutive task instances $\tau_i^{(j)}$ (called jobs) running in parallel and iteratively activated according to their periods T_i. Due to competition among running jobs for platform resources, execution of any job $\tau_i^{(j)}$ may be suspended and then resumed after some period of time. The task response time R_i is the maximal time interval between respective jobs $\tau_i^{(j)}$ starts and terminations which is called "job existence interval".

A key requirement to an RTS software application is called "application feasibility" and is formulated as: $\forall i \; Di \geq R_i$ in all acceptable scenarios of application communication with its environment at the run time. Application feasibility may be checked either through an analytical estimation of the response time for each application task, or through simulation of the application run with an appropriate software tool.

For an RTS designed to run on a single-core processor exact analytical estimations of its feasibility exist since early 70-ies [1-3]; however, for multi-core processors exact analytical estimates are still unknown, while suggested rough methods provide pessimistic results if compared with real RTS behavior[4,5]. Therefore, a software simulation tool is needed to obtain a more exact estimation of application feasibility for RTS on multi-core platforms under various combinations of scheduling modes and access protocols than the known analytical methods. The paper describes the architecture of such software tool [6] and new results of a series of experiments with it.

2. Application Density

A derivative parameter characterizing the task τ_i behavior for the given performance P of processor cores is its utility $U_i = C_i/T_i$**,** which is the portion of the task period used for computing; thus the overall utility U of the application is: $U = (\Sigma_{1< i <n} U_i)/k$, k being the number of processor cores in the given platform, each of the same performance P. The value $1–U$ specifies the portion of the processor time not used by the application (the processor is either idle or is loaded with calculations unrelated to the RTS processing). An increase of the processor performance leads to an increase of the processor idle time for the given RTS software application.

Let's consider an auxiliary task characteristic called hardness: $H_i = Ti/D_i$. If $H_i \geq 1$ then existence intervals of any two consecutive instances of the same task τ_i – jobs $\tau_i^{(j)}$ and $\tau_i^{(j+1)}$ do not intersect. The reverse condition $H_i < 1$ means that existence intervals of such jobs may intersect. If all application tasks are of the same hardness H, then H is called the application hardness. Sometimes the reverse value $H–1$ turned out to be more convenient for consideration.

In [7] the notion of application density as the maximal value of the overall utility of an application was introduced: $Dens = max_P(U)$ for all values P of processor performance which the application is feasible with. Obviously, any correct application is feasible with $P = \infty$ and is not

feasible with $P = 0$. It is also obvious, that if an application is feasible with the performance values P_1 and P_2, where $P_1 < P_2$, then it is feasible for any $P \in [P_1,P_2]$. Therefore, the value *Dens* exists which corresponds to the minimal processor performance P_0 with which the application is still feasible: for all $P < P_0$ the application is not feasible and for any $P \geq P_0$ it is feasible.

The above consideration prompts an efficient algorithm of "catching the lion in desert" for calculating the value *Dens*. Starting with the interval $[a,b]$ of performance values, where $a = 0$ and $b=P_{max}$, P_{max} being large enough for the application to be feasible, application behavior is simulated with the processor performance $P = (b–a)/2$. The next interval will be either $[a,P]$ if this simulation run confirms application feasibility, or $[P,b]$ otherwise. The loop terminates at the interval $[P–\varepsilon,P+\varepsilon]$, P being the resulting performance value with an accuracy of ε. Application density 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 and may be used as a criterion of efficiency of the selected combination.

Current studies were focused on finding dependencies between application hardness and density for various combinations of scheduling modes and access protocols. For that purpose two dissimilar prototypes of the feasibility checker were developed: one in C++/C#, the other in Forth [8, 9], along with a number of application benchmarks. Simulation results obtained with these dissimilar tools differ for less than 0.1 per cent, which is a strong evidence in their trustworthiness.

3. Feasibility Checker

The overall workflow of the simulator for checking application feasibility is presented in Fig. 1. Simulator initialization consists in selecting the desired combination of the scheduling mode and inheritance mode of the access protocol, setting the respective simulator constraints, reading the task description file, and forming the respective resource and task objects. Then the initial list of system events EventList is formed which consists in activation of the all tasks at the moments of system time defined by their phase shifts. Counts for their maximal response times are set to zero and all resources are set to be unlocked.

In the major simulator loop the first group of time-sake events in the ordered EventList is considered, the simulator system time is set to this time moment and all events from this first group are processed one-by-one according to the event type.

1. In case of activating a task, a new job is created from this task and is added to JobList with its priority and planned starting time equal to the current system time; also a new event is added to EventList – to activate the next instance of this task at the moment of time not less than the current time plus the task period T_i.

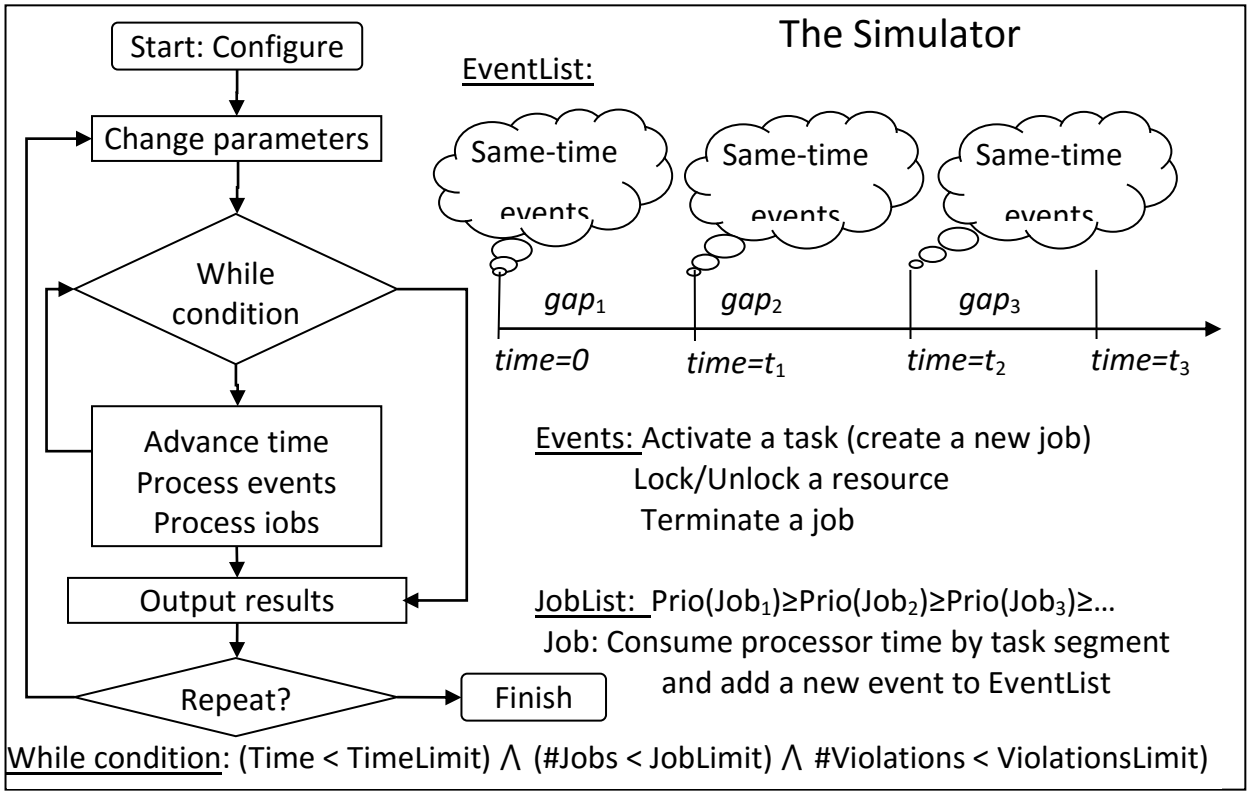

Fig. 1. The overall workflow of the feasibility checker

2. When terminating a job, the response time for the respective task is updated by the maximum of its current value and the existence time of this job. If the job existence time exceeds the task deadline D_i, then a violation of the task feasibility is registered. The considered job is deleted from the JobList.

3. In case of locking/unlocking a resource, if an already locked resource is being locked, the job is moved from JobList to the ordered list of jobs waiting for unlocking of this resource; otherwise the resource becomes locked by this job. When unlocking a resource, if the list of jobs waiting for it is not empty, then the first job form this list is moved back to JobList according to its priority and the resource becomes locked by this job; otherwise, the resource becomes unlocked.

Upon completion of the event processing, the considered event is deleted from EventList. After processing all time-sake events, JobList is considered (it may change as a result of event processing). If it is non-empty, its first element is selected and the time it consumed by this job is updated accordingly, probably generating a new event to terminated this job. If JobList is empty, this means the processor is registered to stay idle for the gap till the next time-sake event group. After that processing the major loop is reiterated. The loop terminates upon exhausting the time limit of the simulation session or when a specified number of created jobs is reached.

The results of simulation – maximum task response time, number of deadline violations, the application density, and other statistics data are displayed. A simulation log may also be displayed.

When any system event is processed, the respective time and other accompanying data are printed-out. All these data may be easily copied into MS Excel for a graphical representation of the obtained results and execution log.

4. Running Experiments with the Simulator

To run experiments, particular configurations of software applications to be analyzed are submitted to the simulator. As a rule, valuable configurations of real-time applications actually used in the practice of industrial programming are confidential proprietary; therefore, experiments were run on configurations of mainly methodological interest, the ones with uniform distribution of utility load and with logarithmical distribution of task periods being among them. Let's consider the known Liu-Layland configuration of 10 tasks which is both utility-uniform and with logarithmical periods of its tasks:

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

The left diagram in Fig. 2 demonstrates how the application density *Dens* depends on the hardness $H=T_i/D_i$ (actually H^{-1}) of its tasks for two classical scheduling modes: Rate Monotonic (RM) and Earliest Deadline First (EDF) on a single core processor. The task hardness H is supposed to be the same for all 10 tasks, and such configuration is called "balanced".

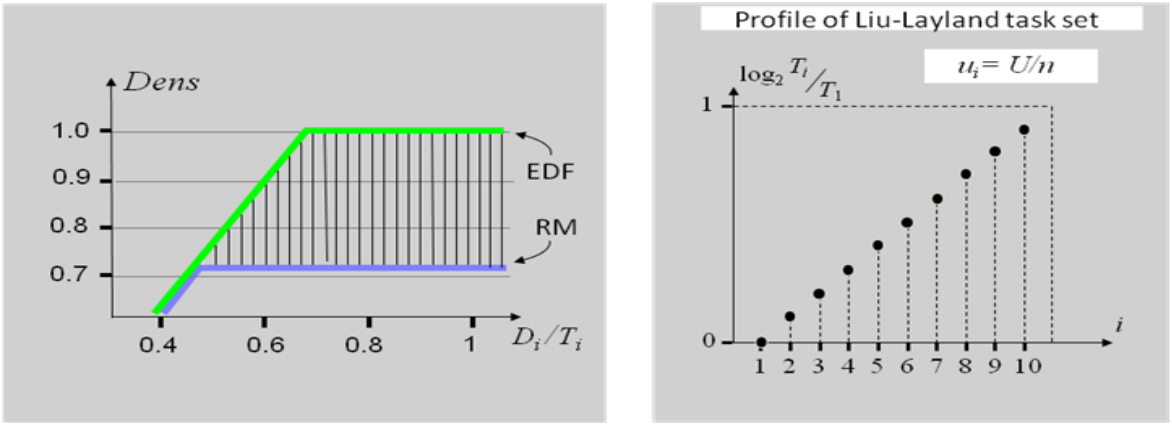

Fig. 2. Liu-Layland balanced configuration of 10 tasks on a single core platform

As one can see, the EDF and RM scheduling modes behave the same when deadline is much less than the period, but then EDF ensures the maximal density of 1 while RM cannot raise over 0.72. The right-side diagram demonstrates the logarithmic dependency of task periods which turns into a pure linear profile.

Results of another experiment with 5 independent tasks with balanced Liu-Layland configuration are presented in Fig. 3.

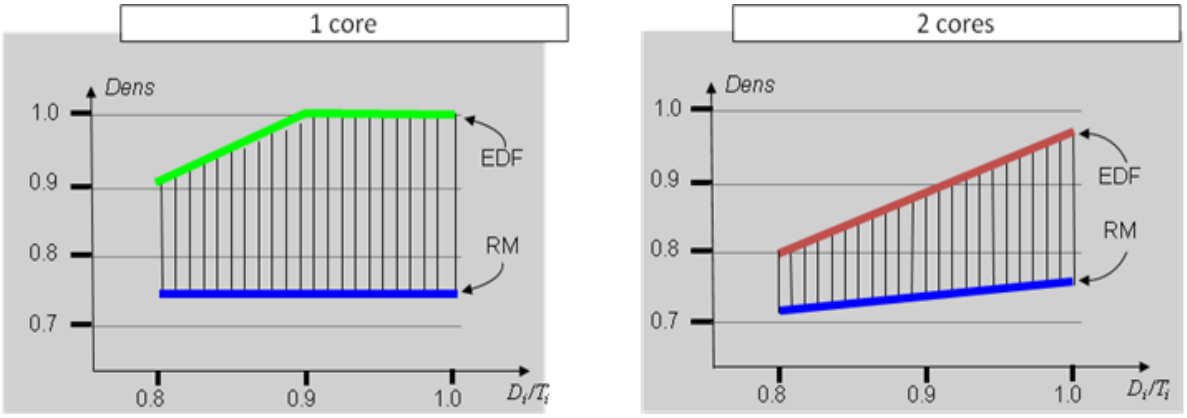

Fig. 3. Liu-Layland balanced configuration of 5 independent tasks

Here two analogous charts for 5 independent tasks with a balanced Liu-Layland configuration are compared when run on a single core and on a two-core platform:

Task	τ_1	τ_2	τ_3	τ_4	τ_5
T_i	1000	1150	1330	1520	1750
C_i	1000	1150	1330	1520	1750

One can notice that the utility load for each task is U_i=1 and the ratio U_i/U=0.2. With the optimal processor performance P using the EDF scheduling mode with application hardness approaching 1 ensures 100% processor load (*Dens*=1) on a single core processor, while on a two-core processor not only RM cannot ensure this efficiency, but EDF fails to reach it as well.

In Fig. 4 the same configuration as in Fig. 3 is studied, but this time the tasks share 5 common resources using mutexes to guard the respective critical intervals and the application runs on a single core platform.

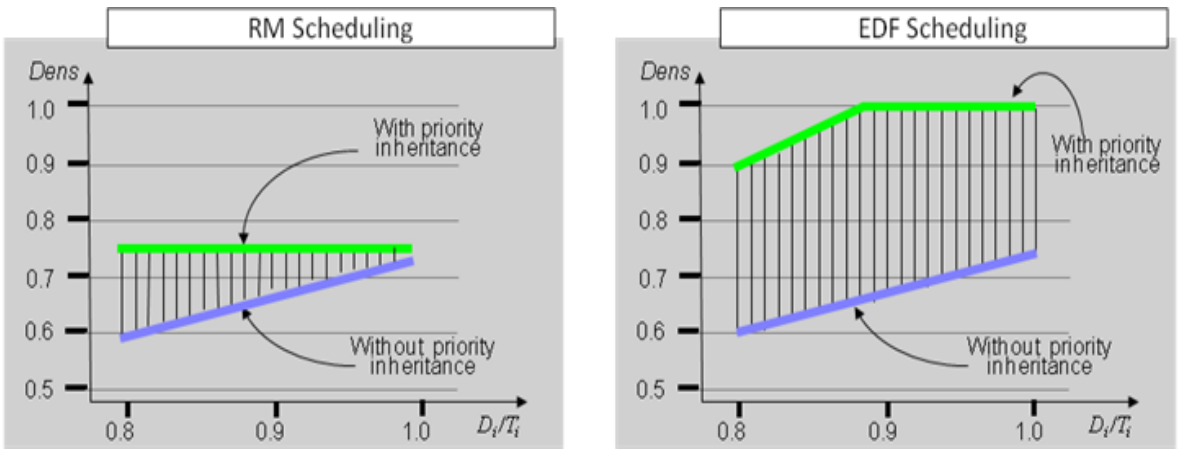

Fig. 4. Liu-Layland balanced configuration of 5 dependent tasks, single core

The charts represent the density/hardness dependency under RM and EDF scheduling modes with and without priority inheritance when tasks perform access to shared resources. Two unexpected facts are worth mentioning in this case:

- when the scheduling mode ignores priority inheritance, the density values for RM and EDF are exactly the same;
- when priority inheritance in any form is added, then density is the same as for independent tasks for both RM and EDF scheduling modes.

When the same application with a priority inheritance mechanism runs on a two-core processor, the advantages of the EDF scheduling mode over the RM one diminish in comparison with a single core platform, especially when the application hardness is close to 1, as one can see in Fig. 5.

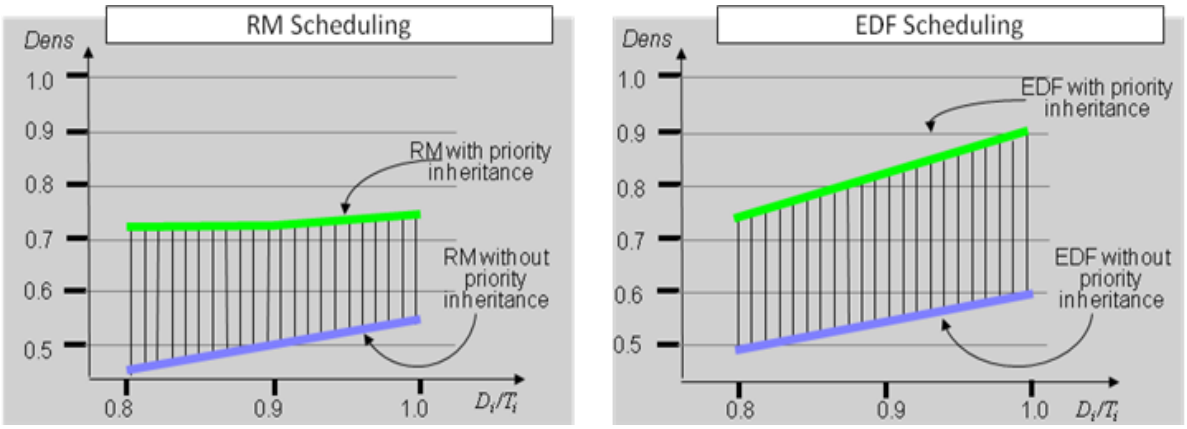

Fig. 5. Liu-Layland balanced configuration of 5 dependent tasks, 2 cores

In the next Fig. 6 one can see how for the same Liu-Layland balanced configuration of 10 independent tasks the difference between RM and EDF diminishes when the number of cores in the processor increases.

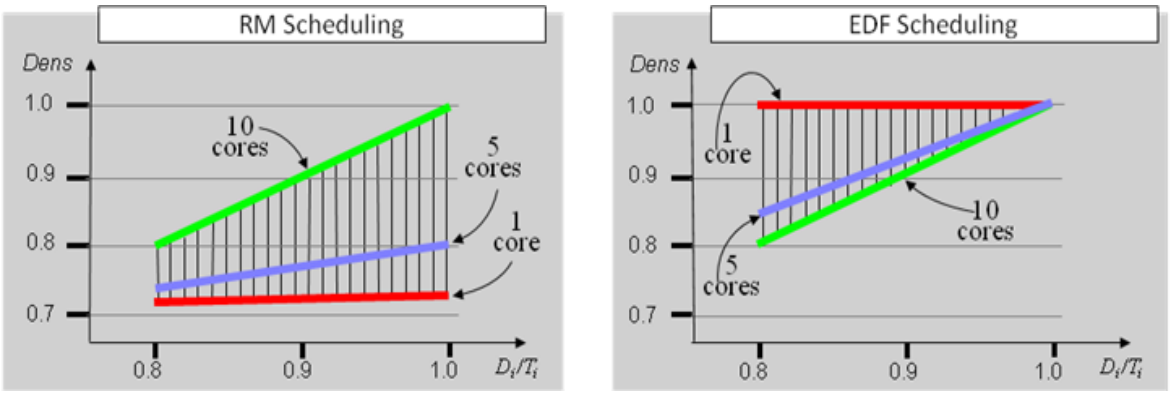

Fig. 6. Liu-Layland balanced configuration of 10 independent tasks, many cores

In Fig. 7 two 4-task configurations with non-uniform utility load are studied. Tasks in the left-hand part are all independent. In the right-hand part tasks 1 and 3 share resource 1, while tasks 3 and 4 share resource 2 and are therefore dependent.

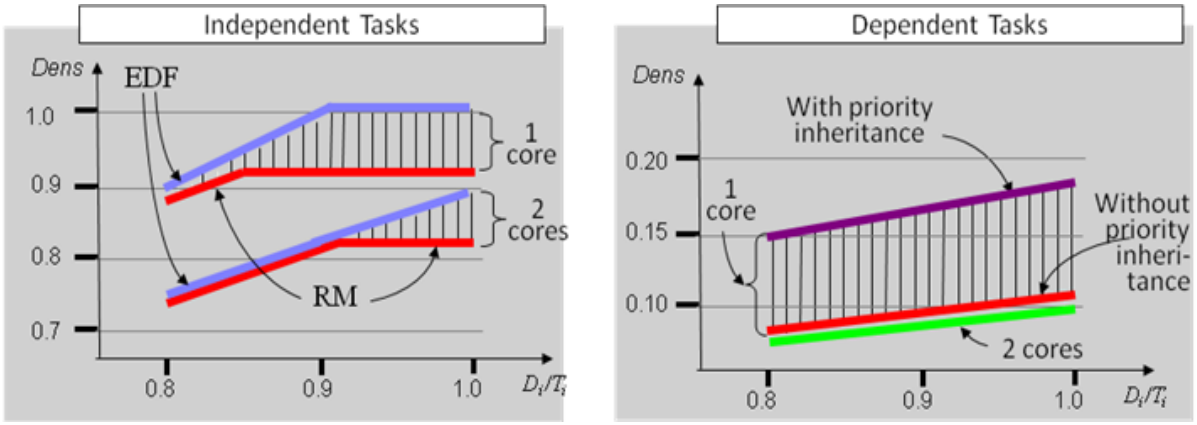

Fig. 7. RM vs. EDF for dependent and independent 4 tasks

For independent task with hardness close to 1, the EDF scheduling mode is much more efficient than RM on a single core platform as well as on a two core one. When deadline diminishes, this difference between EDF and RM disappears. Dependent tasks with shared resources RM scheduling, both with and without priority inheritance, demonstrate the same application density on a two core platform. However, on a single core platform priority inheritance provides substantial gain. The RM and EDF scheduling modes are optimal in their classes of applications when the latter run on a single core platform.

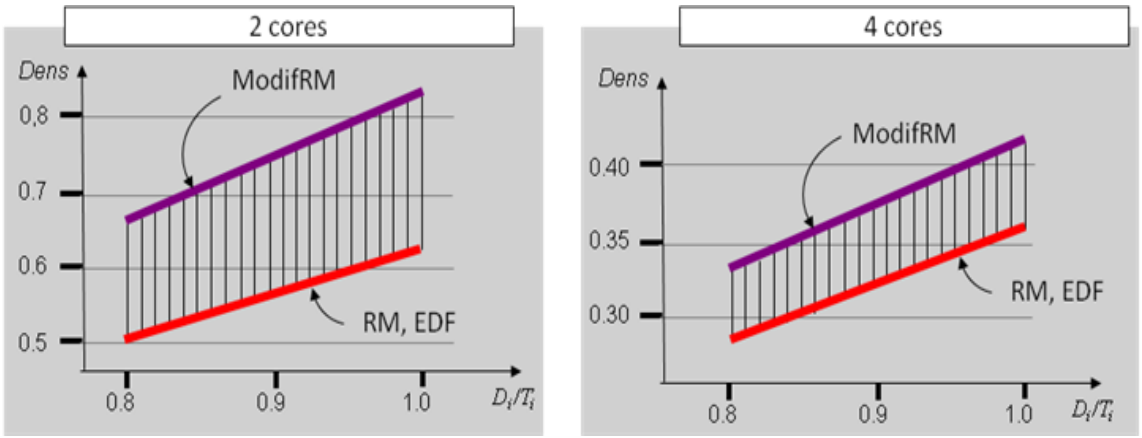

Fig. 8. RM vs. EDF for 5 independent tasks with shifted utility load

In case of multi-core processors, they are not optimal which is pretty well demonstrated by applications with shifted load; i.e., with substantially non-uniform distribution of the total load within the tasks. In the 5 task example in Fig. 8 60% of the total utility load is on task 1: it may be characterized as a "heavy" one, while tasks 2-5 may be called "light". In such cases it's reasonable to use a modified RM scheduling mode: the heavy task is assigned the highest priority, while scheduling among light tasks is governed in the regular RM way. Both charts in Fig. 8 demonstrate the advantages of this ModifRM (Modified Rate Monotonic) scheduling mode on a 2 and 3 core platform, while using RM and EDF scheduling modes is equally inefficient in this case.

5. Conclusions

Using the described methods of software simulation for estimating feasibility of real-time multi-task applications on multicore platforms allows to obtain objective data for selecting an optimal combination of scheduling mode and access protocols. It's noteworthy that analytical methods for such estimates exist for single core platforms only; if used for a multi-core platform they provide too pessimistic results. Therefore, simulation becomes an important tool for searching optimal application structures and platforms for real-time multi-task applications. The developed simulator may be used for feasibility checking of such applications.

Future research will be focused on extending the nomenclature of scheduling modes and access protocols and the profile of applications under study.

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

References

1. Liu, C., Layland, J. Scheduling Algorithms for Multiprocessing in a Hard Real-Time Environment. Journal of the ACM, 20(1), 46-61 (1973)
2. Andersson, B., Baruah, S., Jonsson, J. Static-Priority Scheduling on Multiprocessors. Proc. 22nd IEEE Real-Time Systems Symposium, 193-202 (2001)
3. Laplante, P.A. Real-Time Systems Design and Analysis. John Wiley & Sons, Inc., (2004)
4. Baker, T. Multiprocessors EDF and Deadline Monotonic Schedulability Analysis. Proc. 24th IEEE Real-Time Systems Symposium, 120–129 (2003)
5. Andersson, B. Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%. Proc. 12th International Conference on Principles of Distributed Systems. Luxor, Egypt, 73-88 (2008)
6. Baranov, S.N. Real-Time Multi-Task Simulation in Forth. Proc. 18th Conf. FRUCT, St.Petersburg, Russia, 17-22 (2016)
7. Baranov, S.N., Nikiforov, V.V. Density of Multi-Task Real-Time Applications. Proc. 17th Conf. FRUCT, Yaroslavl, Russia, 9-15 (2015)
8. Baranov, S.N. The Program RTMT for Simulation of Multi-Task Application Run. Certificate of official registration of a computer program No.2016613095, 16 March 2016 (RU), http://www1.fips.ru/wps/portal/Registers/ (in Russian)
9. Forth 200x, http://www.forth200x.org/forth200x.html (2016)