Can Fixed Priority Scheduling Work in Practice?

Raimondas Lencevicius, Alexander Ran
Nokia Research Center
5 Wayside Road, Burlington, MA 01803, USA
Raimondas.Lencevicius@nokia.com Alexander.Ran@nokia.com

1. Introduction and application domain

Last year our research group was requested to study the runtime architecture of mobile phone software in order to understand whether some performance aspects of the phone could be improved. We expected to improve the architecture along several dimensions:

- systematic derivation of task priorities for more effective scheduling
- improve partition into tasks to make the architecture more analyzable in terms of performance

A major part of the runtime architecture improvement work was concerned with the scheduling of the mobile device. We were aware of the large body of research work in fixed-priority system scheduling area [2]-[4]. However, when we tried to apply this research in our real-world situation, we discovered a number of problems described below.

In this paper we are primarily concerned with embedded real-time systems such as mobile phones or personal communication devices. Today personal communication devices are more than voice call terminals. Mobile phones serve as platforms for a variety of mobile applications including text and picture messaging as well as personal information management, including data synchronization with remote servers and desktop computers. Mobile phones host a range of communication-centered applications most of which have real-time constraints. During a voice call speech data must be processed in a timely fashion to avoid jitter. During GPRS session lower layer packets arrive every 10 ms. These are just few examples of system requirements that lead to tight software performance constraints.

To improve the performance of the mobile phone software, we concentrated on the runtime software architecture—a partition of all software functions into concurrent units and a scheduling policy that deliver the best possible service to the user with available resources. The units of concurrency in most products are operating system tasks. Thus partition of software into tasks and allocation of functionality to tasks in the form of objects or functions are the most important decisions in the design of the runtime architecture [5]. In this paper, we focus on the scheduling policy and its parameters. Some problems encountered in our project can be solved by a different task design. However, task redesign often is not feasible in industrial setting since it takes long time and has high costs.

2. Featuresets

Some of the mobile phone applications must be executed concurrently and thus have to be schedulable. However, if all applications were executed at the same time, the system would not be schedulable. Thus an essential concept in the mobile device runtime architecture is a *featureset*. A featureset is a set of concurrently available features. Specification of useful featuresets for a given system is an architecturally significant requirement. All featuresets have to be schedulable.

System designers identify featuresets by first composing a collection of all important use cases. Then designers identify subsets of this collection containing use cases that may overlap in time. Such subsets contain concurrently available features—featuresets. Each use case contains one or more execution scenarios. Each scenario is defined by one or more sequences of events that occur in the scenario. One or more objects handle each event. Each object is allocated to one of the concurrent tasks. Therefore, each featureset has a mapping to a set of tasks (Figure 1).

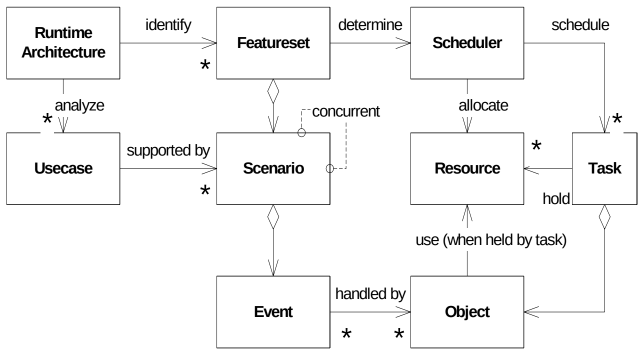


Figure 1. Runtime Architecture Meta Model

Tasks that do not belong to the same featureset do not have to be collectively schedulable. If tasks that belong to the same featureset are not schedulable, some architectural decisions regarding allocation of objects to tasks or resource scheduling policies must be revised.

Since each featureset maps to a set of tasks and no other tasks can run in this featureset, it seems natural to analyze the system and assign priorities separately for each featureset. Such priority assignment in fixed priority scheduling is called a *mode* [4]. Although featuresets can be implemented as modes and mapped one-to-one to their modes, featuresets differ from modes in several aspects. Featuresets are a practical way of finding a minimal set of events or tasks that belong to a mode. This allows to include the smallest number of tasks and therefore the smallest number of real-time constraints into a mode. By reducing a number or tasks and constraints in a mode, the mode becomes more evolvable—additional tasks and constraints can be added easier in the future. This mode minimization is not discussed in the scheduling literature.

3. Task model for scheduling

Whether we are scheduling a mode or a whole system, we need to know certain properties of tasks and resources. To apply fixed priority scheduling [4], we had to produce a list of concurrent tasks and for each task to determine its priority, period, execution time, and deadline or constraint on response time. We had full access to the design documentation, the source code, and extensive execution traces produced from multiple use cases. Unfortunately, the design documentation often does not contain sufficient information regarding the task structure. This is mainly because the initial task structure is often changed at later stages of product integration and fine-tuning when the design documentation is not actively maintained anymore.

Although the source code contains task creation instructions and task priorities, it is not easy to determine from the source code which objects and functions are allocated to which task and how the objects in a given task interact with objects in other tasks. Therefore we mainly focused our efforts on the analysis of execution traces, which proved to be the easiest way to obtain the needed information. In our study we mainly relied on the traces produced by the operating system scheduler. A trace is produced every time a task is scheduled to run.

Task period is defined as the interarrival interval of event(s) causing task's execution. Since such event traces were not available in our case, we approximate the task period by the time interval between task's successive invocations. The execution time of a task is the time between an invocation of a task and the invocation of the next task. We accounted for preemptions by recognizing them in the traces using a trace at the end of each task's response. Unfortunately, we could not identify ways to discover deadlines from analysis of predefined traces. Some of the deadlines are due to protocols of interaction with other systems and cannot be determined from traces in principle. In this study, we had interviews with software designers to elicit the deadline information.

Once we have extracted from the traces the observations of each task's parameters such as invocations and execution times, we immediately noticed a number of issues that are not handled by the fixed priority scheduling methods.

In simple scheduling approaches, each task is characterized by a single period, deadline, and execution time. Of course, it is commonly understood that in many cases the tasks are not perfectly periodic. A typical recommendation from scheduling literature is to consider the worst case: the shortest period and the longest execution time. Unfortunately, in all the use cases we have analyzed such a recommendation is not useful because none of these use cases would be schedulable even though we knew that the system performed fine in practice. Table 1 shows the extracted data of a communication task for the use case of file transfer between a laptop and a server over USB and GPRS. The worst-case period of the task takes 0.4 time units, while the worst-case execution time takes 20 time units.

Table 1. Communication task execution times and periods in file transfer over USB and GPRS

Worst case (longest) execution time	20	Worst case (shortest) period	0.4
Average execution time	1.6	Average period	20.5
Largest cluster execution time	1.3	Largest cluster period	1.6
Execution time clusters		Period clusters	
Time	Size	Time	Size
1.3	4352	1.6	1479
3.1	906	4.3	657
		10.6	562
		20.6	1024
		41.9	1221
		100.9	280

4. Multivariate tasks

The assumption that tasks have a single period and execution time does not hold in our data. Although it appears that the task periods and execution times are not single peak statistical functions, it may be possible to identify a small set of typical ("peak") periods and execution times of a task. We called this a multivariate task hypothesis.

To verify this hypothesis we developed a data-clustering algorithm that identifies the presence of multiple clusters ("peaks") in observations of task periods and execution times. We have applied a modified K-means algorithm with criteria for establishing new clusters and merging not sufficiently separated clusters. The results of cluster analysis confirmed the multivariate task hypothesis—most of the tasks had several characteristic periods and execution times. For example, a communication task (Table 1) has two execution time clusters and six period clusters in the given use case. Most of our real-time tasks could be characterized by a small set of parameter vectors. These data also indicated that such statistical measures, as average period and execution time, are not representative of the mobile device's task behavior. Although the average execution time divided by the average period represents the average processor utilization contributed by a task, this load does not represent a reliable schedulability measure [3]. The average may not represent any single cluster. For example, the average execution time of the task in Table 1 does not correspond to either of the two large execution time clusters. This observation is typical for other tasks too.

In many cases it is impossible to perform the schedulability analysis based on just the largest execution-time and period clusters. For example, the largest period and execution-time clusters of the task in Table 1 show that this task would have utilized the processor at $1.3/1.6 \approx 81\%$ utilization. Since the system has other tasks as well, such a high utilization by one task predicts that the system is unschedulable. However, as we know from experimental data, this is not the case. So the schedulability analysis needs to take into account the relationship between different execution time and period clusters. With such understanding it may be possible to identify pairs of compatible period and execution time values.

What causes multiple period and execution time clusters in a single task? If tasks were designed with runtime architecture and schedulability in mind, a single task should be assigned only objects handling a single event stream [1][5]. However, in real systems, tasks often contain objects handling more than one stream of events. These event streams may be independent and have different periods and execution times. In such case, the observed periods and execution times of the task handling the events will no longer have a single peak, instead they will be a complex superposition of periods and execution times of different event streams and may not display any characteristic period at all. In depth understanding of tasks may allow to separate multiple event streams and determine periods and execution times characterizing each of them. We found that this is difficult to achieve from traces and usually requires domain expert help.

5. Activities

In our system a response to an event may involve multiple tasks of different priority. The invocation of these tasks is deterministic in response to an event and thus has to be seen as a single activity. Furthermore, some responses may have multiple deadlines for different tasks that constitute the response. We call multiple task responses to events *activities*. An activity starts with an external event, for example, timer expiration, interrupt, or another similar event. An activity ends when all tasks involved in the activity are finished with their responses.

It is evident from extracted activities that phone tasks are highly dependent on each other. An external event is handled not by just one task, but by a number of communicating tasks. This means that whole activities need to be considered in real-time analysis and scheduling. Although a single priority cannot be assigned to an activity composed from multiple tasks, we present a few alternative ways of dealing with this issue.

First approach is to analyze activities and to assign task priorities for tasks in activities. Klein et al. [3] and Harbour et al. [2] show how to determine the schedulability of systems with activities. However, we did not find any results that show an optimal priority assignment for systems with activities. Harbour et al. [2] suggest a heuristic of assigning priorities to tasks in activities according to a deadline monotonic algorithm. However, this approach is proved optimal only for a limited set of schedules where all tasks in activities have nonascending priorities. On the other hand, Harbour et al. [2] use the activity's canonical form, which is obtained by converting all task priorities to a nondescending form. It is shown that the activity completion time is the same for original and canonical forms. This seems to argue for the use of the nondescending priority assignment for tasks in an activity. However, no proof is given that such assignment is "good".

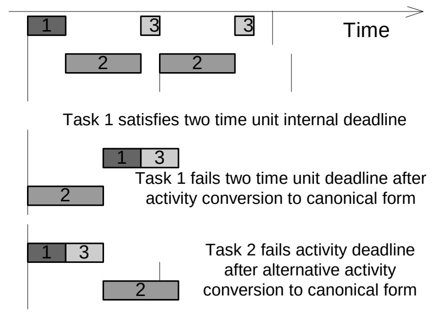
Consider the simplest situation where there are no internal deadlines for the tasks in an activity A and no tasks are shared between activities.

Theorem 1. Conversion of activity A to a canonical form improves schedulability of the system. (Theorem proofs are omitted due to lack of space).

Theorem 1 shows that for every schedule in a system with above constraints there is a better or equal schedule in which all activities are in canonical forms. Such a schedule can be constructed by converting each activity in turn into a canonical form. Therefore the optimal schedule for a system with no intermediate deadlines in activities and no tasks shared between activities is a schedule with all activities in a canonical form.

Unfortunately this result does not hold anymore if the constraints on the system are changed. If tasks in activities have internal deadlines, these deadlines can be broken by the activity conversion to a canonical form. Consider Figure 2. In it C_1 (execution time of task 1) =2, D_1 (deadline of task 1) =2, C_2=4, C_3=2, T_{A1} (activity 1 period) =13, T_{A2}=7. Task 1 satisfies internal deadline of two time units in the original priority assignment, where it has the highest priority of all tasks. However, task 1 fails the deadline when the activity is converted to canonical form and task 1 priority is lowered to the priority of task 3.

Is there an optimal priority assignment in this example such that all activities are in canonical form? No. P_1 (priority of task 1) has to be greater than P_2 for task 1 to satisfy its internal deadline D_1. Which means that $P_3 \geq P_1$ (by canonical form) $> P_2$. But this is the situation at the bottom of the Figure 2, where task 2 fails the activity deadline T_{A2}=7. On the other hand, this task set with a non-canonical form schedule is schedulable as shown in the top of the Figure 2.

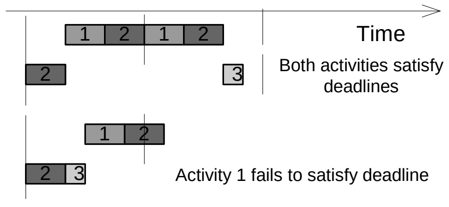


1
3
3
Time
2
2
Task 1 satisfies two time unit internal deadline
1
3
Task 1 fails two time unit deadline after activity conversion to canonical form
2
1
3
Task 2 fails activity deadline after alternative activity conversion to canonical form
2

Figure 2. Conversion of activity with internal deadlines to canonical form

Therefore, if tasks have internal deadlines, the optimal schedule may not have all activities in canonical form.

If tasks are shared between activities, but no tasks have internal deadlines, the usefulness of conversion to a canonical form depends on whether same priority tasks are allowed and how same priority tasks are scheduled. Consider Figure 3. In it C_1=2, C_2=2, C_3=1, T_{A1}=6, T_{A2}=12. Task 2 is shared by both activities. In original priority assignment $P_3<P_1<P_2$. Activity 1 is in canonical form, while activity 2 is not in canonical form. The system is schedulable. If activity 2 is converted to a canonical form, P_2 becomes equal to P_3. However, now activity 1 is not in canonical form: $P_2=P_3<P_1$. Now activity 1 is converted into a canonical form and priorities become $P_1=P_2=P_3$. However, the scheduling of such a system depends totally on the operating system scheduler implementation and in the “bad” case (Figure 3 bottom) activity 1 fails its deadline.



1
2
1
2
Time
2
3
Both activities satisfy deadlines
1
2
2
3
Activity 1 fails to satisfy deadline

Figure 3. Activities with shared tasks

Therefore if activities have shared tasks, the conversion to the canonical form should be used only if scheduling of tasks with the same priority and scheduling of the same task invoked from different activities is well understood. Systems usually are not designed to have tasks with the same priorities because then the internal scheduler implementation determines the processing order of same priority tasks.

Klein et al. [3] suggest assigning shared tasks a priority that is higher than priorities of tasks invoking the shared task. This heuristic avoids the priority inversion between activities, which can occur otherwise. Klein et al. heuristic also avoids the situation when a shared task is invoked by some task that preempted the same shared task. There is no proof that the heuristic always leads to better schedules.

The second approach to activity scheduling is to simplify the system by moving all the functionality performed in an activity into a single task. This allows assigning a single priority to the activity and using scheduling algorithms for the priority assignment. However, such reallocation is not possible if the same task’s actions are needed in different activities. Reallocation is also impossible if tasks in an activity have internal real-time constraints. Finally, reallocation leads to task merging, which, as observed in [2], reduces overall system schedulability.

Theorem 2. Splitting activity into tasks increases system schedulability. Merging tasks decreases system schedulability.

The above theorem assumes zero task-switching overhead. Task splitting increases the number of task switches, which increases the overhead due to task switches. This may become an issue if the task-switching overhead is large.

Harbour et al. [2] proved that any system of two tasks with deadlines equal to periods and system utilization $\leq$ 1 is schedulable by splitting the longer period task into two and assigning appropriate priorities. As far as we know, there is no similar proof for an arbitrary number of tasks. For task sets with deadlines shorter than periods Harbour et al. [2] result does not hold even for two tasks.

It seems that people scheduling a system are left in a quandary: they can stay with activities, but not know the optimal priority assignment, or they can merge tasks decreasing the system schedulability. In our project, we could not merge activity tasks, since tasks were invoked multiple times in an activity and they participated in multiple activities.

6. References

[1] H. Gomaa, *“Designing Concurrent, Distributed, and Real-Time Applications with UML”*, Addison-Wesley, 2000.

[2] M.G. Harbour, M.H. Klein, J.P. Lehoczky, “Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems,” *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 13-28, IEEE Computer Society Press, 1994.

[3] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour, *“A Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems”*, Kluwer Academic Publishers, 1993.

[4] J.W.S. Liu, *“Real-Time Systems”*, Prentice-Hall, 2000.

[5] A. Ran, R. Lencevicius, ”Making Sense of Runtime Architecture for Mobile Phone Software”, *Proceedings of ESEC/FSE’2003*.