ARTOS: Technical Report, No. 1, (VS-R08-2012)

ARTOS: System Model and Optimization Algorithm

Technical Report

Vladimir Nikolov · Matthias Matousek · Dieter Rautenbach · Lucia Draque Penso · Franz J. Hauck ·

Published online: 01 December 2012

Abstract The ARTOS research project at the University of Ulm explores the runtime administration of concurrent soft real-time applications. Available resources need to be assigned to applications, taking into account their priorities and utility. The applications support different modes with associated resource requirements. Higher modes reflect better quality of the outcome.

An algorithm using a dynamic programming approach was designed to find the optimal solution to the problem of assigning resources to applications while achieving the highest possible utility. The algorithm was implemented in Java and in C as part of our preliminary work. Its evaluation shows which factors have the most pronounced impact on the performance of the optimization.

It is investigated to which extent — regarding varying the numbers of applications and their modes, as well as the available resource limits — the algorithm is suitable for deployment in real-time systems.

Keywords ARTOS · Soft Real-Time · Scheduling · QoS · Optimization

V. Nikolov · M. Matousek · F. J. Hauck
Institute of Distributed Systems,
University of Ulm,
Ulm, Germany
E-mail: vladimir.nikolov@uni-ulm.de
E-mail: matthias.matousek@uni-ulm.de
E-mail: franz.hauck@uni-ulm.de

L. D. Penso · D. Rautenbach
Institute of Optimization and Operations Research,
University of Ulm,
Ulm, Germany
E-mail: lucia.penso@uni-ulm.de
E-mail: dieter.rautenbach@uni-ulm.de

1 Problem Statement and Classification

Traditional real-time scheduling theory [18] (e.g. fixed or dynamic priority based preemptive mechanisms) generally lacks methods for fast and robust (i.e. precise and controlled) adaptation to varying resource availability and fluctuating consumer demands. Scheduler relying on that theory are usually required to have stringent knowledge about a set of managed activities (tasks) and their computational requirements in order to guarantee accurate system performance. This condition manifests itself as limitation of the deliverable quality of applications with strict timing constraints but varying execution complexity. Most common examples for the need of flexible and dynamic scheduling come from the area of multimedia [5] and control applications [8]. Those systems actual saturation is affected by the occurance of periodic and sporadic events, e.g. interarrival of external stimuli, intermediate processing results, user interaction, etc. The event-triggered computations may exhibit varying algorithmic complexity depending on unpredictable factors and peculiarity of input signals or data. In such cases scheduling has to deal with executional nondeterminisms while simultanously attempting to retain desired system quality [1].

This illustrates a classical problem in scheduling theory: the trade-off between resource demands of computations and their delivered quality. The examination of an optimal solution for that problem during runtime, i.e. a resource distribution attaining optimal system quality, is the main objective of almost every existing soft real-time scheduler [24]. Techniques used to optimally arrange re-

[1] In real-time literature the term “quality” has various definitions and metrics, e.g. tolerated rate of timing faults, throughput rate of successfully completed tasks, specific numeric or functional QoS factors, etc.

sources with activities have to fit a certain system and task model. Generally, if resources are overprovisioned, e.g. due to optimistic admission or unexpected variations of demands, temporal requirements of tasks and applications are very likely to be missed and consequently system qualities to be jeopardised. On the contrary a slack of resources should naturally allow for a higher admission rate and task throughput, leading to a higher overall system quality.

1.1 Resource Reservations

The most common approach to dynamically control resource utilization and to carry out isolation among multiple concurring computations (consumers) is resource *partitioning*. In terms of shared computational power, partitioning means that each task receives a virtual fraction of a processor over a period of time. This fraction is called *reservation* and reservations can be *static* or *dynamic*. A reservation-based scheduler is responsible for controlling task admission and capacity allocation according to a reservation policy. Moreover, it must enforce the reserved capacities. In case of dynamic reservations, resources must be distributed dynamically, i.e. capacities must be adjusted periodically with respect to the actual task demands and currently available resources. Serveral techniques have already been proposed for that purpose, e.g. *Constant-Bandwidth Servers* [5,6], *Resource Reclaiming* [11,12,16] and *Feedback Scheduling* [8,15,26]. Servers and reclamation techniques are often combined to achieve both, cost enforcement and redistribution of spare capacities. Feedback scheduling incorporates control theoretical mechanisms spanning a feedback loop between reservation adjustment[2] and the observation of some significant system-global [20, 21, 25] or task-local [4, 7] metrics.

Most of these mechanisms have in common that they perform well for transient overload conditions and resource bottlenecks. In fact, slight load peaks can be smoothed by relocating slack resources dynamically to utilization hot-spots while still keeping the system controlable and its QoS feasible. However, on permanent bottlenecks low-priority or even all tasks in the system may suffer from continuous (timing) faults [13]. Plainspoken, cutting of reservations and redistributing spare resources can partly help to treat an overload condition. However, this does not guarantee long-term stability, i.e. temporal correctness and feasible delivered quality of the tasks. Equally, utilization of slack resources while increasing tasks reservation portions does not naturally lead to quality improvement [3]. For these purposes, applications and their tasks must in some way promote controlled quality adjustments with respect to their complexity and resource consumption.

1.2 Quality Adjustments

Several techniques have been proposed for this kind of quality adjustments within the application domain [24], i.e. *job skipping*, *period modifications* and *algorithmic degradation* of tasks' computational complexity. Some of them were incorporated with scheduling strategies, e.g. period variations performed centrally by an *elastic scheduler* [10]. A common approach comprising the above mentioned methods are *application modes*. Therewith, the exact quality adjustment techniques are extracted from system schedulers influence and subjected to application developers.

Usually, application modes incorporate a two-level scheduling scheme. A high-level scheduler makes decisions about the active modes based on their aggregated behaviour. A low-level system scheduler controls the dynamics of the individual application tasks based on a traditional model, e.g. RM [4] or EDF [5]. The distribution of resources is reduced now to an activation scheme comprising an optimization problem, that is the trade-off between the resource requirements of modes and their quality benefits. As already mentioned, quality benefits can be expressed in various ways, e.g. numeric values, or can be computed online considering application-specific quality functions. User preferences on the applications can be accumulated and considered within the optimization as well. Indeed, application modes are not a novel approach in soft real-time theory, and there exist various works dealing with different models, schedulers and optimization algorithms [9, 14, 19, 21, 22].

A crucial point for the performance of a system incorporating multi-mode applications is the need for a precise approximation of each application mode's resource requirements. This information is significant for the computation of an optimal mode configuration. Most of the above cited works require preliminary information about task's average (ACET) or worst-case (WCET) execution time, in order to produce feasible solutions. Some other incorporate predefined functions mapping quality to required resources and vice versa [22]. In case of "unknown" applications some approaches propose *test modes* or *initial estimates* for primarily interpolation of resource usages [17]. However, the situation is much more

[2] in this context a reservation is treated as a *controlled variable*

[3] unless tasks have an "any-time" predicate per definition

[4] Rate-Monotonic

[5] Earliest-Deadline-First

complicated and non-deterministic, when applications are composed of unknown components and services during runtime and characteristics of the target platfrom are unknown until deployment. In this case not only initial estimation is a tough issue, but also the continuous approximation of resource demands and appropriate system adaptation. Due to arbitrary invocation of previously unknown services, execution times may exhibit intensive and unpredictable variations. Hence, preliminarily determined resource demands are utterly inapplicable and actual requirements must be monitored and estimated online.

However, all systems known to us which are monitoring and estimating tasks resource usages during runtime are sensitive to slight load deviations. They react with intensive reconfigurations oftentimes oscillating around short-term optima and partially stable states. In most known cases mode switch costs within applications are disregarded too for simplicity reasons. Brandt et al. [9] for example complain about the diversity of subsequent transient configurations, due to characteristics of their optimization algorithm. This is the only work where the problem of high reconfiguration costs induced by frequent reconfigurations is commented. Thus, the authors propose *skip values* in order to suppress continuous adaptations on single deadline misses or estimates variations, which in fact is a plain approach. In our ongoing research we are working on *defensive adaptation* mechanisms which shall focus more precisely on the mentioned problems. Thereby, the long-term behaviour of applications is recognized for establishment of the most permanent stable state of the system, without losing fast reconfiguration ability and response on crucial load variations and bursts.

1.3 The ARTOS Project

ARTOS is a research project at the *Institute of Distributed Systems* at the University of Ulm, which targets at the development of a framework for multi-mode applications with soft real-time requirements. The framework is build on the basis of OSGi using Real-Time Java (RTSJ[6]). Furthermore, applications are supposed to be dynamically composed of a potentially varying set of available services and components (OSGi bundles) during runtime. Figure 1a illustrates the basic architectural model of an ARTOS platform.

The main subject of ARTOS-Framework is the handling of applications' dynamic lifecycle and fluctuating resource demands, due to their component-based and service-oriented nature. For that purpose, ARTOS incorporates a high-level scheduler which continously determines a feasible resource distribution with respect to maximally achieved quality within the managed platform. Quality adjustments are performed via selection of active application modes. Thereby, the scheduler incorporates a feedback loop between observation of applications' resource demands and a mode-shifting actuator. The basic scheduler scheme is depicted in Figure 1b.

During runtime ARTOS continously measures the demands of a currently effective modes selection (configuration) and computes a statistical approximation of the actual application behaviour. Based on this, a decision component evaluates the necessity for a reconfiguration, taking into account available resources and the actual occurrence of timing faults. If a reconfiguration is required the estimated resource requirements are fed into an optimization component, which creates an new optimal selection of application modes. Finally, a mode-switching component toggles the applications to the appropriate modes of the new configuration, following a certain strategy (*switch-protocol*).

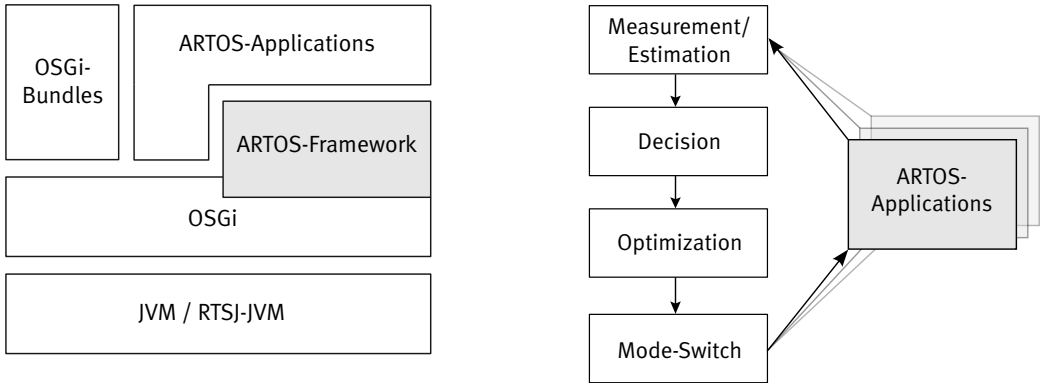
Several aspects are targeted by our ongoing research work, especially initial measurement and estimation strategies for unknown applications during runtime, detection of periodic bottlenecks and smoothing via phase shifts of applications' threads, damping mechanisms for oscillating reconfigurations. These aspects are contributed to the development of an advanced high-level scheduler, which on one hand behaves defensively regarding system reconfigurations but on the other hand ensures fast response on internal load variations. However, these research topics are beyond the scope of this paper.

The purpose of this paper is to present the current development status of ARTOS, focusing on its system model and on early evaluations of the central optimization component which is the core of our high-level scheduler. In the following Section 2 we give a formal definition of our system model and the resource optimzation problem as well as some examples of the application approximation techniques used by the framework during runtime. Our solution of the optimization problem based on a knapsack algorithm with dynamic programming is then presented in Section 3. Section 4 deals with implementation strategies of the algorithm, while performance evaluations of our current implementation are presented in Section 5 for a set of test cases. Finally, in Section 6 we conclude our actual investigations.

2 Definition of System Model and Optimization Problem

At any time during system execution we assume a dynamic set of n applications $\mathcal{A} = \{A_1, A_2, ..., A_n\}$, which are persumed to run concurrently and not to be related to

[6] Real-Time Specification for Java



(a) ARTOS Architecture (b) ARTOS High-Level Scheduler

Fig. 1: ARTOS System and Scheduler Model

each other. Applications can be activated respectively in different application modes. For each application $A_i \in \mathcal{A}$ there is a set of m_i modes $\mathcal{M}_i = \{M_{i,1}, M_{i,2}, ..., M_{i,m_i}\}$. It is assumed for simplicity that $\mathcal{M}_i \cap \mathcal{M}_j = \emptyset$ for distinct applications A_i and A_j. To each mode belongs an associated resource requirement $R_{i,j}$ and utility $\mathrm{U}_i(M_{i,j})$. Both are assumed to be non-negative numeric values, with $R_{i,j} \in \mathbb{N}_0 \; \forall \; i \in \{1, 2, ..., n\} \wedge j \in \{1, 2, ..., m_i\}$ and $\mathrm{U}_i : \mathcal{M}_i \rightarrow \mathbb{N}_0$. The delivered utility is defined by an application specific and platfrom independent benefit function $\mathrm{U}_i(M_{i,j})$ which is subjected by all possible application modes. Every set of modes $\mathcal{M}_i$ of an application A_i contains a mode $M_{i,0}$ with zero resource requirements ($R_{i,0} = 0$) which corresponds to the application not running at all. Of course the utility gained in this case is also zero ($\mathrm{U}_i(M_{i,0}) = 0$). Adding such a mode ensures that an optimal resource distribution always exists. Applications can be weighted by users and their importance a_i must be normalized and considered by the system too. Further we assume monotone applications, i.e. a higher application mode yields a higher application utility by possibly consuming a higher amount of resources. For the moment we expect, that applications are independent and their demands $R_{i,j}$ contain all required CPU resources. The total system resource is bounded by $R \in \mathbb{N}_0$, corresponding to 100% resource utilization. The value R could also correspond to less than 100% utilization in order to obtain a buffer of slack resources.

For a given set of applications and their finite sets of modes, the resource allocation in our system shall be established by merely setting up an optimal configuration of application modes. In extreme cases applications can be even deactivated. This can be described as an optimization problem which aims to maximize the gained utility so that the total system resource boundary R is observed. It is comparable with the well known knapsack problem [23] which considers a knapsack with a fixed weight constraint. A number of objects — all assigned a weight and a value — must be chosen to fit into the knapsack without exceeding its limits, yet assuring a maximum value of the knapsack objects. One common solution uses a dynamic programming algorithm [23].

The optimization task in our system can be defined as presented in Table 1. For the ideal theoretical case an optimal resource distribution can be calculated and set up in advance (offline). A recalculation is merely necessary on variations of user preferences or when the set of active applications changes dynamically. However, in practice this is not suitable since:

1. Resource demands $R_{i,j}$ are unknown or not applicable to the actual platform.
2. $R_{i,j}$ are not constant but subject to application-specific variations.
3. An external consumer outside of the observed system steals resources[7].

Our system is not reliant on preliminary knowledge about execution times, not least because those can not be predicted for all platforms an application should run on. Hence, possible $R_{i,j}$ must be estimated as a basis for the optimization. Initial estimation of applications is a significant part of our research work, but out of the scope of this paper.

In order to cope with fluctuating applications' demands, their actual demands $S_{i,j}$ are measured. Consequently, moving averages $L_{i,j}$ of the execution times are calculated. Similarly, the deviation of $S_{i,j}$ from $L_{i,j}$ is computed as the moving average $J_{i,j}$ (jitter). All measured and computed values are relative to the actually available resources for ARTOS. A first approach for the estimation is:

$$R_{i,j} = \lfloor L_{i,j} + \Phi_i \cdot J_{i,j} \rfloor \,, \qquad \text{with } \Phi \in \mathbb{R}_{>0}. \tag{1}$$

[7] e.g. other processes running on the same operating system

Table 1: Problem Definition

$$\begin{aligned} \text{Find a selection} \quad & J = \{M_{1,j_1}, M_{2,j_2}, ..., M_{n,j_n}\}, M_{i,j_i} \in \mathcal{M}_i, \\ \text{maximizing} \quad & \sum_{i=1}^{n} a_i \cdot \mathsf{U}_i(M_{i,j_i}) \,, \\ \text{subject to} \quad & \sum_{i=1}^{n} R_{i,j_i} \leq R \text{ and } |J \cap \mathcal{M}_i| = 1 \,\forall\, \mathcal{M}_i \,. \end{aligned}$$

Through variation of Φ_i the probability for an overrun of the real costs with respect to the estimated $R_{i,j}$ can be adjusted. A lower probability leads to an overprovisioning of resource demands. This is a relative limitation of executable applications, but leads to less frequent resource bottlenecks in the established application modes configuration. Furthermore, factor Φ_i controls the adaptation velocity of the estimates $R_{i,j}$ with respect to occurring changes within the measured demands. Thus, a higher value of Φ_i leads to a faster approximation of an application burst and also to a higher overprovisioning of its actual resource demands.

In that context it is interesting to mention, that it depends on the sampling frequency of the $S_{i,j}$ whether periods of applications[8] or of single tasks are noticeable within the estimated $R_{i,j}$. Hence, it is quite possible that optimizations operate with estimations reflecting a microscopic rather than a long term behaviour of the applications. As a consequence, series of reoptimizations and even cyclic reconfigurations are possible. This problem is in fact a part of our ongoing research work. A conceivable strategy could be based on period-wise aggregations of measured parameters and estimates recomputations. Such an approach aims at smoothing the reconfiguration probability within application periods, while tolerating a certain approximation error and amount of timing faults. Further investigations are in progress.

3 Optimization Algorithm

Similar to the dynamic programming approach for the knapsack problem, the algorithm which we developed solves the problem of choosing a mode $M_{i,j}$ for every application A_i so that the total gained utility $u = \sum_{i=1}^{n} a_i \cdot \mathsf{U}_i(M_{i,j_i})$ is at its maximum under the constraint $\sum_{i=1}^{n} R_{i,j_i} \leq R$. Our actual knapsack-based solution is independent of the nature of the utility functions and always produces an optimal solution which is calculated in one algorithm iteration. At each iteration step the algorithm recurs over every feasible combination of modes (including the zero-modes $M_{i,0}$) and choses a partial solution with maximal utility for the next step. The complexity of the entire solution depends on the number of applications and their modes. A quantitative evaluation is presented in Section 5 based on a first implementation of the algorithm which is explained next.

Two tables, u and J, are used as data structures for the computation.

For $k \in \{0, 1, ..., n\}$, $r \in \{0, 1, ..., R\}$ and $M_{i,j_i} \in \mathcal{M}_i$ let

$$\mathsf{u}(k, r) = a_1 \cdot \mathsf{U}_1(M_{1,j_1}) + a_2 \cdot \mathsf{U}_2(M_{2,j_2}) + \ldots + a_k \cdot \mathsf{U}_k(M_{k,j_k}) \quad (2)$$

denote the maximum utility such that $R_{1,j_1} + R_{2,j_2} + \ldots + R_{k,j_k} \leq r$.

Let further

$$\mathsf{J}(k, r) = \{M_{1,j_1}, M_{2,j_2}, ..., M_{k,j_k}\} \quad (3)$$

be an optimum realizing selection satisfying the given constraints.
Table $\mathsf{u}(k, r)$ vividly contains the maximum possible utility for the first k applications and a total system resource boundary of r, as $\mathsf{J}(k, r)$ holds the selection which led to the gained maximum utility.

The tables are initialized with $\mathsf{u}(0, r) = 0$ and $\mathsf{J}(0, r) = \emptyset$ for all $r = 0, 1, ..., R$. For $k \in \{1, 2, ..., n\}$ and $r \in \{0, 1, ..., R\}$ the following recursion applies:

$$\mathsf{u}(k, r) = \max_{M_{k,j} \in \mathcal{M}_k} \{\mathsf{u}(k-1, r-R_{k,j}) + a_k \cdot \mathsf{U}_k(M_{k,j})\} \quad (4)$$

while with

$$\mathsf{J}(k, r) = \mathsf{J}(k-1, r-R_{k,j}) \cup \{M_{k,j}\} \quad (5)$$

an optimum realizing selection is given. Checking if $r - R_{k,j}$ is non-negative and therefore still resides within the tables ensures observation of the constraint $R_{1,j_1} + R_{2,j_2} + ... + R_{n,j_n} \leq R$. This definition allows for the formulation of the algorithm in pseudo code (see Algorithm 1).

[8] i.e. coming up from the temporal distribution of applications (periodic) tasks

Algorithm 1 Optimization algorithm

```
 1: for r = 0 → R do
 2:     u[0][r] ← 0, J[0][r] ← ∅
 3: end for
 4: for k = 1 → n do
 5:     u[k][0] ← 0
 6:     for r = 1 → R do
 7:         max ← 0
 8:         for M_{k,j} ∈ M_k do
 9:             b ← u[k − 1][r − R_{k,j}] + a_k · U_k(M_{k,j})
10:             if b > max then
11:                 max ← b
12:                 J[k][r] ← J[k − 1][r − R_{k,j}] ∪ {M_{k,j}}
13:             end if
14:         end for
15:         u[k][r] ← max
16:     end for
17: end for
18: return J[n][R]
```

3.1 Priority and Utility

As one can see, application priorities a_k are implied in line 9 in order to express user preferences. They have direct impact on the utility $\mathsf{U}_k(M_{k,j})$ gained by the application modes and must be normalized in order to be treated equally for all applications.

So far the utility function $\mathsf{U}_i(M_{i,j})$ was described as a function $\mathsf{U}_i : \mathcal{M}_i \rightarrow \mathbb{N}_0$, but its semantic meaning has not been specified further. In fact, the realization of U is not regulated by the system framework at all; it is for the developers to decide how the utility is defined for their individual applications. This may necessitate normalizing the result of the utility function to ensure that every $\mathsf{U}_i, i \in \{1, 2, ..., n\}$ maps into the same range and thereby achieving comparable values. It is also beneficial to choose a function that can be computed quickly, since it will be invoked several times for each entry in $\mathsf{u}(k, r)$. This frequent invocation could lead to a serious negative impact on the execution time of the algorithm, if a utility function consists of complex calculations. Consequently, it might be useful to pre-compute and store the utilities for all modes $M_{i,j} \in \mathcal{M}_i$ and therefore being free of delays.

In our experiments we used precomputed utility values for application modes, which were normalized within a predefined numeric range and delivered all at once to the optimization algorithm. On this way, no further invocations of $\mathsf{U}_i(M_{i,j})$ were necessary during optimization.

3.2 Theoretical Considerations

As the knapsack problem is known to be NP-complete [23], no algorithm which computes the exact solution can be expected to run in polynomial time. However, dynamic programming provides a pseudo-polynomial time algorithm. Since the number of computed values in $\mathsf{u}(k, r)$ is $n * (R + 1)$ and for every entry there are at most $m_{max} = \max_{i \in \{1,2,...,n\}}\{|\mathcal{M}_i|\}$ modes, the described algorithm runs in pseudo-polynomial time. However, if all involved numerical values are kept within reasonable ranges, this should not constitute a serious problem.

For example, the value denoting the total system resource boundary R can be set to any non-negative integer value. It is obviously beneficial for the computing time to keep R — and therefore the resource requirements $R_{i,j}$ for the modes of the applications — small. The increase of computing time with a larger R is evaluated in Section 5.

Finding the maximum, as it is carried out in Lines 8 to 15 in Algorithm 1, causes the optimization to prefer applications with a lower index over applications with a higher index if more than one optimal solution exists. Replacing the "$>$" in Line 11 with a "$\geq$" leads to a preference of applications with high index. In order to circumvent this characteristic the algorithm can be easily extended to keep track of all detected optimal solutions and to choose e.g. a "middle" one as the new optimum. The following extension exemplifies the solution:

Algorithm 2 Extended optimization algorithm

```
 1: ...
 2: for k = 1 → n do
 3:     u[k][0] ← 0
 4:     for r = 1 → R do
 5:         max ← 0
 6:         nmax ← 0
 7:         for M_{k,j} ∈ M_k do
 8:             b ← u[k − 1][r − R_{k,j}] + a_k · U_k(M_{k,j})
 9:             if b > max then
10:                 max ← b
11:                 J[k][r] ← J[k − 1][r − R_{k,j}] ∪ {M_{k,j}}
12:             else
13:                 if b = max then
14:                     maxima[nmax] ← M_{k,i}
15:                     nmax ← nmax + 1
16:                 end if
17:             end if
18:         end for
19:         if nmax > 0 then
20:             J[k][r] ← J[k − 1][r − R_{k,j}] ∪ maxima[nmax/2]
21:         end if
22:         u[k][r] ← max
23:     end for
24: end for
25: ...
```

As can be seen in Lines 13 to 16, multiple optimal solutions are stored temporarily in an array *maxima*. The selection of a middle optimum is carried out in line 20. Choosing a middle path through the solution space pro-

vides a more uniform distribution of available resources between applications and a potentially higher admission rate. Other strategies are possible here as well.

4 Implementation

Over the course of implementing the algorithm, many decisions need to be made, especially regarding data structures and the large potential for efficiency optimization. The choice of the value for the total system resource boundary R has been addressed already and is further evaluated in Section 5.

Taking a closer look at the recursion in Algorithm 1 used to fill table u(k, r), one might notice that once a row — that is, a particular k — has been computed, the row above that one is no longer needed for the rest of the computation to complete. Therefore, it is entirely sufficient to store only a table of $2 \times (R + 1)$ values for u(k, r). This can be realized efficiently with a two-dimensional array containing integer values.

For the second table J(k, r), the situation is considerably more complex. It is possible to realize J as an array with dimensions $2 \times (R+1)$ as well, but the definition (5) of J(k, r) shows that every entry would need to hold a selection of k modes which adds an additional dimension to the array. The last row would then contain $R + 1$ selections with n modes[9] each, leading to a total size of about $2 \times (R+1) \times n$. It is significantly easier to manage a two-dimensional array sized $n \times (R + 1)$, where each entry in J(k, r) holds the index or ID of the selected mode of application A_k that led to the gained utility in u(k, r). The selected modes of each application $A_i, i < k$ can be retrieved by following the entries in reverse order. When, for example, J$(k, r) = 2$ indicates that mode $M_{k,2}$ of application A_k was selected, $M_{k-1,\mathrm{J}(k-1,r-R_{k,j})}$ is the selected mode of application A_{k-1} and so forth. It is necessary to store all of the selected modes, in order to be still able to return the complete selection at the end of the computation; no entries can be abandoned as in the implementation of u(k, r).

All implementations completed so far use arrays of signed 32 bit integer types. Choosing different types might have lead to faster execution times, but was not realized in order to be able to compare and profile different implementations more easily.

Further optimizations are possible, for example the omission of computations of the last row ($k = n$); only the last entries u(n, R) and J(n, R) need to be calculated. This optimization and others of similar nature exist, but were not implemented yet for the sake of simplicity and comparability of the different implementations.

[9] where n is the total number of applications in the system

Listing 1: ApplicationImpl.java

```
public class ApplicationImpl
             implements Application {
  // these are the Ri,j
  private int[] resRequirements;
  private int priority;
  public ApplicationImpl(int[] modes,
       int priority){...}
  public int[] getResRequirements() {...}
  public int getPriority() { ... }
  void setPriority(int priority) {...}
  // utility function
  public int utility(int mode) {...}
}
```

4.1 Java

Java was used for the first implementation of the optimization algorithm. With the considerations made earlier in this chapter, the implementation is fairly straightforward.

Applications are implemented as objects of type `Application` (see Listing 1). Each `Application` has one field for the priority of the application and one field holding an array of resource requirements.

The value at position `0` of `resRequirements` corresponds to the mode `0` of the application and so forth. Both tables u(k, r) and J(k, r) are stored in two-dimensional `int`-arrays where `apps` is an array of type `Application[]`:

```
u = new int[2][R+1];
J = new int[apps.length][R+1];
```

Using modifiers such as `static`, `private` and `final` allows the compiler to perform better optimizations and takes advantage of characteristics of the Java Virtual Machine. Values are stored in local variables wherever possible and the creation of complex objects is avoided in order to speed up access to the data.

Java's `System.nanoTime()` is used to profile computation time. More precise and more sophisticated alternatives[10] certainly exist for this task. The choice was made because it seems to be the time measuring function which is best comparable with the chosen C equivalent[11].

4.2 RTSJ

The non-determinism of Java is a serious issue for real-time systems. Using RTSJ features for the implementation does not make the execution faster, but it helps

[10] like profiling tools or libraries

[11] see Section 4.3

to make the runtime of the algorithm more deterministic. The greatest concern is Java's Garbage Collection. The Real-Time Specification for Java defines so-called scoped memory and immortal memory areas. In addition to the common heap and stack, scoped and immortal memory can also be used to allocate memory. The difference is that these memory areas are not garbage-collected. Scoped memory is freed when all threads have left the memory area. Immortal memory is not freed at all. This allows for certain threads — the `NoHeapRealtimeThread`s (NHRT)— never to be interrupted by the Garbage Collector[12].

In the actual implementation, a NHRT is created inside of the immortal memory area, which is accessible through the `ImmortalMemory` class (see Listing 2). Executing the optimization algorithm is then done via the NHRT.

Listing 2: Immortal Memory

```
// do all allocations in immortal memory
ImmortalMemory.instance().enter(
  new Runnable() {
    public void run() {
      final Testcase testcase = Testcase.
          loadTestcase(file);

      // switch to an instance of NHRT
      NoHeapRealtimeThread nhrt =
      new NoHeapRealtimeThread(null,
        ImmortalMemory.instance()) {
        public void run() {
          Simulation.testAlgorithm(testcase);
        }
      };

      nhrt.start();
    }
});
```

Everything in the `Simulation` and `Optimization`[13] classes is done without explicit use of RTSJ features.

Since `ImmortalMemory` is never freed, the memory will simply fill up over the course of executing this implementation, eventually leading to an `OutOfMemoryError` if the consumed memory gets too high. This does not constitute a problem during our experiments because the program terminates after the invocation of the optimization, but the algorithm should be implemented using `ScopedMemory`[14] when it is deployed in the actual framework.

[12] if they are configured accordingly

[13] which is being called by `Simulation.testAlgorithm(…)`

[14] which is freed when all threads have left the memory area

Listing 3: Application struct

```
typedef struct {
        int priority;
        int nmodes;
        int *modes;
} app;
```

To ensure optimal results, the real-time JVM[15] is instructed to compile the optimization methods at class-initialization-time instead of using the Just-in-Time compiler or executing interpreted code.

4.3 C

To be able to form an opinion on whether the optimization algorithm is suitable for deployment in a soft real-time environment, a native version was implemented for comparison with the Java version. The implementations are much alike, since Java and C have very similar syntax. Differences arise, of course, in the management of the data around the algorithm. Because C is not an object-oriented programming language, applications need to be represented differently. A `struct`, as shown in Listing 3, is used for that purpose.

It contains the same information as the `ApplicationImpl` class[16] in the Java implementation. The `modes` array is the equivalent to the `resRequirements` array. The number of modes for this application is stored in `nmodes`.

The C code is compiled with `-march=native` and `-O2` to instruct the compiler to do optimizations. The `-march=native` switch enables optimizations for the local machine and `-O2` enables most supported optimizations [2]. The additional optimization level `-O3` did not lead to any measurable speed benefits.

The POSIX function `clock_gettime` is used to profile the C implementation of the algorithm. Just like Java's `System.nanoTime()` it returns a time with nanosecond resolution. The program needs to be linked with `-lrt` in order to access the function.

4.4 JNI

A third approach is the implementation in Java but using the *Java Native Interface*[17] (*JNI*). Consequently, the opti-

[15] In our experiments we used the Sun Java Real-Time System 2.2 HotSpot Virtual Machine. See section 5.1.

[16] see Listing 1

[17] `http://download.oracle.com/javase/6/docs/technotes/guides/jni/`

Listing 4: Native Function

```
public class NativeOptimization {
    private static native
    int[] nativeFindSetOfModes(
            int R,
            int[][] apps,
            int[][] utilities,
            int[] priorities);
    // ...
}
```

mization algorithm is implemented in C and mostly similar to the pure native implementation. The difficulty is to pass the application representations from Java to native without strongly affecting the efficiency. For that reason the `Application`[18] objects are not passed to the native function; instead arrays of integers containing the same information are used, which saves many JNI function calls. Listing 4 shows the signature of the JNI method used to invoke the native optimization code.

The variable `R` corresponds to the total system resource boundary, while the `priorities` array — as hinted by its name — contains the application priorities. The array `apps` consists of an array of resource requirements for each application; thus, `apps[i][j]` corresponds to the resource requirement $R_{i,j}$ for the mode $M_{i,j}$ of application A_i. The `utilities` array contains the precomputed utility values of all application modes. Within the native optimization code Java arrays are accessed using JNI functions defined in `jni.h` as shown in Listing 5.

Access to primitive arrays is performed via the 'critical' versions of the JNI functions, in order to avoid potential blocking effects within the JVM during optimization [19]. Naturally, this kind of accessing the Java types adds overhead to the execution of the optimization. The impact of the overhead as compared to that of a pure native implementation is evaluated in Section 5.

5 Evaluation

Three variables have major impact on the computation time of the algorithm. The pseudo code in Algorithm 1 (on page 6) shows three nested `for`-loops. The first one iterates over all applications, the second loops R times and the third iterates over each mode of the current application. These three values — the number of applications, the total system resource boundary and the average number of modes per application — determine the actual execution time of the algorithm. Since the utility function is

[18] see Listing 1

[19] see http://java.sun.com/docs/books/jni/html/objtypes.html

Listing 5: Accessing Java arrays in C

```
{...} nativeFindSetOfModes( JNIEnv *env,
  jclass clazz, jint R, jobjectArray apps,
  jobjectArray utilities, jintArray priorities)
{
...
// get a pointer to the priorities
prior = (*env)->
    GetPrimitiveArrayCritical(env,
        priorities, NULL);

// get the number of applications
napps = (*env)->GetArrayLength(env, apps);

// get the modes of the applications
modes = (jint**)malloc(napps * sizeof(jint*));
nmodes = (jsize*)malloc(napps * sizeof(jsize));

for (i = 0; i < napps; i++) {
  app = (*env)->
    GetObjectArrayElement(env, apps, i);

  modes[i] = (*env)->
    GetPrimitiveArrayCritical(env, app, NULL);

  nmodes[i] = (*env)->
    GetArrayLength(env, app);
}
...
```

called for each entry in u(k, r), it is an additional factor to the execution time. However, as mentioned in Section 3.1, it can be enforced to pre-compute the values of the utility function. Therefore, it is not regarded in the evaluation of the algorithm.

The objective of this Section is to evaluate the effect of the relevant variables on the algorithm. All three implementations — Real-Time Java[20], C, and Java with native calls — are assessed.

5.1 Test setup

5.1.1 Test Cases

In order to evaluate the algorithm, testing and profiling needs to be carried out. For that reason each implementation is set up to accept test cases which contain the total system resource boundary and a set of applications. Applications consist of a priority and a set of modes represented by their resource requirements. The individual test cases are defined as *JavaScript Object Notation* (*JSON*) [3] objects such as the following example in Listing 6:

[20] further simply referred as "Java"

Listing 6: JSON sample test case

```
{
  "systemResourceBoundary":10,
  "apps":[
      {   "priority":1,
          "resourceRequirements":[0,3,5,7] },
      {   "priority":5,
          "resourceRequirements":[0,3,5] },
      {   "priority":2,
          "resourceRequirements":[0,4,5] }
  ]
}
```

The test cases are loaded into the Java implementations[21] via the Flexjson[22] library. For the C implementation, the JSON objects are converted by a small Python script to simple textual descriptions of the tests, which are then delivered to the standard input stream of the native executable.

5.1.2 Test Execution

Test results are printed to the standard output stream and contain information such as the total system resource boundary, the number of applications, the resource requirements of each application mode, the average number of modes, the resulting modes selection, as well as the execution time of the algorithm in nanosecond resolution. It is not guaranteed that nanosecond precision is indeed obtained, but the most precise clock available is expected to be used, according to the documentation of `System.nanoTime()` in Java and `clock_gettime` in C.

Each test is executed hundred times and the results are averaged to compensate fluctuations. The execution of the test cases and the aggregation of the results are handled by Python scripts.

The compiled Java program is executed in the Sun Java Real-Time System 2.2 HotSpot Virtual Machine[23]. It is started with the command-line arguments

```
-XX:+RTSJBuildCompilationList
-Drtsj.precompile=nhrt.precompile
-XX:+RTSJBuildClassInitializationList
-Drtsj.preinit=itc.preinit
```

which lead to pre-compilation and pre-initialization of the classes described in the files `nhrt.precompile` and `itc.preinit` [1]. The files contain — among others — the optimization classes.

Since RTSJ features are used a "regular" Java Virtual Machine — such as Sun's HotSpot VM or OpenJDK — is not sufficient. Unfortunately, other real-time JVMs like the JamaicaVM[24] by aicas had problems running the utilized libraries, due to differing Java versions, and therefore could not be tested.

5.2 System Characteristics

The machine used for the tests contains an Intel(R) Core(TM)2 Duo CPU with 2×1.4GHz and runs a 64 Bit Arch Linux operating system with kernel version 3.0 and glibc version 2.14. Since ARTOS targets common desktop or mobile computing environments, the tests should have representative results. Tests are executed on an otherwise mostly idle system, avoiding other simultaneously running processes, which could cause load and distort the measurements.

5.3 Variation of System Resource Boundary

The value R denotes the limit with which the total system resources are bounded. Naturally, each resource is limited in some kind, but the numerical value which declares that boundary can be chosen freely.

Since R directly represents the size of one dimension of the table u(k, r), it is the most obvious factor for the execution time of the algorithm along with the number of applications.

The first test case consists of 10 applications with 5 modes each. This is a realistic scenario, albeit a modest one. These applications are tested with various numeric values for the total system resource boundary. Several different values are chosen and the resource requirements of the individual modes are adapted proportionally. Higher numeric values for the resource requirements are a direct consequence of a higher numeric value for R. The tested values range from 128 to 65536, referring to the values that can be stored in one or two bytes respectively.

The table in Figure 2a shows the results for the Java program. The memory consumption does neither include the applications nor their resource requirements nor other data; only allocations for the data structures used by the algorithm are considered. Doubling R results in approximately twice as much execution time and memory consumption. This is plausible as doubling R means doubling the size of u(k, r) and therefore there are twice as many values to calculate and store. The same statement applies to the native implementation, as shown in Figure 2b. However, the native code runs considerably faster than the Java version, while the execution times of the optimization with JNI, as shown in Figure 2c, lie in between. Figure 3 illustrates the linear increase of execution

[21] both pure Java and Java with native calls

[22] `http://flexjson.sourceforge.net/`

[23] `http://java.sun.com/javase/technologies/realtime/index.jsp`

[24] `http://www.aicas.com/jamaica.html`

R	time[ms]	mem[kB]
128	0.50	7
256	1.07	13
512	2.13	25
1024	3.81	49
2048	7.56	97
4096	15.12	193
8192	30.25	385
16384	60.80	769
32768	122.63	1537
65536	242.85	3073

(a) Real-Time Java

R	time[ms]	mem[kB]
128	0.17	6
256	0.31	12
512	0.62	24
1024	1.18	48
2048	2.32	96
4096	4.71	192
8192	9.31	384
16384	18.82	768
32768	38.02	1536
65536	74.67	3072

(b) Native

R	time[ms]
128	0.39
256	0.69
512	1.23
1024	2.40
2048	4.88
4096	9.79
8192	19.38
16384	38.49
32768	76.27
65536	152.10

(c) JNI

Fig. 2: Measurement results for different R, with 10 apps and 5 modes each

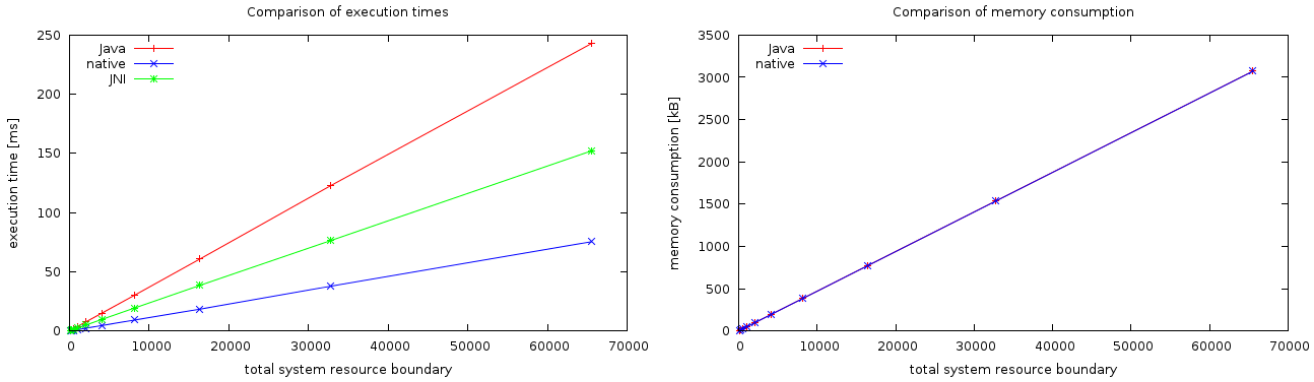


Fig. 3: Comparison of Java, native and JNI implementations for different R

time and memory consumption in all three implementations. Java needs about 240 milliseconds to complete the optimization task with a system resource boundary of 65536. The native implementation is faster with about 75 milliseconds, but such a response time is still unacceptable for an algorithm that assigns resources to real-time applications. Figure 3 shows the gap between the execution time of the Java program compared to that of the native implementation, while the memory consumption is very similar. Unless a faster JVM is used or better optimization of the code leads to faster execution, a native implementation of the optimization is advised.

Further optimizations might shorten the execution time additionally, but keeping the data structures small by choosing a small R proves to be a very efficient way of optimizing. In theory, a larger R leads to more precision, but the fluctuation of the resource requirements of the applications will prevent the system from reaching a very high precision. When, for example, the resource requirement of a mode of an application varies by about 10% of the total system resource present, only 5 different modes are possible at a maximum. Thus, a system resource boundary which is much bigger than 100 seems inadequate.

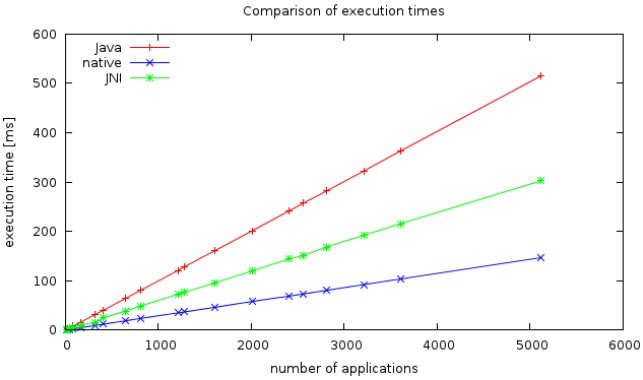
5.4 Variation of the number of applications

Like the system resource boundary, the number of applications directly affects the size of the data structures of the algorithm. At least it affects the size of J(k, r), since u(k, r) only stores the information about two applications at a time. However, the execution times do not benefit from this, as even though a lot of memory can be saved through an efficient data structure, the algorithm still iterates over all applications. With each added application comes a set of modes leading to even more complexity, as each mode needs to be treated. This means that in reality the complexity grows by the average number of modes with each additional application; thus, much faster than by increasing R.

Considering that the results of the test in Section 5.3 suggest a small numerical value for the system resource boundary, the second test uses a value of 256 for R. The test cases range from 10 applications up to about 5000 applications with 5 modes each. The depicted results in Figure 4 confirms the assumptions. Again, doubling the

apps	time(Java)	time(native)	time(JNI)
10	1.04	0.33	0.75
20	2.00	0.63	1.04
40	3.96	1.18	2.48
80	7.74	2.32	4.81
160	15.44	4.72	9.66
320	31.14	9.20	15.14
410	40.09	11.70	24.56
640	63.29	18.30	37.98
810	80.48	23.38	48.39
1210	120.44	34.60	72.44
1280	128.02	36.45	76.21
1610	160.84	45.89	95.45
2010	201.06	57.75	119.75
2410	241.86	68.70	144.21
2560	257.06	72.98	150.80
2810	281.70	80.22	168.14
3210	322.26	91.90	191.97
3610	363.34	103.39	215.66
5120	515.33	146.68	302.48

(a) Execution times in ms



(b) Comparison of Execution times

Fig. 4: Results for different numbers of applications, with 5 modes per app and $R = 255$

number of applications doubles the execution time, but the values increase much faster[25] than in section 5.3.

The execution time of the native implementation ceases to be reasonable at about 50 to 100 applications[26]. This is not a very high number, but the question is rather how many applications are realistic. There is no way anyone but the end-user would have any control over the number of applications present in the system. One could certainly come up with an example where lots of applications are judicious, but considering an average desktop computing environment, more than 20 applications will execute rarely.

5.5 Variation of the Number of Modes per Application

The third `for`-loop of Algorithm 1 (on page 6) iterates over each mode of an application and therefore is the third major factor affecting the execution time. Increasing the average number of modes by one means an additional step for each position in u(k, r) and J(k, r) respectively.

It is impossible for the number of modes to exceed R since the resource requirement can not be higher than the total system resource boundary, at least if integer values are used. Thus, testing with many modes requires to raise the system resource boundary. Figure 5 show the results of a test with a system resource boundary R of 3000, 10 applications and numbers of modes ranging from 50 to about 2000. Even 50 modes per application already cause very long execution times in both Java and native implementations. This is not very surprising, as Section 5.3 already shows that increasing R causes considerably longer execution times itself. Increasing the average number of modes multiplies the execution times by the number of applications and the number of modes. The results of this test are far beyond acceptable response times for a soft real-time system.

However, very high numbers of modes are not to be expected. Even when the resource requirements of individual modes fluctuate much less than in the example in the last paragraph of Section 5.3, more than 20 different modes per application are not likely. Another test with a system resource boundary of 256, 10 applications and up to 50 modes per application results in much lower execution times, as Figure 6 illustrates.

It is to be mentioned that despite of the increase of the number of modes per application, the consumed memory for the last two test cases stays the same. The algorithm does not need to allocate more memory to handle the additional modes, because all the information is already there. The data structures only grow if the resource boundary or the number of applications is increased.

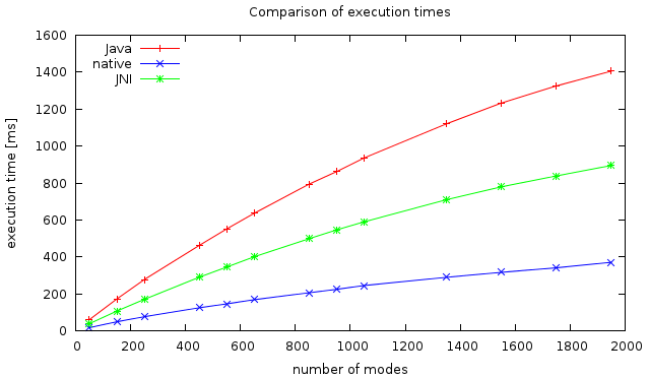
6 Conclusion

This paper presented and evaluated a dynamic programming algorithm with the purpose of optimal resource assignment to concurrent soft real-time applications, using various test scenarios. The algorithm uses a similar approach to a well-known algorithm solving the knapsack problem. As ARTOS aims to be a framework for applications with soft real-time requirements, it was necessary

[25] this applies also for the memory consumption of this test case
[26] see Figure 4a

modes	time(Java)	time(native)	time(JNI)
50	60.2	17.6	38.9
150	171.3	49.6	107.4
250	276.6	76.8	170.4
450	467.9	125.3	290.2
550	549.6	146.3	345.3
650	634.3	168.8	401.0
850	792.9	206.4	498.7
950	862.2	224.9	546.2
1050	937.3	245.5	589.9
1350	1124.7	290.0	709.6
1550	1225.3	316.9	779.4
1750	1323.5	341.2	838.3
1950	1404.1	371.0	894.6

(a) Execution times in ms

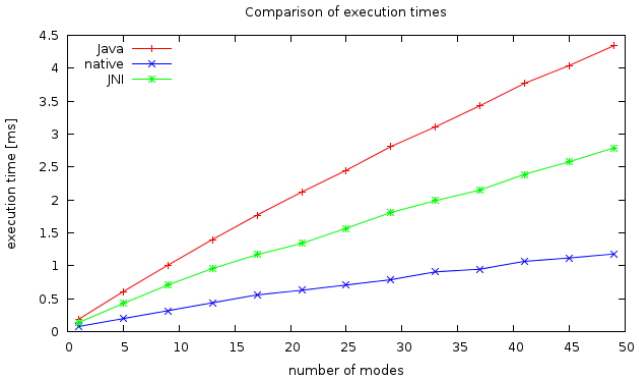


(b) Comparison of execution times

Fig. 5: Results for different numbers of modes, with 10 apps and $R = 3000$

modes	time(Java)	time(native)	time(JNI)
1	0.19	0.08	0.14
5	0.62	0.20	0.43
9	0.99	0.32	0.71
13	1.36	0.44	0.96
17	1.79	0.56	1.17
21	2.10	0.63	1.34
25	2.48	0.71	1.57
29	2.78	0.79	1.81
33	3.10	0.91	1.99
37	3.43	0.95	2.15
41	3.72	1.07	2.39
45	4.05	1.12	2.58
49	4.34	1.18	2.79

(a) Execution times in ms



(b) Comparison for execution times

Fig. 6: Results for up to 50 modes per application, with 10 apps and $R = 256$

to analyze the characteristics of the pseudo-polynomial time optimization algorithm.

All tests evince that the Java implementation of the algorithm takes longer for the optimization than the native version. This applies at least to the utilized Sun Java Real-Time System HotSpot Virtual Machine. Quite possibly a faster JVM can be found, but it still might be favorable to choose a native implementation for the optimization part of the framework. Calling the algorithm via the *Java Native Interface* adds overhead, but the tests were still faster than a pure Java implementation. The pros and cons — regarding portability, determinism and speed — still need to be reassessed in further investigations.

The actual execution time of the algorithm depends on three major factors. One is the value defining the total system resource boundary, another one is the number of applications which are present in the system, and the third one is the number of modes per application. All three values have substantial impact on the performance.

The variable R denoting the total system resource boundary is arbitrary, thus a small numerical value is an obviously beneficial choice for a superior execution time. Tests show that small three-digit values should not be exceeded in order to be able to achieve acceptable response times. Using one byte for the resource requirements — and therefore 255 as system resource boundary — might be a sensible choice.

It can not be foreseen how many applications will be running concurrently. The natively implemented optimization algorithm can handle up to about 50 applications with 5 modes each, and still have response times of a few milliseconds. In reality less than about 10 to 20 applications are expected — considering the field of application, which is a regular desktop computing environment with multimedia applications.

A high number of modes per application affects the performance the most, but examples show that more than

roughly 10 modes do not make much sense, since such a precision can not be reached due to system limitations.

What response times are acceptable for the optimization algorithm depends strongly on how often the optimization is triggered. The task of how to decide when the system needs to be reconfigured is part of our ongoing investigations and research. Should this prove to be relatively seldom, the algorithm could handle even more applications, and additionally, the obstruction of the applications would be minimized. It might even be possible to compute parts of the optimization beforehand in the form of lookup tables. Whether this is feasible would require further investigation.

References

1. Sun java real-time system 2.1 compilation guide, 2011-10-05.
2. Gnu compiler documentation – options that control optimization, 2011-10-09.
3. Javascript object notation, 2012-11-30.
4. L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proc. of the 6th IEEE Int. Conf. on Emb. and Real-Time Comp. Sys. and Appl.—RTCSA*, 1999.
5. L Abeni and G. C. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 4–13, 1998.
6. L. Abeni and G. C. Buttazzo. Resource Reservation in Dynamic Real-Time Systems. *Real-time Systems*, 27:123–167, 2004.
7. L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the 23th IEEE Real-Time Sys. Symp.—RTSS*, 2002.
8. K.-E. Arzen, A. Cervin, J.2 Eker, and L. Sha. An introduction to control and scheduling co-design. In *Conference on Decision and Control*, volume 5, 2000.
9. S. A. Brandt and G. J. Nutt. Flexible soft real-time processing in middleware. *Real-time Systems*, 22:77–118, 2002.
10. G. C. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-Time Sys.*, 23:7–24, 2002.
11. M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. of the 21st IEEE Real-Time Sys. Symp.—RTSS*, pages 295–304, 2000.
12. M. Caccamo, G. C. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Trans. Comp.*, 54, February 2005.
13. T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari. A robust mechanism for adaptive scheduling of multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 10(4):1–24, 2011.
14. T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari. On the integration of application level and resource level qos control for real-time applications. *IEEE Transactions on Industrial Informatics*, 6:479–491, 2010.
15. J. Eker, P. Hagander, and K. Arzen. A feedback scheduler for real-time controller tasks. *Proceedings of The IEEE*, 2000.
16. G. Lipari and S. K. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proc. of the 12th Euromicro Conf. on Real-Time Sys.—ECRTS*, pages 193–200, 2000.
17. G. Lipari, T. Cucinotta, F. Checconi, D. Faggioli, L. Abeni, L. Palopoli, and P. Valente. Design and implementation of a middleware for qos management - ii, 2005.
18. C. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of The ACM*, 1973.
19. J. W. S. Liu, W. K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proc. of The IEEE*, 82:83–94, 1994.
20. C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. Hyuk Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proc. of the 21st IEEE Real-Time Sys. Symp.—RTSS*, pages 13–24, 2000.
21. C. Lu, J. A. Stankovic, S. Hyuk Son, and G. Tao. Feedback control real-time scheduling: framework, modeling, and algorithms. *Real-Time Sys.*, 23:85–126, 2002.
22. R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. A resource allocation model for qos management. In *Proc. of the 18th IEEE Real-Time Sys. Symp.—RTSS*, pages 298–307, 1997.
23. Uwe Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001.
24. L. Sha, T. F. Abdelzaher, K.-E. Arzen, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: a historical perspective. *Real-Time Sys.*, 28:101–155, 2004.
25. Jack A. Stankovic, Chenyang Lu, Sang Hyuk Son, and Gang Tao. The case for feedback control real-time scheduling. In *Euromicro Conference on Real-Time Systems*, pages 11–20, 1999.
26. P. Zhou, J. Xie, and X. Deng. Optimal feedback scheduling of model predictive controllers. *J. of Control Theory and Appl.*, 4:175–180, 2006.