Finite Horizon QoS Prediction of Reconfigurable Firm Real-Time Systems*

Mehdi Amirijoo, Aleksandra Tesanovic, Torgny Andersson
Dept. of Computer and Information Science
Linköping University, Sweden
{meham,alete,-}@ida.liu.se

Jörgen Hansson
Software Engineering Institute
Carnegie Mellon University, USA
hansson@sei.cmu.edu.

Sang H. Son
Dept. of Computer Science
University of Virginia, USA
son@cs.virginia.edu

Abstract

Updating real-time system software is often needed in response to errors and added requirements to the software. Stopping a running application, updating the software, and then restarting the application is not suitable for systems with high availability requirements. On the other hand, dynamically updating a system may increase the execution time of the tasks, thus, degrading the performance of the system. Degradation is not acceptable for performance-critical real-time systems as there are strict requirements on the performance. In this paper we present an approach that enables dynamic reconfiguration of a real-time system, where the performance of the system during a reconfiguration satisfies a given worst-case performance specification. Evaluation shows that the presented method is efficient in guaranteeing the worst-case performance of dynamically reconfigurable firm real-time systems.

1. Introduction

Successful deployment of real-time and embedded systems strongly depends on low development costs, a short time to market, and a high degree of reconfigurability. Component-based software development (CBSD) [6] has been developed specifically to address the same type of problems that real-time software development is facing today. Namely, CBSD enables systems to be developed out of pre-defined software components to fit a specific application. This enhances reusability and reduces the resources allocated to the development of the system. Moreover, using CBSD principles software systems are developed with plug-and-play capabilities, which implies that system reconfiguration can be done dynamically by simply plugging new or additional components to the system during run-time, thereby ensuring that systems are able to efficiently evolve.

However, the real-time research community has not yet fully capitalized on CBSD benefits due to the unpredictable nature of dynamic reconfiguration in real-time environments. Namely, a dynamic reconfiguration of a real-time system changes the temporal properties of the tasks in a system, which in turn affects the quality of service (QoS) negatively. For example, changing the software of the system may result in an increase in the execution time of the tasks, thus, permanently increasing the utilization or deadline miss ratio. For these reasons, the majority of current component-based real-time systems are monolithic and, hence, are not dynamically reconfigurable, e.g., [5, 8]. However, reconfiguring a system on-line is desirable for embedded real-time systems that require continuous hardware and software upgrades in response to technological advancements, environmental change, or alteration of system goals during system operation. Consequently, there are strong reasons for enabling dynamic reconfiguration of real-time systems, but only under the condition that QoS can be guaranteed in the reconfigured system.

In our previous work we have shown that reconfigurability can indeed be achieved for firm and soft real-time systems, even though the execution time of the tasks vary when adding, removing, or changing components [7]. We addressed the problem of changing temporal properties of

* This work was funded, in part by CUGS (the National Graduate School in Computer Science, Sweden), Swedish Foundation for Strategic Research (SSF) via the SAVE project (SAfety critical component-based VEhicular systems), CENIIT (Center for Industrial Information Technology) under contract 01.07, and NSF grants IIS-0208578 and CCR-0329609.
Aleksandra Tesanovic is currently at Philips Research Laboratories, Eindhoven, The Netherlands, e-mail: aleksandra.tesanovic@philips.com.

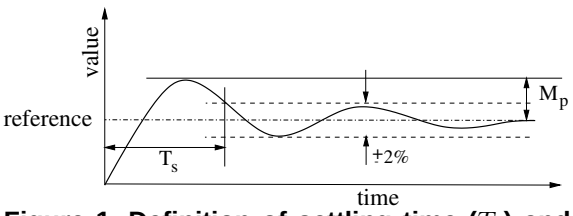


Figure 1. Definition of settling time (T_s) and overshoot (M_p)

tasks by using feedback control [3] to continuously force the QoS to converge toward a reference level, see Figure 1. Although the QoS of the system reaches the reference in the steady-state, significant overshoots (M_p) and long settling times (T_s) could be observed for reconfiguration instances that heavily affected the execution time and the arrival pattern of the tasks, e.g., we noted great overshoots in deadline miss ratio for reconfigurations that resulted in increased execution time of the tasks. However, for performance-critical soft and firm real-time systems, e.g., telecommunication, video streaming, and web services, it is of paramount importance that the overshoot and the settling time are within acceptable limits, as a large overshoot in deadline miss ratio may not be acceptable since too many deadlines are missed.

According to the discussions above and our previous studies [7], we have concluded that using feedback-based QoS management for guaranteeing the performance of dynamically reconfigurable real-time systems is beneficial but not enough. We need to ensure that the reconfiguration can take place without violating a given QoS specification in terms of a maximum tolerable overshoot and longest acceptable settling time. One way to ensure that the QoS does not violate the QoS specification is to determine the overshoot and settling time before the actual reconfiguration, i.e., to predict the overshoot and settling time, and only carry out the reconfiguration if it is possible to meet the given specification. If the QoS specification cannot be met, then the reconfiguration specification, i.e., the choice of components for adding, removing, or exchanging, has to be altered. This way the system is reconfigurable, meaning that the constituents of the software are alterable, and the reconfiguration can be safely executed.

The problem that we consider in this paper is how to predict the overshoot and settling time of QoS, defined by the deadline miss ratio $m(k)$ and utilization $u(k)$, in the face of a reconfiguration. We contribute by introducing a prediction framework that given a prediction specification consisting of (see Figure 4) (i) the desired QoS specification of the system in terms of the maximum tolerable overshoot and longest acceptable settling time for $m(k)$ and $u(k)$, (ii) the reconfiguration specification that contains the information about components that are going to be added, removed,

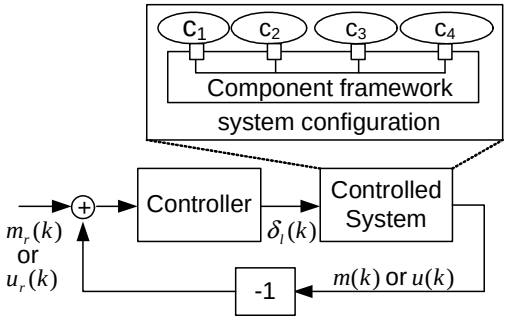


Figure 2. Feedback loop structure.

or exchanged in the system, and (iii) the prediction interface of the system that contains the specification of components and tasks that already exist in the system, is able to forecast the overshoot and the settling time of the reconfigured system. The framework returns a positive answer (i.e. OK) if the system can undergo the reconfiguration without violating its QoS specification, otherwise a negative answer (i.e. NotOK) is returned. The forecast is done using a method that predicts QoS over a finite horizon.[1] To the best of our knowledge, this is the first paper describing a QoS prediction methodology for firm reconfigurable real-time systems.

The remainder of this paper is organized as follows. In Section 2 we give an overview on dynamic reconfiguration of real-time systems. Section 3 presents the prediction framework, and in Section 4 we evaluate the accuracy of the prediction framework. Conclusions are given in Section 5.

2. Overview on Reconfiguration

This section briefly describes the reconfiguration mechanisms normally used by component-based systems residing in real-time and embedded environments. In this paper we focus on QoS prediction and, therefore, we refer to [1,7] for a detailed description on how reconfiguration is carried out.

To facilitate reconfiguration, component-based systems have a middleware layer that handles inter-component communication and stores states of components during system reconfiguration (see system configuration in Figure 2). Additionally, for ensuring substitutability of components, each component has functional interfaces, where a set of operations that a component provides to other components in the system is declared. A system can be reconfigured by (i) removing a component, (ii) adding a component, and (iii) exchanging a component with another. We illustrate what happens in a system under reconfiguration by giving a basic example where a current version of a component c_1, denoted c_1^1, is exchanged with a new version, c_1^2, in the sys-

1 By finite horizon we mean a finite interval of time in the future.

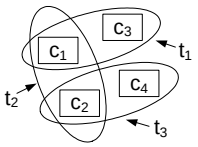


Figure 3. An example of the relation between tasks and components.

tem configuration in Figure 2. Note that, an exchange of a component encompasses all the dimensions of the dynamic reconfiguration as it also includes removal of the current version of a component and the addition of a new version. When the reconfiguration is initiated, the system undergoes three phases: pre-reconfiguration, reconfiguration, and post-reconfiguration phase.

In the pre-reconfiguration phase (pre-RP), the new version of the component c_1^2 is *loaded* into the memory. At this point, c_1^2 is not added to the system, but we load c_1^2 into the memory to carry out the reconfiguration as fast possible by minimizing the delay of removing c_1^1 and adding c_1^2. Now, we have to ensure that the system can undergo reconfiguration without interrupting the execution of the tasks. This means that tasks using the operations of component c_1^1 have to be completed before the reconfiguration is carried out. If the components $c_1, \ldots, c_4$ would be executed by the three tasks $t_1, \ldots, t_3$, as shown in Figure 3, then exchanging c_1^1 running on tasks t_1 and t_2 requires both of these tasks to complete executing before the system can enter the reconfiguration phase (RP).

Pre-RP ends when there are no executing tasks using c_1^1. Hence, at the beginning of RP the state of c_1^1 will not change since the state of c_1^1 changes only through operation calls. Therefore, we can now remove c_1^1 from the system by *exporting* its state into the middleware layer. The new component c_1^2 is then added to the system by *importing* the state of c_1^1, i.e., we transfer the state of c_1^1 to c_1^2. The pointers to operations of c_1^1 are redirected to the operations of c_1^2, hence, all future operations calls to c_1^1 are directed to c_1^2. Updating the operation pointers marks the end of RP. In the post-reconfiguration phase (post-RP), c_1^1 is *unloaded* from the memory, and tasks that are using the component c_1 are allowed to execute since c_1^2 has been loaded and added into the system. Note that the reconfiguration has to be completed as soon as possible for the system to resume normal operation.

3. Approach

In this section we present the assumed system and reconfiguration model, followed by the definition of the prediction specification, which gives how to specify the worst-case QoS, reconfiguration, and the knowledge of the components and tasks in the system. Finally, we give the method for predicting the overshoot and settling time of the controlled variable, which represents the QoS.

3.1. System and Reconfiguration Model

As the goal of this work is to develop a prediction method for determining the behavior of a component-based real-time system before the reconfiguration takes place, we first need to establish a model of the component-based real-time system. The model should be general enough to ensure that existing and future component-based real-time approaches are captured, and, consequently, can effectively use the presented prediction method. We first define a real-time system as a collection of interacting components and tasks, and then elaborate on each of the system constituents. Finally, we discuss the way reconfiguration is modeled in the system.

Definition 1 (System) *A component-based real-time system $\mathcal{R}$ is a tuple $\langle \mathcal{T}, \mathcal{C} \rangle$, where $\mathcal{T}$ is the set of all tasks that can execute in the system, and $\mathcal{C}$ is the set of components constituting the system.*

The following definition of a component captures the fact that a component provides operations to other components in the system and maintains an internal state, which needs to be preserved under reconfiguration.

Definition 2 (Component) *A component $c \in \mathcal{C}$ is a tuple $\langle \mathcal{S}, \mathcal{O} \rangle$, where $\mathcal{S}$ is the internal state of the component, and $\mathcal{O}$ is a set of operations $\{o_1, o_2, ..., o_P\}$ provided by component c.*

We now establish the definition of a task with regard to its necessary temporal characteristics and components it executes. We consider a firm real-time system as the controlled system, where there is one CPU as the main processing element. Hence, deadline misses are allowed, however, late results are of no value and are discarded. A task $\tau \in \mathcal{T}$ is classified as either a periodic or an aperiodic task and is defined as follows.

Definition 3 (Task) *Let $\mathcal{R}$=$\langle \mathcal{T}, \mathcal{C} \rangle$ be a real-time system. A task $\tau \in \mathcal{T}$ is a tuple $\langle x, \hat{x}, x^i, d, i, \hat{i} \rangle$, where x is the actual execution time, $\hat{x}$ is the estimated execution time, x^i is the internal execution time not related to calling operations, d is the relative deadline, i is the actual inter-arrival time, and $\hat{i}$ is the estimated inter-arrival time of τ.*

If $\tau \in \mathcal{T}$ is periodic, then the actual inter-arrival-time equals the estimated inter-arrival time, i.e., $i = \hat{i}$. If $\tau \in \mathcal{T}$ is aperiodic we define i as the mean inter-arrival-time, and $\hat{i}$ as the estimated mean inter-arrival-time. The actual execution time x of a task τ is not known in advance. We say that

a task completes when it has finished its execution, and we say that it terminates when it completes or misses its deadline. We now give the definition of an active task, which is needed later (in Section 3.3.2) for deriving the changes in load in response to a reconfiguration.

Definition 4 (Active Task) *Let $\tau \in \mathcal{T}$ be a task and let a denote the time when τ is admitted. The task τ is active at time t if and only if $a \leq t < a + \hat{i}$.*

We consider the following admission model. Upon arrival to the system, an instance of a task is inserted in an arrival queue, which is sorted according to the arrival time. Task instances are removed from the front of the arrival queue and admitted if and only if the sum of the task instance load and the load of admitted tasks is less than the requested load of admitted tasks. The deadline miss ratio is computed over admitted tasks only.

Without loss of generality we hereafter focus primarily on the reconfiguration of one component in the system; all introduced notions are easily applicable to an arbitrary number of components as we show in our experimental evaluations in Section 4. Recall that the system reconfiguration is of type add, remove, or exchange (of components). The reconfiguration, therefore transforms the original system $\mathcal{R}$ into the modified system $\mathcal{R}'$. We introduce ρ to denote the actions involved in the reconfiguration of the system. The impact of reconfiguration is straightforward. When adding or exchanging a component, a set of new tasks $\mathcal{T}^+$ may be added to the system. An exchange of component $c \in \mathcal{C}$ with c_ρ in $\mathcal{R} = \langle \mathcal{T}, \mathcal{C} \rangle$, would result in a new system $\mathcal{R}' = \langle \mathcal{T}', \mathcal{C}' \rangle$, where $\mathcal{T}' = \mathcal{T} \cup \mathcal{T}^+$ and $\mathcal{C}' = \mathcal{C} \cup \{c_\rho\} \setminus \{c\}$. To meet the requirements on urgency of reconfiguration discussed in Section 2, we assume that reconfiguration ρ of $\mathcal{R}$ is carried out by a separate non-preemptive task, τ_ρ, executing in the system with the highest priority. Let the start of the pre-RP, RP, and post-RP be denoted with the sampling instants k_{pre}, k_{rp}, and k_{post}. The reconfiguration task τ_ρ is released and executed at k_{rp} and τ_ρ completes at k_{post}. We consider post-RP, and therefore the overall reconfiguration, completed when the system reaches steady-state (see T_s in Figure 1), which occurs at the sampling instant k_{end}.

We assume the following feedback control scheduling architecture (see Figure 2). The controlled system is composed out of components forming a specific configuration. The deadline miss ratio $m(k)$ or the utilization $u(k)$ is controlled using the feedback structure. Let $y(k)$ denote the controlled variable, i.e., $m(k)$ or $u(k)$. Input to the controller is the difference between the reference $y_r(k)$, representing the desired performance of the controlled system, and the actual system performance given by the measured variable $y(k)$. Based on the performance error $y_r(k) - y(k)$ the controller computes a change $\delta_l(k)$ to the estimated admitted workload $l(k)$, such that the difference between the desired performance and the actual performance is minimized.

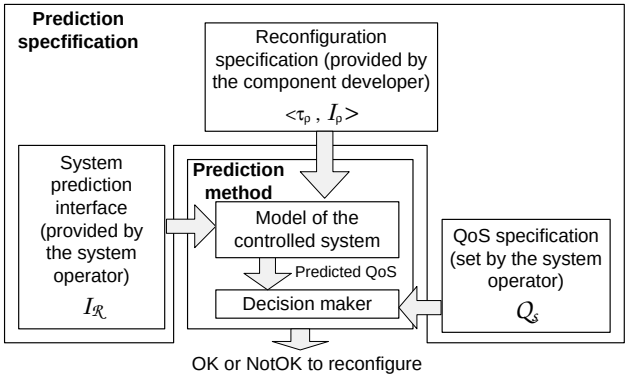


Figure 4. The prediction framework.

3.2. Prediction Specification

The prediction of the system behavior, as shown in Figure 4, is done using the specification consisting of (i) the system prediction interface $\mathcal{I}_\mathcal{R}$ that contains the necessary system information needed for reconfiguration prediction, (ii) the reconfiguration specification $\langle \tau_\rho, \mathcal{I}_\rho \rangle$ containing information about the reconfiguration in terms of the reconfiguration task τ_ρ and the component prediction interface $\mathcal{I}_\rho$ of the new component, and (iii) the QoS specification $\mathcal{Q}_S$, describing the worst-case tolerable system performance during a reconfiguration. In this section we give definitions of the constituents of the prediction specification as follows.

Definition 5 (Component Prediction Interface) *Let $\mathcal{R} = \langle \mathcal{T}, \mathcal{C} \rangle$ be a component-based real-time system. A prediction interface $\mathcal{I}_c$ of a component $c = \langle \mathcal{S}, \mathcal{O} \rangle \in \mathcal{C}$ is defined as a tuple $\langle \hat{x}_{o_1}, \hat{x}_{o_2}, \ldots, \hat{x}_{o_P} \rangle$, where $\hat{x}_{o_i}$ is the estimated execution time of the operation $o_i \in \mathcal{O}$ provided by c.*

To define the system prediction interface, we first define the prediction interface of a task as follows.

Definition 6 (Task Prediction Interface) *Let $\mathcal{R} = \langle \mathcal{T}, \mathcal{C} \rangle$ be a component-based real-time system. A prediction interface $\mathcal{I}_\tau$ of a task $\tau \in \mathcal{T}$ is defined by a tuple $\langle x^i, \hat{i}, \mathcal{C}_\tau, \mathcal{F} \rangle$, where x^i is the internal execution time of τ, $\hat{i}$ is the estimated inter-arrival time of τ, $\mathcal{C}_\tau \subseteq \mathcal{C}$ is the set of components executed by τ, and $\mathcal{F} = \{\langle o, n \rangle \mid o$ is an operation of some $c \in \mathcal{C}_\tau$, and n is the number of times τ calls $o\}$.*

Task prediction interface, therefore, captures the effects of mapping components to tasks. This is achieved by having the internal execution time x^i to account for the time it takes to execute the internal code of the task, i.e., the code handling the operations, and the function $\mathcal{F}$ that gives both

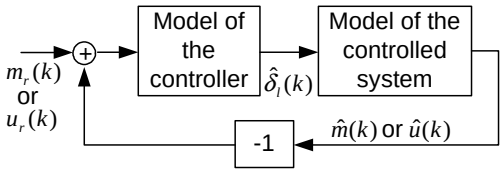


Figure 5. Prediction of QoS.

operations executed by the task and the number of times an operation is called by the task.

Definition 7 (System Prediction Interface) *Let $\mathcal{R} = \langle \mathcal{T}, \mathcal{C} \rangle$ be a component-based real-time system consisting of components $\mathcal{C}$, running on a set of tasks $\mathcal{T}$. Let each task $\tau_j \in \mathcal{T}$ be associated with a task prediction interface $\mathcal{I}_{\tau_j}$ and each component $c_k \in \mathcal{C}$ be associated with a component prediction interface $\mathcal{I}_{c_k}$. The prediction interface of a system $\mathcal{I}_\mathcal{R}$ is defined as a tuple $\langle \mathcal{I}_{\tau_1}, \ldots, \mathcal{I}_{\tau_N}, \mathcal{I}_{c_1}, \ldots, \mathcal{I}_{c_M} \rangle$, where N is the number of tasks and M is the number of components.*

The QoS specification is compared to the predicted overshoot and settling time of the utilization or deadline miss ratio in order to determine whether or not the predicted behavior is acceptable (as shown in Figure 4). The QoS specification is defined as follows.

Definition 8 (QoS specification) *Let $\mathcal{R} = \langle \mathcal{T}, \mathcal{C} \rangle$ be a component-based real-time system. A QoS specification $\mathcal{Q}_S$ is a member of $\{Q_M, Q_U\}$, where,*

- *$Q_M = \langle M_P^M, T_s^M \rangle$ gives the maximum overshoot M_P^M and the longest settling time T_s^M allowed for the deadline miss ratio, and*
- *$Q_U = \langle M_P^U, T_s^U \rangle$ gives the maximum overshoot M_P^U and the longest settling time T_s^U allowed for the utilization.*

Hence, $\mathcal{Q}_S$ is expressed in terms of deadline miss ratio or utilization. If the predicted overshoot and the settling time are less or equal to the overshoot and settling time specified in $\mathcal{Q}_S$, then the reconfiguration satisfies $\mathcal{Q}_S$. Otherwise, the reconfiguration does not satisfy $\mathcal{Q}_S$.

3.3. QoS Prediction Method

In this section we discuss the prediction method consisting of a decision maker and a model of the controlled system, see Figure 4. For the remainder of this paper, let T denote the prediction step and let $z(k)$ be the value of z at time kT.

3.3.1. Decision Maker The decision maker utilizes the information from the model of the controlled system to decide whether a reconfiguration satisfies a QoS specification $\mathcal{Q}_S$ (given by Definition 8). The model of the controlled system describes the effect of the manipulated variable $\delta_l(k)$ on the controlled variables deadline miss ratio $m(k)$ and utilization $u(k)$, i.e., the model predicts the deadline miss ratio or utilization given $\delta_l(k)$. Let $\hat{m}(k)$ denote the prediction of $m(k)$, and $\hat{u}(k)$ denote the prediction of $u(k)$. We insert the model of the controlled system into the feedback loop as shown in Figure 5, along with the model of the controller, i.e., a relation between the performance errors $m_r(k) - m(k)$ and $u_r(k) - u(k)$, and the manipulated variable $\delta_l(k)$. The decision maker then predicts the values of $m(k)$, $u(k)$, and $\delta_l(k)$, i.e., $\hat{m}(k)$, $\hat{u}(k)$, and $\hat{\delta}_l(k)$, for each sample k. This is done by starting with an initial value of the deadline miss ratio $\hat{m}(0)$ and utilization $\hat{u}(0)$, e.g., $\hat{m}(0) = 0$ and $\hat{u}(0) = 0$, and computing $\hat{\delta}_l(1)$. Having $\hat{\delta}_l(1)$, we can compute $\hat{m}(1)$ and $\hat{u}(1)$, thus, we obtain $\hat{\delta}_l(2)$, and so on. The computation of $\hat{\delta}_l(k)$, $\hat{m}(k)$, and $\hat{u}(k)$ is continued until the post-RP ends, which is marked by the point in time k_{end} when $\hat{m}(k)$ and $\hat{u}(k)$ have settled around the reference (see the definition of the settling time in Figure 1). By computing $\hat{m}(k)$ and $\hat{u}(k)$ for all k, we can also compute the predicted overshoot by finding the greatest $\hat{m}(k)$ and $\hat{u}(k)$. The predicted deadline miss ratio overshoot $\hat{M}_P^M$ and utilization overshoot $\hat{M}_P^U$ are then computed as the percentage by which $\hat{m}(k)$ and $\hat{u}(k)$ are greater than the corresponding reference, see Figure 1. The predicted settling time for deadline miss ratio $\hat{T}_s^M$ and utilization $\hat{T}_s^U$ are computed as the time it takes for $\hat{m}(k)$ and $\hat{u}(k)$ to settle around the reference, see Figure 1.

The decision maker returns OK if and only if the predicted overshoot and predicted settling time are smaller than the overshoot and settling time given by the QoS specification $\mathcal{Q}_S$. If $m(k)$ is the controlled variable, then OK is returned if and only if $\hat{M}_P^M \leq M_P^M$ and $\hat{T}_s^M \leq T_s^M$. If $u(k)$ is the controlled variable, then OK is returned if and only if $\hat{M}_P^U \leq M_P^U$ and $\hat{T}_s^U \leq T_s^U$. Otherwise NotOK is returned.

3.3.2. Model of the Controller and the Controlled System Any controller that can be realized in discrete time using difference equations (including non-linear equations) may be used in combination with the prediction method. This means that commonly used controllers such as proportional integral derivative (PID) controllers, state space controller, and linear quadratic controllers [3] are compatible with the prediction method. For example, a utilization PI controller, which computes the control signal according to $\delta_l(k) = \delta_l(k-1) + K_P((K_I + 1)e(k) - e(k-1))$, where $e(k) = u_r(k) - u(k)$, may be used since it is in discrete time form.

We now model the controlled system and we start with defining the estimated load of a task. Let the estimated load of a τ be $\hat{l} = \frac{\hat{x}}{\hat{i}}$, where $\hat{x}$ and $\hat{i}$ are the estimated execution time and inter-arrival time of τ, respectively. The esti-

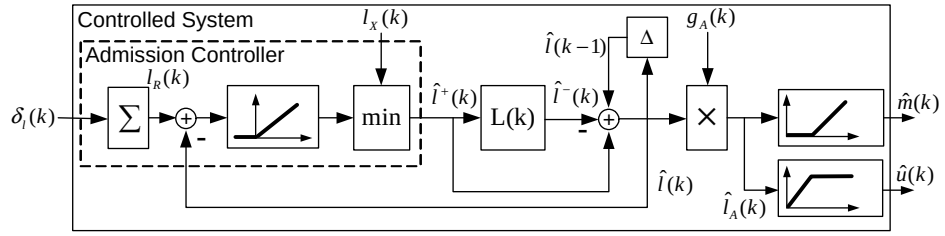


Figure 6. The Model of the Controlled System.

mated execution time of τ is computed as the sum of the internal execution time x^i of τ and the execution time $\hat{x}_o$ of each operation o called by τ. Recall that the task prediction interface (see Definition 6) of τ, gives the internal execution time x^i of τ and the number of times n a certain operation o is called by τ. The component prediction interface (see Definition 5) gives the estimated execution time $\hat{x}_o$ of the component operation o. Hence, by using the system prediction interface (see Definition 7), which consists of the task prediction interface and the component prediction interface, we can compute the estimated execution time of the tasks. This gives that the load is computed as follows:

$$\hat{l} = \frac{\hat{x}}{\hat{i}} = \frac{x^i + \sum_{\forall f \in \mathcal{F}} \hat{x}_o n}{\hat{i}}. \quad (1)$$

Figure 6 gives the model used for predicting QoS.[2] Let us start from the output from the controller, namely the desired change in the estimated admitted workload denoted with $\delta_l(k)$ (compare with Figure 2). The requested admitted workload $l_R(k)$ is the summation over $\delta_l(k)$, i.e., $l_R(k) = l_R(k-1) + \delta_l(k)$. The admission controller has to ensure that the level of admitted workload is kept at $l_R(k)$. Therefore the requested admitted workload is compared to the estimated admitted workload $\hat{l}(k)$ and the difference between the two is computed.[3] If $l_R(k)$ is less than $\hat{l}(k)$, then we have to reject all arriving tasks and wait until some of the admitted tasks are terminated, since we cannot reject any of the already admitted tasks. Recall from Section 2 that tasks using a component that is to be exchanged or removed are not eligible for execution. Therefore, if $l_R(k)$ is greater than $\hat{l}(k)$, then the amount of estimated workload that is admitted, denoted with $\hat{l}^+(k)$, is the minimum of the $l_R(k) - \hat{l}(k)$ and $l_X(k)$, where $l_X(k)$ is the total workload of tasks eligible for execution and that arrive during the time interval $[(k-1)T, kT]$. The workload $l_X(k)$ is derived from the system prediction interface, by summing the estimated workload of tasks that do not use the component that is to be removed or exchanged.

Let $\hat{l}^-(k)$ denote the aggregated workload of tasks that become inactive during the time interval $[(k-1)T, kT]$ (see Definition 4). Hence, $\hat{l}^-(k)$ models the amount of workload leaving the system during the time interval $[(k-1)T, kT]$. Now, the amount of admitted workload at time kT is the sum of the admitted load at time $(k-1)T$ and the amount of estimated workload admitted during $[(k-1)T, kT]$, minus the amount of estimated workload of tasks that leave the system during the time interval $[(k-1)T, kT]$, i.e.,

$$\hat{l}(k) = \hat{l}(k-1) + \hat{l}^+(k) - \hat{l}^-(k). \quad (2)$$

We note that $\hat{l}^+(k)$ and $\hat{l}^-(k)$ are related, since tasks that are admitted leave the system at some time in the future. The number of tasks leaving the system increases as the number of admitted tasks increases, consequently, an increase in $l^+(k)$ results in an increase in $l^-(k+b)$ for $b \geq 0$. We use $L(k)$ to relate the estimated workload that leaves the system and the estimated workload that is admitted into the system, i.e., $L(k)$ relates $\hat{l}^-(k)$ and the previously admitted workload given by $\hat{l}^+(k-b)$, where $b \geq 0$. The relation $L(k)$ is derived using the estimated load and the inter-arrival time of the tasks, which is obtained using the system prediction interface. For details on how to compute $L(k)$ we refer to [1]. At this point we have expressions for $\hat{l}^+(k)$ and $\hat{l}^-(k)$, and it is straightforward to compute the estimated load in the system $\hat{l}(k)$ by using equation (2).

A model describing the QoS of a reconfigurable system must encompass changes in the temporal properties of the tasks and, as such, the load in the system. After a reconfiguration, the actual load in the system may increase or decrease depending on the increase or decrease in the execution time of the tasks. Recall that using the system prediction interface we can compute the estimated execution time of the tasks using equation (1). Since we have the estimated execution time $\hat{x}_o$ of the operations of the new component, which is to be added or exchanged, we can compute the estimated execution time of the tasks after a reconfiguration, again using equation (1). Having the estimated execu-

2 Note, the model can be further reduced in size. However, a reduction implies a more detailed and complicated discussion and, thus, for the sake of simplicity, we have not reduced the model.

3 Since we do not have access to accurate execution times, we have to resort to estimations of execution time and, hence, we use the estimated admitted workload.

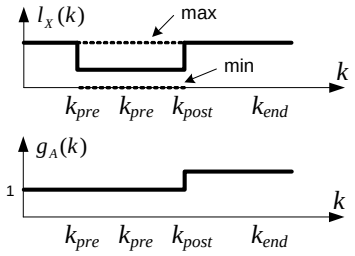


Figure 7. An example of how model variables vary during each RP.

tion times before and after a reconfiguration, we can compute the load of the tasks before and after the reconfiguration. We model the change in the total workload by introducing a factor $g_A(k)$ that gives the changes in load, based on the new component and the tasks that use it. Specifically, $g_A(k) = \frac{\hat{l}_A(k)}{\hat{l}(k)}$, where $\hat{l}_A(k)$ is the estimated admitted workload after RP and $\hat{l}(k)$ is the estimated admitted workload before and during RP. If a reconfiguration results in an increase in load, then $g_A(k) > 1$. Conversely, if a reconfiguration results in a decrease in load, then $g_A(k) < 1$.

We finally model the relationship between $\hat{l}_A(k)$, $\hat{m}(k)$, and $\hat{u}(k)$. We adopt the model presented by Lu et al. [4], which we briefly describe below. The relationship between the admitted workload $\hat{l}_A(k)$ and the utilization $\hat{u}(k)$ is nonlinear. As shown in Figure 6, when $\hat{l}_A(k)$ is less or equal to one, i.e. the CPU is underutilized, then $\hat{u}(k)$ is equal to $\hat{l}_A(k)$. However, when $\hat{l}_A(k)$ is greater than one, then $\hat{u}(k)$ remains at one, as $\hat{u}(k)$ per definition cannot be greater than one. Continuing with the deadline miss ratio $\hat{m}(k)$, let l_{tm} be the greatest workload threshold for which admitted tasks are schedulable. We note that $\hat{m}(k)$ is zero when $\hat{l}_A(k) \leq l_{tm}$, since all tasks meet their deadlines. However, when $\hat{l}_A(k) > l_{tm}$, then $\hat{m}(k)$ increases with $\hat{l}_A(k)$.

In summary we have developed a model of the controlled system, from the input $\delta_l(k)$ to the outputs $\hat{m}(k)$ and $\hat{u}(k)$. Next, we study how $l_X(k)$ and $g_A(k)$ vary during the reconfiguration phases, as well as how the length of each phase is predicted. An example of how $l_X(k)$ and $g_A(k)$ vary during each reconfiguration phase is given in Figure 7.

3.3.3. Reconfiguration Phases During the pre-RP the system has to make sure that admitted tasks do not use the component c that is to be removed or exchanged. This means that newly arrived tasks using c cannot be admitted and the system has to wait until previously admitted tasks that use c and are executing, will terminate. As such, the workload of executable tasks $l_X(k)$ decreases during pre-RP if at least one task uses c. This is shown in Figure 7, where tasks that use c must be rejected. The length of the pre-RP is the time it takes for admitted tasks that use c to terminate. When adding a new component c_ρ, no tasks have to be rejected, hence, $l_X(k)$ does not decrease. Considering $g_A(k)$, at this point the execution time and the load of the tasks have not changed, since the reconfiguration has not been implemented yet. Therefore, the variable $g_A(k)$ is one.

During the reconfiguration phase, the reconfiguration task τ_ρ is released and executed. This does not affect the set of executable tasks $l_X(k)$. Similarly, the execution time of the tasks do not change until k_{post}, hence, $g_A(k)$ does not change during RP and is equal to one.

During post-RP, the execution time of the tasks changes due to the reconfiguration. Therefore, $g_A(k)$ may increase or decrease according to the discussion in Section 3.3.2. If a component is exchanged, then the tasks that were rejected during the pre-RP and RP are eligible for execution again during post-RP. Therefore, if $l_X(k)$ decreases during pre-RP, then during post-RP $l_X(k)$ increases to the same level as before pre-RP, as illustrated in Figure 7. For component removal, $l_X(k)$ stays unchanged and is equal to $l_X(k)$ during pre-RP and RP.

4. Performance Evaluation

The goal of the performance evaluation is to determine the accuracy of the prediction method (shown in Figure 4). The reliability of the decision maker, in terms of providing a correct answer, increases as the accuracy of the controlled system model increases. Below we describe the method we use to evaluate the precision of the controlled system model, followed by the simulation setup and the result of the experiments.

4.1. Evaluation Method

We validate the model of the controlled system by establishing the difference between the predictions and the actual outcome of the controlled system (i.e., QoS) when carrying out reconfigurations under feedback control. If the difference between the predictions and the actual outcome is small, then we say that the model of the controlled system is accurate.

The performance evaluation is undertaken by a set of simulation experiments where we have varied the increase in execution time after a reconfiguration and the number of tasks in the system. Considering the first parameter, we stated earlier in Section 1 that a reconfiguration may result in an increase in the execution time of the tasks, causing overshoots in deadline miss ratio and utilization. Since our aim is to derive a method for predicting the overshoot and settling time, we consider the case when the execution time

increases as a result of a reconfiguration. Further, the behavior of a real-time system depends on the number of tasks in the system. Therefore, we study the prediction when the number of tasks are varied.

Recall from Section 1 that an exchange of a component encompasses all the dimensions of the dynamic reconfiguration as it includes also a removal of the old version of a component and the addition of a new version. Therefore, we evaluate the prediction method when exchanging components, as the exchange procedure is more intricate and has greater effects on the QoS. Further, in the experiments we define QoS in terms of utilization, as utilization gives us more information when analyzing the results. For example, it is easier to estimate the increase or decrease in load when analyzing utilization rather than deadline miss ratio. We refer to [1] for experiments on removing and adding components, and prediction of deadline miss ratio.

4.2. Simulation Setup

In the experiments we use two different task sets denoted $\mathcal{T}^A$ and $\mathcal{T}^B$, see Table 1 (U refers to a uniform distribution). Furthermore, each one of these task sets consists of two subsets, where we denote the individual subsets by appending a subscript to the name of the task set, e.g., $\mathcal{T}_1^A$ and $\mathcal{T}_2^A$ denote the two subsets of $\mathcal{T}^A$. We consider a task τ to have the following properties: estimated execution time in the pre-RP $\hat{x}^{\text{PRE}}$, estimated execution time in the post-RP $\hat{x}^{\text{POST}}$, estimated inter-arrival time, $\hat{i}$, and the number of tasks N in the task set. The actual execution time of τ in the pre-RP and post-RP is distributed according to $U : (\hat{x}^{\text{PRE}}(1 - x_V),\ \hat{x}^{\text{PRE}}(1 + x_V))$ and $U : (\hat{x}^{\text{POST}}(1 - x_V),\ \hat{x}^{\text{POST}}(1 + x_V))$, respectively. The variation x_V introduces additional uncertainty in the actual execution times, thus, making the prediction more challenging. Similarly, the actual inter-arrival time is distributed as $U : (\hat{i}(1 - i_V),\ \hat{i}(1 + i_V))$, where i_V is uniformly distributed. We set the execution time of τ_ρ to $0.030s$ (see [1] for more details). The tasks used in this evaluation have varying actual execution times, where the actual execution times are not known. Further, the arrival times of the tasks are unpredictable, which in combination with inaccurate execution time estimates, simulate a realistic real-time system.

The prediction step T is set to $0.1s$ for all experiments, i.e., the admission controller is invoked and $\hat{u}(k)$ computed every $0.1s$. The sampling period of the utilization is set to $10s$, i.e., the utilization is sampled and the controller invoked every $10s$. Earliest deadline first (EDF) is used to schedule the tasks (e.g. see [2]). To quantify overshoots we need to set the reference such that $u(k)$ is less than 100% at all times. If we set the utilization close to 100%, then the utilization stays 100% when $u(k)$ overshoots, hence, we cannot see the extent of the overshoot. Therefore for the purpose of the evaluation we set the utilization reference to 70%, which gives us some slack to 100%. This way we are able to fully examine the accuracy of the controlled system model.

Using the task set $\mathcal{T}^A$ and $\mathcal{T}^B$ we are able to evaluate the prediction method when the execution time of the tasks increases after RP (compare the distribution of $\hat{x}^{\text{PRE}}$ and $\hat{x}^{\text{POST}}$). An increase in execution time represents the worst-case scenario since this causes overshoots in deadline miss ratio and utilization. Hence, by increasing the execution time, we evaluate the prediction capability of the model during a system state that gives overshoots. Also, by increasing the execution time of all the tasks we simulate the case when multiple components are exchanged, since exchanging several components in the worst case increases the execution time of all tasks. Further, $\mathcal{T}^A$ and $\mathcal{T}^B$ differ in that the number of tasks in task set $\mathcal{T}^A$ is less than the number of tasks in $\mathcal{T}^B$. Hence, we also capture the case when the number of tasks in the system vary. This means that using $\mathcal{T}^A$ and $\mathcal{T}^B$, we satisfy the goal of the performance evaluation.

4.3. Results

In this section we evaluate the accuracy of the controlled system model and we show that the difference between the predictions and the actual outcome of the QoS is very small, i.e., we achieve very accurate predictions.

Figures 8(a) and 8(b) show the results when $\mathcal{T}^A$ and $\mathcal{T}^B$ are used. The grey lines represent the actual QoS outcome, while the dark dashed lines represent the QoS prediction obtained from the model of the controlled system. At time $10s$ we start the pre-RP. Pre-RP and RP are rather short, since allowing the tasks, which use the component under exchange, to terminate and executing τ_ρ is carried out quickly. At the beginning of the post-RP the system is reconfigured, which results in an increase of the execution time of the tasks. Consequently, the load of the tasks increases, causing an overshoot in utilization. This can be seen by studying the actual QoS outcome in Figures 8(a) and 8(b).

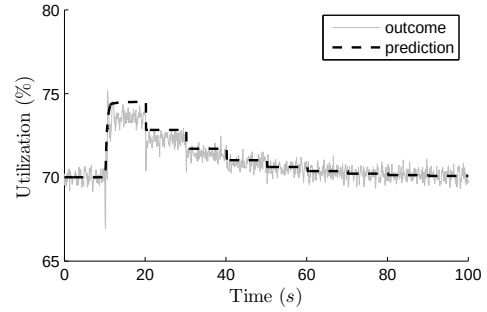
We also see that the utilization drops at times $20s$, $30s$, $40s$, and so on. We have found out that this is caused by the admission controller, trying to reduce the workload such that $\hat{l}(k)$ equals $l_R(k)$, see Figure 6 (refer to [1] for details). The drops in utilization at times $20s$, $30s$, and $40s$ are not important for the model to capture since we are focusing on overshoots and settling times of utilization and deadline miss ratio. We can therefore ignore modeling this particular behavior.

Considering the predictions in Figures 8(a) and 8(b), we see that the predicted QoS, represented by the dark dashed lines, follows the actual outcome closely. This means that the model of the controlled system is very accurate in cap-

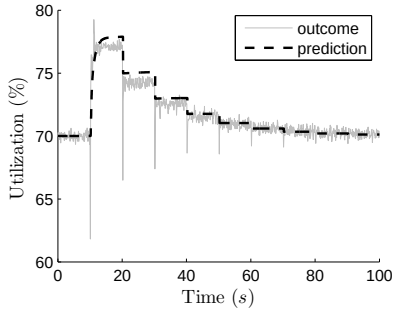
Property	$\mathcal{T}_1^A$	$\mathcal{T}_2^A$
$\hat{x}^{\text{PRE}}$ (s)	$U : (0.020,\ 0.050)$	$U : (0.010,\ 0.020)$
$\hat{x}^{\text{POST}}$ (s)	$U : (0.040,\ 0.050)$	$U : (0.015,\ 0.022)$
x_V	$U : (0.10,\ 0.20)$	$U : (0.10,\ 0.20)$
$\hat{i}$ (s)	$U : (0.2,\ 1.5)$	$U : (0.5,\ 10)$
i_V	0	$U : (0.03,\ 0.1)$
N	20	15

Property	$\mathcal{T}_1^B$	$\mathcal{T}_2^B$
$\hat{x}^{\text{PRE}}$ (s)	$U : (0.015,\ 0.025)$	$U : (0.020,\ 0.050)$
$\hat{x}^{\text{POST}}$ (s)	$U : (0.018,\ 0.028)$	$U : (0.030,\ 0.060)$
x_V	$U : (0.10,\ 0.20)$	$U : (0.10,\ 0.20)$
$\hat{i}$ (s)	$U : (0.2,\ 5)$	$U : (0.1,\ 50)$
i_V	$U : (0.01,\ .1)$	$U : (0.01,\ 0.1)$
N	80	40

Table 1. Task sets used in the experiments.



(a) Task set $\mathcal{T}^A$



(b) Task set $\mathcal{T}^B$

Figure 8. Prediction of the utilization.

turing the behavior of QoS when carrying out reconfigurations. An accurate model implies that the decision maker is fed with a prediction of high quality, and that the output from the decision maker is of high confidence. Thus, our evaluation shows that the model of the controlled system is accurate and, as such, the result of the decision maker is very reliable.

5. Conclusions

In this paper we introduced a framework for predicting the QoS of a component-based firm real-time system that is about to undergo dynamic reconfiguration, i.e., adding, removing, or exchanging components on-line. The framework ensures that reconfigurations can take place without violating a given QoS specification of a system in terms of a maximum tolerable overshoot and longest acceptable settling time. This is done by determining, i.e., predicting, the overshoot and settling time before an actual reconfiguration is made. The framework returns a positive answer if the system can undergo the reconfiguration without violating its QoS specification, or a negative answer if the QoS specification cannot be satisfied. With our framework we enable dynamic reconfiguration of component-based firm real-time systems, by making it possible to guarantee system performance when adding, removing, and exchanging components.

In our future work we intend to develop a feedback-feedforward control structure where we use the predicted QoS to proact, rather than react to changes in QoS due to a reconfiguration. This way we are able to improve QoS by suppressing overshoots and achieve lower settling time.

References

[1] M. Amirijoo, A. Tesanovic, T. Andersson, J. Hansson, and S. H. Son. Finite horizon QoS prediction of reconfigurable firm real-time systems. Technical Report XYZ, University of Virginia, Computer Science Department, 2006.

[2] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[3] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[4] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Real-time Systems*, 23(1/2), July/September 2002.

[5] D. B. Stewart, R. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12), 1997.

[6] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[7] A. Tesanovic, M. Amirijoo, D. Nilsson, H. Norin, and J. Hansson. Ensuring real-time performance guarantees in dynamically reconfigurable embedded systems. In *IFIP Conference on Embedded And Ubiquitous Computing (EUC)*, 2005.

[8] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th international conference on Software engineering*, pages 255–265, Orlando, Florida, USA, May 2002. ACM Press.