A Feedback Scheduling Framework for Component-Based Soft Real-Time Systems

Nima Khalilzad[1], Fanxin Kong[2], Xue Liu[2], Moris Behnam[1], Thomas Nolte[1]
[1]MRTC/Mälardalen University, Sweden
[2]McGill University, Canada
nima.m.khalilzad@mdh.se

***Abstract*—Component-based software systems with real-time requirements are often scheduled using processor reservation techniques. Such techniques have mainly evolved around hard real-time systems in which worst-case resource demands are considered for the reservations. In soft real-time systems, reserving the processors based on the worst-case demands results in unnecessary over-allocations.**

In this paper, targeting soft real-time systems running on multiprocessor platforms, we focus on components for which processor demand varies during run-time. We propose a feedback scheduling framework where processor reservations are used for scheduling components. The reservation bandwidths as well as the reservation periods are adapted using MIMO LQR controllers. We provide an allocation mechanism for distributing components over processors. The proposed framework is implemented in the TrueTime simulation tool for system identification. We use a case study to investigate the performance of our framework in the simulation tool. Finally, the framework is implemented in the Linux kernel for practical evaluations. The evaluation results suggest that the framework can efficiently adapt the reservation parameters during run-time by imposing negligible overhead.

I. Introduction

Multiprocessors are becoming increasingly more widespread computing platforms. Thanks to the computational capacity of the multiprocessors, previously segregated software systems can now be integrated on a shared hardware platform. Component-Based Software Engineering (CBSE) provides a modular approach for designing and developing complex software systems. CBSE provides means and techniques for integration of independently developed software components.

When it comes to real-time systems, timing constraints of software components have to be considered at the integration phase. We consider component models in which a real-time software component corresponds to a set of real-time tasks. A component also has an intra-component scheduler which coordinates task executions. Processor reservation and hierarchical scheduling techniques are often used to provide timing guarantees to the components in component-based systems (e.g., [1], [2]). Therefore, from the real-time scheduling perspective, the problem of component integration is reduced to creating adequate processor reservations for hosting the components.

Real-time tasks can either have hard deadlines where deadline misses are absolutely unacceptable or soft deadlines where occasional deadline misses can be tolerated. A hard real-time component is a component with hard real-time tasks. The size of processor reservations assigned to the hard real-time components is derived from the Worst-Case Execution Time (WCET) of the component's inner tasks. For instance in [3] and [4], targeting multiprocessor platforms, the authors provided analysis frameworks in which the reservation properties are extracted from intra-component schedulers and task parameters. Such analyses result in pessimistic allocations. The over-allocation is due to two reasons. Firstly, WCET is unlikely to happen in reality. Secondly, the analysis that derives the processor reservation sizes based on the WCET of tasks is pessimistic. Soft real-time components are software components consisting of soft real-time tasks. When integrating soft real-time components, pessimistic allocations are not justifiable. This is because pessimistic allocations do not permit an efficient processor utilization. In addition, in a group of soft real-time tasks the processor demand is subjected to large variations during run-time. For instance, the execution time of video decoder tasks can significantly vary depending on the content of the video frames. As a result, the processor demand of a real-time component consisting of such dynamic tasks may change significantly during run-time. Therefore, assigning a fixed-size processor reservation (for instance based on the average processor demands) results in an unacceptable number of timing violations.

Adaptive reservation techniques are widely used in single-processor platforms for scheduling soft real-time tasks with dynamic execution times (e.g., [5], [6]). In this paper, however, we focus on soft real-time components integrated on multiprocessor platforms. In our model, the components may be spread over multiple processors. As a result, the component's inner tasks are scheduled using a multiprocessor global scheduling algorithm. We propose a feedback scheduling framework which is built upon adaptive reservations. In our framework, the component demand is monitored during run-time. The processors reservations hosting the component are, then, adjusted according to the current demand. The processor allocations are also reconfigured to cope with the current state of the components. More specifically, in this paper, we present the following contributions: (i) a feedback scheduling scheme that uses Multiple Input Multiple Output (MIMO) controllers to regulate both period and budget of the periodic servers simultaneously (ii) an approximate model of the reservation dynamics through system identification (iii) a component allocation heuristic that maps software components to the processors and evaluating it against the optimal solution (iv) optimal compression algorithms that provide compressed bandwidths in overload situations (v) simulation-based evaluation of our MIMO controllers in TrueTime (vi) implementation and evaluation of our framework in the Linux kernel.

II. Preliminaries

System model. We assume a multiprocessor platform consisting of M identical processors. n components are running on the multiprocessor platform. We consider an open system in which components are allowed to join and/or leave the platform. As a result, n varies in run-time. The set of components

which are active in the system at any given time t is denoted using $\Gamma(t)$.

We target a component-based software development model in which the following two roles are defined: (i) component developer (ii) system integrator. The component developer is responsible for developing real-time tasks and selecting an appropriate scheduling policy for them. Then, the component requirements are abstracted using a number of interface parameters. When it comes to components with hard real-time requirements, a component interface represents the minimum amount of resource needed for guaranteeing the schedulability of the component. Such an interface is calculated using the WCET of the component's inner tasks. In our framework, however, we are targeting soft real-time components with dynamic workloads. Therefore, a component interface expresses an interval in which the processor demand of the component may vary during run-time. Basically, instead of the worst-case resource demands, the aggregate behavior of all tasks with respect to the processor requirement is expressed in the component interface. The system integrator, on the other hand, receives a number of components and he/she is responsible for integrating the components such that the requirements specified in the interface parameters are respected. The integrators' responsibility involves (i) identifying an approximate model of the component's resource requirements within its operating region (ii) designing controllers that adapt the resource provisions to the components during run-time. In this paper, we focus on the component integration.

Component model. Component $\mathfrak{C}_j$ consists of a number of real-time tasks, where $j \in [1, n]$ is the index of the particular component. The components also have an intra-component scheduler which is responsible for scheduling component's inner task. The relative importance of components with respect to the other components that are composed together on one platform is represented by ζ_j. The importance value is used when the system is overloaded. In such a situation, the components that can better contribute to the overall value of the system are preferred to the ones that have less impact on the total system value. Components can be assigned to one or more processors. We use period and bandwidth for specifying processor requirements of components. The bandwidth indicates the processor portion that a component requires, while the period indicates the granularity of the CPU provisioning. The component developers specify the operating range of their components, that is, a processor demand interval that the component will operate at run-time. The bandwidth requirement is denoted using $\bar{\alpha_j}$ and σ_{α_j}, where $\bar{\alpha_j}$ is the operating bandwidth and σ_{α_j} indicates the amount of deviation from the operating bandwidth. Similarly, the period requirement is specified using an operating period $\bar{T_j}$ and its deviation σ_{T_j}. The component interface $< \bar{\alpha_j},\ \bar{T_j},\ \sigma_{\alpha_j},\ \sigma_{T_j},\ \zeta_j >$ denotes that the component will require a bandwidth between $\bar{\alpha_j} - \sigma_{\alpha_j}/2$ and $\bar{\alpha_j} + \sigma_{\alpha_j}/2$. Similarly, the period may be changed from $\bar{T_j} - \sigma_{T_j}/2$ to $\bar{T_j} + \sigma_{T_j}/2$. The system integrator develops a model in the operating range of the component that is used for adaptation purposes. Note that $\bar{\alpha_j}$ and $\bar{T_j}$ do not need to be exact values, rather they are estimations of the component's processor requirements. We will adapt the resource provisioning to the components during run-time to compensate for the resource requirement estimation errors.

Task model. Our scheme supports periodic/sporadic task models τ_i<p_i, $c_i(l)$, D_i>, where i is the index of the particular task, p_i is the task period or the minimum inter-arrival time, $c_i(l)$ is the execution cost of the l^{th} instance of the task and D_i is the task deadline. Each instance of task execution is called a job. Note that the execution cost of tasks is time-varying and may be different from job to job. Throughout the paper and for simplicity we use an implicit deadline periodic task model, i.e. $p_i = D_i$. We do not assume any predefined execution cost $c_i(l)$ for tasks, however, we assume a task is not allowed to run in parallel, hence $\forall t\ c_i(l) \leq p_i$. The jobs of a task are executed sequentially, i.e., each job of a task is only allowed to run if all of the previous jobs of the same task have finished their executions. When tasks miss their deadlines, they continue their execution until the end. The goal of our framework is to provide a predictable Quality of Service (QoS) to the tasks, while efficiently utilizing the processor capacity. We use the number of deadline violations as a metric for measuring the QoS.

Virtual clusters and virtual processors. The computation capacity of the multiprocessor platform becomes available to the components through Virtual Clusters (VC). A particular VC i, denoted by Π_i, is a set of Virtual Processors (VP) $\Pi_i = \{\pi_{i,1}, \pi_{i,2}, \cdots\}$, where $\pi_{i,j}$ is the j^{th} VP of Π_i. A VP is created by partitioning a single physical processor in time. We use idling periodic servers compatible with the periodic resource model [7] for partitioning a single physical processor. When the server is active while there is no ready task to run, the idling servers idle their budget. The deadline of servers implementing the VPs is assumed to be equal to their corresponding periods. $\pi_{i,j}$ receives $q_{i,j}$ units of the physical processor time every T_i time units, where $q_{i,j} \leq T_i$. The periods of all VPs belonging to Π_i is equal to T_i. The bandwidth of a VP is defined as $\rho_{i,j} = q_{i,j}/T_i$. We assume Π_i can have at most one VP on any given physical processor. Π_i receives B_i time units every T_i units, where $B_i = \sum_{j \in [1 \ldots M]} q_{i,j}$. In this summation, we assume $q_{i,j} = 0$ for the case where the VC has less than M VPs and $\pi_{i,j}$ does not exist. The bandwidth of Π_i is defined as the following: $\alpha_i = B_i/T_i$. We have n VCs hosting n components at each point in time, i.e., the number of VCs in the system is equal to the number of components n.

Multiple VPs that belong to distinct VCs may share a physical processor. We use the partitioned EDF scheduling algorithm for scheduling the VPs. For scheduling the tasks within the components, however, we use a global multiprocessor scheduler. In other words, when a VC is spread over multiple processors, tasks within the VC may migrate from a processor to another processor. The intra-cluster scheduler (task scheduler) can be either global fixed-priority or global EDF. Considering the two levels of scheduling, our scheme can be seen as a two-level hierarchical scheduling framework.

Operational modes. We consider the following two mutually exclusive operating modes for the system: *normal mode* and *overload mode*. In the normal mode the components can receive their desired processor bandwidths because the total required processor is less than the available processor time, i.e., $\sum_{i \in \Gamma} \alpha_i \leq M$. In the normal mode we use a number of independent MIMO controllers to regulate the bandwidths and the periods of the VCs. In the overload mode, however, the total required bandwidth is larger than the available processor capacity. In this mode the system will suffer, i.e., real-time tasks will inevitably miss their deadlines. Our goal, in the overload mode, is to distribute the total bandwidth among

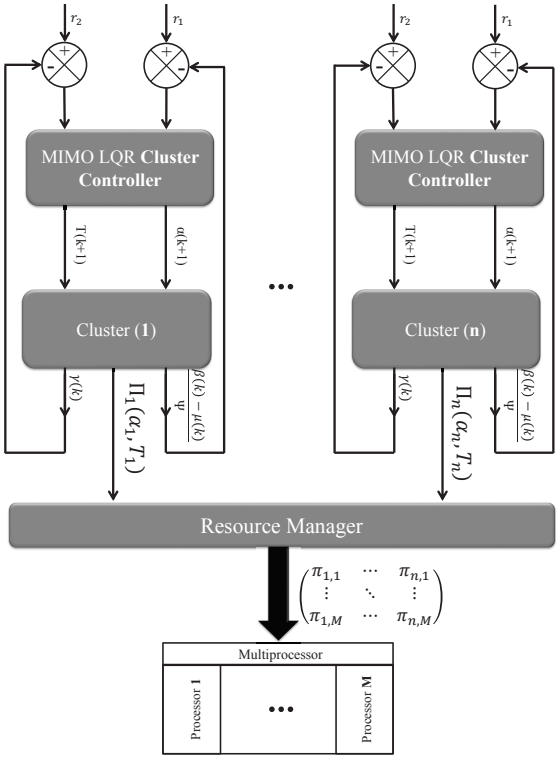


Fig. 1: The architecture of our adaptive component-based scheduling framework.

components in such a way that the overall value of the system is maximized. Therefore, utilizing the importance value of the components (ζ_i for each component $\mathfrak{C}_i$), we use a centralized controller to distribute the total bandwidth among components. If a system operates in the overload mode most of the time, then the system is poorly designed and the integrator should remove some of the components to reduce the load. We assume that the overload mode happens transiently, and the system mostly operates in the normal mode.

Overview of the framework. Figure 1 depicts the architecture of our adaptive framework. The framework is comprised of two types of elements: (1) cluster controllers (2) a resource manager. The cluster controllers monitor the state of the VCs and adapt their bandwidths and periods to deal with components' dynamic resource requirements. The cluster controllers are designed using control theory. Section III describes the cluster controllers in detail. The resource manager, on the other hand, is responsible for allocating components on the processors. The resource manager receives n VCs and it allocates each VC on a number of VPc. The resource manager adds, removes and adjusts VPs dynamically to respond to the needs of the VCs. Section IV addresses the design of the resource manager.

III. Modeling and Design of Cluster Controllers

In this section we focus on adapting the parameters of a single VC serving a component. Therefore, for simplicity, we drop index i when referring to parameters associated with Π_i. Throughout this section we assume that the system is in the normal mode. The cluster dynamics are sampled and adapted periodically. The sampling time is denoted using k. The time distance of two consecutive samples is referred as a *sampling interval* and its length is denoted using Ψ.

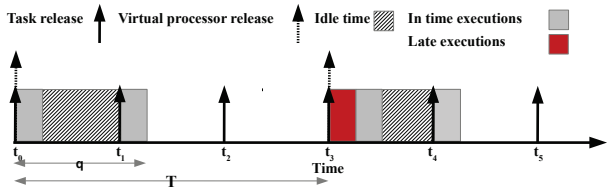


Fig. 2: Alignment problem: VP's period is not aligned with the task period. Task is released at time t_2 while the VP is inactive. The task has to wait until the next VP release which is after task's deadline. Therefore, the task misses its deadline while the VP budget is idled.

In control theory, control inputs are variables that are used for manipulating the plant. We consider the VCs as our plant. Our objective is to make sure that the VCs provide sufficient processor capacities to the components at each point in time. Therefore, we choose T and α as our control inputs. We use the parameters expressed in the component's interface as the operating points and we take the distance from the operating points as our control inputs. Therefore we have:

$$\mathbf{u}(k) = \begin{bmatrix} \alpha(k) - \bar{\alpha} \\ T(k) - \bar{T} \end{bmatrix},$$

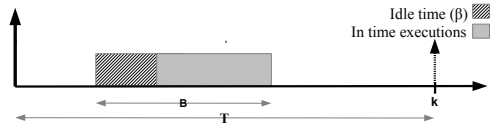
where $\mathbf{u}(k)$ is the control input at sampling time k. We construct our model around the operating points of the system. The reason behind using the operating points is that, the plant's behavior can be approximated in the vicinity of these points using a linear model.

A. Why should the cluster periods be adapted?

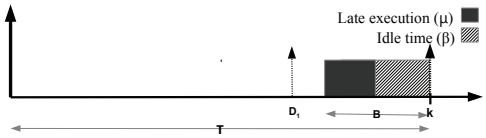
At first glance it might appear that changing the VC bandwidths through adapting their budgets might be sufficient. However, there are a number of good reasons for adapting the VC periods as well. Let's first discuss the problems associated with two extremes of period assignment, i.e., extremely short periods and extremely large periods. As the VC period decreases, the number of preemptions in a given time interval increases. Therefore, considering the overhead penalty associated with preemptions, it is desirable to assign periods as large as possible. Extremely large periods, on the other hand, impose insignificant overhead. However, when tasks are faster than their VPs, the VP bandwidths have to be significantly larger than the task set's processor utilization, because the budget provisioning may not be aligned with the task executions. We refer to this problem as the *alignment problem* which is illustrated in Figure 2. The importance of the granularity of a resource partition is also studied in [8]. In addition, in case of sporadic task models, the tasks may occasionally use their minimum inter-arrival times. Hence, assigning the VC period based on the minimum inter-arrival times will impose unnecessary overhead to the system.

Measurable variables. For controlling the VC parameters, we need a number of variables that can describe the dynamics of the VCs. We should choose parameters that (i) can be easily measured (ii) be an indication of the workload and task frequencies. In fact, we consider the changes in workload as a disturbance and our control objective is to compensate for it.

The cluster Π is assigned $B(k)$ time units every $T(k)$ time units. It idles $\beta(k)$ time units of its budget due to unavailability of workload and utilizes the rest of its budget ($B(k) - \beta(k)$). Tasks inside the VC either finish their executions before their deadlines or after them. The part of task's execution time



(a) Idle time (β).



(b) Late execution (μ).

Fig. 3: Visualization of the measurable variables.

executed after task's deadline is called a *late execution*. The part of the VC budget consumed by the late executions is denoted by $\mu(k)$. At any sampling time k, the cluster controller can measure $\beta(k)$ and $\mu(k)$. Note that $\beta(k)$ and $\mu(k)$ are respectively the aggregate values of the idled budget and the late executions happened in a sampling interval. These parameters are illustrated in Figure 3a and Figure 3b. The number of jobs that have missed their deadlines is another variable that can be monitored by the cluster controller. The number of deadline misses happened between sampling times $k-1$ and k is denoted by $\gamma(k)$.

State variables. We intend to use a linear model for modeling the dynamics between the inputs and the state variables. Thus, we are interested in variables that their changes, with respect to the changes of the inputs, are as close as possible to linear. Since both $\beta(k)$ and $\mu(k)$ are saturated at zero we use the following linear combination of them as our first state variable: $x_1(k) = (\beta(k) - \mu(k))/\Psi$. Note that this state variable is normalized by the sampling length Ψ. The processor resource over-allocation ($x_1(k) > 0$) and under-allocation ($x_1(k) < 0$) to the components is revealed by $x_1(k)$. However, when the idle time is equal to the late execution time ($\beta(k) = \mu(k)$), or when the late execution time is significantly smaller than the idle time ($\mu(k) \ll \beta(k)$), components may suffer from deadline misses while $x_1(k)$ is not revealing the state of the VCs. To address this problem, we choose to further monitor the number of deadline misses happened in a sampling interval. Thus, the second state variable is: $x_2(k) = \gamma(k)$. This variable can further express the state of VCs when $x_1(k)$ is not expressive. In summary, we use the following state variables:

$$\mathbf{x}(k) = \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} = \begin{bmatrix} \frac{\beta(k)-\mu(k)}{\Psi} \\ \gamma(k) \end{bmatrix}.$$

Suppose that $\alpha^*(k)$ and $T^*(k)$ are a bandwidth and a period in which $x_1(k) = r_1$ and $x_2(k) = r_2$, where r_1 and r_2 are desired values of $x_1(k)$ and $x_2(k)$ respectively. Assuming that $T(k) = T^*(k)$ if $\alpha(k) > \alpha^*(k)$ the VC will waste some of its budget, hence $x_1(k) > 0$. If $\alpha(k) < \alpha^*(k)$ the VC will suffer from a budget deficiency and $x_1(k) < 0, x_2(k) > 0$. Assuming that $\alpha(k) = \alpha^*(k)$, if $T(k) > T^*(k)$ the VC will suffer from the alignment problem, therefore $x_1(k), x_2(k) > 0$. If $T(k) < T^*(k)$, the VC will impose some overhead due to short periods and $x_2(k) < 0$. Note that in order to detect this case, i.e., $T(k) < T^*(k)$ we have to set r_2 to a small number greater than zero. As discussed above, the designed state variables reveal the internal states of the VCs.

B. Modeling the cluster dynamics

We are interested in deriving a model which captures the relation between the control inputs and the state variables. Throughout our experiments we observed that this relation is not linear due to (i) queuing effects of **u** on **x** (ii) saturation of **x**. The queuing effect is due to task scheduling. For instance, increasing the VC bandwidth does not necessarily reduce the number of deadline misses. This is because some tasks may have backlogs from the previous sampling interval. Therefore, increasing the bandwidth will allow them to execute in the next sampling interval. Saturation of **x** happens due to the nature of our system. For example, at most all of jobs of all tasks within the VC can miss their deadlines. Therefore, in such a condition that all jobs miss their deadlines, decreasing the bandwidth will not have any influence in the number of deadline misses. Despite the non-linearity nature of our system, linear models often work well for nonlinear systems specially when the purpose is to regulate the system output based on a number of control inputs [9]. We use the so called "black box" approach for modeling the relation between **u** and **x**. We employ the Auto-Regressive with eXogenous variables (ARX) model to describe the relation between the state variables and the inputs. Therefore, the state space system model is as follows:

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k), \end{aligned} \tag{1}$$

where **A**, **B** and **C** are 2×2 matrices, $\mathbf{u}(k)$ is the control input, $\mathbf{y}(k)$ is the system output, and for simplicity we assume $\mathbf{C} = \mathbf{I}$ (**I** is the identity matrix). Matrix **A** indicates the dependency between the next value of outputs to their previous values. Matrix **B**, however, expresses the functional dependency between the control inputs and the system outputs.

C. System identification

We use system identification for identifying matrices **A** and **B** of the model presented in Eq. 1. System identification uses statistical tools to estimate the model parameters. The identification processor is as follows. First the components are executed on the target hardware platform. The control inputs (**u**) are modified throughout the execution of the components and the system outputs (**y**) are noted. Finally, parameter estimation is performed given **u** and **y**. Similar to WCET analysis which is hardware dependent, the identified parameters may depend on the characteristics of the target hardware platform.

The system outputs $\mathbf{y}(k)$ can be highly variable due to stochastics of workload needed to be processed during a sampling interval. This effect can make it difficult to model the relation between the control inputs $\mathbf{u}(k)$ and the outputs $\mathbf{y}(k)$. The amount of workload submitted during each sampling interval depends on (i) number of job releases (ii) execution time of each job (iii) amount of backlog, i.e., job executions that are released in the previous sampling interval but not completed in the same sampling interval. The effect of the number of job releases can be counteracted by choosing an appropriate sampling length. For instance, if a component consists of periodic tasks, we can use the least common multiple of tasks to counteracted for the problem of number of releases. The sampling length should be large enough to accommodate multiple job releases. However, sampling infrequently may result in slow responses to changes. On the other hand, sampling too frequently may impose considerable

overhead to the system. Therefore, choosing an appropriate sampling length is of paramount importance that requires careful study and investigations.

We use the Root Mean Square Error (RMSE) for evaluating our identified model parameters[1]. RMSE is scale dependent, therefore we use it for comparing different models representing same data. When comparing two models, the one that has smaller RMSE is better. We also evaluate the variability explained by the model using R^2. In general, models that their $R^2 > 0.8$ are considered to be an acceptable fit to the system [9]. Models that their R^2 is closer to one better explain the identified system. We use sine functions for changing inputs to excite the system and observe the outputs. First we excite the system by only altering α. Then we alter T while keeping α unchanged. Finally, we use all samples for system identification. The model parameters ($\mathbf{A}$ and $\mathbf{B}$) are estimated using least squares. Note that since the periodic servers provide timing isolation, system identification for each component can be done independently. The identified system model will still be applicable after integration with other components because the processor provision to the component will not be affected by other components in the normal mode. Recall that in this section we assume that the system is in the normal mode.

D. Controller design

Linear-Quadratic Regulators (LQR) let us to trade-off between control speed and over reaction. In contrast to the well-know PID controllers in which the gain values are directly quantified by designers, the LQR controllers allow designers to focus on the cost of control actions as well as control errors. In general, we prefer unaggressive control actions which provides slow reactions to sudden changes to avoid overreacting to transient stochastics. We define error $\mathbf{e}(k)$ as: $\mathbf{e}(k) = \mathbf{r} - \mathbf{y}(k) = \mathbf{r} - \mathbf{x}(k)$, where $\mathbf{r}$ is the reference value for the output ($\mathbf{r} = [r_1 r_2]^T$). The dynamics of the control system based on $\mathbf{e}(k)$ is as follows:

$$\begin{aligned}\mathbf{e}(k+1) &= \mathbf{r} - \mathbf{A}\mathbf{x}(k) - \mathbf{B}\mathbf{u}(k) \\ &= \mathbf{A}\mathbf{e}(k) - \mathbf{B}\mathbf{u}(k) + (\mathbf{I} - \mathbf{A})\mathbf{r}.\end{aligned}$$

Instead of directly using the model presented in Eq. 1, we use the system model based on error for the controller design. In addition to $\mathbf{e}(k)$, we also use integral states: $\mathbf{e_I}(k+1) = \mathbf{e_I}(k) + \mathbf{e}(k)$, where $\forall k \leq 0$ we have $\mathbf{e_I}(k) = 0$. Hence, the augmented state space model is:

$$\begin{bmatrix}\mathbf{e}(k+1) \\ \mathbf{e_I}(k+1)\end{bmatrix} = \begin{bmatrix}\mathbf{A} & \mathbf{0} \\ \mathbf{I} & \mathbf{I}\end{bmatrix}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} + \begin{bmatrix}-\mathbf{B} \\ \mathbf{0}\end{bmatrix}\mathbf{u}(k) + \begin{bmatrix}\mathbf{I} - \mathbf{A} \\ \mathbf{0}\end{bmatrix}\mathbf{r}.$$

We use dynamic feedback, that is:

$$\mathbf{u}(k) = -\mathbf{K}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} = -\begin{bmatrix}\mathbf{K}_P & \mathbf{K}_I\end{bmatrix}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix},$$

where $\mathbf{K}_P$ and $\mathbf{K}_I$ are 2×2 matrices. By substituting the control law in the state space model we obtain the following closed-loop system model:

$$\begin{bmatrix}\mathbf{e}(k+1) \\ \mathbf{e_I}(k+1)\end{bmatrix} = \left(\begin{bmatrix}\mathbf{A} & \mathbf{0} \\ \mathbf{I} & \mathbf{I}\end{bmatrix} - \begin{bmatrix}-\mathbf{B} \\ \mathbf{0}\end{bmatrix}\begin{bmatrix}\mathbf{K}_P & \mathbf{K}_I\end{bmatrix}\right) \begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} + \begin{bmatrix}\mathbf{I} - \mathbf{A} \\ \mathbf{0}\end{bmatrix}\mathbf{r}.$$

[1]The model evaluation metrics used in this paper (i.e., RMSE and R^2) are explained in Chapter 2.4.4 of [9].

In LQR control design we are looking for gain values ($\mathbf{K}_P$ and $\mathbf{K}_I$) that minimize the following quadratic cost function:

$$J = \sum_{k=1}^{\infty} [\mathbf{e}(k)\mathbf{e_I}(k)]^T \mathbf{Q}\begin{bmatrix}\mathbf{e}(k) \\ \mathbf{e_I}(k)\end{bmatrix} + \mathbf{u}(k)^T \mathbf{R}\mathbf{u}(k),$$

where $\mathbf{Q}$ specifies the cost of error and $\mathbf{R}$ quantifies the cost of control action. The responsibility of the system integrator is to choose suitable error and control cost matrices.

IV. Resource manager

Thus far, we have considered adapting the parameters of a single VC. In this section we consider the whole system. The resource manager has the following responsibilities. (1) Admission control based on the minimum resource requirements; (2) cluster compression, when the average resource requirements can not be met; (3) allocation of the VCs to processors, i.e., mapping the VCs to the VPs; (4) adjusting the parameters of VPs and dealing with overloads. In the rest of this section the above responsibilities are explained in detail. At each sampling point k, the resource manager allocates the suggested parameters by the cluster controllers to the VCs. In this section we focus on a single sampling point. Therefore, for simplicity, we drop sampling time k when referring to the output of the cluster controllers.

Admission. The resource manager creates $\{\Pi_1, \cdots, \Pi_n\}$ for hosting $\{\mathfrak{C}_1, \cdots, \mathfrak{C}_n\}$. The system integrator is allowed to admit components such that the sum of components' minimum bandwidth, specified in the component interfaces, is less the available multiprocessor bandwidth: $\sum_{i\in[1\ldots n]} \bar{\alpha_i} - \sigma_{\alpha_i}/2 \leq M$. In doing so, we can guarantee minimum $\bar{\alpha_i} - \sigma_{\alpha_i}/2$ resource provisioning for $\mathfrak{C}_i$.

Cluster compression. The resource manager performs an allocation based on the operating bandwidths $\bar{\alpha_i}$ specified in the component interfaces. Since the admission is done based on the minimum required bandwidths, it is possible to have $\sum_{i\in[1\ldots n]} \bar{\alpha_i} > M$. In such a case, the resource manager first performs a cluster bandwidth compression, that is, compressing the cluster bandwidths such that the total required bandwidth is less than or equal to M. The VCs will receive partial bandwidths after the compression. α'_i and λ_i denote the compressed bandwidth and the compression factor of Π_i respectively, where $\lambda_i \alpha_i = \alpha'_i$. Our objective is to maximize $\sum_{i=1}^{n} \lambda_i \zeta_i$ when performing the compressions. In doing so, components which have less impact on the total value of the system will be subjected to more compressions. Let $\bar{\alpha_i} - \sigma_{\alpha_i}/2 = \phi_i$ and $\frac{\zeta_i}{\alpha_i} = \Delta_i$. The compression problem is formulated as the following:

$$\textbf{Maximize}: \quad \sum\nolimits_{i=1}^{n} \alpha'_i \Delta_i, \tag{2a}$$

$$\textbf{Subject to}: \quad \alpha_i \geq \alpha'_i \geq \phi_i \qquad \forall i \in [1 \ldots n], \tag{2b}$$

$$\sum\nolimits_{i=1}^{n} \alpha'_i \leq M. \tag{2c}$$

We use Algorithm 1, which has polynomial time complexity ($O(n^2)$), for solving the cluster compression problem. The algorithm treats VCs in the order of Δ_i. Each VC receives at least ϕ_i. In addition, it receives $\alpha_j - \phi_j$ bandwidth if the remaining multiprocessor capacity ($\mathfrak{M}$) is sufficient. Otherwise, the remaining capacity is added to the VC's bandwidth.

Theorem 1. *Algorithm 1 is optimal, i.e., the compression factors produced by this algorithm maximizes the total system value.*

The formal proof of the above theorem is presented in the appendix.

Algorithm 1: Cluster compression algorithm.

Input: $\{\Delta_1, \ldots, \Delta_n\}$ and $\{\phi_1, \ldots, \phi_n\}$.
Output: $\{\alpha'_1, \ldots, \alpha'_n\}$.
1: G = $\{\Delta_1, \ldots, \Delta_n\}$;
2: $\forall i, \alpha'_i = \phi_i$;
3: $\mathfrak{M} = M - \sum_i \phi_i$;
4: **while** $G \neq \emptyset$ AND $\mathfrak{M} > 0$ **do**
5: $\quad \Delta_j = \max(G)$;
6: $\quad \alpha'_j = \phi_j + \min(\mathfrak{M}, \alpha_j - \phi_j)$;
7: $\quad G = G - \Delta_j$;
8: $\quad \mathfrak{M} = \mathfrak{M} - \alpha'_j + \phi_j$;
9: **end while**

Allocation. The resource manager performs allocations assuming that the overall required bandwidth is less than or equal to the multiprocessor bandwidth. The allocation algorithm creates at most M VPs for each VC such that all VPs collectively provide B_i units of the processor time to Π_i. We use the partitioned EDF algorithm for scheduling the VPs. The allocation algorithm has three objectives. First of all, the number of VPs should be minimized. This is because when components are split, their inner tasks will migrate between the processors. Hence, the components will require extra bandwidth to compensate for the migration overhead. In addition, we favor balanced allocations that is, fairly distribution of the slack time over all processors. The reason behind preferring balanced distributions is to give the cluster controllers more freedom to adapt the VC bandwidths. When a processor is overloaded, more important components can steal bandwidth from the less important ones that coexist with them on the same processor. Hence, we favor an allocation approach that co-allocates more important components with less important ones. This approach gives more freedom to the more important components to adapt their bandwidth in the overload situations. Hence, the third objective of the allocation algorithm is to achieve a balanced importance distribution. Thus, the allocation problem formulation is as follows:

$$\textbf{Maximize}: \quad w_1 \left(nM - \sum_{i=1}^{n} \sum_{j=1}^{M} f_{i,j} \right) + w_2 z_2 + w_3 z_3, \tag{3a}$$

$$\textbf{Subject to}: \quad z_2 \leq \sum_{i=1}^{n} \rho_{i,j} \qquad \forall j \in [1 \ldots M], \tag{3b}$$

$$z_3 \leq \sum_{i=1}^{n} \frac{\rho_{i,j}}{\alpha_i} \zeta_i \qquad \forall j \in [1 \ldots M], \tag{3c}$$

$$\sum_{i=1}^{n} \rho_{i,j} \leq 1 \qquad \forall j \in [1 \ldots M], \tag{3d}$$

$$\sum_{j=1}^{M} \rho_{i,j} = \alpha_i \qquad \forall i \in [1 \ldots n], \tag{3e}$$

$$\frac{\rho_{i,j}}{\alpha_i} \leq f_{i,j} \qquad \forall i \in [1 \ldots n], \forall j \in [1 \ldots M], \tag{3f}$$

$$f_{i,j} \in \{0, 1\}, \rho_{i,j} \in \mathbb{Z}_{\geq 0}, \tag{3g}$$

where z_2 and z_3 correspond to load balancing, and importance balancing objectives respectively. z_2 represents the maximum load assigned to one processor. While, z_3 represents the maximum importance available on one processor. w_1, w_2 and w_3 are the weights of the three aforementioned objectives. $f_{i,j}$ is equal to one when $\pi_{i,j}$ exists, i.e. $\rho_{i,j} > 0$. Note that the allocation algorithm assumes that $\sum_{i=1}^{n} \alpha_i \leq M$.

The optimization problem formulation presented in Eq. 3 is a mixed integer linear programming problem. The complexity of solving this problem is exponential in the number of processors and the number of components. Hence, solving it for large $n \times M$ may become intractable. Therefore, we present an allocation heuristic to partition components in polynomial time. The allocation heuristic is presented in Algorithm 2. Let $v_i = \zeta_i \alpha_i$ denote the value of $\mathfrak{C}_i$. In the algorithm $\{\alpha\}$, $\{v\}$ and $\{\rho\}$ represent the set of VC bandwidths, values and VP bandwidths respectively. First we sort the VCs based on their values. The result bandwidth set is descending in value, i.e., $v_i \geq v_{i+1}$. Then we try to allocate each VC to a processor without splitting it. We use the worst fit allocation, i.e., among all candidate processors that can accommodate the current VC, we choose the one that after allocation it will leave the largest slack time. If the allocation fails, then we split the VC, i.e., we create a number of VPs for the VC. For splitting, we start with a processor that has the largest slack time. We allocate all of the slack time to Π_i and move to a processor with the next largest slack. This process continues until all of the bandwidth of the VC is assigned.

Algorithm 2: Heuristic algorithm for allocating the VCs on processors.

Input: set of cluster bandwidths $\{\alpha\}$ and component values $\{v\}$.
Output: matrix of virtual processor bandwidths $\{\rho\}$.
1: sort the active components (Γ) based on their values $\{v\}$
2: **for** $i \in \Gamma$ **do**
3: $\quad$ **if** `WorstFit`(α_i, $\{\rho\}$) = **false then**
4: $\quad\quad$ `Split`(α_i, $\{\rho\}$)
5: $\quad$ **end if**
6: **end for**

Adjusting VCs. When a cluster controller suggests a new bandwidth and a new period for a VC, the resource manager is responsible to adjust the parameters of the VPs associated with that VC. First of all, the resource manager checks if the suggested values are within the operating range of the component. If the values are beyond the operating region, the resource manager overwrites the suggested values with the boundary of the operating region that is closer to the suggested values. The suggested period is assigned to all of the corresponding VPs. However, the suggested bandwidth is distributed among them. Our goal in distributing the total bandwidth among the VPs is to minimize the number of VPs that are assigned to the VC. This is because in the system identification step the components are identified independently using a minimum number of processors. Hence, we start from the largest slack processor and we allocate its slack bandwidth to the VC. If the VC still needs more bandwidth we move to the second largest slack processor. This process continues until the suggested bandwidth is assigned to the VC.

Dealing with overloads. Assume that the cluster controller of Π_i wants to adapt its bandwidth to α_i^{new}. If the slack time on all processors is not enough to accommodate Π_i with its

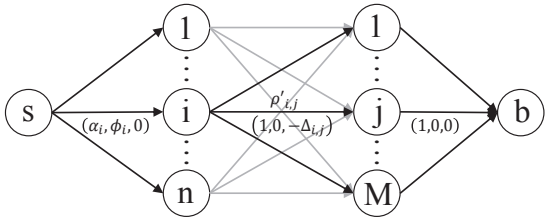


Fig. 4: The maximum flow formulation of the VP compression algorithm. Node S and b represent the source and the sink nodes respectively. The labels on the edges represent the capacity, demand and cost respectively $(u_{i,j}, d_{i,j}, \kappa_{i,j})$.

new bandwidth we have to perform a bandwidth compression. We prefer performing the compression without conducting cluster reallocations. This is because reallocation may force VP migrations which in turn incur overhead costs. In this situation, each VP affected by the compression will receive a portion of its original bandwidth: $\rho'_{i,j} = \lambda_{i,j}\rho_{i,j}$, where $\lambda_{i,j}$ is the compression factor of $\pi_{i,j}$ and $\rho'_{i,j}$ is the overload bandwidth of $\pi_{i,j}$. Our objective is to maximize $\sum_{j=1}^{M}\sum_{i=1}^{n}\lambda_{i,j}\zeta_i$. Let $\frac{\zeta_i}{\rho_{i,j}} = \Delta_{i,j}$. The VP bandwidth compression is formulated as the following optimization problem:

$$\textbf{Maximize}: \quad \sum_{j=1}^{M}\sum_{i=1}^{n}\rho'_{i,j}\Delta_{i,j}, \tag{4a}$$

$$\textbf{Subject to}: \quad \alpha_i \geq \sum_{j=1}^{M}\rho'_{i,j} \geq \phi_i \qquad \forall i \in [1\ldots n], \tag{4b}$$

$$\sum_{i=1}^{n}\rho'_{i,j} \leq 1 \qquad \forall j \in [1\ldots M]. \tag{4c}$$

This problem can be mapped to the "maximum flow minimum cost with edge demands" problem. Let $H = (V, E)$ be a directed graph with cost $\kappa_{i,j}$, demand $d_{i,j}$ and capacity $u_{i,j}$ associated with every edge $(i, j) \in E$. Figure 4 illustrates our model. The edges connecting the source to the n nodes corresponding to the components have a capacity equal to the component bandwidth, a demand equal to the minimum bandwidth of the component and a cost equal to zero. These edges apply the constraint expressed in Eq. 4b. The edges connecting the n component nodes to the M processor nodes have a capacity equal to one (maximum bandwidth of a processor), a demand equal to zero and a cost equal to $-\Delta_{i,j}$. The edges connecting the M processor nodes to the sink have a capacity equal to one (to apply the constraint of Eq. 4c), a demand and a cost equal to zero. We use the cycle canceling algorithm for solving this problem in polynomial time [11]. Once the problem is solved, the flows of the edges that connect the n component nodes to the M processors will be selected as the compressed VP bandwidths $(\rho'_{i,j})$.

Since the component's bandwidth requirements may change during run-time and new virtual processors may be created, the initial allocation might become inefficient after some time. Hence, once in a while, the components need to be reallocated. However, in this paper we do not address this problem and we leave it for the future work. We provide some guidelines for selecting the sampling length, operating regions and importance values in the appendix.

Mode change. In our scheme, the cluster parameters are adapted during run-time. This phenomena is referred as mode change in the multi-mode real-time system literature. A potential problem that can happen in mode changes is that, even if the schedulability condition is satisfied before and after the mode change, the system is not necessarily schedulable during the transient mode. Since in this paper we focus only on soft real-time components in which occasional deadline misses can be tolerated, we intentionally neglect this problem. However, if hard real-time components coexist with soft real-time components, the transient overloads can be avoided by introducing a mode change delay similar to [10], and the rest of our method can be directly applied.

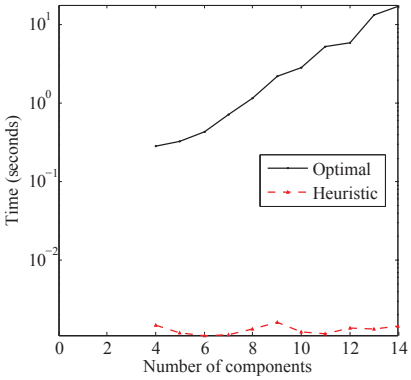


Fig. 5: Execution time of the allocation algorithms. Note that y-axis is in logarithmic scale.

V. Evaluations

In this section we first evaluate the allocation heuristic. We, then, present a case study consisting of two components. The components are identified using our simulation tool. Thereafter, the performance of the closed-loop system is evaluated both in the simulation tool as well as in our Linux implementation.

A. Allocation heuristic

We have evaluated the allocation heuristic against the optimal solution. In our evaluations we assumed $M = 4$. We set the total system utilization to two. We changed the number of components from four to 14. For each n we generated 100 random systems. The total utilization was divided among n components using the UUnifast algorithm [12]. Finally, the average achieved objective for each n is reported in Figure 6. We have compared five algorithms in the evaluation: (i) optimal load balancing algorithm (ii) optimal importance balancing algorithm (iii) optimal split algorithm (iv) optimal combined objective algorithm (v) our heuristic. We used the CVX solver for solving the optimal algorithms. Each graph in Figure 6 illustrates a certain objective achieved by the five algorithms. In all of our evaluations we assumed $w_1 = 1/4(n-1)$, $w_2 = 4/\sum_{i=1}^{n}\alpha_i$ and $w_3 = 4/\sum_{i=1}^{n}\alpha_i\zeta_i$. The figures show that (1) except the optimal combined objective, all other algorithms have poor performance with respect to some objective, (2) our heuristic outperforms the combined optimal algorithm in the split objective, while the combined optimal algorithm outperforms the heuristic in the rest of the algorithms, (3) our heuristic outperforms all optimal algorithms that consider only one of the three objectives. Figure 5 illustrates the execution time of our heuristic allocation against the optimal solution. Each point in the figure is the average of 100 random systems. As shown in the figure, the execution time of the optimal algorithm increases exponentially when increasing the number

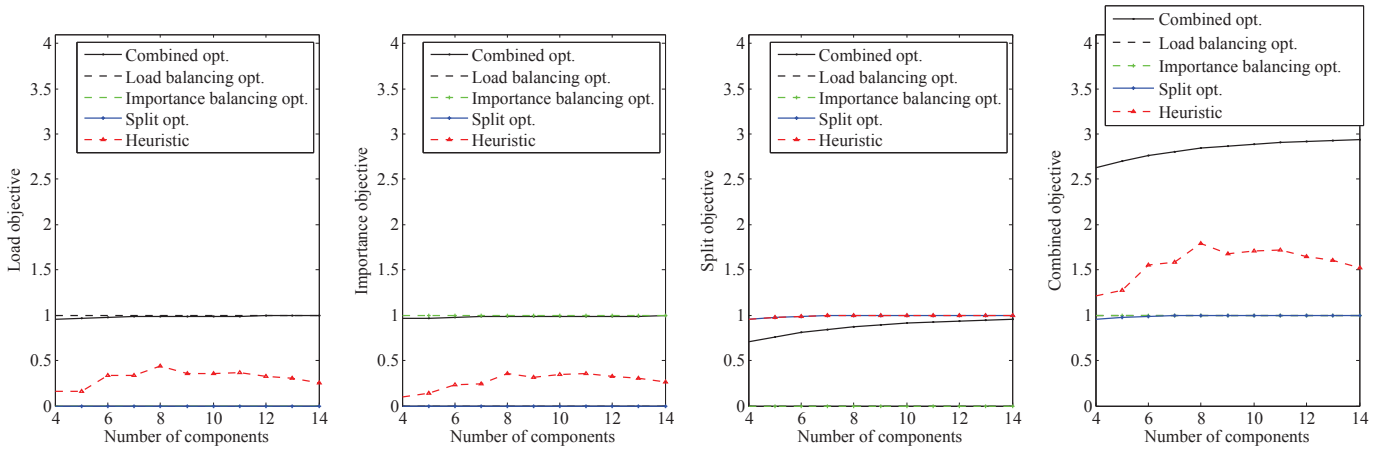
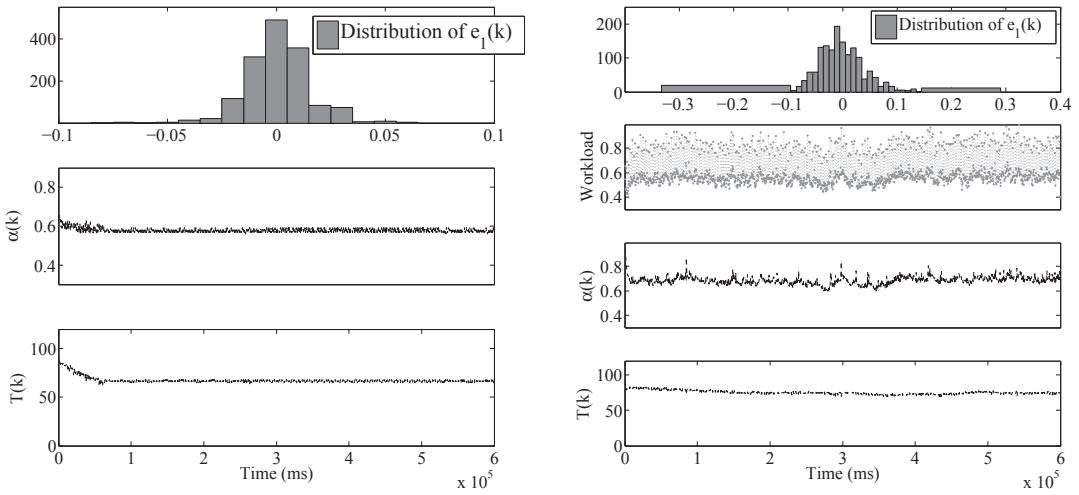


Fig. 6: Four different objectives achieved by the five allocation algorithms.



(a) Static component.

(b) Dynamic component.

Fig. 7: Distribution of e_1, workload variation and parameter adaptations.

of components, while the heuristic has approximately constant execution time.

B. Case study

We have modified the TrueTime [13] simulation tool such that two level hierarchical scheduling is implemented[2]. For system identification and controller design we used our modified TrueTime simulator. We first present two example components. Thereafter, using the examples we explain our modeling approach. Note that a simulation based system identification is only valid if the task execution times are gathered from running tasks on the target hardware platform.

Component 1 (Static component)**.** *Consider a component consisting of three periodic tasks with the following periods* $\{p_1 = 40, p_2 = 50, p_3 = 100\}$ *and the following execution times* $\{c_1 = 12, c_2 = 10, c_3 = 5\}$.

Component 2 (Dynamic component)**.** *Assume a component consisting of three periodic tasks with the following periods* $\{p_1 = 40, p_2 = 50, p_3 = 100\}$. τ_1 *and* τ_2 *are video decoder tasks that decode their input video stream. Therefore, their execution time is highly variable.* τ_3 *is a static task, i.e. its execution time is fixed* ($c_3 = 5$)*. Average bandwidth consumed by the video decoder tasks* τ_1 *and* τ_2 *are* 0.29 *and* 0.25 *respectively. This component could, for instance, represent a robotic vision system consisting of two cameras where one real-time task is assigned for decoding video streams captured from each camera. The third task, however, is performing analysis over the decoded streams by executing a predefined number of instructions. The execution time distribution of* τ_1 *and* τ_2*, and the workload distribution of this component is illustrated in Figure 8a.*

Let us consider the parameter identification for Component 1, we assumed the task parameters were not available. Instead of the task parameters, the component designer had provided the following interface: $<\bar{\alpha}$=0.65, $\bar{T}$=90, σ_α=0.15, σ_T=100$>$. Therefore, we changed the bandwidth of the VC in the following region $[\bar{\alpha} - \sigma_\alpha/2, \bar{\alpha} + \sigma_\alpha/2]$ while $\forall k\ T(k)$=40. In another experiment we changed $T(k)$ in the following region $[\bar{T} - \sigma_T/2, \bar{T} + \sigma_T/2]$ while keeping $\alpha(k)$=58. The simulation duration was 10 minutes of the tasks' executions. The sampling length was assumed to be 400. We performed

[2]The source code of our modified version is available at: https://github.com/nimazad/TrueTime-HSF.

parameter estimations over the observed data. The value of matrices **A** and **B** are reported in Table II. Evaluating the result model on the training data (i.e., same data which was used for parameter estimation), we found the following properties: R^2=0.91 RMSE=4.63. In another experiment we altered both $\alpha(k)$ and $T(k)$ at the same time to assess how much our model can explain the new dataset. As a result we had R^2=0.92 and RMSE=5.37. The identified model parameters were used to design a MIMO LQR controller for adapting the VC parameters.

For the dynamic component we used the following interface: $<\bar{\alpha}$=0.72, $\bar{T}$=90, σ_α=0.26, σ_T=50$>$. We followed the same steps (with the same sampling length) as the static component case. The value of matrices **A** and **B** are reported in Table II. Evaluating the result model on the training data we had R^2=0.95, RMSE=4.53. Evaluating the model on a test dataset in which both bandwidth and period were changed simultaneously we had: R^2=0.98, RMSE=1.23. In summary, we conclude that the identified model parameters, for both static and dynamic component, can explain the training dataset as well as the test dataset to an acceptable extend (see Section III-C for more details on R^2 and RMSE).

Afterwards, we considered Component 1 which its parameters were already identified. We set **R**=diag$(10, 10)$ and **Q**=diag$(1, 1, 0.1, 0.1)$. The value of the result gain matrix is reported in Table II. We ran the closed-loop system using the obtained gain matrix. The reference values were **r**=$[0.02,\ 1]^T$. Figure 7a illustrates the probability distribution of e_1 as well as the bandwidth and period adaptation for the cluster that is serving the static component. Since the task parameters were not changed, the controller stabilized $\alpha(k)$ and $T(k)$ in almost constant values. The average number of deadline misses in this experiment was 1.25 which is very close to the set reference value. The standard deviation of e_2 was 1.66. In another experiment we designed a LQR controller for Component 2 which was identified previously. We set the same **R** and **Q** matrices as the static component example. The value of the result gain matrix is reported in Table II. The reference values were **r**=$[0.06,\ 1]^T$. The VC parameter adaptations are illustrated in Figure 7b. The depicted workload variation in the figure is collected independently running the component with the full processor capacity. Since the task execution times were changed, the cluster parameters were also adapted based on the current demand at each time point.

Linux experiments. We have implemented our adaptive framework in the Linux kernel. Inspired by [14], we used kernel loadable modules to implement our scheme[3]. In the rest of this section, we present the results of our Linux evaluations. We used Intel Core i5-3550 processor clocked at 3.3 GHz. Our loadable module can utilize all four cores on this processor. However, for imposing overload situations, we limited the number of available processors to two. The cluster controllers and the resource manager are developed as user space tasks. The controller tasks were attached to a different VC than the component VCs. We considered the two components that we have designed cluster controllers for them using our simulations. The resource manager created two clusters Π_1 and Π_2 for hosting the static component and the dynamic component respectively. Cluster Π_3 was also created for hosting the cluster controller tasks. The bandwidth of Π_3 was equal to 0.05 and it was constant throughout the experiment. All task parameters described in the definition of the components were assigned in milliseconds. We ran the experiment for 10 minutes. Figure 8c illustrates the adaptations for this experiment. The observed distribution of e_2 is slightly different than the simulations. The difference is due to the fact that the simulation does not take into account the overhead of scheduling, adaptation and operating system related interferences. The average observed e_2 was -0.12 for Π_1 and -0.07 for Π_2.

[3]The source code is available at: http://nimazad.github.io/FS-CBRTS.

Adaptation overhead. We created a periodic task associated with each cluster which was ran within Π_3. The period of these tasks was equal to 400 (sampling length). The LQR controller as well as the resource manager functionalities are implemented in these tasks. In the above experiment the maximum observed execution time for the controller task of Π_1 and Π_2 were $0.101ms$ and $0.081ms$ respectively. Given that we had two processors available, each adaptive cluster cost approximately 0.01 % of the multiprocessor time. The total adaptation overhead is proportional to the number of adaptive components.

In another set of experiments, to impose overload situations, we created a dummy cluster (Π_4) and assign the following bandwidth to this cluster: $\alpha_4 = 0.42$. With the existence of Π_4, it was not possible to perform reservations based on the worst-case demands anymore. The importance of the clusters were set as follows: $\zeta_1 = 200$, $\zeta_2 = 300$, $\zeta_3 = 2000$ and $\zeta_4 = 3000$. Therefore, the resource manager created two VPs for Π_1 in the beginning of the experiment. Splitting this VC imposes migration overhead to Π_1. We considered three different setups: (1) adaptation was turned off for the both VCs while we assigned $\bar{\alpha}$ and $\bar{T}$ to the VCs; (2) both VCs were adapted; (3) we used the average assigned bandwidth and period observed in the second setup and repeated the experiment with those values. We ran the experiment for 10 minutes. The average observed e_1, e_2 and their standard deviations are reported in Table I. Note that $e_1 < 0$ means that the cluster was idling its bandwidth more than the reference value (r_1) and $e_2 < 0$ means that the number of deadline misses observed at each sampling time was more than one. The results presented in Table I show that fixed allocation based on the operating points specified in the component interface was inefficient. Note that the operating points are based on the average workloads. When both VCs were adapted, the VP compression was performed 32 times, whereas cluster adaptation was performed 1500 times. Therefore, the additional overhead due to the compressions was insignificant. Since Π_1 has the lowest importance, the compression did not provide extra bandwidth to it. In the adaptive case, the average bandwidth and period assigned for Π_1 and Π_2 were 0.61, 50.65, 0.76 and 74.41 respectively. Hence, in average, there was 0.16 slack bandwidth in the system which permits the admission control to admit new components if required. In the third setup, we used the average bandwidths and periods assigned by the cluster controller in setup 2, and we assigned them as fixed values to the VCs. The average number of deadline misses as well as the standard deviation of the deadline misses for Π_2 in comparison to the second setup were increased. In addition, in the second setup 3 % of the VC bandwidth was wasted, whereas in the third setup 8 % of the VC bandwidth was idled. The results suggest that adaptation helps when the workload is subjected to unpredictable disturbances such as migration overhead and

execution time variations.

	Π_1				Π_2			
Exp.	$\bar{e_1}$	σ_{e_1}	$\bar{e_2}$	σ_{e_2}	$\bar{e_1}$	σ_{e_1}	$\bar{e_2}$	σ_{e_2}
1	0.16	0.04	-3.19	1.64	1.00	0.36	-12.94	9.87
2	0.17	0.04	0.31	1.96	0.54	0.07	-0.17	2.17
3	0.62	0.008	0.18	0.91	0.53	0.13	-0.30	3.85

TABLE I: Mean and standard deviation of e_1 and e_2 for the three setups.

Step response experiment. Figure 8b illustrates the response of the static component to a step workload change. The experiment was performed using the same setup as described above. For this experiment the execution time of τ_1 was set to 10 before time $2 \times 10^5 ms$. Afterwards, it was increased to 14. This execution time change caused a 10 % change in the workload. Note that the reference value for $x_1(k)$ was 0.02. Therefore, the cluster controller provided more bandwidth than the workload. In addition, the cluster controller had to compensate for the workload disturbances such as context switches and scheduling overheads.

VI. Related work

Feedback control has found its way in computing systems for helping system designers to deal with uncertainties and dynamicity. For instance, in high-performance computing load is unpredictable and dynamic. A MIMO controller is used to control CPU and memory utilizations in an Apache web server [15]. In [16] a MIMO LQR controller is used to solve a load balancing problem. The controller equalizes the load among different resources to improve response times as well as the throughput.

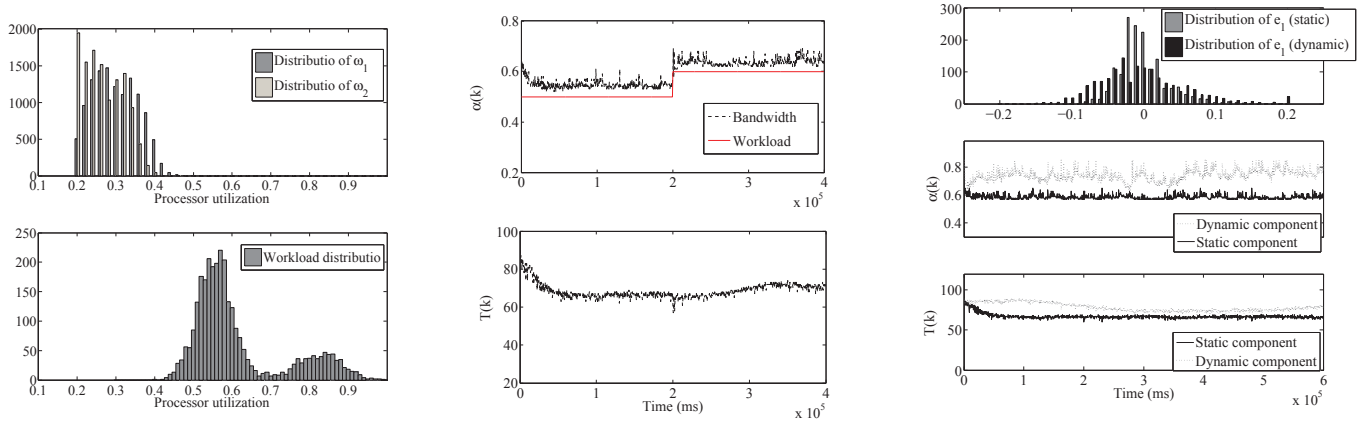
In the context of real-time scheduling, Lu *et al.* proposed a feedback scheduling scheme to cope with unpredictable workloads [17]. In their framework the deadline miss ratio and the system utilization is used as sensors, while the admission control is used as an actuator. The problem of task reweighting under multiprocessor scheduling algorithms is studied in [18] and [19]. In these papers it is assumed that, tasks ask for a new processor utilization during run-time. A number of reweighting rules for partitioned and global scheduling algorithms are presented. In [20] task reweighting is combined with feedback loops that estimate the weight of the next job. In distributed real-time systems, utilization control is performed through rate adaptation to provide quality of service guarantees [21]. In [22] service levels are adapted based on monitoring the number of deadline misses and the processor utilizations. Utilization control is coupled with processor frequency adjustment in [23] and [24]. Targeting end-to-end task models, DEUCON [25] employs a decentralized approach in which task rates (periods) are adapted using MIMO model predictive controllers. The control objective is to minimize the difference between the utilization set points and current utilizations. The main difference of our paper with the aforementioned works is the following. Since we consider component-based systems in which a component is comprised of a set of tasks, a reservation-based scheduling policy is needed to isolate the timing behavior of the components in run-time. While this separation of run-time behavior for components is not supported by the above frameworks.

Adaptive reservation schemes, first introduced in [5], are powerful approaches for controlling the amount of processor allocated to individual tasks that demonstrate dynamic processor requirement. The mathematical model of a such scheme using Constant Bandwidth Servers (CBS) is derived in [26]. PI controllers are used for controlling the bandwidth of CBS. In [27] stochastic controllers are used for the same purpose. Regarding adaptive reservations in which multiple parameters are adapted, in [28] both periods and budgets of the CBS are adapted. This framework targets legacy tasks which do not communicate with the scheduler. Two different components are used (i) period detector (ii) budget estimator. One centralized controller is used for adapting the periods and the budgets. In the context of the ACTORS project [29], a cascade controller is used on top of CPU reservations for adapting their bandwidths. Our work is different from the above reservation-based approaches in the following main aspects. (i) Except ACTORS, all aforementioned frameworks target single processors. While we target multiprocessors. In contrast to ACTORS, our framework allows spreading VCs (components) over multiple physical processors. This feature allows running components which their utilization is more than one, i.e., component that can not be executed using only one processor. (ii) In the above schemes (including ACTORS) the distance between the task finishing time and its corresponding CBS deadline is used as the sensor. However, we consider a more general component model in which multiple tasks may be in a single component. In our model, the intra-component scheduler coordinates the execution sequence of the tasks inside a component. Hence, the control input used by the above frameworks is not applicable to our model. (iii) Except [28], the other frameworks only adapt reservation budgets, while we adapt the period and budget simultaneity. Our framework is different from [28] in aspect (i) and (ii). In addition, in contrast to [28], we consider software components that are developed using API functions that inform the scheduler when the tasks start executing, finish execution and wait until their next period.

Finally, we proposed adaptive reservation schemes for hierarchical real-time systems in [30], [31]. In our previous works we have addressed single processors while in this paper we consider multiprocessor platforms. Considering multiprocessors resulted in introducing a mechanism for distributing components over the processors. Moreover, we adapt both period and budget of the reservations using MIMO controllers, whereas we only investigated adapting the budgets in the aforementioned publications. In addition, in this paper, we solved the problem of bandwidth distribution in overload situations optimally.

VII. Conclusions

We proposed a feedback scheduling framework for component-based soft real-time systems. We targeted software components consisting of multiple real-time tasks which exhibit significant processor demand variation during run-time. Our framework uses processor reservations for providing processor time to the components. A component may be distributed over several processors. Hence, the intra-component tasks are scheduled using a global multiprocessor scheduler. First we showed that it is important to adapt both period and bandwidth of the reservations. We, then, used a case study and evaluated our MIMO LQR controllers in the TrueTime simulation tool. Finally, we implemented our framework in the Linux kernel and evaluated the case study in practice. The evaluations show that our framework can efficiently adapt the reservations to deal with the workload disturbances.



(a) Execution times distribution of τ_1 and τ_2, and workload distribution of the dynamic component.

(b) Response of the static component to a step workload.

(c) Distribution of e_1, bandwidth adaptation and period adaptation for the two component clusters.

Fig. 9: Evaluation results (a) running the two sample component (b) for the step workload change.

	Static component	Dynamic component
A	$\begin{bmatrix} 0.3711 & -0.5503 \\ 0.1798 & 1.106 \end{bmatrix}$	$\begin{bmatrix} 0.7035 & -0.4138 \\ 0.0582 & 1.033 \end{bmatrix}$
B	$\begin{bmatrix} 0.8887 & -0.0413 \\ -0.2952 & 0.0160 \end{bmatrix}$	$\begin{bmatrix} 0.8443 & -0.0336 \\ -0.2421 & 0.0138 \end{bmatrix}$
K	$\begin{bmatrix} -0.0390 & 0.6150 & -0.0832 & 0.0260 \\ -0.3985 & -1.2376 & -0.0311 & -0.0949 \end{bmatrix}$	$\begin{bmatrix} -0.3506 & 1.1139 & -0.0871 & 0.0090 \\ -0.0850 & -0.7983 & -0.0117 & -0.0992 \end{bmatrix}$

TABLE II: The value of different matrices corresponding to the case study.

In the future, we will investigate the problem of reallocating components systematically by introducing a new metric to understand when it is necessary to perform reallocations. We are also contemplating the elimination of the system identification step by utilizing an adaptive control scheme that can develop the plant model during run-time.

REFERENCES

[1] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in *RTSS'97*, December 1997, pp. 308–319.

[2] G. Lipari and S. Baruah, "A hierarchical extension to the constant bandwidth server framework," in *RTAS'01*, May 2001, pp. 26–35.

[3] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *ECRTS'08*, July 2008, pp. 181–190.

[4] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation," in *RTSS'10*, December 2010, pp. 249–258.

[5] L. Abeni and G. Buttazzo, "Adaptive bandwidth reservation for multimedia computing," in *RTCSA'99*, December 1999, pp. 70–77.

[6] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA-adaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, January 2009.

[7] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *RTSS'03*, December 2003, pp. 2–13.

[8] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *RTAS'01*, May 2001, pp. 75–84.

[9] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[10] L. Santinelli, G. Buttazzo, and E. Bini, "Multi-moded resource reservations," in *RTAS'11*, April 2011, pp. 37–46.

[11] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[12] E. Bini and G. Buttazzo, "Biasing effects in schedulability measures," in *ECRTS'04*, June 2004, pp. 196–203.

[13] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzen, "How does control timing affect performance? analysis and simulation of timing using Jitterbug and TrueTime," *Control Systems, IEEE*, vol. 23, no. 3, pp. 16–30, June 2003.

[14] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An external CPU scheduler framework for real-time systems," in *RTCSA'12*, August 2012, pp. 240–249.

[15] N. Gandhi, D. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh, "MIMO control of an apache web server: modeling and controller design," in *ACC'02*, vol. 6, 2002, pp. 4922–4927.

[16] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano, "Using MIMO linear control for load balancing in computing systems," in *ACC'04*, vol. 3, 2004, pp. 2045–2050.

[17] C. Lu, J. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, pp. 85–126, 2002.

[18] A. Block, J. H. Anderson, and U. C. Devi, "Task reweighting under global scheduling on multiprocessors," *Real-Time Systems*, vol. 39, no. 1-3, pp. 123–167, 2008.

[19] A. Block, J. Anderson, and G. Bishop, "Fine-grained task reweighting on multiprocessors," in *RTCSA'05*, 2005, pp. 429–435.

[20] A. Block, B. Brandenburg, J. Anderson, and S. Quint, "An adaptive framework for multiprocessor real-time system," in *ECRTS'08*, July 2008, pp. 23–33.

[21] J. Yao, X. Liu, X. Chen, X. Wang, and J. Li, "Online decentralized adaptive optimal controller design of CPU utilization for distributed real-time embedded systems," in *ACC'10*, June 2010, pp. 283–288.

[22] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu, "Feedback control scheduling in distributed real-time systems," in *RTSS'01*, December 2001, pp. 59–70.

[23] X. Wang, X. Fu, X. Liu, and Z. Gu, "PAUC: Power-aware utilization control in distributed real-time systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 3, pp. 302–315, Aug 2010.

[24] X. Chen, X.-W. Chang, and X. Liu, "SyRaFa: Synchronous rate and frequency adjustment for utilization control in distributed real-time embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 1052–1061, May 2013.

[25] X. Wang, D. Jia, C. Lu, and X. Koutsoukos, "DEUCON: Decentralized end-to-end utilization control for distributed real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 7, pp. 996–1009, July 2007.

[26] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *RTSS'02*, December 2002, pp. 71–80.

[27] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni, "Adaptive reservations in a linux environment," in *RTAS'04*, May 2004, pp. 238–245.

[28] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Adaptive real-time scheduling for legacy multimedia applications," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 4, pp. 86:1–86:23, January 2013.

[29] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino, "Resource management on multicore systems: The ACTORS approach," *Micro, IEEE*, vol. 31, no. 3, pp. 72–81, May-June 2011.

[30] N. M. Khalilzad, M. Behnam, and T. Nolte, "Multi-level adaptive hierarchical scheduling framework for composing real-time systems," in *RTCSA'13*, August 2013, pp. 320–329.

[31] N. M. Khalilzad, M. Behnam, G. Spampinato, and T. Nolte, "Bandwidth adaptation in hierarchical scheduling using fuzzy controllers," in *SIES'12*, June 2012, pp. 148–157.

[32] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

APPENDIX

Guidelines for parameter selections. Our adaptive framework spares engineers from conducting the WCET analysis. However, the component developers and the system integrator have to carefully design some parameters for maximizing the performance of the framework. Here we provide some guidelines for some of the parameters.

The system integrators have to study the dynamics of the components using different sampling lengths. The choice of the sampling length affects the accuracy of the identified models. This is because, the observed system dynamics are different for different sampling lengths. The larger the sampling length, the smoother the output. However, a too large samplings length results in slow reactions to the changes.

The operating regions of the components are provided by the component developers. These parameters have to be selected based on the task parameters as well as experimental studies. For instance, the operating point of the component period depends on the period of the tasks within the component. While, the bandwidth can be extracted by running the component and profiling its processor usage. Throughout the system identification step, the observed state space values have to be stored. In doing so, the system integrators can plot the feasible combinations of the state space values. The desired set points are, then, selected from the feasible set. For instance, the system integrator may choose a very small r_2. However, the price for having a too small r_2 often is to select a large r_1 which essentially means that we have to waste some bandwidth in order to achieve a very small number of deadline misses.

Our model of the component importance is quiet flexible. Here we consider two type of systems. (1) Systems in which the contribution of each component to the overall value of the system is clear for the integrator; (2) systems in which the relative importance is relevant, e.g., $\mathfrak{C}_i$ is always prioritized over $\mathfrak{C}_j$ in all overload conditions. For type (1), it is easy to assign the importance values. For instance if $\mathfrak{C}_1$ contributes 10 % to the total system value, we can assign $\zeta_1 = 10$ provided that $\sum_{i=1}^{n} \zeta_i = 100$. For type (2), the system integrator should first sort the components based on their desired priority at run-time. Assume that components are sorted based on their desired run-time overload priority, i.e., for $i \in [1, \ldots, n]$, $\mathfrak{C}_i$ has to be prioritized to $\mathfrak{C}_{i+1}$. The process of importance assignment starts from $\mathfrak{C}_n$. The designer assigns a small number to this component. $\mathfrak{C}_{n-1}$, then, we will have the following condition:

$$\zeta_{n-1} > \zeta_n \frac{\bar{\alpha}_{n-1} + \sigma_{\alpha_{n-1}}}{\bar{\alpha}_n - \sigma_{\alpha_n}}.$$

This condition is because the compression algorithm takes the size of the components into account when compressing the components. In this approach, $\mathfrak{C}_i$ will have $n - i$ conditions for its importance. To summarize we have:

$$\zeta_i > \max_{i+1<j<n} \left(\zeta_j \frac{\bar{\alpha}_i + \sigma_{\alpha_i}}{\bar{\alpha}_j - \sigma_{\alpha_j}} \right).$$

Proof of Theorem 1:

Proof: We prove using the Lagrangian duality [32]. The Lagrangian is

$$L = \sum\nolimits_i \delta_i \Delta_i - \theta(\sum\nolimits_i \delta_i - M) - \sum\nolimits_i \overline{\chi}_i(\delta_i - \alpha_i) + \sum\nolimits_i \underline{\chi}_i(\delta_i - \phi_i)$$

where $\theta, \overline{\chi}_i, \underline{\chi}_i$ are Lagrange multipliers. The Karush-Kuhn-Tucker (KKT) conditions are:

$$\Delta_i - \theta - \overline{\chi}_i + \underline{\chi}_i = 0, \tag{5}$$

$$\overline{\chi}_i(\delta_i - \alpha_i) = 0,\ \overline{\chi}_i \geq 0, \tag{6}$$

$$\underline{\chi}_i(\delta_i - \phi_i) = 0,\ \underline{\chi}_i \geq 0, \tag{7}$$

$$\sum\nolimits_i \delta_i = M,\ \theta \geq 0. \tag{8}$$

In the following, we prove that the solution by Algorithm 1 satisfies the KKT conditions. After Algorithm 1, the set G is partitioned into three subsets as follows: $S_1 = \{i | \delta_{i \in S_1} = \phi_i\}, S_2 = \{i | \delta_{i \in S_2} \in (\phi_i, \alpha_i)\}, S_3 = \{i | \delta_{i \in S_3} = \alpha_i\}$. Set $\Delta_{left} = \max\{\Delta_{i \in S_1}\}, \Delta_{right} = \min\{\Delta_{i \in S_3}\}$. From the searching process by the ***while*** loop in Algorithm 1, we know that: $(i)\Delta_{left} < \Delta_{right}, (ii)\Delta_{i \in S_1} \leq \Delta_{left}, (iii)\Delta_{left} < \Delta_{i \in S_2} < \Delta_{right}, (iv)\Delta_{i \in S_3} \geq \Delta_{right}$. For $i \in S_2$, $\overline{\chi}_i = \underline{\chi}_i = 0$; so, by Eq. (5), we have: $\theta = \Delta_i$, $\Delta_{left} < \theta < \Delta_{right}$. For $i \in S_1$, $\overline{\chi}_i = 0$; so, by Eq. (5), we have:

$$\underline{\chi}_i = \theta - \Delta_i > \theta - \Delta_{left} \geq 0.$$

For $i \in S_3$, $\underline{\chi}_i = 0$; so, by Eq. (5), we have:

$$\overline{\chi}_i = \Delta_i - \theta > \Delta_{right} - \theta \geq 0.$$

Therefore, all KKT conditions are satisfied and the solution is optimal. ■