Multi-Level Adaptive Hierarchical Scheduling Framework for Composing Real-Time Systems

Nima Moghaddami Khalilzad, Moris Behnam and Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
nima.m.khalilzad@mdh.se

Abstract

Processor partitioning and hierarchical scheduling have been widely used for composing hard real-time systems on a shared hardware platform while preserving the timing requirements of the systems. Due to the safety critical nature of the hard real-time systems for deriving the sufficient partition size often conservative analysis is used. Applying the exact same analysis for deriving the partition sizes for soft real-time systems result in unnecessary processors overallocation and consequently waste of the CPU resource.

In this paper, to address the problem of composing soft and hard real-time systems on a resource constrained shared hardware, we present a multi-level adaptive hierarchical scheduling framework. In our framework, we adapt the processor partition sizes of soft real-time systems according to their need at each time point by on-line monitoring their processor demand. Furthermore, we implement our adaptive framework in the Linux kernel and show the performance of our framework using a case-study.

I. Introduction

The recent advances in hardware technologies enable a new opportunity to compose previously separated (both hardware and software) systems into a single hardware platform. To this end, an enormous number of works has been done on developing mechanisms to ease the composition process. When dealing with real-time systems, the composition of a number of such systems should hold the established timing properties of the individuals before the composition. This issue has been addressed in the hierarchical scheduling literature in recent years where a number of methods has been proposed. In these methods, the composition can be done safely with respect to the timing requirements of the real-time systems.

A popular approach to deal with this problem is to divide the CPU into a number of CPU reservations (often called CPU partitions) and to assign each real-time system to one reservation. In doing so, we can derive the minimum supply of the CPU resource to each partition and compare that with the CPU demand of the system assigned to that partition. The CPU demand of systems is derived based on the Worst Case Execution Time (WCET) of their inner tasks.

While the problem of composing hard real-time systems is well studied, composition of soft real-time systems together or with hard real-time systems has not received as much attention. There exists a group of real-time tasks in which the WCET is not known a prior and even if it is known, the worst case is far much larger than the average case execution time. For instance, a video decoder application where its execution time is depending on the content of the input video may experience significantly large variation in its execution time depending on which video that is being played. Or consider a control task controlling a physical environment, where depending on its sensor value, its CPU demand may significantly change. Therefore, given that occasional deadline misses are acceptable in most of these systems, reserving the CPU resource based on the WCET may result in a great deal of resource overallocation. This overallocation may be infeasible in resource constrained embedded systems.

Moreover, dynamic real-time tasks may be organized hierarchically in systems. For instance consider virtual operating systems running on a processor where in order to guarantee the timing requirements of the virtual operating systems, they are running inside a CPU partition. In addition, inside each operating system there may be a number of independent real-time applications running in parallel. In order to isolate the timing behavior of the applications, the CPU share of the operating systems might also be partitioned, and each application may be assigned to a CPU partition of its parent operating system. Similarly, applications may contain sub-applications which can result in further CPU partitioning.

In this paper we propose a multi-level hierarchical scheduling scheme for composing hard and soft real-time systems. We allocate static CPU reservations to hard

real-time systems and adaptive reservations to soft real-time systems. We propose an adaptation mechanism for adjusting the CPU reservation sizes at each time point based on the CPU demand of the components inside the CPU reservations. Then, we show that using our adaptation techniques, adapting the soft real-time partition sizes does not affect the timing behavior of the hard real-time partitions. Thereafter, we first implement the framework as a Linux kernel loadable module and then evaluate the performance of our proposed framework using a case-study. In the evaluation, we use real applications such as video player and image processing applications which exhibit wide CPU demand variations.

The rest of the paper is organized as follows. In Section II we review the related work. Section III presents our adaptive framework and reveals the details of our adaptation mechanism. The overview of the Linux implementation is presented in IV. We present a case-study in Section V and show the performance of our framework. Finally, we conclude the paper in Section VI.

II. Related work

Hierarchical scheduling through CPU reservation emerged in 90's [1], where the idea was to partition the CPU into a number of partitions (CPU reservations) and assign a partition to a group of tasks. In doing so, the task groups experience temporal isolation, i.e., the timing behavior of each task group (also called a component) can be studied independently. The partitioning also paves the way for adding and removing components without jeopardizing the timing requirements of other components in the system.

When partitioning the CPU, we need a model that captures the amount of provided resource at each time point. The resource partition model is presented in [2]. Schedulability analysis for hierarchical scheduling with fixed priority at the global level and local EDF is presented in [3]. In [4] Shin and Lee present the periodic resource model in which the CPU reservation is achieved using periodic servers. The periodic servers provide their inner components with Q units of the CPU time each P time units. We use the periodic resource model in our adaptive scheme.

When using the CPU partitioning approach, the common assumption is that the CPU demand of the real-time tasks (WCET) are known a prior. Given this demand, a sufficient partition size can be calculated such that the timing requirement of real-time tasks are not violated. However, we assume that for soft real-time tasks the task demands and therefore the sufficient partition size are unknown.

Since Stankovic *et al.* introduced the idea of closed-loop real-time scheduling [5], there has been a growing interest in adopting feedback control techniques in the context of real-time scheduling. In [6] Feedback Control EDF (FC-EDF) is presented in which there is a PID controller on top of the EDF scheduler. The controller monitors the deadline miss ratio and based on that adjusts the tasks requested CPU utilization value. This adjustment affects the available CPU utilization and therefore the admission control. Targeting control tasks Cervin *et al.* presented a feedback-feedforward scheme in which by adapting the sampling period of tasks the quality-of-control is regulated [7].

In order to ensure that the multimedia applications receive enough CPU bandwidth, resource partitioning is used for scheduling multimedia tasks [8]. Due to the dynamic nature of multimedia applications (with respect to execution time), it is desirable to have adaptive reservations for these kinds of applications. Abeni *et al.* proposed using a PI controller on top of Constant Bandwidth Servers (CBS) which adapts the CBS bandwidth such that it tracks the current workload of tasks attached to the CBS.

Utilizing adaptive CBS (with a new control scheme), Palopoli *et al.* present the AQuoSA framework [9]. This scheme only considers the existence of one task per each CBS. In order for it to work in hierarchical settings where multiple tasks exist in each server, a new controlled variable should be defined.

In the context of the ACTORS project [10] a cascade controller is used on top of the hard CBS scheduling algorithm for adapting the CBS bandwidth. In this work alike the AQuoSA framework the authors also use one task per CPU reservation.

Finally, we studied the problem of budget adaptation using PI controllers [11] and Fuzzy controllers [12]. In our aforementioned previous work we have investigated two-level hierarchical scheduling, however, in this paper we present a new budget adaptation scheme which supports any arbitrary level of hierarchy in hierarchical scheduling. The new scheme also takes the existence of hard real-time systems into account and we show that adapting the soft real-time partitions does not harm the hard real-time partitions. Moreover, in our previous papers we performed simulation-based evaluations whereas in this paper we evaluate our framework by implementing our multi-level adaptive hierarchical scheduling framework in the Linux kernel and by running real applications.

III. Framework

In this section we present the structure of our adaptive framework and our adaptation mechanism.

A. Application model

An application ($\mathcal{A}_j$) consists of m_j real-time tasks (τ_i^j) and n_j sub-applications (A_κ^j). When referring to both tasks

and applications in $\mathcal{A}_j$, we use the term "inner components" of $\mathcal{A}_j$. Each application is assigned a periodic server S_j with period P_j, budget B_j and importance ζ_j. The periodic server provides the application with B_j units of the CPU time every P_j time units. The importance value ζ_j shows the relative importance of applications with respect to other applications that coexist in the system belonging to the same parent. The importance value is only used when the system is overloaded in which we have to serve the more important applications at the price of sacrificing less important applications. Tasks and applications may join and/or leave the application during run-time. We define the bandwidth of applications (α_j) as follows: $\alpha_j = B_j/P_j$.

We distinguish two types of applications in our framework: Hard Real-Time (HRT) and Soft Real-Time (SRT) applications. Inner components of applications inherit their parent type, meaning that e.g., if an application is HRT then its children components are also HRT. HRT applications receive fixed CPU reservations during their life-time in the system which is calculated using the analysis provided in [4], whereas SRT applications receive dynamic CPU reservations. Therefore, the budget in an SRT application is changing and it is a function of time $B_j(t)$.

B. Task model

We consider a periodic task model in which the following parameters are associated with a task τ_i^j that belongs to application $\mathcal{A}_j$: period T_i^j, deadline D_i^j, priority pr_i^j and the worst case execution time C_i^j. Without loss of generality, we assume implicit deadline model, i.e., $T_i^j = D_i^j$. Note that C_i^j is only available for hard real-time tasks, however, we assume that the soft real-time task execution times are unknown a prior and that they possibly have significant variations during tasks life-time in the system. The soft real-time tasks may miss their deadlines due to insufficient CPU allocation, then the remaining execution time ϵ_i^j has to be executed in the next task periods under its parent partition. A new instance of tasks is allowed to be executed only when its previous instances has completed its execution. One instance of the task execution is called a "job".

C. System model

We assume a single processor system composed of n SRT applications and m HRT applications. As mentioned in the application model, an application itself may be composed of a number of sub-applications and/or tasks. Therefore, our systems model is a multi-level hierarchical model. The scheduling is also done hierarchically. The global scheduler is responsible for scheduling the global level periodic servers which serve the global level applications. Each application in turn has its own local scheduler which distributes the application's assigned CPU portion among its inner components. In the case of SRT application servers, there is a budget controller component attached to each server which is responsible for adapting the budget of servers. Figure 1 illustrates our system model. We refer to the set of all, SRT and HRT applications at the root level using $\mathcal{A}^r$, $\mathcal{A}^{srt}$ and $\mathcal{A}^{hrt}$ respectively.

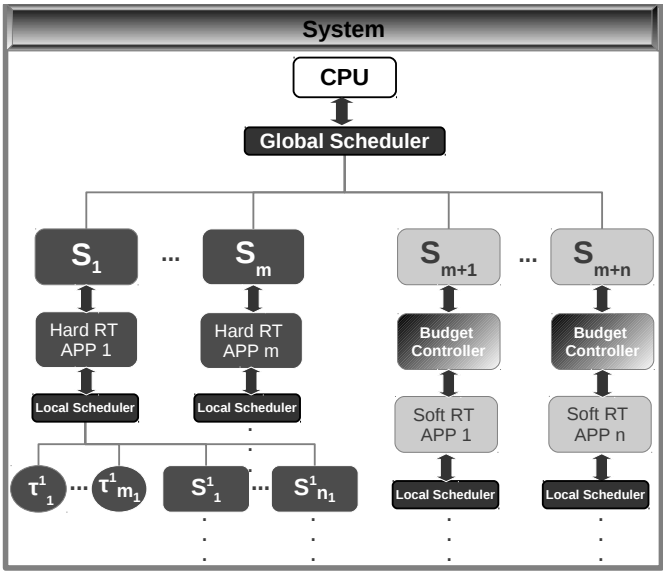

Fig. 1. Visualization of the system model.

D. Adaptation model

Assuming that the system designer selects the application periods, we keep the server periods fixed, and we only adapt the budgets such that each application receives just enough CPU time for scheduling its inner components. For instance the period of an application consisting of video decoder tasks is set based on the requirement on the minimum number of frames needed per second. However, the budget totally depends on the content of the video at each time point. Note that only SRT applications are subjected to the adaptation.

The budget adaptation is done periodically at each P_j^{Ctrl} time units by the budget controller component. We assume that the controller period P_j^{Ctrl} is proportional to its corresponding server period $P_j^{Ctrl} = \mu \times P_j$. Therefore, at every μ server release events we call the controller function and adapt the server budgets.

We call each trigger of the controller component a "control event" and we represent the control event sequence by k where $k \geq 1$. The first control event for each server S_j ($k_j = 1$) happens at time $t = P_j^{Ctrl}$. Let us define $\Psi_j^{k_j}$ as the k'th "control period" of $\mathcal{A}_j$ which represents the following time window:

$$\Psi_j^{k_j} = \Big((k_j - 1)P_j^{Ctrl}, k_j P_j^{Ctrl}\Big].$$

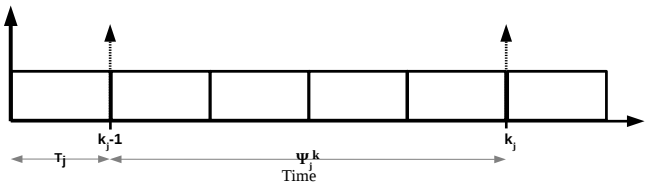

Fig. 2. Visualization of the control period Ψ_j^k and the control event k_j.

The first control period of server S_j (Ψ_j^1) is as follows: $\Psi_j^{k_j} = (0, P_j^{Ctrl}]$. The control event and the control period concepts are visualized in Figure 2. In the figure the controller period is four times the server period ($\mu = 4$).

For keeping the equations simple, we drop index j when referring to k_j and Ψ_j^k. At each control event k monitoring the CPU demand of application $\mathcal{A}_j$, we assign a sufficient budget $B_j(k)$ to the application server S_j. Assuming that we know the CPU demand of the application $\mathcal{A}_j$ inner components during the next control period Ψ^{k+1}, the sufficient budget for S_j is as follows:

$$B_j(k) = \lambda \times \Big(b_j(k) + r_j(k)\Big), \tag{1}$$

where $b_j(k)$ is the amount of total required budget for Ψ^{k+1} (we assume we know it) and $r_j(k)$ is the amount of work, in terms of the CPU time, that should have been processed during Ψ^k but it is postponed to Ψ^{k+1}. We explain how to measure the remaining work $r_j(k)$ in the rest of this section. λ is a coefficient that specifies how much of the total budget should be allocated at each server period. If the control period is equal to the server period ($P_j^{Ctrl} = P_j$) then $\lambda = 1$. However, in order to decrease the control overhead we may assign a control period that is larger than the server period. Therefore, in general case $\lambda = \mu^{-1}$.

E. Controlled parameters

In order to realize the workload of the applications we need to monitor some parameters while doing the scheduling, these parameters are called "controlled parameters".

Since the periodic servers idle their budget when the server is active and there is no running task or active application in the server, we monitor the server's budget to see how much of the assigned budget is used in practice. This parameter is called the actual required budget β_j. For instance when the controller realizes that $\beta_j(k) < \mu \times B_j(k)$ it may adapt the budget of application $\mathcal{A}_j$ at the next control event $k+1$ and assign a lower budget to this application. $\beta_j(k)$ is visualized in Figure 3.a. In this figure $\mu = 2$, therefore at time k the controller will be triggered and will see that $\beta_j(k) = x_1 + x_2$.

There should also exist a mechanism for detecting budget deficiency. There are two different sources of budget

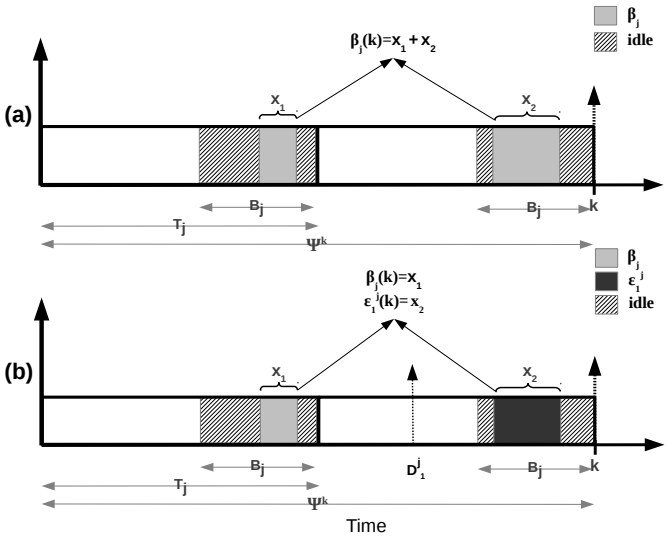

Fig. 3. Visualization of the controlled parameters: a) $\beta_j(k)$ b) $\epsilon_i^j(k)$.

deficiency: i) an application's inner tasks are suffering, i.e., tasks are missing their deadlines ii) an application's inner applications are suffering, i.e., they do not receive their assigned budget at each period. For monitoring (i) we measure the amount of task execution time after passing the deadline (ϵ_i^j) which can be translated as the amount of budget deficiency of the parent. For instance, assume that in Figure 3.b τ_1^j is the only component in $\mathcal{A}_j$ and that it misses its deadline at D_1^j, therefore the amount of consumed budget after the deadline $\epsilon_1^j(k)$ is equal to x_2.

On the other hand, at each server period when a new instance of the server is being released, if the controller observes that the server's previously assigned budget is not exhausted (it is neither used by its inner components nor idled) it realizes that the server's parent is suffering from a budget deficiency. We represent the amount of unallocated budget to $\mathcal{A}_\kappa^j$ using δ_κ^j.

1) Remaining workload ($r_j(k)$): Recall that we assume $T_i^j = D_i^j$. Therefore, when task τ_i^j in control period Ψ^k misses its deadline and then it consumes ϵ_i^j time unites of the budget, it actually consumes the budget of its next instance. Therefore, even if we know the exact amount of tasks required budget in Ψ^{k+1} and we assign this budget to S_j at k, the task will still miss its deadline because ϵ_i^j amount of its budget is used by its previous instance and this domino effect will affect all of the later jobs of τ_i^j. Hence, in order to stop this deadline miss domino at each control event we compensate the postponed workload by adding $r_j(k)$ units to $B_j(k)$. On the other hand, $\mathcal{A}_\kappa^j$ may receive lower budget than its assigned budget, and therefore (δ_κ^j) workload will be postponed from Ψ^k to Ψ^{k+1}. Note that the servers treat their inner components equally, i.e., when a server receives δ_κ^j budget units less than its assigned budget, for the parent server

this case is equal to the case that a task misses its deadline and consumes ϵ_i^j units of budget after its deadline. This abstraction makes it possible to have any arbitrary levels of hierarchy in the system. Therefore, we can derive the amount of remaining workload of $\mathcal{A}_j$ from Ψ^k that should be compensated in Ψ^{k+1}:

$$r_j(k) = \sum_{i \in S_j} \epsilon_i^j(k) + \sum_{\kappa \in S_j} \delta_\kappa^j(k). \quad (2)$$

2) When $r_j(k) > 0$ and $\beta_j(k) < \mu \times B(k-1)$: In the following two circumstances a server can idle its budget while its inner components are suffering from low CPU allocation: i) when the server period and its inner components periods are not aligned, ii) when the workload is decreased at the beginning of the control period and increased later at the same control period. In these circumstances the controller should provide more budget to the application. To this end, we take the following steps at each control event:

- The remaining workload $r_j(k)$ is calculated using Equation 2.
- If $r_j(k) > 0$ we overwrite the actual required budget such that $\beta_j(k) = \mu \times B_j(k) + \lambda \times r_j(k)$. This action means that whenever there is a budget deficiency then the actual required budget was $\lambda \times r_j(k)$ units more than the previously assigned budget. Since $\beta_j(k)$ is used for estimating the future workload, this overwriting the history indeed is a signal to the controller to increase the budget for Ψ^{k+1}.

F. Estimating the future workload

In Equation 1 we assume that we know the next workload, however, in reality we have no information about the next workload and predicting the exact amount of the workload is rather difficult. Inspired by the AQuoSA framework [9], we use a workload predictor component that estimates the next workload and based on that prediction we assign a budget to the applications. Figure 6 shows the CPU utilization percentage (U %) at each server period by applications in the case-study used in our evaluations in Section V. For instance $\mathcal{A}_3$ is composed of three tasks that have dynamic CPU demands. Note that to capture this figure we have assigned almost 100 % of the CPU time to the applications, in order to observe the CPU demand of these applications. The figure depicts large variations in the CPU demand which in practice makes the job of the workload estimator component difficult. When there is only one task in an application, and we can model the task execution, then the model can be used in predicting the future workload. However, when increasing the number of tasks in each application then the execution pattern becomes more fuzzy and the workload prediction becomes more difficult.

Our approach for estimating the future workload is based on the assumption that the future follows the trend of the past. Of course when the future is totally different compared to the past (e.g., an application or a task leaves or joins the system), then our estimator will make mistakes and the system may suffer for a while until the estimator makes "good enough" estimations. We model the CPU demand of the tasks using the Autoregressive (AR) model, therefore considering h previous $\beta_j(k)$ we have:

$$b_j(k) = \sum_{k-h}^{k-1} w_k \beta_j(k) + e_k, \quad (3)$$

where w_k is the weight of observation k and e_k is Gaussian white noise.

G. Dealing with overload situations

The budget controller first calculates the available budget av_κ^j for its application by excluding the bandwidth of the other applications from the bandwidth of its parent α_j:

$$av_\kappa^j = \alpha_j - \sum_{q \in \mathcal{A}_j \wedge q \neq \kappa} \alpha_q.$$

Assuming that we use EDF at the root level, the total bandwidth at the root level is 1. The controller also calculates the maximum possible budget ma_κ^j by only excluding the bandwidth of higher importance applications from its parent's bandwidth:

$$ma_\kappa^j = \alpha_j - \sum_{q \in \mathcal{A}_j \wedge \zeta_q > \zeta_\kappa} \alpha_q.$$

Afterwards, based on Equation 1, the budget controller computes a new budget for applications new_κ^j. Then the following three situations can happen:

1) $new_\kappa^j \leq av_\kappa^j$: The controller assigns the new budget to $\mathcal{A}_\kappa$: $B_\kappa^j(k) = new_\kappa^j$.
2) $av_\kappa^j < new_\kappa^j \leq ma_\kappa^j$: The controller assigns the new budget to the application ($B_\kappa^j(k) = new_\kappa^j$) but it compensates $\Delta B = B_\kappa^j(k) - av_\kappa^j$ by taking the bandwidth from lower importance applications. The budget stealing process is done in the reverse order of the application importances.
3) $new_\kappa^j > ma_\kappa^j$: The controller assigns ma_κ^j to the application ($B_\kappa^j(k) = ma_\kappa^j$) and similar to case (2) it compensates ΔB by stealing budget from the lower importance applications.

The aim behind stealing budget from lower importance applications is to make sure at each k we have:

$$\sum_{j \in \mathcal{A}^r} \alpha_j \leq 1.$$

Example 1: Assume a system composed of three SRT applications $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ where $\zeta_1 > \zeta_2 > \zeta_3$,

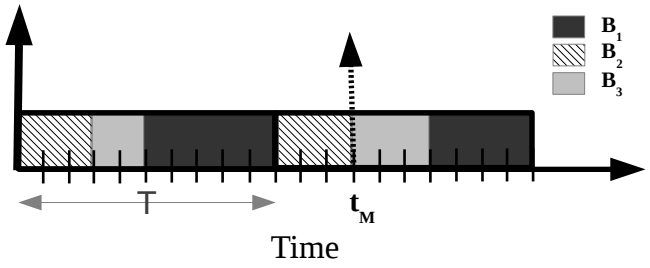

Fig. 4. Mode change hazard.

$P_1 = P_2 = P_3 = 10$, $B_1(k-1) = 5$, $B_2(k-1) = 3$ and $B_3(k-1) = 2$. At control event k the controller decides to increase the budget of $\mathcal{A}_2$ to four. Therefore, we have: $new_2 = 4$, $av_2 = 3$ and $ma_2 = 5$. In this situation, the controller assigns $B_2(k) = new_2 = 4$ and decreases $\Delta B = 1$ from $B_3(k)$, hence, we will have $B_1(k) = 5$, $B_2(k) = 4$ and $B_3(k) = 1$.

H. Mode change

The act of adapting the application budgets can be seen as changing the system mode where in one mode the system has a set of budgets associated with the applications and in the next mode a new set of budgets are assigned to the applications. When adapting the budget of SRT applications, we should make sure that the HRT applications are not influenced by the adaptation. In this subsection we present a potential hazard that adapting the budget of SRT applications can create and we propose two approaches to address this problem.

Example 2: Assume a system composed of an HRT application ($\mathcal{A}_1$) and two SRT applications ($\mathcal{A}_2$ and $\mathcal{A}_3$) which all share the same period $P_1 = P_2 = P_3 = 10$. The schedule of this system is depicted in Figure 4. The initial budgets are as follows: $B_1 = 5$, $B_2 = 3$ and $B_3 = 2$, however, at time t_M the controller decides to change the budgets as follows: $B_2 = 2$ and $B_3 = 3$. As a result of this adaptation, B_1 which is an HRT application does not receive its five budget units in time. This example shows that although we do not change the budget of HRT applications, adapting SRT application budgets (if not done carefully at right moments) may affect the HRT applications.

1) Using only one SRT application at the root level: A simple way to avoid this problem is to have only one SRT node at the root level ($\mathcal{A}_s$) and make sure that $B_s \leq U' \times P_s$ where assuming EDF at the root level

$$U' = 1 - \sum_{j \in \mathcal{A}^{hrt}} \alpha_j,$$

and $\mathcal{A}^{hrt}$ is the set of all HRT applications at the root level of the hierarchy.

2) Mode change protocol: If we use more than one SRT application at the root level, then the budgets should be updated based on a protocol that guarantees safe mode changes. In multi-mode real-time system literature [13] there are two types of protocols: i) synchronous protocols in which the new-mode tasks are not released until the old-mode tasks are finished ii) asynchronous protocols where both the old-mode and the new-mode tasks can run together. Type (i) protocols guarantee the mode change by introducing offset value for the release of the new-mode tasks. This offset is not desirable in our framework since our objective is to achieve the best possible performance. In type (ii) protocols, schedulability analysis is used to check whether the mode change is safe or not. However, we present a protocol that does not introduce offsets for release of servers, and by following that the mode change is guaranteed to be done safely.

Assuming that we use EDF as our global scheduler, first we should show that the system is schedulable in all modes. Hence, the first condition for a safe budget adaptation is that the controller adapts the budgets in such a way that:

$$\forall\, k \sum_{j \in \mathcal{A}^r} \frac{B_j(k)}{P_j} \leq 1.$$

However, since the utilization of HRT applications is fixed we can exclude it and only check the condition for the utilization of the SRT application. Therefore:

Condition 1:

$$\forall\, k \sum_{j \in \mathcal{A}^{srt}} \frac{B_j(k)}{P_j} \leq U'.$$

We should also make sure that the transition is safe, hence, assuming that the mode change happens at time t_M, we should show that in all time windows $(t_0, t_1]$ where $t_1 - t_0 = L$ and $t_0 < t_M < t_1$, the following condition holds:

Condition 2:

$$\texttt{dbf}(\mathcal{A}^{srt}, L) \leq U' \times L,$$

where `dbf` returns the demand bound function of its input application set and its input duration. Given that we know the budgets in each mode, we can find an upper bound for `dbf`:

$$\texttt{dbf}(\mathcal{A}^{srt}, L) \leq L \times \sum_{j \in \mathcal{A}^{srt} \,\wedge\, k \in (t_0, t_1]} \frac{max\Big(B_j(k)\Big)}{P_j}.$$

Consequently, it is sufficient to show:

$$\sum_{j \in \mathcal{A}^{srt} \,\wedge\, k \in (t_0, t_1]} \frac{max\Big(B_j(k)\Big)}{P_j} \leq U'.$$

Since we use periodic servers it is safe to evaluate time windows with the following length range:

$$0 \leq L \leq LCM(\mathcal{A}^{srt}),$$

where LCM returns the least common multiple of the periods of its input set. The LCM might be a large number, therefore similar to [14] we can find a smaller range for L. However, this problem is out of the scope of this paper. Condition 2 reflects the fact that in the transition mode, the demand depends on the maximum of the budgets assigned in all modes.

We enforce Condition 1 by applying the overload control mechanism as explained earlier in this section. As a result, we only need to address Condition 2 in our mode change protocol.

The Decrease, Wait, Increase (DWI) protocol: In each mode change there are two types of changes: i) the budget of some applications should get increased ($\mathcal{A}^{inc}$) ii) the budget of some other applications should get decreased ($\mathcal{A}^{dec}$). We immediately apply the budget decreases, then we wait for LCM of $\mathcal{A}^{srt}$ periods, finally we apply the budget increases. Note that the DWI protocol does not delay the release of the servers and it only delays the budget increases.

Lemma 1: The DWI protocol fulfills Condition 2.

Proof: We prove the lemma by contradiction. Assume that the system is changing its mode from mode one with budget B_j to mode two with budget B'_j, and also assume that Condition 2 does not hold, hence:

$$\exists\, \mathcal{W} \;\Big|\; \mathcal{W} \leq LCM(\mathcal{A}^{srt}) \wedge \sum_{j \in \mathcal{A}^{srt} \wedge k \in \mathcal{W}} \frac{max\Big(B_j(k)\Big)}{P_j} > U'.$$

Given that Condition 1 holds in both modes:

$$\exists\, \mathcal{W}, i, j, k, k' \;\Big|\; \mathcal{W} \leq LCM(\mathcal{A}^{srt}) \wedge\; k, k' \in \mathcal{W}$$

$$\wedge\, B_j^{dec}(k) = B_j \;\wedge\; B_i^{inc}(k) = B'_i,$$

however, according to the DWI protocol there is at least $LCM(\mathcal{A}^{srt})$ time unites distance between B_j and B'_i and they can not happen in the same $\mathcal{W}$. Hence, Lemma 1 is proved. ■

IV. Implementation

Inspired by [15] we have implemented our adaptive hierarchical scheduling framework as a Linux kernel loadable module. We first load our module in the Linux kernel and then each application needs to register itself to the module. Our framework supports any arbitrary levels of periodic servers hierarchy in which each server has its independent ready queue i.e, the local schedulers of the servers may be different. We support both EDF and FP scheduling algorithms [1]. We have a data structure for storing task and server information. There is a one to one relation between the Linux tasks and the tasks defined in our framework.

In summary, the user first needs to define the structure (tasks, servers and their position in the hierarchy) of the target hierarchical system using a number of API functions. Thereafter, when a task attaches itself to one of the predefined tasks in our loadable module, the module changes its scheduling policy to "SCHED_FIFO". Whenever the task has to run, using "wake_up_process(task)" the module runs the task and whenever the task has to stop by changing its state to "TASK_UNINTERRUPTIBLE" it is stopped.

Since we assume a periodic task model, there is also an API function that the tasks have to call whenever they finish their execution period to inform the module that they need to sleep until their next release. We use the "timer_list" structure in the Linux kernel to handle the scheduling events such as server release, server budget depletion and task release.

The controller is also implemented as a function in the kernel loadable module. At each server release event the controller function is called and based the value of μ it either directly returns or adapts B_j.

A. Monitoring the controlled parameters

When running servers and tasks we store the start time in their corresponding data structure. When stopping them, we calculate the distance between the stop time and the start time. In doing so we update β of their parent application. When a task is stopped (it is either end of its job or a preemption), we compare the stopped time by its deadline. If the deadline is missed we update the value of ϵ_i, otherwise, we only update their parent β.

V. Evaluation

In this section we evaluate our framework by conducting a case-study. We first present the tasks and the system used for the case-study, and then we present the results.

A. Tasks

We use two types of tasks that have highly varying execution times: i) MPlayer video player [2] ii) image processing task developed using openCV libraries which basically filters a range of colors of its input video. On the

[1] In the case of using FP scheduling algorithm, the overload control mechanism should be modified such that the EDF utilization bound is replaced by the FP utilization bound in the equations.

[2] http://www.mplayerhq.hu

	Hard-Soft	Type	$P_j - T_i^j$	$B_j(0)$
S_1	HRT server	-	100	40
τ_1^1	HRT task	static	200	-
S_2	SRT server	-	10	6
S_3	SRT server	-	20	8
τ_1^2	SRT task	MPlayer	40	-
τ_2^2	SRT task	image processing	200	-
τ_3^2	SRT task	image processing	250	-
S_4	SRT server	-	100	20
τ_1^3	SRT task	image processing	350	-
τ_2^3	SRT task	image processing	200	-

TABLE I. Specification of servers and tasks of the case-study.

other hand for simulating tasks with constant execution times we use a C program with a loop that increments a variable a each iteration. This type of tasks is called a "static task".

B. Evaluation setup

We use Ubuntu 12.04.2 with Linux kernel version 3.8.2 on an Intel CORE i5 processor in which only CPU 0 is active. The scheduler resolution (system tick) is set to one milliseconds.

The weight values in the workload predictor component (Equation 3) are all set to $1/h$. We set $e_k = 1/2 \times std(\beta_j(k), h)$ where std returns the standard deviation of its h previous $\beta_j(k)$. Since the controller has negligible overhead (see Section V-D) we set $\mu = 1$. Based on the suggestion in [4], the server periods are set to half of their shortest inner component period.

C. Case study

In order to show the performance of our framework we design a sample system consisting of four applications. The structure of the sample system and its specification are presented in Figure 5 and Table I respectively. The period and initial budget values are all in milliseconds. We assume that the applications are ordered based on their importance, i.e, $\zeta_3 > \zeta_4$. Therefore, in overload situations the controller will take bandwidth from $\mathcal{A}_4$ and give it to $\mathcal{A}_3$. Note that we use different local schedulers in the applications. We assume that the tasks in $\mathcal{A}_4$ are ordered according to their priority, i.e, $pr_1^4 > pr_2^4$.

The workload variations for $\mathcal{A}_1$, $\mathcal{A}_3$ and $\mathcal{A}_4$ are shown in Figure 6. In the figure we show the percentage of the CPU demand (U) by each application. For getting these values we have executed the applications alone and measured their CPU demand. Therefore when composing all of them, their peaks in CPU utilization might match at some points in time and cause overload situation. In addition to the execution time changes there is another source of workload variations in the sample system. τ_3^3

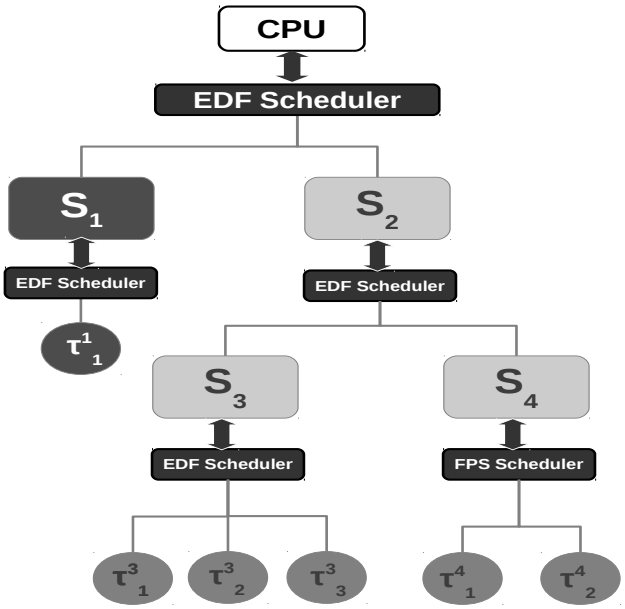

Fig. 5. Structure of the sample system used in the case-study.

and τ_1^4 leave the system after around 62 and 104 seconds respectively which causes a drop in the workload of their applications, and hence the budget of the application is decreased by the budget controller (see Figure 7).

We ran the system for two minutes and here we report the results for different configurations. Table II shows the Deadline Miss Ratio (DMR_j) and the average bandwidth ($\overline{\alpha_j}$) of $\mathcal{A}_j$ for different values of h. Recall from Equation 3 that the predictions of the workload estimator is based on h previous observations. The DMR_j is calculated by dividing the sum of job deadline misses of the tasks in an application by the total number of completed jobs in the same application. The results suggest that for $h \geq 3$ the estimator component is able to make acceptable predictions and hence the system reaches an overall deadline miss ratio less than nine percent. For $h \geq 3$ there is no significant changes, however, $h = 10$ gives slightly better deadline miss ratio. At the same h value $\overline{\alpha_j}$ is also minimum among other configurations. Note that lower $\overline{\alpha_j}$ means that the budget controller has assigned tighter budgets. When the applications have tighter budgets, then the admission control may accept more applications into the system and the lower importance SRT applications may receive more bandwidth if they need.

Figure 7 shows the bandwidth adaptation of the two SRT applications that are inside $\mathcal{A}_2$. Note that $\mathcal{A}_2$ is also adapting its bandwidth, however, since its children require all U' bandwidth most of the time, B_2 is almost fixed during the two minutes experiment.

For a group of real-time tasks, even missing the deadline point may be acceptable if the tasks finish their job execution close enough to their deadline point. To this end,

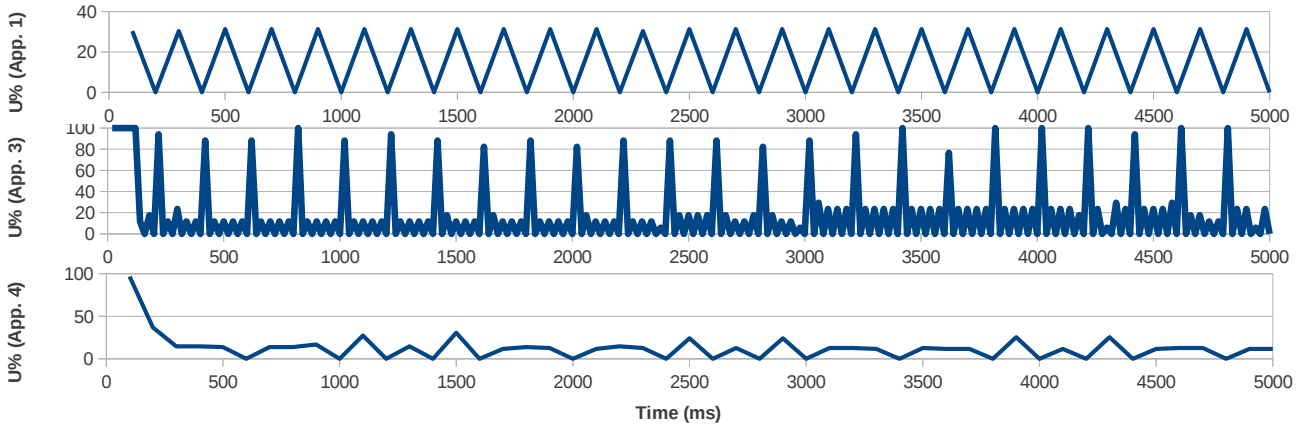

Fig. 6. Workload of $\mathcal{A}_1$, $\mathcal{A}_3$ and $\mathcal{A}_4$ in terms of CPU Utilization (U).

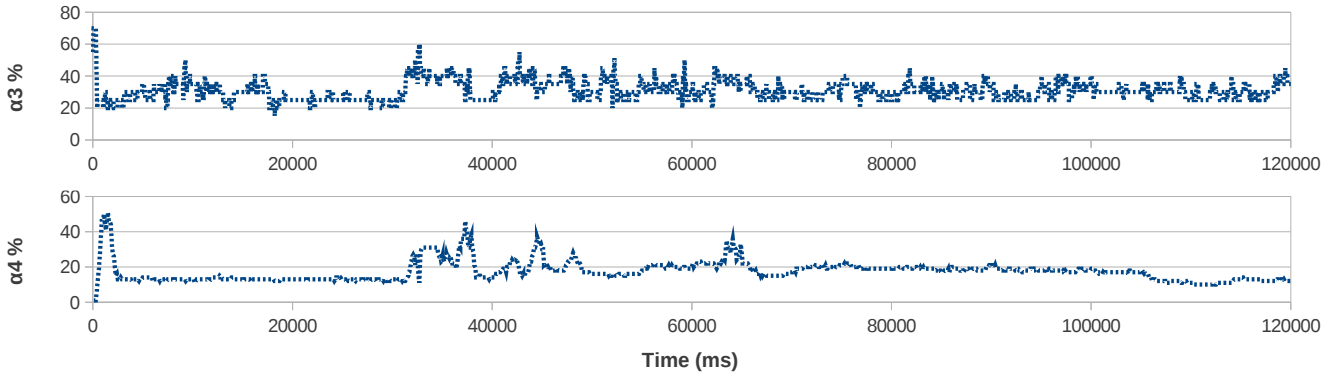

Fig. 7. Bandwidth adaptation of $\mathcal{A}_3$ and $\mathcal{A}_4$.

h	server	DMR_j	$\overline{\alpha_j}$
1	S_3	3.99 %	31.61 %
	S_4	52.43 %	14.91 %
2	S_3	0.50 %	30.92 %
	S_4	20.01 %	15.57 %
3	S_3	0.67 %	31.06 %
	S_4	8.89 %	16.55 %
5	S_3	0.20 %	32.15 %
	S_4	9.12 %	18.00 %
10	S_3	0.12 %	30.88 %
	S_4	8.12 %	17.79 %
20	S_3	0.39 %	31.61 %
	S_4	8.89 %	18.10 %

TABLE II. DMR_j and $\overline{\alpha_j}$ of the case-study.

we use the notion of job tardiness (Θ_i) [16] which shows the distance of a job deadline point and its execution end time. Note that $\Theta_i < 0$ means that the task has missed its deadline while $\Theta_i \geq 0$ means that the task has finished its execution in time. Figure 8 shows the tardiness of all tasks in the case-study during the two minutes experiment. Note that τ_3^3 and τ_1^4 leave the system before the experiment finishes. The figure illustrates that in the higher importance application $\mathcal{A}_3$, τ_1^3 experience the worst tardiness, however, $\Theta_1^3 \geq -40$ which means that in the worst case a video frame is decoded and displayed one frame later than its originally supposed time (the task displays one frame each 40 milliseconds which results in 25 frame per second).

Now assume that before running the system we already know the average required bandwidth of each application. From Table II we observe that in the best case $\alpha_3^{av} \simeq 31$ % and $\alpha_4^{av} \simeq 18$ %. Based on these bandwidth estimations, in another experiment with the exactly same setting as the first experiment we assign fixed budgets for the applications. The result is a 0.67 % deadline miss ratio for $\mathcal{A}_3$ while a 30.23 % deadline miss ratio for $\mathcal{A}_4$. Note that since $\mathcal{A}_3$ has shorter deadline than $\mathcal{A}_4$ it is more likely that $\mathcal{A}_3$ will be the winner when competing with $\mathcal{A}_4$ to get the CPU (their parent scheduler is EDF). This experiment reveals that even with the knowledge of the average workload, the bandwidth adaptation is a necessity due to workload variations.

D. Overhead of the adaptation

We have measured the overhead of calling the controller function at each server release event. The overhead in average is around one microsecond per each function call. The changes of overhead for $1 \leq h \leq 20$ is not significant.

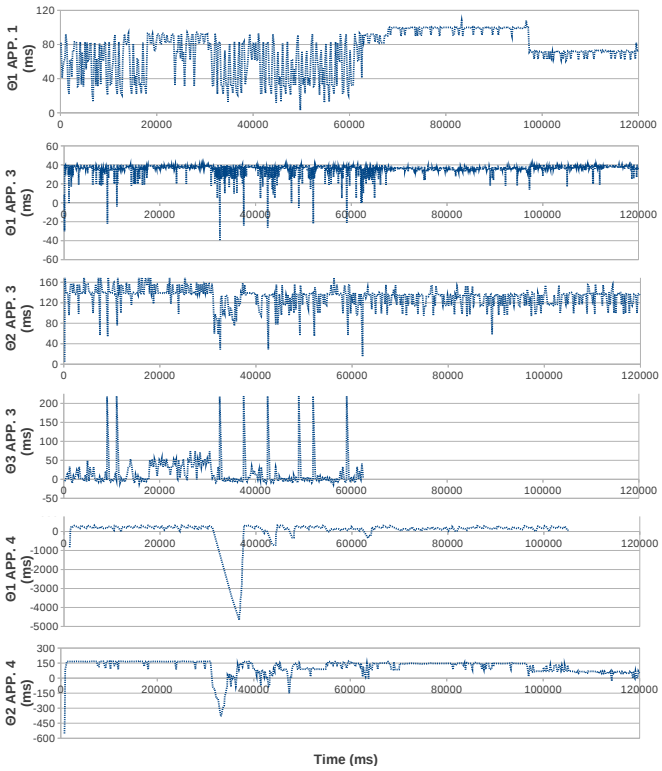

Fig. 8. Tardiness of the tasks.

Therefore, considering the sample system in the case-study the total overhead is around 0.016 % of the CPU time.

VI. Conclusion

In this paper we presented a multi-level adaptive hierarchical scheduling framework in which based on feedbacks from the system we assign adaptive CPU partition sizes to soft real-time applications. We showed that by imposing a negligible overhead (less than 0.02 % of the CPU time) we are able to serve real-time applications (especially the more important application) such that they reach an acceptable deadline miss ratio.

We intend to extend our work to multiprocessor platforms in which we will adapt the interface parameters of the virtual clusters introduced in [17]. We also want to study the use of more sophisticated workload estimator components in our framework to examine whether we could better serve the applications by more accurate workload estimations.

References

[1] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, December 1997, pp. 308 –319.

[2] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS '01)*, May 2001, pp. 75–.

[3] F. Zhang and A. Burns, "Analysis of hierarchical EDF pre-emptive scheduling," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, December 2007.

[4] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium, (RTSS '03)*, December 2003.

[5] J. Stankovic, C. Lu, S. Son, and G. Tao, "The case for feedback control real-time scheduling," in *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, June 1999.

[6] C. Lu, J. Stankovic, G. Tao, and S. Son, "Design and evaluation of a feedback control EDF scheduling algorithm," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS '99)*, December 1999.

[7] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzen, "Feedback-feedforward scheduling of control tasks," *Real-Time Systems*, pp. 25–53, 2002.

[8] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998, pp. 4–13.

[9] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA-adaptive quality of service architecture," *Softw. Pract. Exper.*, vol. 39, no. 1, pp. 1–31, January 2009.

[10] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K. Arzen, 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.

[11] N. M. Khalilzad, T. Nolte, M. Behnam, and M. Asberg, "Towards adaptive hierarchical scheduling of real-time systems," in *Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '11)*, September 2011.

[12] N. M. Khalilzad, M. Behnam, G. Spampinato, and T. Nolte, "Bandwidth adaptation in hierarchical scheduling using fuzzy controllers," in *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, June 2012.

[13] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, 2004.

[14] B. Andersson, "Uniprocessor EDF scheduling with mode change," Polytechnic Institute of Porto (ISEP-IPP), Tech. Rep., January 2008.

[15] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "Exsched: An external CPU scheduler framework for real-time systems," in *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, August 2012.

[16] A. Srinivasan and J. Anderson, "Efficient scheduling of soft real-time applications on multiprocessors," in *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, July 2003, pp. 51–59.

[17] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *Proceedings of the Euromicro Conference on Real-Time Systems, (ECRTS '08)*, July 2008, pp. 181 –190.