Swift mode changes in memory constrained real-time systems

Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Eindhoven University of Technology
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands
m.holenderski, r.j.bril, j.j.lukkien@tue.nl

***Abstract*—A method for "preempting memory" is presented, where (parts of) the memory allocated to an active task may be reallocated to another task, without corrupting the state of the active task's job. The method is based on combining scalable components with Fixed-Priority Scheduling with Deferred Preemption (FPDS).**

Real-time systems composed of scalable components are investigated. A scalable component can operate in one of several modes, where each mode defines a certain trade off between the resource requirements and output quality. The focus of this paper is on memory constrained systems, with modes limited to memory requirements. During runtime the system may decide to reallocate the resources between the components, resulting in a mode change. The latency of a mode change should satisfy timing constraints expressed by upper bounds.

A modeling framework is presented combining scalable components with FPDS. A quantitive analysis comparing Fixed-Priority Preemptive Scheduling (FPPS) and FPDS is provided, showing that FPDS sets a lower bound on the mode change latency. The analytical results are verified by simulation. The results for both FPPS and FPDS are applied to improve the existing latency bound for mode changes in the processor domain.

The presented protocol is especially suited for pipelined applications, allowing to perform the mode change without the need to first clear the whole pipeline.

I. Introduction

A real-time *application* is modeled by a set of tasks, where each *task* specifies part of the application workload associated with handling an event. Scalable applications are those which can adapt their workload to the available resources. Tasks use *components* to do the work. A task can be regarded as the control flow through the components it uses. In this paper we consider scalable applications which can operate in one of several predefined modes. A *system mode* specifies the resource requirements of the application task set, in particular the resource requirements of the components used by the task set. In a resource constrained system, not all components can operate in their highest modes (i.e. modes requiring the most resources) at the same time. Some components will need to operate in lower modes. During runtime the system may decide to reallocate the resources between the components, resulting in a *mode change*. Many applications pose a requirement on the *latency* of a mode change. In this paper we investigate bounding this latency in memory constrained real-time system.

As an example application we consider a video content analysis system in the surveillance domain [1], shown in Figure 1.

The Network In component receives the encoded frames from the network and places them in a buffer for the Decoder. The decoded frames are then used by the VCA component, which extracts 3D model metadata, and places the metadata in another buffer. This metadata is then used by the Alert component to perform a semantic analysis and identify the occurrence of certain events (e.g. a robbery). The Display component reads the decoded video stream, the metadata and the identified events, and shows them on a display.

When a robbery is detected the system may change into a different mode, e.g. in addition to decoding the video stream it may also transmit it over the network to a portable device. In a memory constrained system, such a mode change will require reallocating some memory from the buffers along the video processing chain to the buffer between the Network In and Network Out components, and reducing the modes of the video processing components, e.g. the Decoder. A mode change is orchestrated by the Quality Manager Component (QMC), which is aware of the tradeoff between the quality and resource requirements of individual components.

Another example of an application with frequent mode changes and tight latency bound requirements is scalable video decoding in a TV, where a mode change represents the reallocation of (memory) resources between competing video processing chains, e.g. when swapping the channels destined for the main screen and a Picture in Picture [2].

In this document we investigate the problem of improving the latency bound of system mode changes, in memory constrained real-time systems composed of scalable components. We focus on component modes describing the memory requirements of the components.

Contributions

- A method for reallocating the memory allocated to the active task, without corrupting the state of the active task's job.
- The combination of Fixed-Priority Scheduling with Deferred Preemption (FPDS) and the capacity to choose a subtasks path online, according to current system memory requirements.
- A modeling framework combining scalable components (Section III) with FPDS (Section IV) forming a basis for an efficient mode change protocol.

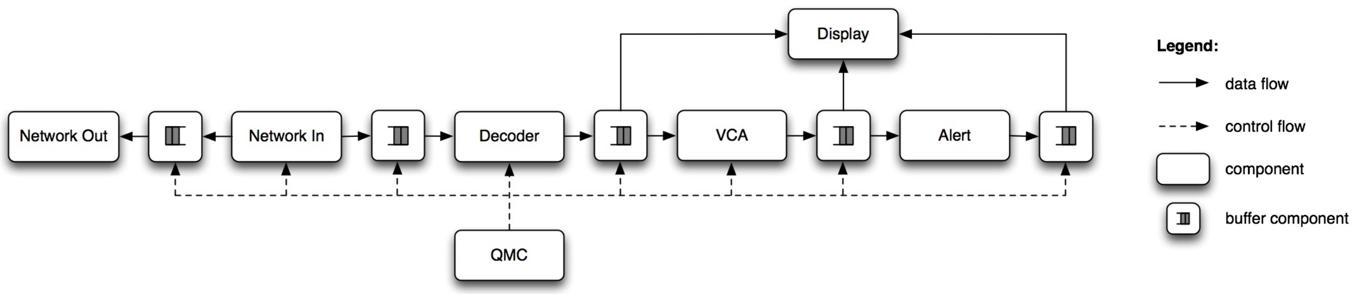

Figure 1. Quality control flow in a video content analysis system.

- A simple mode change protocol based on FPDS (Section V).
- A quantitate analysis of mode change latency under Fixed-Priority Preemptive Scheduling (FPPS) and FPDS, and an improvement on the existing latency bound for mode changes under FPPS (Section VI) supported by simulation results (Section VII).

II. Related work

Scalable multimedia applications

Multimedia processing systems are characterized by few but resource intensive tasks, communicating in a pipeline fashion. The tasks require processor time for processing the video frames (e.g. encoding, decoding, content analysis), and memory space for storing intermediate results between the pipeline stages. The pipeline nature of task dependencies poses restrictions on the lifetime of the intermediate results in the memory. The large gap between the worst-case and average-case resource requirements of the tasks is compensated by buffering and their ability to operate in one of several predefined modes, allowing to trade off their *processor* requirements for quality [3], or *network bandwidth* requirements for quality [4]. In this document we investigate trading *memory* requirements for quality. We assume there is no memory management unit available.

Modes and mode changes

[5], [6] define a *(system) mode* as the current task set, with the task model limited to *processor requirements* (period, deadline and computation time), and a *(system) mode change* as the addition or deletion of tasks from the task set.

In this document we define (system) mode in terms of the *task parameters* of all the components in the system. Unlike the existing literature where a mode expresses the requirements for processor [6], [3] or network bandwidth [4], we let a mode describe the *memory* requirements of the components.

[6] present a survey of mode change protocols for FPPS on a single processor, and propose several new protocols along with their corresponding schedulability analysis and configuration methods. They consider a standard fixed priority sporadic task model, with task τ_i specified by its priority i, minimum interarrival period T_i, worst-case execution time C_i, and deadline D_i, with $0 < C_i \leq D_i \leq T_i$.

They classify existing mode change protocols according to three dimensions:

1) Ability to abort old-mode tasks upon a mode change request:
    a) All old-mode tasks are immediately aborted.
    b) All old-mode tasks are allowed to complete normally.
    c) Some tasks are aborted.
2) Activation pattern of unchanged tasks during the transition:
    a) Protocols with periodicity, where unchanged tasks are executed independently from the mode change in progress.
    b) Protocols without periodicity, where the activation of unchanged periodic tasks may be delayed until the mode change has completed (limiting thus the interference with the tasks involved in the mode change).
3) Ability of the protocol to combine the execution of old- and new- mode tasks during the transition
    a) Synchronous protocols, where new-mode tasks are never released until all old-mode tasks have completed their last activation in the old mode.
    b) Asynchronous protocols, where a combination of both old- and new-mode tasks are allowed to be executed at the same time during the transition.

[7] present a synchronous mode change protocol without periodicity, where upon a mode change request the active old-mode tasks are allowed to complete, but are not released again (if their next periodic invocation falls within the mode change), classified in the three dimensions as (1b, 2b, 3a). Their algorithm bounds the mode change latency by

$$\mathcal{L} = \sum_{\tau_i \in \Gamma_{del_c}} C_i \tag{1}$$

where Γ_{del_c} is the set of tasks in the old mode which are to be deleted, but have to complete first, and $\mathcal{L}$ is the upper bound on the latency of a mode change, i.e. the time interval between a *mode change request* and the time when all the old-mode tasks have been deleted and all the new-mode tasks, with their new parameters, have been added and resumed.

The (1b, 2a, 3a) mode change protocol by [8] is applicable when tasks access shared resources according to any resource access policy. Upon a mode change request the protocol waits until an idle instant. When an idle instant is detected, the activation of old-mode tasks is suspended, the mode is changed (e.g. in case of priority ceiling protocol the ceilings of the shared resources are adjusted) and the new-mode tasks are released. This protocol is very simple, however it suffers from a large mode change latency, bounded by the worst-case response time of the lowest priority task.

In this document we present a (1b, 2b, 3a) mode change protocol. We assume that the old-mode tasks cannot be aborted at an arbitrary moment, but only at preliminary termination points. Each task is assumed to have a predefined set of preliminary termination points and upon an abortion request a task is allowed to execute until its next preliminary termination point.

Unlike [6], we allow a system mode change to be preempted by a new mode change request. Since at any time during a mode change the intermediate system mode is as valid as any other system mode, preemption of a mode change means simply setting a new target for the new mode change.

III. Modeling scalable components

We assume a platform without a memory management unit. We employ the concept of *memory reservations* to provide protection between memory allocations of different components. The remainder of this section describes our application model.

Component

We assume a set of n interconnected components $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$. A component $\mathcal{C}_i$ encapsulates a data structure with accompanying interface methods for modifying it and communicating with the system and other components. It can be regarded as a logical resource shared between different tasks. For example, a Decoder component encapsulates the encoded frame data which it reads from the input buffer and provides methods for decoding the frames and placing it in the output buffer.

Scalable components

We investigate a system running a single application composed of n *scalable components* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$, where each component $\mathcal{C}_i$ can operate in one of m_i predefined *component modes* in $\mathcal{M}_i = \{\mathcal{M}_{i,1}, \mathcal{M}_{i,2}, \ldots, \mathcal{M}_{i,m_i}\}$. Non-scalable components are considered as a degenerated case of a scalable component which can only operate in a single mode.

We also assume the existence of a *Quality Management Component* (QMC), which is responsible for managing the individual components and their modes to provide a system wide quality of service. The QMC may request a component to operate in a particular mode. Upon such a request the component will change its mode to the requested mode (within a bounded time interval).

Component modes

Each component mode $\mathcal{M}_{i,j} \in \mathcal{M}_i$ represents a tradeoff between the output quality of the component $\mathcal{C}_i$ and its resource requirements. In this document we focus on the memory resource. At any particular moment in time each component is assumed to be set to execute in one particular mode, referred to as the *target component mode*. Note, however, that a component may be executing in a different *current component mode* than the target mode it is set to, e.g. during the time interval between a request from the QMC to change its mode and the time the change is completed.

Memory reservations

A component expresses its memory requirements in terms of *memory reservations*, where each reservation specifies the size of a contiguous memory space to be used exclusively by the requesting component. A memory reservation $\mathcal{R}_{i,k}$ is specified by a tuple $(\mathcal{C}_i, s_k)$, where $\mathcal{C}_i$ is the component requesting the reservation, and s_k is the size of the *contiguous* memory space requested. A component may request several reservations, and in this way distribute its memory requirements among several reservations.

A component mode $\mathcal{M}_{i,j}$ is expressed as a set of reservations $\mathcal{M}_{i,j} \subseteq \mathcal{R}_i$, where $\mathcal{R}_i = \{\mathcal{R}_{i,1}, \mathcal{R}_{i,2}, \ldots\}$ is the set of all reservations ever requested by component $\mathcal{C}_i$.

Reservations are managed (i.e. allocated, monitored and enforced) by the *Resource Management Component* (RMC). Before starting to operate in a particular target mode, the component will request the corresponding memory reservations. Upon a request the RMC will check if the reservation can be granted, considering the total memory size and the current reservations of other components.

A component may also discard previously requested (and granted) reservations, if its memory requirements are reduced in the new mode, in order to make space for reservations of other components.

System modes

The product of all target (or current) component modes at a particular moment in time defines a *target* (or *current*) *system mode* $\mathcal{S}$. In a system comprised of n components a system mode is an n-tuple ranging over the component modes, i.e $\mathcal{S} \in \mathcal{M}_1 \times \mathcal{M}_2, \times \ldots \times \mathcal{M}_n$.

Mode changes

While the system is operating in mode $\mathcal{S}$ the QMC may request a *system mode change*. A system mode change is defined by the *current mode* and the *target mode*. During a system mode change the QMC may request any component to change its mode, in order to make sure that in the end the whole system will be operating in the target mode. In terms of memory reservations, some components will be requested to decrease their memory requirements and discard some of their memory reservations, in order to accommodate the increase in the memory provisions to other components.

A resource constrained real-time system has to guarantee that all required modes are feasible, as well as all required *mode changes*. The *mode change latency*, defined as the time interval between a *mode change request* and the time when the resources have been reallocated, should satisfy timing constraints expressed by upper bounds.

IV. MODELING FPDS

Application

An application $\mathcal{A}$ performs one or more control and monitoring functions, and is defined by a set of m tasks $\{\tau_1, \tau_2, \ldots, \tau_m\}$. We assume several applications in our system, e.g. in the surveillance example in Section I we can identify one application processing the incoming video frames, and one application transmitting the incoming packets over the network.

Task and job

A task τ_i describes the work needed for handling an event. Some of these tasks are executed in response to external events (e.g. arrival of a network packet) and others are executed in response to events in other tasks (e.g. completion of decoding of a video frame). Example of tasks are the decoding a frame or changing the system mode. A task can be regarded as the control flow through the components it uses to do the work, as shown in Figure 2.

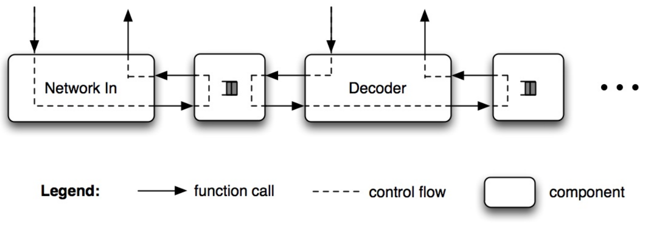

Figure 2. An example of a network task and video decoding tasks, represented by the paths traversing the components.

A component can be used by several tasks. For example, the left buffer in Figure 2 is used by both the network and the decoder task. More precisely, a component $\mathcal{C}_i$ can be accessed by set of tasks given by $\phi(\mathcal{C}_i)$, which is determined at compile time or whenever a task enters or leaves the system. We assume mutually exclusive access to components.

Each task τ_i is assigned fixed priority i. The priority assignment is arbitrary, i.e. not necessarily rate or deadline monotonic.

A *job* σ_i represents the execution of a task τ_i. In the remainder of this document we refer to jobs as tasks, when the meaning is not ambiguous.

Subtask and subjob

The task may be composed of several *subtasks*, dividing the task into smaller units of computation. A subtask $\tau_{i,j}$ of task τ_i is described by its worst-case execution time $C_{i,j}$. A subtask differs form a task as it does not have a period nor an explicit deadline.

A *subjob* $\sigma_{i,j}$ represents the execution of a subtask $\tau_{i,j}$. In the remainder of this document we refer to subjobs as subtasks, when the meaning is not ambiguous.

A subjob can lock a shared resource, e.g. via a critical section. We assume that a critical section does not span across subjob boundaries, i.e if a subjob locks a shared resource then it will also unlock it before it completes. Subjobs are preemptive in case of FPPS, and non-preemptive in case of FPDS.

Decision point and task graph

Decision points mark the places in the execution of a task, when the following subtask is selected.

The subtasks are arranged in a directed acyclic graph, referred to as the *task graph*, where the edges represent subtasks and the nodes represent decision points. An example is shown in Figure 3, showing the subtasks $\tau_{i,j}$ for task τ_i.

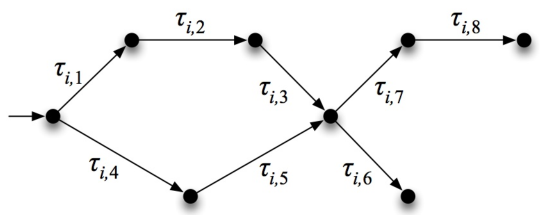

Figure 3. An example of a task graph with edges representing subtasks $\tau_{i,j}$ and nodes representing decision points.

During its execution the job follows a path in the task graph. When it arrives at a decision point the following subtask is selected. If more than one edge is leaving the corresponding node in the task graph, a single edge is selected (i.e. branching in the graph behaves like an if-then-else statement).

Traditionally the path is selected based on the *available data* and the internal *state* of the component. We extend the execution model and allow the path to be influenced *externally*, e.g. by the QMC controlling the modes of the components used by the task.

Note that the component mode may change while the task is residing in a decision point (e.g. if requested by the QMC). However, the mode is guaranteed to be set before the next subtask starts.

Preemption point

In case of FPDS, we limit preemption only to predefined preemption points, at the boundaries of subtasks. We assume a one-to-one mapping between the preemption points and the decision points in the task graph.

An actual preemption at a preemption point will occur only when there is a higher priority subjob ready. If a job was preempted at a preemption point, it postpones the decision of which subtask to follow (in case it is possible to follow several subtasks) until it resumes.

In case of FPPS, there are no explicit preemption points and a subjob may be preempted by a higher priority subjob.

Preliminary termination point

Preliminary termination points identify places during a job execution, where the processing can be terminated (producing a *usable* result, however of lesser quality), by skipping the remaining subjobs in the task graph. As a consequence, at these points the component mode can be changed to one with lowest resource requirements.

In case of FPDS we assume that the preliminary termination points coincide with the preemption points (and the decision points), meaning that at every preemption point the job processing can be terminated, and vice-versa. The preliminary termination points thus unite the scalable components with FPDS.

V. The swift mode change protocol

Upon a mode change request the system needs to know *when* the old mode tasks are ready to release their resources (i.e. when they have completed) before reallocating these resources to the new mode tasks, and it must deal with task interdependencies, in case they access shared resources. Our solution based on FPDS offers a simple implementation of mode changes, from the perspective of component design and system integration, while improving on the existing mode change latency bound.

FPDS guarantees that at any moment in time at most one subtask is executing. All other ready tasks are residing at their preemption points. By coinciding preemption points with preliminary termination points we avoid having to wait until the tasks currently accessing those components involved in the mode change have completed, and wait only until the currently running task reaches its next preemption point.

Upon a mode change request, tasks need to execute mode change routines responsible for adapting the resources requirements of the components they use. *We let the QMC task τ_{qmc} (released upon a mode change request and responsible for managing the mode change) execute the component mode change routines on behalf of the tasks and assign τ_{qmc} the highest priority.* After the currently running subtask completes, τ_{qmc} performs the mode changes for all components. Since τ_{qmc} has the highest priority other tasks will not be able to interfere with the mode change, thus reducing the mode change latency.

Our approach is especially suited for applications composed of scalable components which produce incremental results, e.g. a scalable MPEG decoder component, which incrementally processes enhancement layers to produce higher quality results. After each layer the decoding can be aborted, or continued to improve the result. We assume a scalable MPEG decoder [9], [10], with a layered input stream. Every frame in the stream has a *base layer*, and a number of additional *enhancement layers*, which can be decoded to increase the quality of the decoded video stream.

VI. Analysis

Every system mode change is defined by the current (or old) system mode $\mathcal{S}_c$ and the target (or new) system mode $\mathcal{S}_t$. The system may decide to change the mode of a component at an arbitrary moment in time during the execution of a subtask. When this happens, the component will first complete the current subtask, before changing its mode, even if the new mode discards the results produced by the subtask.

It may seem unreasonably expensive to first wait to complete the work and then throw away the result. However, since arbitrary termination is not supported (according to our assumption in Section IV), by coinciding the preliminary termination points with preemption points, this allows us to reduce the latency of mode changes.

The presented mode change protocol may delay the unchanged tasks until the mode change has completed. The interference on these tasks is bounded by the mode change latency and thus it can be taken into account in the schedulability analysis of all tasks.

Component mode change latency

Two parties are involved in a component mode change: the system for (re)allocating the resources, and the component itself, for adapting its control flow to the new mode. We assume that the system (i.e. the RMC) can allocate and deallocate resources in a bounded amount of time.

In order to provide a bounded system mode change latency, we therefore have to guarantee that a component responds to a mode change request within a bounded amount of time. The latency of changing the mode of a single component can be divided into three parts:

$$l(\mathcal{C}_i) = l_W(\mathcal{C}_i) + l_P(\mathcal{C}_i) + l_R(\mathcal{C}_i) \qquad (2)$$

where

- $l(\mathcal{C}_i)$ is the component mode change latency of component $\mathcal{C}_i$.
- $l_W(\mathcal{C}_i)$ is the time needed for $\mathcal{C}_i$ to become ready for the mode change. It is the time needed by the task accessing $\mathcal{C}_i$ to reach its next preliminary termination point, and therefore it is bounded by the duration of the longest subtask in any task accessing $\mathcal{C}_i$.

- $l_P(\mathcal{C}_i)$ is the time required by $\mathcal{C}_i$ to do any pre- and post-processing. We assume for every $\mathcal{C}_i$ there exists a worst case $C_{i,p}$.
- $l_R(\mathcal{C}_i)$ is the time needed to (re)allocate the reservations of $\mathcal{C}_i$ involved in the mode change. Since this overhead is relatively small compared to the other two terms, we will represent it with a constant C_{sys}, representing the worst-case system overhead for (de)allocating the affected reservations.

The following equation refines Equation (2)

$$l(\mathcal{C}_i) = \max_{\tau_j \in \phi(\mathcal{C}_i)} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) + C_{i,p} + C_{sys} \tag{3}$$

where τ_j is a task accessing component $\mathcal{C}_i$, $\tau_{j,k}$ is a subtask of task τ_j, and $C_{\tau_{j,k}}$ is the worst-case computation time of subtask $\tau_{j,k}$. We assume that the RMC overhead is accounted for in $C_{\tau_{j,k}}$, $C_{i,p}$ and C_{sys}.

System mode change latency

For the system mode change latency we have to consider all the components involved in the mode change. We therefore introduce

$$\mathcal{L}(\gamma) = \mathcal{L}_W(\gamma) + \sum_{\mathcal{C}_i \in \gamma} l_P(\mathcal{C}_i) + \sum_{\mathcal{C}_i \in \gamma} l_R(\mathcal{C}_i) \tag{4}$$

where γ is the set of components involved in the mode change, $\mathcal{L}(\gamma)$ is the latency of a system mode change involving components in γ, and $\mathcal{L}_W(\gamma)$ is the time interval between the mode change request and the time when all tasks accessing components in γ have reached their preliminary preemption point. $\mathcal{L}_W(\gamma)$ depends on the scheduling algorithm and is discussed in the following two sections.

Fixed-Priority Preemptive Scheduling

In a preemptive system, a subtask may be preempted by a higher priority task. This higher priority subtask may again be preempted by a subtask with an even higher priority. Therefore, due to the arbitrary preemption, several subtasks may be preempted and active at the same time. In the worst case we can have a chain of subtasks which are preempted just after they became active.

Since a subtask has to complete before allowing its components to change their mode, for FPPS the $\mathcal{L}_W(\mathcal{C}_i)$ term in Equation (4) is bounded by the *sum* of durations of *longest* subtasks of tasks accessing the components involved in the mode change:

$$\mathcal{L}_W(\gamma) = \sum_{\tau_j \in \Phi(\gamma)} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) \tag{5}$$

where $\Phi(\gamma) = \bigcup_{\mathcal{C}_i \in \gamma} \phi(\mathcal{C}_i)$ is the set of all tasks involved in the mode change (the set union gets rid of duplicates due to tasks accessing several components simultaneously).

Combining Equations (3), (4) and (5) we get

$$\mathcal{L}^{FPPS}(\gamma) = \sum_{\tau_j \in \Phi(\gamma)} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) + \sum_{\mathcal{C}_i \in \gamma} (C_{i,p} + C_{sys}) \tag{6}$$

Note that we implied here a synchronous mode change protocol without periodicity, where the current subtasks of tasks involved in the mode change are forced to complete first (e.g. by raising their priority) possibly interfering with the tasks not involved in the mode change.

Reallocating memory between the components involved in the mode change may not always be possible, e.g. due to memory fragmentation. Consequently, the memory allocated to other components may need to be reallocated as well. We guarantee system integrity by raising the priority of the tasks using these components. We can take the reallocation overhead into account by including these indirectly affected components in γ. Note that the set of the affected components depends on the memory allocation algorithm, and in the worst case it is equal to Γ.

Fixed-Priority Scheduling with Deferred Preemption

When using FPDS with non-preemptive subtasks, a task may be preempted only at subtask boundaries, i.e. at the preemption points. This implies that at most one subtask may be active at a time (the currently running one); all other tasks will be waiting at one of their preemption points. Therefore, for FPDS the $\mathcal{L}_W(\mathcal{C}_i)$ term in Equation (4) is bounded by the *duration of longest subtask* among *all* tasks in the system:

$$\mathcal{L}_W(\gamma) = \max_{\tau_j \in \Gamma} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) \tag{7}$$

where Γ is the complete task set.

Combining Equations (3), (4) and (7) we get

$$\mathcal{L}^{FPDS}(\gamma) = \max_{\tau_j \in \Gamma} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) + \sum_{\mathcal{C}_i \in \gamma} (C_{i,p} + C_{sys}) \tag{8}$$

Improving the bound for synchronous mode change protocols, without periodicity

As indicated in Section II, the currently best known bound on the mode change latency in a synchronous mode change protocol without periodicity, due to [7], is equal to

$$\mathcal{L} = \sum_{\tau_i \in \Gamma_{del_c}} C_i \tag{9}$$

Their algorithm behaves similarly to the one we described for the FPPS case, in Section VI. We can apply our results for FPDS to the processor mode change domain and reduce this mode change latency bound.

[6] assume the task set can be divided into to subsets: those than can be aborted upon a mode change request (Γ_a), and those which need to run till completion (Γ_c). It is the tasks that need to complete, which contribute to the mode change latency, i.e. $\Gamma_{del_c} \subseteq \Gamma_c$.

If we make the tasks $\tau_i \in \Gamma_c$ non-preemptive, then at most one task will be running at a time, and the mode change latency will be reduced to

$$\mathcal{L} = \max_{\tau_i \in \Gamma_{del_c}} C_i \tag{10}$$

Of course, the lower bound comes at a price of a tradeoff between the latency bound and the schedulability of the task set, where the non-preemptive tasks may increase the blocking time of higher priority tasks.

On the one hand, it is likely that a task will be required to complete after a mode change request, in order to avoid the corruption of shared resources. Therefore the increase in blocking time may not be significant. On the other hand, there are more efficient resource access protocols, with lower bounds on the maximum blocking time, rather than Fixed-Priority Non-preemptive Scheduling (FPNS).

Note that FPNS can be considered as the most pessimistic configuration of FPDS: if tasks can be subdivided into shorter subtasks with preemption points coinciding with preliminary termination points, then the mode change latency can only be reduced.

VII. Simulation results

The setup was similar to the example in Section I, with a single application consisting of three periodic tasks $\tau_{network}$, $\tau_{decoder}$ and $\tau_{renderer}$, all sharing the same period T. There were three processing components $\mathcal{C}_{network}$, $\mathcal{C}_{decoder}$ and $\mathcal{C}_{renderer}$, communicating via two buffer components: $\mathcal{C}_{encoded}$ between $\mathcal{C}_{network}$ and $\mathcal{C}_{decoder}$ storing the encoded frames, and $\mathcal{C}_{decoded}$ between $\mathcal{C}_{decoder}$ and $\mathcal{C}_{renderer}$ storing the decoded frames. The simulation setup is shown in Figure 4, with $\phi(\mathcal{C}_{network}) = \{\tau_{network}\}$, $\phi(\mathcal{C}_{decoder}) = \{\tau_{decoder}\}$, $\phi(\mathcal{C}_{renderer}) = \{\tau_{renderer}\}$, $\phi(\mathcal{C}_{encoded}) = \{\tau_{network}, \tau_{decoder}\}$ and $\phi(\mathcal{C}_{decoded}) = \{\tau_{decoder}, \tau_{renderer}\}$.

A mode change request was generated periodically with period $T_{qmc} = 1.1T$. Running the simulation for $20T$ ensured that mode changes were requested at different points during the execution of tasks. Every mode change involved all 5 components.

The setup was embedded in the realtime operating system μC/OS-II [11], which was extended with support for periodic tasks, FPNS, FPDS, and the architecture described in [12] (the QMC and the corresponding component interfaces). The simulations were run in Ubuntu 8.10, running in the VMWare Fusion virtual machine on an Apple MacBook Pro, 1.83 GHz Intel Core Duo.

Following the observation that FPNS is the most pessimistic configuration for FPDS, we had each task consist of a single subtask, and compared the mode change latency under FPPS vs. FPDS by scheduling the subtasks preemptively vs. non-preemptively.

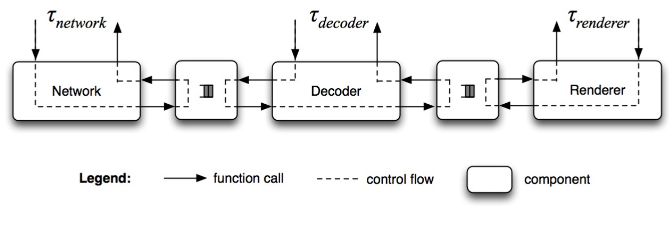

Figure 4. Simulation setup

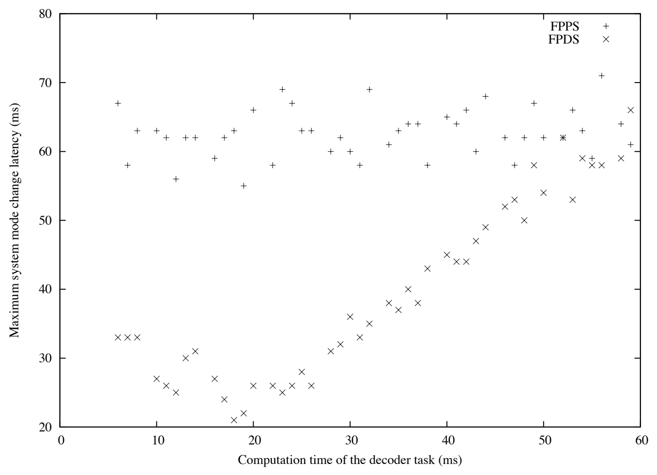

Figure 5. Computation time $C_{\tau_{decoder}}$ (ms) vs. maximum system mode change latency (ms). $T = 100$, $C_{\tau_{network}} = C_{\tau_{renderer}} = 60 - C_{\tau_{decoder}}/2$

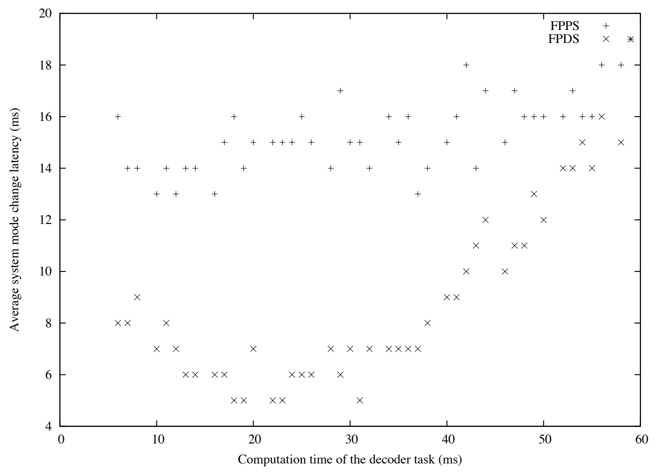

Figure 6. Computation time $C_{\tau_{decoder}}$ (ms) vs. average system mode change latency (ms). $T = 100$, $C_{\tau_{network}} = C_{\tau_{renderer}} = 60 - C_{\tau_{decoder}}/2$

In Figures 5 and 6 the period T was set to 100ms and the total utilization of all three tasks was fixed at 0.6. We varied $C_{\tau_{decoder}}$ and distributed the remaining computation time equally between $C_{\tau_{network}}$ and $C_{\tau_{renderer}}$. The "V" shape of the FPDS results is due to the mode change latency being dominated by $\tau_{network}$ and $\tau_{renderer}$ for lower values of $C_{\tau_{decoder}}$.

In Figures 7 and 8 the utilizations of tasks were fixed at $U_{\tau_{network}} = 0.12, U_{\tau_{decoder}} = 0.3$ and $U_{\tau_{renderer}} = 0.18$. We varied the period T, thus proportionally increasing the computation times of tasks.

The results in Figures 5, 6, 7 and 8 show an improvement of FPDS over FPPS in both worst case and average case system mode change latency.

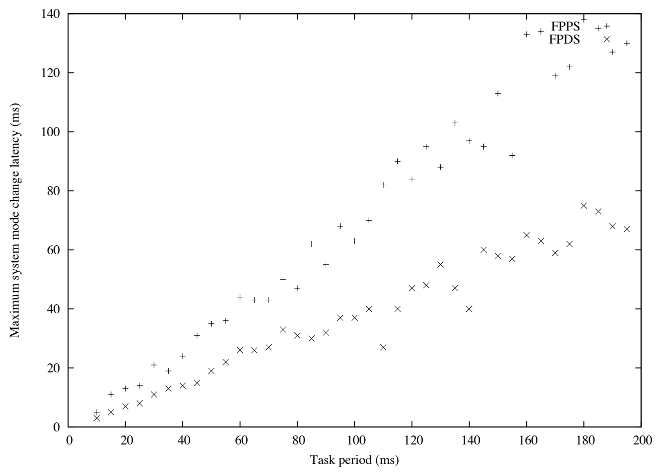

Figure 7. Period T (ms) vs. maximum system mode change latency (ms). Utilization of tasks was $U_{\tau_{network}} = 0.12, U_{\tau_{decoder}} = 0.3, U_{\tau_{renderer}} = 0.18$

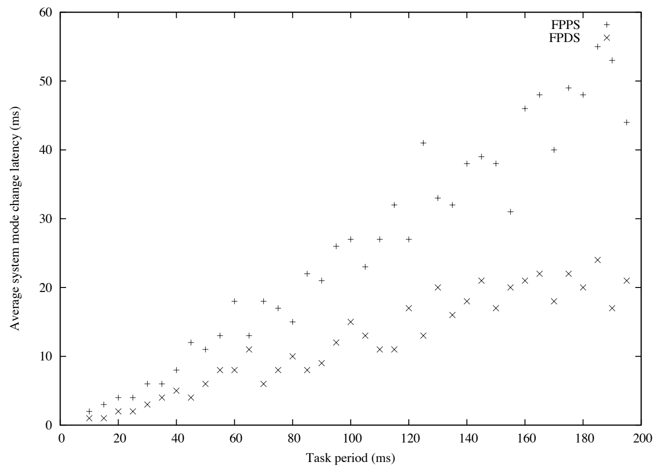

Figure 8. Period T (ms) vs. average system mode change latency (ms). Utilization of tasks was $U_{\tau_{network}} = 0.12, U_{\tau_{decoder}} = 0.3, U_{\tau_{renderer}} = 0.18$

VIII. Conclusions

In this document we have investigated mode changes and the mode change latency bound in real-time systems comprised of scalable components. We compared two approaches, based on FPPS and FPDS. We showed that combining scalable components with FPDS (exploiting its non-preemptive property and coinciding preliminary termination points with preemption points), guarantees a shorter worst case latency than FPPS, and applied these results to improve the existing latency bound for mode changes in the processor domain.

The presented mode change protocol is simple and avoids the complication and overheads associated with task signaling during a mode change, necessary in existing mode change protocols.

IX. Future work

[13] show how to exploit the unused processor capacity during runtime. It remains to be investigated how the notion of spare processor capacity, such as gain and slack time, can be translated to the memory domain, and how these "gain space" and "slack space" can be exploited during runtime, e.g. using FPDS and the fact that when a task is preempted at a preemption point (with the assumption that shared resources are not locked across preemption points), parts of its memory space could be used by other components.

So far we considered only the memory resource. It is to be investigated whether the techniques described in this document can be adapted to systems where tasks explicitly express their requirements for *simultaneous* access to *several resources*.

References

[1] CANTATA, "Content Aware Networked systems Towards Advanced and Tailored Assistance," 2006. [Online]. Available: http://www.hitech-projects.com/euprojects/cantata

[2] R. J. Bril, E. F. M. Steffens, and G. S. C. van Loo, "Dynamic behavior of consumer multimedia terminals: System aspects," in *Proc. IEEE International Conference on Multimedia and Expo*. IEEE Computer Society, 2001, pp. 597–600.

[3] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "Qos control strategies for high-quality video processing," *Real-Time Systems*, vol. 30, no. 1-2, pp. 7–29, 2005.

[4] D. Jarnikov, P. van der Stok, and C. Wust, "Predictive control of video quality under fluctuating bandwidth conditions," in *Proceedings of the International Conference on Multimedia and Expo*, vol. 2, June 2004, pp. 1051–1054.

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

[6] 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.

[7] J. Real, "Protocols de cambio de mondo para sistemas de tiempo real (mode change protocols for real time systems)," Ph.D. dissertation, Technical University of Valencia, 2000.

[8] K. Tindell and A. Alonso, "A very simple protocol for mode changes in priority preemptive systems." Universidad Politecnica de Madrid, Tech. Rep., 1996.

[9] B. G. Haskell, A. Puri, and A. N. Netravali, *Digital Video: An introduction to MPEG-2*. Chapman & Hall, Ltd., 1996.

[10] D. Jarnikov, "Qos framework for video streaming in home networks," Ph.D. dissertation, Eindhoven University of Technology, 2007.

[11] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 1998.

[12] M. Holenderski, R. J. Bril, and J. J. Lukkien, "Swift mode changes in memory constrained real-time systems," Eindhoven University of Technology, Tech. Rep. CS-09-08, 2009. [Online]. Available: http://w3.win.tue.nl/en/research/research_computer_science/

[13] N. C. Audsley, R. I. Davis, and A. Burns, "Mechanisms for enhancing the flexibility and utility of hard real-time systems," in *Proceedings of the Real-Time Systems Symposium*, 1994, pp. 12–21.