Swift mode changes in memory constrained real-time systems

Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands
{m.holenderski, r.j.bril, j.j.lukkien}@tue.nl*

July 16, 2009

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.

Next to improving the latency bound of mode changes, the presented architecture offers a simple protocol and avoids the complication and overheads associated with tasks signaling during a mode change, necessary in existing mode change protocols. The architecture is especially suited for pipelined applications, allowing to perform the mode change without the need to first clear the whole pipeline.

1 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 [CANTATA, 2006], 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 video content analysis VCA component, which

*This work has been supported in part by the Information Technology for European Advancement (ITEA2), via the CANTATA project.

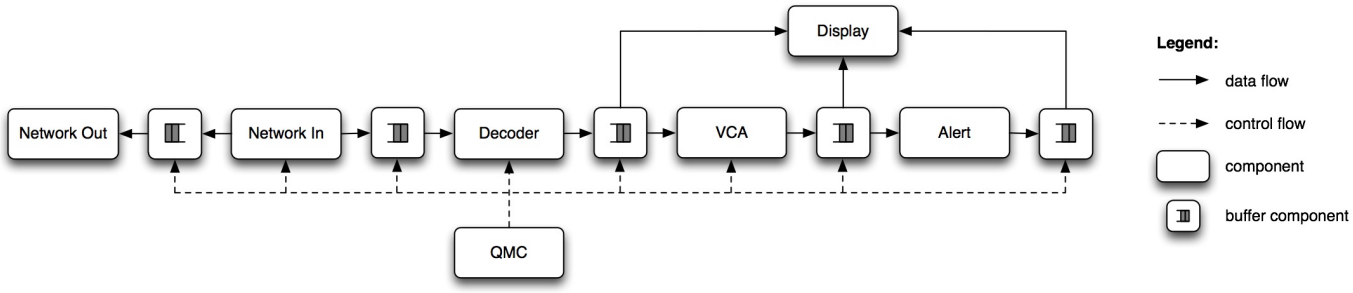

Figure 1: Quality control flow in a video content analysis system.

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 [Bril et al., 2001].

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 3) with FPDS (Section 4) forming a basis for an efficient mode change protocol.
- A simple mode change protocol based on FPDS (Section 5) , and the accompanying architecture description (Section 6).
- 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 7) supported by simulation results (Section 8).

2 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 [Wüst et al., 2005], or *network bandwidth* requirements for quality [Jarnikov et al., 2004]. In this document we investigate trading *memory* requirements for quality. We assume there is no memory management unit available.

Modes and mode changes

[Liu, 2000, Real and Crespo, 2004] 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 [Real and Crespo, 2004, Wüst et al., 2005] or network bandwidth [Jarnikov et al., 2004], we let a mode describe the *memory* requirements of the components.

[Real and Crespo, 2004] 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.

[Real, 2000] 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.

[Sha et al., 1988] consider periodic task executing according to the Rate Monotonic schedule and sharing resources according to the priority ceiling protocol. Their (1b, 2a, 3b) mode change protocol

provides a complicated set of rules for determining when the priority ceilings of shared resources can be changed without conflicting with the delayed release of the new-mode tasks.

The (1b, 2a, 3a) mode change protocol by [Tindell and Alonso, 1996] 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 [Real and Crespo, 2004] , who assume that a mode change request may not occur during an ongoing mode change, 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. Our protocol is as simple as [Tindell and Alonso, 1996], yet improves the mode change latency bound of [Real, 2000].

Resource reservations

Processor reservations

Resource reservations have been introduced by [Mercer et al., 1994], to guarantee resource provisions in a system with dynamically changing resource requirements. They focused on the processor and specified the reservation budget by a tuple (C, T), with capacity C and period T. The semantics is as follows: a reservation will be allocated C units of processor time every T units of time. When a reservation uses up all of its C processor time within a period T it is said to be *depleted*. Otherwise it is said to be *undepleted*. At the end of the period T the reservation capacity is *replenished*.

They identify four ingredients for guaranteeing resource provisions:

Admission When a reservation is requested, the system has to check if granting the reservation will not affect any timing constraints.

Scheduling The reservations have to be scheduled on the global level, and tasks have to be scheduled within the reservations.

Accounting Processor usage of tasks has to be monitored and accounted to their assigned reservations.

Enforcement A reservation, once granted, has to be enforced by preventing other components from "stealing" the granted budget.

[Rajkumar et al., 1998] aim at a uniform resource reservation model and extended the concept of processor reserves to other resources, in particular the disk bandwidth. They schedule processor reservations according to FPPS and EDF, and disk bandwidth reservations according to EDF. They extend the reservation model to (C, T, D, S, L), with capacity C, period T, deadline D, and the starting time S and the life-time L of resource reservation, meaning that the reservation guarantees of C, T and D start at S and terminate at $S + L$.

Note that [Rajkumar et al., 1998] apply their uniform resource reservation model only to *temporal resource*, such as processor or disk-bandwidth. They do not show how their methods can be applied to *spatial resources*, such as memory.

Memory reservations

[Nakajima, 1998] apply memory reservations in the context of continuous media processing applications. They aim at reducing the number of memory pages wired in the physical memory at a time by wiring only those pages which are used by threads currently processing the media data in the physical memory.

Unlike the traditional approaches which avoid page faults by wiring all code, data and stack pages, they introduce an interface to be used by the realtime applications to request reservations for memory pages. During runtime, upon a page fault, the system will load the missing page only if it does not exceed the applications reservation. Thus the number of wired pages is limited.

[Eswaran and Rajkumar, 2005] implement memory reservations in Linux to limit the time penalty of page faults within the reservation, by isolating the memory management of different applications from each other. They distinguish between *hard* and *firm* reservations, which specify the *maximum* and *minimum* number of pages used by the application, respectively. Their reservation model is hierarchical, allowing child reservations to request space from a parent reservation. Their energy-aware extension of memory reservations allows to maximize the power savings, while minimizing the performance penalty, in systems where different hardware components can operate in different power levels and corresponding performance levels. They also provide an algorithm which calculates the optimal reservation sizes for the task set such that the sum of the task execution times is minimized.

In this paper we use (hard) memory reservations as containers for memory allocations, allowing components to request memory and guarantee granted requests during runtime, avoiding static memory allocation in absence of paging support. Memory reservations are granted to components only if there is enough space in the common memory pool, and memory allocations are granted only if there is enough space within the corresponding reservation.

3 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.

4 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 1 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.

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.

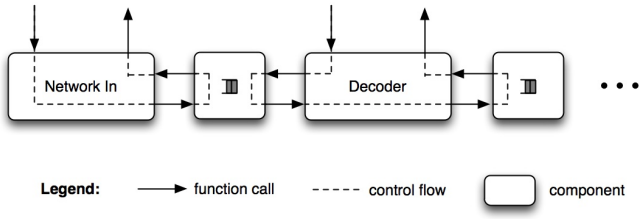

Figure 2: An example of a network task and video decoding tasks, represented by the paths traversing the components.

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.

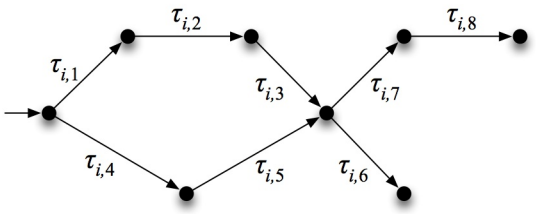

Figure 3: An example of a task graph with edges representing subtasks $\tau_{i,j}$ and nodes representing decision points.

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.

5 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 [Haskell et al., 1996, Jarnikov, 2007], 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.

6 Architecture description

This section describes the proposed architecture, introducing the necessary components and their interactions. In this section we consider only FPDS.

RMC

The RMC provides an interface for requesting and discarding memory reservations. Upon a request for a reservation it does a feasibility check. It is also responsible for guaranteeing that once a reservation is

granted to a component, memory within the reservation will not be allocated to other components.

The RMC is a framework component. It does not have an associated task, but rather offers interfaces to the components to manage resources, similar to system calls in a traditional operating system.

QMC

The QMC task τ_{qmc}, released upon a mode change request, uses the QMC for managing the modes of other components. During runtime when components change their modes, their reservations are reallocated, possibly leading to memory fragmentation. The fragmentation may lead to infeasible reservation allocations, even if there is enough total memory space available. In order to alleviate this problem, the QMC also decides *where* to allocate the reservations.

Each component $\mathcal{C}_i$ has a *component mode table*, which specifies all possible modes $\mathcal{M}_{i,j}$ for component $\mathcal{C}_i$. Each entry in the table is a *set of* tuples (o_k, s_k, r_k) representing the set of reservations in $\mathcal{M}_{i,j}$, where

- o_k is the memory offset for the reservation, i.e. the starting address of the reservation in the memory. The memory offset allows the QMC to guide the RMC in allocating reservations in the memory, in order to make sure that the reservations needed for the target mode can indeed be allocated. This approach is similar to the static memory segmentation taken by [Geelen, 2005].
- s_k is the size of the reservations.
- r_k is a reference to the corresponding reservation. This reference is used by the QMC to discard reservations on behalf of the components.

The QMC is aware of the possible component modes. We are not concerned with how the QMC selects system modes, and for sake of simplicity we therefore simply assume that the set of components is fixed and that the mode selection is table driven, based on a table pre-computed offline. Our mode change approach, however, is applicable also to open systems, where components may enter and leave the system during runtime.

Each system mode in the *system mode table* contains a list of component modes. In Section 3 a component mode specified a list of reservations, where each reservation was a pair $(\mathcal{C}_i, s_i)$. In order to guarantee feasibility of the requested reservations (in spite of possible memory fragmentation) the QMC has to guide the RMC when allocating the reservations. An alternative would be a negotiation between the QMC and the RMC, where the QMC would keep discarding and requesting reservations until the RMC would grant them, however, this would come at a cost of additional performance overhead and loss of predictability of the mode change latency.

Let δ_i be the time interval during the ith system mode change when the QMC is performing the mode changes on behalf of other components. We assume the following invariant holds: at any time during runtime (except during an interval δ_i) every component is in one of the modes specified in its component mode table, and the system is in one of the modes specified in the system mode table.

The QMC determines the necessary component mode changes by computing the "difference" between the target system mode and the current system mode in the system mode table.

The following sections describe the QMC interface methods and how they are used, also shown in Figures 4 and 5. All these methods return a status code, which indicates whether the method call was successful allowing the QMC to adapt the mode change in case a component fails to comply with a request.

Interface between QMC and components

During the initialization, each component registers its component mode table at the QMC. During runtime the QMC uses this mode table to communicate any mode changes back to the component.

Moreover, each component implements `QMC_BeforeModeChange()` and `QMC_AfterModeChange()` methods to be called by the QMC upon a mode change. The implementation of these two methods may change during task execution, e.g. depending on the current component mode.

The `QMC_BeforeModeChange()` method is called before the QMC reallocates memory reservations between the components involved in the mode change, offering the component to gracefully scale down

its memory requirements (e.g. scale down the resolution of frames in a buffer). It takes one argument: a reference to the entry in the component's mode table corresponding to the new mode.

The `QMC_AfterModeChange()` method is called after the QMC has discarded or requested new reservations on behalf of the component and is intended to update any memory references that have changed during the mode change (e.g. due to reservation reallocation). It takes one argument: a reference to the entry in the component's mode table corresponding to the old mode.

Interface between QMC and RMC

The QMC requests and discards reservations via the RMC, using the interface described in [Holenderski, 2008]. In order to help the RMC with allocation of reservations, the QMC communicates the offset for the reservation in the memory.

Initialization

1. The components register at the QMC, by calling `QMC_RegisterComponent()`. This method binds the components to the placeholders in the system mode table, stored by the QMC. Before this point, the entries in the table, specifying the mapping of reservations to components, contain place holders for the components.

2. The QMC sets all components to their initial mode, by following the same protocol as used for mode changes (described in the following section in steps 3-6).

3. The components are started.

A mode change

A mode change is defined by the current mode and the target mode, and follows the following protocol:

1. Upon a system mode change request (e.g. user input) the QMC task τ_{qmc} is inserted into the ready queue.

2. The currently running subtask (if any) is allowed to complete.

3. When τ_{qmc} is activated (at the highest priority), the fact that we are using FPDS guarantees that all other tasks are remaining at a preemption point, allowing the QMC to change the component modes on behalf of the tasks. The QMC therefore compares the current system mode and the target system mode and identifies those component modes which need to change, resulting in a set of reservations which need to be discarded or requested.

4. The QMC then calls the `QMC_BeforeModeChange()` in the identified components, which allow the components to do any processing before the mode is changed, e.g. it allows to reduce quality/resolution and thus the size of the frames stored in a buffer before shrinking the buffer and reallocating its memory.

5. The QMC then proceeds to discard and request these reservations, by calling the `RMC_DiscardMemoryBudget()` and `RMC_RequestMemoryBudget()`. The QMC may also move an existing reservation to a new location, by calling `RMC_MoveMemoryBudget()`. Note that the RMC may return with an error from each of these method calls, if the call would result in overlapping reservations (the QMC has enough information though to avoid these errors).

6. The QMC calls the `QMC_AfterModeChange()` in each affected component to allow it to do some post processing after the mode change, e.g to increase the resolution of decoded frames or to update any memory references. In the latter case the component uses the reservation references r_k in the entry in its component mode table corresponding to its new mode, to update the memory pointers to existing memory allocations (in case the reservation has been reallocated), by calling `RMC_GetPointerForAllocation()` in the RMC.

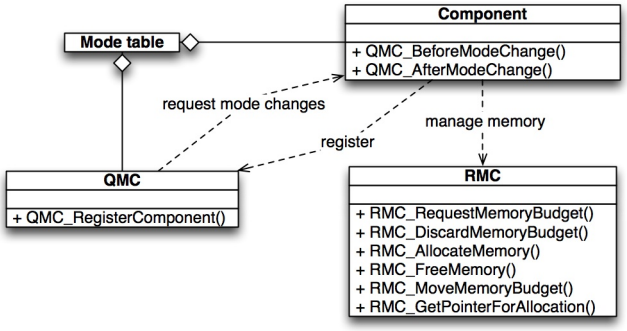

Figure 4: UML class diagram for the QMC, the RMC and a regular component.

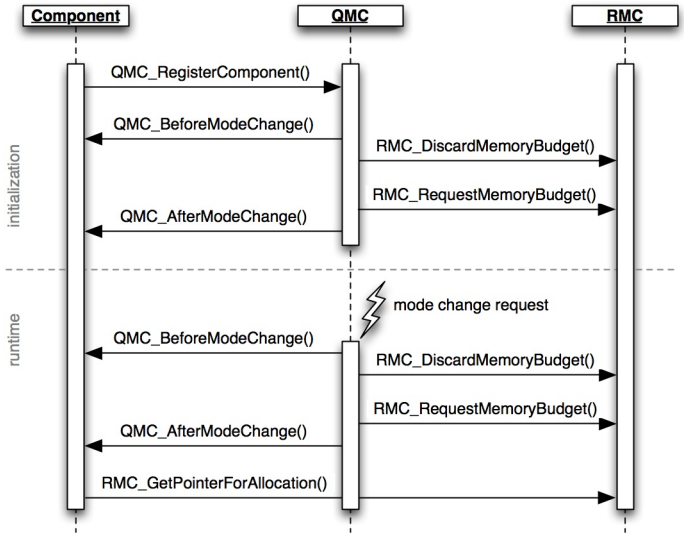

Figure 5: UML message sequence showing the interaction between a regular component, the QMC and the RMC.

7 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 4), 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:

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

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

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

Intermezzo Until now we have assumed that under FPDS subtasks are non-preemptive. We can relax this assumption and distinguish between two classes of tasks: *application tasks* and *framework tasks*. An application task is non-preemptive with respect to other application tasks, but it can be interrupted by a framework task *as long as* they do not share common resources. From the perspective of the interrupted task the framework tasks behave similarly to interrupt service routines, resuming the preempted task upon completion[1]. If we implement the QMC task τ_{qmc} as a framework task we can reduce the mode change latency by considering only those tasks which are involved in the mode change, rather than considering all tasks in the system, since a subtask not sharing any resources with tasks involved in the mode change can be preempted by a framework task. The $\mathcal{L}_W(\mathcal{C}_i)$ term in Equation (4) therefore becomes

$$\mathcal{L}_W(\gamma) = \max_{\tau_j \in \Phi(\gamma)} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) \tag{9}$$

Combining Equations (3), (4) and (9) we get

$$\mathcal{L}^{FPDS}(\gamma) = \max_{\tau_j \in \Phi(\gamma)} (\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}) + \sum_{\mathcal{C}_i \in \gamma} (C_{i,p} + C_{sys}) \tag{10}$$

Improving the bound for synchronous mode change protocols, without periodicity

As indicated in Section 2, the currently best known bound on the mode change latency in a synchronous mode change protocol without periodicity, due to [Real, 2000], is equal to

$$\mathcal{L} = \sum_{\tau_i \in \Gamma_{del_c}} C_i \tag{11}$$

Their algorithm behaves similarly to the one we described for the FPPS case, in Section 7. We can apply our results for FPDS to the processor mode change domain and reduce this mode change latency bound.

[Real and Crespo, 2004] 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. $\Gamma_{del_c} \subseteq \Gamma_c$.

[1] Further elaboration falls outside the scope of this report.

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

$$\mathcal{L} = \max_{\tau_i \in \Gamma_{del_c}} C_i \tag{12}$$

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.

8 Simulation results

The setup was similar to the example in Section 1, 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 6, 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 [Labrosse, 1998], which was extended with support for periodic tasks, FPNS, FPDS, and the architecture described in this paper (the RMC, 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.

In Figures 7 and 8 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 9 and 10 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 7, 8, 9 and 10 show an improvement of FPDS over FPPS in both worst case and average case system mode change latency.

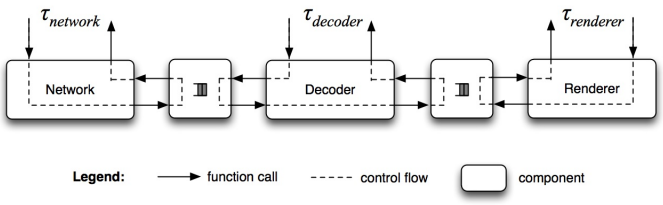

Figure 6: Simulation setup

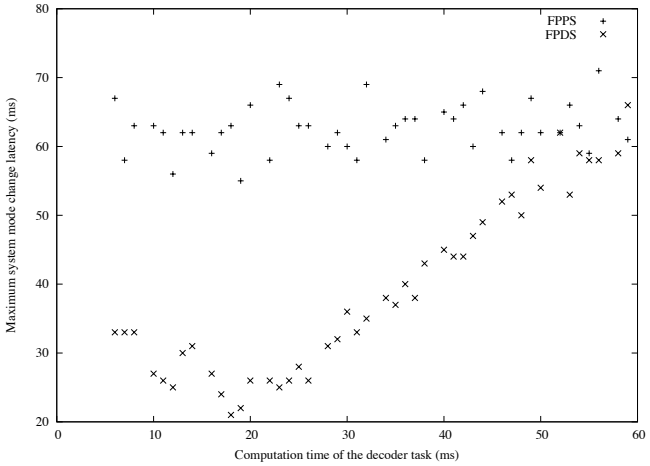

Figure 7: Computation time $C_{\tau_{decoder}}$ (ms) vs. maximum system mode change latency (ms). $T = 100$, $C_{\tau_{network}} = C_{\tau_{renderer}} = 60 - C_{\tau_{decoder}}/2$

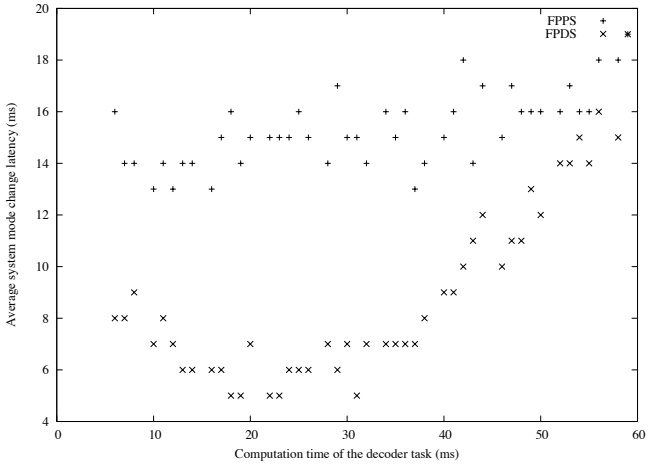

Figure 8: Computation time $C_{\tau_{decoder}}$ (ms) vs. average system mode change latency (ms). $T = 100$, $C_{\tau_{network}} = C_{\tau_{renderer}} = 60 - C_{\tau_{decoder}}/2$

9 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.

By adopting the presented architecture, describing the interaction between the QMC, RMC and the components, a mode change in a pipelined application can be performed almost instantaneously, without the need to clear the whole pipeline, further reducing the mode change latency.

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.

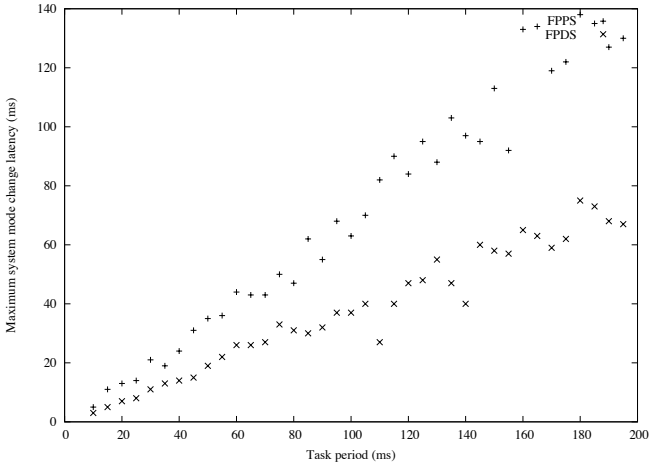

Figure 9: Period T (ms) vs. maximum system mode change latency (ms). Utilization of tasks was $U_{\tau_{network}} = 0.12, U_{\tau_{decoder}} = 0.3, U_{\tau_{renderer}} = 0.18$

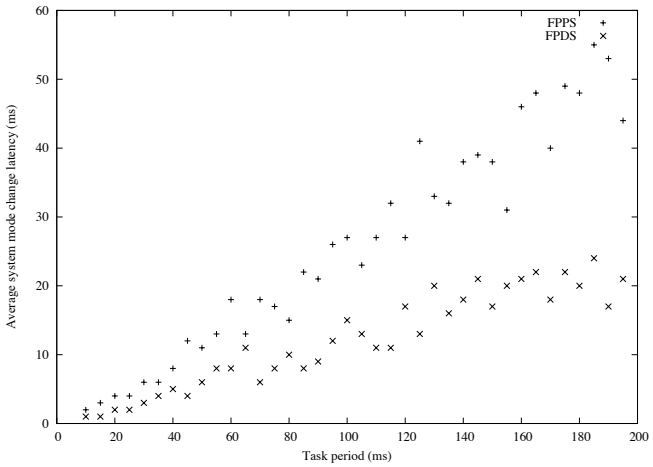

Figure 10: Period T (ms) vs. average system mode change latency (ms). Utilization of tasks was $U_{\tau_{network}} = 0.12, U_{\tau_{decoder}} = 0.3, U_{\tau_{renderer}} = 0.18$

10 Future work

[Audsley et al., 1994] 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*.

Acknowledgments

We would like to thank Pieter Cuijpers and Ugur Keskin for their feedback on earlier versions of this document.

References

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

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

[CANTATA, 2006] CANTATA. *Content Aware Networked systems Towards Advanced and Tailored Assistance.* 2006. URL `http://www.hitech-projects.com/euprojects/cantata`.

[Eswaran and Rajkumar, 2005] A. Eswaran, R. R. Rajkumar. *Energy-aware memory firewalling for qos-sensitive applications.* In *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 11–20. IEEE Computer Society, Washington, DC, USA, 2005.

[Geelen, 2005] T. Geelen. *Dynamic loading in a real-time system: An overlaying technique using virtual memory.* Tech. rep., Eindhoven University of Technology, 2005.

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

[Holenderski, 2008] M. Holenderski. *Resource management component: Design and implementation.* 2008. URL `http://www.win.tue.nl/san/projects/cantata/`.

[Jarnikov, 2007] D. Jarnikov. *QoS Framework for Video Streaming in Home Networks.* Ph.D. thesis, Eindhoven University of Technology, 2007.

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

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

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

[Mercer et al., 1994] C. Mercer, R. Rajkumar, J. Zelenka. *Temporal protection in real-time operating systems.* In *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, pp. 79–83. 1994.

[Nakajima, 1998] T. Nakajima. *Resource reservation for adaptive QoS mapping in real-time mach.* Parallel and Distributed Processing, pp. 1047–1056, 1998.

[Rajkumar et al., 1998] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa. *Resource kernels: A resource-centric approach to real-time and multimedia systems.* In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking.* 1998.

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

[Real and Crespo, 2004] J. Real, A. Crespo. *Mode change protocols for real-time systems: A survey and a new proposal.* Real-Time Systems, vol. 26(2):pp. 161–197, 2004.

[Sha et al., 1988] L. Sha, R. Rajkumar, J. Lehoczky, K. Ramamritham. *Mode change protocols for priority-driven preemptive scheduling.* Tech. Rep. CMU/SEI-88-TR-34, Carnegie Mellon University, 1988.

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

[Wüst et al., 2005] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, C. Hentschel. *Qos control strategies for high-quality video processing.* Real-Time Systems, vol. 30(1-2):pp. 7–29, 2005.