Swift mode changes in memory constrained real-time systems

Mike Holenderski

Third draft

Abstract

Method for “preempting memory” is presented, where (parts of) the memory allocated to an active job of a task may be reallocated to a job of another task, without corrupting the active job’s state.

Real-time systems composed of scalable components are investigated. A scalable component can operate in one of several modes. Each of these modes defines a certain trade off between the resource requirements of the components and their output quality. The focus is on memory constrained systems, with modes limited to memory requirements. The latency of a mode change, defined as the time needed to re-allocate memory between components, should satisfy timing constraints.

A modeling framework for component and system modes, and mode changes is presented. A quantitative analysis for Fixed Priority Preemptive Scheduling (FPPS) and Fixed Priority Scheduling with Deferred Preemption (FPDS) is provided, showing that FPDS sets a lower bound on the mode change latency. The results for both FPPS and FPDS are applied to improve the existing latency bound for mode changes in the processor domain.

Furthermore, interfaces are described for specifying component modes and controlling mode changes during runtime.
10 Quality Manager Component

11 Protocol between components and the QMC
  11.1 Three phases to changing a system mode
  11.2 Registration of the mode graph at the QMC
  11.3 Request a mode change
  11.4 Confirm a mode change

12 Analysis
  12.1 Component mode change latency
  12.2 System mode change latency
    12.2.1 Fixed Priority Preemptive Scheduling
    12.2.2 Fixed Priority Scheduling with Deferred Preemption
  12.3 Improving the bound for synchronous mode change protocols,
      without periodicity

13 Prototype

14 Conclusions

15 Future work
  15.1 Aborting a component at an arbitrary moment
  15.2 Relaxed mapping between subjobs and component modes
  15.3 Extension to arbitrary levels of component modes
  15.4 Speeding up system mode transitions
  15.5 Apply FPDS to reduce the latency of a processor mode change
  15.6 Pruning infeasible mode transitions
  15.7 Interface between the QMC and the RMC
  15.8 Extrapolation to processor mode changes
  15.9 Multi processors
  15.10 Extension of component modes to multiple resources
  15.11 Exploiting spare memory capacity
  15.12 Efficient representation of the system mode transition graph
1 Introduction

In this document we investigate real-time systems composed of scalable components, which can operate in one of several modes. Each of these modes defines a certain trade off between the resource requirements of the components and their output quality. A resource constrained real-time system has to guarantee that all required modes are feasible, as well as all required mode changes. The latency of a mode change, defined as the time needed to re-allocate memory between components, should satisfy timing constraints.

1.1 Context

Multimedia processing systems are characterized by few but resource intensive tasks, communicating in a pipeline fashion. The tasks require processor for processing the video frames (e.g. encoding, decoding, content analysis), and memory 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 [Wust et al. 2004], or network bandwidth requirements for quality [Jarníkov et al. 2004]. We assume there is no memory management unit available.

1.1.1 Example application

We consider a video content analysis system in the surveillance domain, shown in Figure 1. The boxes represent components and the thick arrows represent data flow between the components, which communicate via shared buffers. An arrow can be regarded as a shared buffer, where the components pointed by the arrows are the ones reading from and writing to the buffer.

The video stream from a camera monitoring a bank office enters the system through the Video In component, which places it in a buffer. The Encoder reads the raw frames from the buffer and encodes them for sending over the network. The Network Out component takes the encoded frames and sends them over the network to the video processor.

The Network Out component will read the encoded frames from the buffer and transmit them only when network bandwidth is available. We assume that the network is shared with other applications, and therefore the network may not always be available for Network Out due to congestion. Moreover, the time intervals when the network is available are not predictable. Even if we can assume a certain average bandwidth, we cannot predict at which moments in time it will be available.

On the video processor side, 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 a 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 the occurrence of an event in the video is identified, the system may change into a different mode. For example, when a robbery occurs, we may want to transmit the video stream over the network to a portable device, e.g. a PDA, allowing offsite and onsite police officers to gain insight into the robbery. This will require taking some resources from the VCA or the Alert component to allow the Network Out component to transmit the video.

1.1.2 Scalable components

We investigate a system running a single application composed of $n$ scalable components $C_1, C_2, \ldots, C_n$, where each component $C_i$ can operate in one of $m_i$ predefined component modes $M_{i,1}, M_{i,2}, \ldots, M_{i,m_i}$. We assume a one-to-one mapping between components and tasks, with component $C_i$ being mapped to task $\tau_i$.

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. Figure 2 shows an example of the interaction between QMC and the Decoder component.

The QMC may request a component to operate in a particular mode. Upon such a request the component will change its state to the requested mode (within a bounded time interval).
The QMC interacts in the same way with all scalable components in the application, as shown in Figure 3. We assume that buffers are implemented as proper components, and that the Network Out, Decoder, VCA and buffer components are scalable.

![Figure 3: Interaction between the QMC and the scalable components.](image-url)

**Component and system modes** Each component mode $M_{c,i}^j$ specifies a tradeoff between the output quality of the component $C_i$ and its resource requirements. 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 actual 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.

The product of all target (or actual) component modes at a particular moment in time defines a system mode $M_s^i$. In a system comprised of $n$ components a system mode is an $n$-tuple ranging over the component modes, i.e $M_s^i \in M_{c,1}^j \times M_{c,2}^j \times \ldots \times M_{c,n}^j$. The product of target component modes is called a target system mode, and the product of actual component modes is called an actual system mode.

Since we are interested in the resource management aspect of quality of service, in the remainder of this document we will use component and system modes to refer to the corresponding resource requirements. The quality settings serve merely as labels for the requirements.

**Memory reservations** A component mode specifies the memory requirements of a component while it is operating in this particular mode. To guarantee the required memory during runtime, the component expresses its memory requirements in terms of memory reservations. Before starting to operate in a particular mode, the component will request memory reservations, where each reservation specifies the size of a contiguous memory space to be used exclud-
sively by the requesting component. A component may request several memory reservations, thus distributing its total memory requirements among several reservations. Upon a request the framework will check if the reservation can be granted, considering the total memory size and the current reservations.

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.

**Mode changes** While the system is executing in mode $M_i$ 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 individual component to change its mode, in order to make sure that in the end the system will be operating in the target mode. In terms of memory reservations, some components are 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, which will request additional reservations.

**Example: a scalable MPEG decoder** As a leading example of a scalable component we will consider the Decoder component and assume it to be an instance of a scalable MPEG decoder [Haskell et al. 1996, Jarnikov 2007]. An MPEG video stream consists of a stream of I, P and B frames, with interdependencies shown in Figure 4.

![Figure 4: Inter-frame dependencies in an MPEG video stream, in an IBBPBBPBBP Group of Pictures.](image)

The incoming video stream is assumed to be layered, as shown in Figure 5. 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.

The Decoder processes incoming frames one by one. It starts with decoding the base layer $B$. If the enhancement layer $E1$ is available, and the Decoder is “allowed” to process the $E1$ layer it does so, otherwise it returns the decoded base layer and continues to decode the incoming stream with the base layer of the next frame. The decision whether or not to process an enhancement layer is taken based on the availability of data (internal control) and the available resources, in particular memory space (external control, e.g. the QMC).

There are different scalable video coding techniques, such as spatial, temporal, or SNR scalability [Jarnikov 2007]. Figure 6 shows an example of temporal
Figure 5: Frame processing in a layered MPEG decoder.

Figure 6: Processing of an IBBPBBPBBP Group Of Pictures, encoded with temporal scaling.

Note that we assume that every frame in the base layer is decoded, and only frames in the enhancement layers may be skipped.

1.2 Problem statement

In this document we investigate the problem of bounding the latency 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.

1.3 Our approach

Traditionally, if we were to request a mode change from a set of components we would have to wait until all their tasks complete their current invocation, before changing their mode and reallocating memory between the components affected by the mode change. To speed up the process, we divide the tasks into smaller subtasks and allow to preempt a job, together with its allocated memory, at subtask boundaries. This allows us to preempt a job and reallocate its memory before the job completes.

Our approach is suited for applications composed of scalable components, which produce incremental results, e.g. the 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.

1.4 Scope of this document

In this document we present a method for swift system mode changes. By swift we mean the mode change latency satisfies specified timing constraints. We restrict our investigation to uni-processor systems and compare two scheduling algorithms: Fixed Priority Preemptive Scheduling (FPPS) [Liu and Layland 1973] and Fixed Priority Scheduling with Deferred Preemption (FPDS) [Burns 1994, Burns et al. 1994, Gopalakrishnan and Parulkar 1996, Burns 2001, Burns and Wellings 2001, Bril et al. 2007].

Several ingredients are required for implementing a system supporting swift mode changes. Global interfaces have to be defined which are used between the components and the underlying system to facilitate the component and system mode changes. After the system decides on a particular mode change, it has to manage the individual component modes in a way which will make the system mode change possible. In particular it has to allocate the reservations comprising the component modes. As we will see later, the reservation allocation scheme has an impact on which system mode transitions are possible and which are not.

• We would like to gain insight into the costs of mode changes (latency, memory space for intermediate modes and administration). How does reservation allocation affect these costs? Is it possible to bound the costs, if so, what is the bound?

• We would like to investigate allocation schemes which take into account future mode changes. How can we specify the desired future mode transitions? Can we assume a static mode transition “schedule”?

• Our ideas about swift mode changes will be implemented and validated on top of / next to the Resource Management Component (RMC) [Holenderski 2008], developed within the CANTATA project.

1.5 Contributions

• Method for “preempting memory”, where the memory allocated to an active job may be reallocated to a different job, without corrupting the active job’s state.

• Modeling framework for mode changes in systems composed of scalable components.

• Analysis of system mode change latency, with respect to memory allocation, for Fixed Priority Preemptive Scheduling and Fixed Priority Scheduling with Deferred Preemption.

• Tighter bound on the latency of mode changes in the processor domain, under the assumption of Fixed Priority Non-preemptive Scheduling, compared to the existing bound for Fixed Priority Preemptive Scheduling.
• Observation of the similarities between processor and memory reservations.

• Interface description for specifying component modes and controlling mode changes during runtime. Extension of the control dependencies in the sub-job graphs with external control (e.g. the QMC).

1.6 Outline
2 Related work

2.1 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.

[Martins and Burns 2008] provide more general definitions for mode and mode changes in the real-time systems domain. They define mode as

> “the behavior of the system, described by a set of allowable functions and their performance attributes, and hence by a single schedule, containing a set of processes and their timing parameters”,

and mode change as:

> “a change in the behavior exhibited by the system, accompanied by a change in the scheduling activity of the system”.

They distinguish between the mode as perceived from the system perspective, and the perspective of the application or end-user.

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, Wust et al. 2004] or network bandwidth [Jarnikov et al. 2004], we let a mode describe the memory requirements of the components.

In the remainder of this section we will use $L$ for 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.

[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 the standard fixed priority sporadic task model, with task $\tau_i$ 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:

- **Ability to abort old-mode tasks upon a mode change request:**
  - All old-mode tasks are immediately aborted.
  - All old-mode tasks are allowed to complete normally.
  - Some tasks are aborted.

- **Activation pattern of unchanged tasks during the transition:**
  - Protocols with periodicity, where unchanged tasks are executed independently from the mode change in progress.
  - Protocols without periodicity, where the activation of unchanged periodic tasks may be delayed.
• Ability of the protocol to combine the execution of old- and new-mode tasks during the transition
  – Synchronous protocols, where new-mode tasks are never released until all old-mode tasks have completed their last activation in the old mode.
  – Asynchronous protocols, where a combination of both old- and new-mode tasks are allowed to be executed at the same time during the transition.

[Sha et al. 1989][Liu 2000, p. 103, 326] provide a synchronous mode change protocol without periodicity and bound the latency of a mode change, for rate monotonic priority assignment with $D = T$. They consider mode changes in priority-driven systems with independent tasks, and with dependent tasks for arbitrary resource access protocols. [Liu 2000] model the execution of a mode change as an aperiodic or sporadic mode-change job and bound the mode change latency by
\[ L = \max_{\tau_i \in \Gamma} T_i \]  
where $\Gamma$ is the complete task set.

[Sha et al. 1989] provide a tighter bound for the priority ceiling protocol. They observe that during a mode change the priority ceilings of some semaphores involved in the mode change many need to be raised, before the new-mode tasks resume. Since a semaphore may be locked before the mode change request, the time needed to raise the priority of a semaphore is bounded by the period of the task with the priority equal to its old priority ceiling. For rate monotonic priority assignment, the lower the priority of a task, the longer its period. Therefore, [Sha et al. 1989] bound the latency of a mode change by
\[ L = \max(T_s, \max_{\tau_i \in \Gamma_{del}} T_i) \]  
where $\Gamma_{del}$ is the set of the tasks to be deleted, and $T_s$ is the period of the task with the priority equal to the lowest priority ceiling of all the semaphores whose ceilings need to be raised.

[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). This algorithm bounds the mode change latency by
\[ L = \sum_{\tau_i \in \Gamma_{del,c}} C_i \]  
where $\Gamma_{del,c}$ is the set of tasks in the old mode which are to be deleted, but have to complete first.

[Real and Crespo 2004] devise an asynchronous mode change protocol with weak periodicity, applicable to systems where only some old-mode tasks can be aborted. We refer to their protocol as having weak periodicity, because in case the new-mode tasks which are released before the mode change completes interfere with the periodic unchanged tasks, rendering the schedule unfeasible, then their protocol delays the next activation of the unchanged tasks.
In this document we present a synchronous mode change protocol, without periodicity. Periodic unchanged tasks are not activated until the mode change has completed (limiting thus the interference with the tasks involved in the mode change). We assume that the old-mode tasks cannot be aborted at an arbitrary moment, however, they do not have to complete either. Each task is assumed to have defined a 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.

[Nelis and Goossens 2008] provide a sufficient condition on a mode change in a real-time system with identical multiprocessors. They assume that tasks are independent, and that during a mode change from old mode to new mode, all tasks in the old mode need to be disabled, and can either be aborted or required to run until completion, before the tasks in the new mode are resumed. They present a mode change protocol based on the Minimum Single Offset Protocol [Real and Crespo 2004], and derive an upper bound on the mode change delay, equal to

$$L = \begin{cases} \max_{\tau_i \in \Gamma_{delc}} T_i & \text{if } |\Gamma_{delc}| \leq m \\ \frac{1}{m} \sum_{\tau_i \in \Gamma_{delc}} T_i + (1 - \frac{1}{m}) \max_{\tau_i \in \Gamma_{delc}} T_i & \text{otherwise} \end{cases}$$

where $m$ is the number of processors in the system.

[Almeida et al. 2007] present a two phased approach for mode changes, where possible modes are evaluated offline and limited to those with highest utilization. The QoS component then uses this pre-computed information to change the modes during runtime.

2.2 Resource reservations

2.2.1 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.

[Audsley et al. 1994] present a method for exploiting the spare processor capacity during runtime. They consider a task set composed of hard and soft tasks, where the hard tasks are assumed to be schedulable, and the soft tasks may be schedulable, if enough spare capacity can be guaranteed during runtime. They identify two forms of spare capacity: gain time being the processor time allocated to hard tasks off-line but not used at runtime, and slack time being the processor time not allocated to hard tasks. They work under the assumptions that \( D \leq T \), and that tasks are independent.

[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 FPSPS and EDF, and disk bandwidth reservations according to EDF. For disk bandwidth scheduling they optimize the overhead of arbitrary preemption with a kind of slack stealing algorithm granting disk access to tasks closer to the current head position. However, the slack stealing optimization does not perform any better than pure 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 \( S + L \).

They distinguish between hard, firm and soft reservations, depending on how the gain-time and slack-time are allocated.

- Hard reservations are assumed to be allocated according to their worst case requirements, and therefore they do not exploit any gain-time. When depleted they cannot be scheduled until replenished.

- Firm reservations are allocated less than their worst case requirements, and can exploit the gain-time due to hard and firm reservations. When depleted they can be scheduled only if all other hard and firm reservations are depleted.

- Soft reservations share the slack-time, which was not allocated to other hard and firm reservations. They can also exploit the gain-time due to hard, firm and soft reservations. When depleted soft reservations can be scheduled along with other depleted hard, firm and soft reservations.

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 space.

[Abeni and Buttazzo 2004] investigate processor reservations in the context of EDF scheduling. They propose the Constant Bandwidth Server for serving periodic and aperiodic tasks. Its goal is to provide feasibility (no deadline must be missed), efficiency (tasks should be served in order to reduce the mean tardiness) and flexibility (the server should handle tasks with variable or even unknown execution times and arrival patterns). They achieve these by modifying the deadline of the running task: when a task exhausts the budget of its server its deadline is postponed (and hence its priority is reduced). This way
the task releases the processor to a higher priority task, but still remains ready to consume any unused capacity by the higher priority tasks.

2.2.2 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 reduced.

[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.

2.3 Memory allocation schemes

[Saksena and Wang 2000] present an algorithm which selectively disables pre-emption in order to minimize the maximum stack size requirement while respecting the schedulability of the task set. They make the observation that the total amount of memory used by a task set can be limited, by allowing preemption to occur only between selected task groups. They assign a static preemption threshold \( \gamma_i \) to every task \( \tau_i \) and introduce the notion of mutually non-preemptive tasks, if for tasks \( \tau_i \) and \( \tau_j \) with priorities \( \pi_i \) and \( \pi_j \), we have \((\pi_i \leq \gamma_i) \land (\pi_j \leq \gamma_j)\). Then they define a non-preemptive group as a subset of tasks \( G \subseteq T \), where for every pair of tasks \( \tau_i, \tau_j \in G \), \( \tau_i \) and \( \tau_j \) are mutually non-preemptive. They show that the maximum number of task frames on the stack is equal to the number of groups, and observe that a small number of groups leads to lower requirement for the stack size.

[Gai et al. 2001] consider multi processors systems, where tasks communicate via shared memory, and propose an efficient implementation of the pre-emption thresholds [Saksena and Wang 2000] using SRP [Baker 1991], and a method for assigning the preemption thresholds in such a way that the total memory requirement is minimized.

[Kim et al. 2002] integrate the preemption threshold scheduling with priority inheritance and priority ceiling protocols.
2.4 Resource access protocols

2.4.1 Priority Ceiling Protocol

The Priority Ceiling Protocol (PCP) has been introduced by [Sha et al. 1990] to bound the priority inversion, and prevent deadlocks and chained blocking, when tasks can access shared resources.

To avoid multiple blocking a task is allowed to enter a critical section only if it will not be blocked by an already locked resource before completing the critical section. It assigns a priority ceiling to every resource, which is equal to the highest priority among the tasks which can ever lock it. During runtime it maintains a system ceiling, equal to the highest priority ceilings among all the currently locked resources. A task is allowed to lock a resource only if its priority is higher than the system ceiling, otherwise the task is blocked. Upon locking a resource the task inherits the resource’s priority ceiling.

2.4.2 Stack Resource Protocol

[Baker 1991] improve on PCP and propose the Stack Resource Protocol (SRP). In its FPPS form (when we equate the preemption thresholds to task priorities), SRP is similar to PCP, with the difference that it compares the priority of the task to the system ceiling already when the tasks wants to start executing (rather than waiting until the task tries to lock a resource).

A nice property of SRP is that the task executions are perfectly nested. It means that if a task \( \tau_i \) preempts task \( \tau_j \), then \( \tau_j \) will not execute again before \( \tau_i \) has finished, a property that does not hold for PCP. As a consequence a single stack can be used for executing both tasks.

Additional advantages of SRP with respect to PCP include:

- Simpler management of priority ceilings. In PCP the priority of a task is changed when it inherits the priority of a resource it locks, while in SRP the task priorities are static.
- Applicability to Dynamic Priority Scheduling, e.g. EDF.
- Applicability to multi-unit resources.

2.4.3 Banker’s algorithm

In its original formulation [Dijkstra 1964] the Banker’s algorithm is intended to manage a single multi-unit resource. The analogy is made to a pot of money managed by a banker, which wants to guarantee that all the money he loans will be paid back eventually. It addresses the problem of deadlock between borrowers, who may not be able to complete their business if they do not receive all the funds they request, which could occur if portions of the money pot are distributed over several borrowers. The algorithm assumes that if a borrower is granted the requested loan, it will complete its business within a finite amount of time and will return all the borrowed money upon completion.

In a formulation closer to the real-time domain, the banker’s algorithm can be used to manage access to a single multi-unit resource among several tasks. The banker corresponds to the scheduler responsible for deciding which task
can be running. The scheduler is invoked every time a process tries to access a shared resource. In this sense the bankers algorithm is similar to PCP.

On the other hand, since PCP is limited to single-unit resources, the Banker’s algorithm resembles SRP, which avoids deadlock when sharing a multi-unit resource.

While both PCP and SRP rely on the priority ceilings for resources pre-computed offline, the Banker’s algorithm requires only the information already available in the task model, of course at the cost of larger runtime overhead, since upon every resource access it will have to check for deadlock against all other tasks.

2.4.4 Wait-free synchronization

The wait-free synchronization [Herlihy 1991, Herlihy and Moss 1993] offers an alternative protocol for accessing shared resources, which do not require mutual-exclusion.

[Herlihy and Moss 1993] define the notion of a transaction for accessing a number of shared (single-unit) resources, e.g words in a shared memory. A transaction is a finite sequence of resource-access instructions, executed by a single process, satisfying the following properties:

**Serializability:** From the semantic perspective, transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another.

**Atomicity:** Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making its changes visible to other processes (effectively) instantaneously, or it aborts, causing its changes to be discarded.

The resource-access instructions include instructions for reading from and writing to the shared memory. The transaction’s data set is the set of accessed memory locations, divided between: read set (those that are read), and write set (those that are written).

The tentative changes will not take effect until the transaction is committed. Upon a commit, the transaction tries to execute the tentative changes. It succeeds only if no other transaction has updated any location in the transactions data set, and no other transaction has read any location in this transactions write set. If it succeeds, the transactions changes to its write set become visible to other processes. If it fails, all changes to the write set are discarded.

The wait-free synchronization allows to share resources without the need for acquiring and releasing locks, with the associated blocking. Their simulation results show, however, that the wait-free approach works well for transactions that have short durations and small data sets. The longer the transaction duration and the larger the data set, the more likely it will be aborted due to an interrupt or a synchronization conflict.

[Holman and Anderson 2006] show how to apply wait-free synchronization in Pfair scheduling in multi processor systems.
2.5 Implementation of scalable applications

[Geelen 2005] investigate applications which can change their functionality during runtime, by varying their sets of components. Their notion of dynamic functionality is similar to the concept of scalability discussed in Section 2.1, where the current task set is determined by the current system mode. They propose a method for dynamic loading of memory in real-time systems. They present a case study of a DVD player platform, with a memory management unit present but disabled, which uses static memory partitioning for memory management.

They propose a method with the memory management unit enabled for dividing the memory requirements into overlays, grouping together the memory requirements of several components belonging to a particular component mode (which they refer to as a “use-case”). They also show how to manage the loading and storing of overlays between RAM and hard disk by directly controlling the entries in a TLB, based on an implementation in a particular DVD player product. They identify several stages during the system runtime, and limit the dynamic memory allocation to the initialization phase. They assume soft deadlines on the mode changes, aiming at providing “smooth” transitions.

3 Our approach

In this section we introduce our approach for bounding the latency of system mode changes, in memory constrained real-time systems composed of scalable components.

3.1 Mapping between component modes and subtasks

Figure 7 shows a task graph, a component mode graph and the mapping between the subtasks and the component modes for a scalable Decoder introduced in Figure 5. Note that this is an example of a Decoder, and other implementations are possible.

![Diagram](image)

Figure 7: A scalable MPEG decoder example: (a) task graph, (b) component mode graph, (c) mapping of subtasks to component modes.

For each frame the Decoder may decide (or be requested by the QMC), to provide a specific video quality. It will first always process the base layer. Then,
Upon reaching the next preemption point it will follow the selected path: either return the decoded base layer frame, or continue with decoding the first enhancement layer. $\sigma_{\text{return}}$ indicates that the base layer can be simply returned, without additional processing effort. If the decoder continues to decode the enhancement layer, upon reaching the next decision point it will again decide whether to return the currently encoded layers or continue with decoding the next enhancement layer. If it decides to return, it has the choice between returning the base layer by following $\sigma_{\text{return}}$, or first merging the base and enhancement layers before returning the result, indicated by $\sigma_{\text{merge}}$.

The Decoder component is scalable, meaning that it can operate in one of several predefined modes, as shown in the component mode graph in Figure 7.b. The nodes indicate component modes, while the edges indicate the order between the modes, with the arrowhead pointing to a mode requiring more resources.

Each subtask requires a particular set of resources and therefore operates in a particular component mode, with the subtask-to-mode mapping shown in Figure 7.c. At a preemption point the component is assumed to be in the component mode mapped to the last executed subtask.

3.2 Fixed Priority Scheduling with Deferred Preemption

We employ FPDS allowing preemptions only at subjob boundaries, referred to as preemption points. Assuming the preemption points are selected in such a way that critical sections do not span across the preemption points, we can exploit FPDS as a resource access protocol. With non-preemptive subjobs we can be sure that subjob can access shared resources without corrupting them.

Assuming that every task invocation is independent of previous invocations, we can bound the component mode change latency by the duration of the longest path in its task graph\(^1\). However, if we make sure that every preemption point coincides with a preliminary termination point (see Assumption 7), we can bound the component mode change latency by the duration of the longest subtask.

3.3 Resource reservations

Resource reservations place the responsibility for resource management somewhere between the component and the system. On the one hand, they require additional work from the programmer to specify and negotiate the reservation parameters, before actually requesting the resources. On the other hand, reservations allow exploiting the component specific knowledge to guide the resource management process and thus reduce the overhead found in fully automatic resource management.

We consider a system running several concurrent components. In the resource reservation model [Mercer et al. 1994, Rajkumar et al. 1998] a component may request a budget for a particular resource, such as processor time, memory space or hard disk bandwidth. In this document we focus on the memory resource. Memory reservations aim at guaranteeing memory provisions by

\(^1\)In case of an MPEG decoder, a frame may depend on all frames in a Group Of Pictures (typically 12 frames). In this case the mode change latency (for providing or taking the memory) is bounded by 12 times the duration of a single task invocation.

**Note:** May need more explanation
allowing components to negotiate memory budgets before allocating the memory.

We specify a memory budget by the size of contiguous memory that is being requested. A component may request several memory budgets, thus distributing its total memory requirement among several reservations. Note that all the reservations of a component do not need to occupy contiguous memory space. This simplifies the memory (re)allocation and allows the system to allocate the memory more efficiently.

In a system comprised of several components there is a natural tradeoff when allocating resources to individual components. The budget negotiation and allocation process should therefore support the greater goal of managing a system wide quality of service, through system modes and system mode changes. We consider a centralized approach for managing QoS, with the QMC component playing the managing role.

3.3.1 Similarity between processor and memory reservations

Allocating processor to tasks is similar to allocating memory in the sense, that in both cases a particular portion of silicon is allocated to a particular task for a particular time interval. In case of processor scheduling it is the silicon comprising the logic and memory (registers, cache, etc.), while in case of memory “scheduling” it is the storage elements comprising the allocated memory space.

Allocation vs. access The main purpose of the memory reservations is to guarantee future memory allocations to a component by preventing other components from allocating memory within the reserved memory space. Note that memory reservations do not address the problem of components accessing (i.e. reading from or writing to) memory locations outside of their reservations. It is similar to a processor reservation, where the reservation guarantees a certain processor time to a component, but it does not prevent higher priority tasks (e.g. interrupt service routines) from corrupting the state of the processor. Just like interrupt service routines are assumed to properly save and restore the registers to keep the processor state in tact, we assume that components are implemented carefully, ensuring the integrity of memory allocations. In this document a memory reservation guarantees only that a component will be able to allocate the reserved memory size.

To make the similarities between memory and processor reservations more explicit, we can define complementary notions for the four processor reservations ingredients (see Section 2.2):

Admission When a memory reservation is requested, the system has to check if granting the reservation will not affect any spatial constraints, i.e. if the requested memory can indeed be allocated together with the current reservations of other components.

Scheduling Scheduling in the processor domain is a way of allocating the processor time among tasks. In the memory domain, this corresponds to allocating memory space among tasks. The similarity becomes apparent when we compare the graphical representation of a cyclic processor schedule where different time units are allocated to different tasks, with that
of memory blocks in a linear memory space being allocated to different tasks.

**Accounting** In order to determine if a task can be granted a memory allocation request, the memory 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, i.e. preventing them to allocate memory outside their reserved memory space.

## 4 Application model

This section defines our application model. We assume a single application in our system.

**Application** An application $\mathcal{A}$ is a set of $n$ components $\{C_1, C_2, \ldots, C_n\}$.

**Component** A component $C_i$ defines the interfaces used for communicating with the system and other components. We distinguish between two classes of components: application components (e.g. the Decoder in Section 1.1.1) and framework components (e.g. the QMC in Section 1.1.2).

**Task** A task $\tau_i$ describes the work to be done by component $C_i$. There is a one-to-one mapping between tasks and components.

The task may be composed of several subtasks, dividing the task into smaller units of computation. The subtasks are arranged in a directed acyclic graph, referred to as the task graph, where the edges represent subtasks and the nodes represent subtask boundaries, referred to as decision points. An example is shown in Figure 8, showing the subtasks $\sigma_{ij}$ for task $\tau_i$.

![Figure 8: An example of a subjob graph.](image)

The execution of a task is called a job. 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.
A task $\tau_i$ is further specified by a fixed priority $i$. The priority assignment is arbitrary, i.e. not necessarily rate or deadline monotonic.

We consider periodic tasks, where a task is also described by a period $T$, which specifies the fixed inter-arrival time between consecutive jobs.

We assume a one-to-one mapping between tasks and components, and sometimes use $\tau_i$ also to refer to the component corresponding to task $\tau_i$.

**Subtask** A subtask $\tau_{ij}$ describes the work to be done by a single step on the task graph of a task $\tau_i$. Each subtask is mapped to a component mode, specifying the resources it requires for execution.

A subtask is further described by its worst-case execution time $C_{ij}$.

**Job** A job $\sigma_i$ represents the execution of a task $\tau_i$. It consists of a chain of subjobs, corresponding to a path through the subtask graph of the corresponding task.

**Subjob** A subjob $\sigma_{ij}$ represents the execution of a subtask $\tau_{ij}$.

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.

In this document we use subjobs as subtasks interchangeably, when the meaning is not ambiguous.

**Decision point** Decision points mark the places in the execution of a task, when the following subtask is selected. They are the nodes in the task graph. 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. We assume a one-to-one mapping between the preemption points and the decision points in the task graph. A subjob is non-preemptive with respect to other application components and may be preempted only by a higher priority subjob belonging to a framework component (assuming no resource sharing between application and framework components).

In case of FPPS, there are no explicit preemption points and a subjob may also be preempted by a higher priority subjob belonging to an application component.

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.

**Preliminary termination point** Preliminary termination points identify places during a job execution, where the processing can be terminated, skipping the remaining subjobs in the subjob 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.

In case of FPPS we assume that preliminary termination points can be also placed within preemptive subjobs.

4.1 Arbitrary termination

Sometimes it may be convenient to terminate a job before it completes. For example, when the Decoder component is requested to reduce its mode while it is processing an enhancement layer, it knows that upon the next preemption point the result of current processing will be discarded. Therefore, to utilize the resources more efficiently, it can be beneficial to “prematurely” terminate the processing of a job. However, it may not always be possible to terminate a job at an arbitrary moment, otherwise e.g. it may leave some shared resources in an inconsistent state. We assume that a job may terminate only at preliminary termination points. Note that termination is not the same as preemption. A subjob can not be preempted under FPDS, but it may be terminated.

There are several options on how to deal with arbitrary termination. The choice depends on the application.

4.1.1 Continue until next preliminary termination point

Upon a request to reduce its mode, a component may be required to continue until the next preliminary termination point. For example, if FPDS is used as a resource access protocol using non-preemptive subtasks to guard the access to shared resources, upon a termination request a job will be required to continue until the next preemption point (coinciding with the next preliminary termination point).

4.1.2 Terminate and jump forward

If a subjob is not using mutually exclusive resources, upon a request to reduce its mode, it may be able to abort its computation and jump to the next decision point. For example, if as the consequence of the mode reduction the data for processing the current subjob is no longer available, then it is not necessary to continue with processing the current subjob, and the job can jump to the next decision point to continue with the following subtask.

4.1.3 Terminate and restore a roll-back state

If jumping forward immediately is not possible, upon a request to reduce its mode, the subjob may terminate, restore a previously saved roll-back state, and jump either to the next or the previous decision point in the task graph (coinciding with a preliminary termination point).

This option requires an approach similar to wait-free synchronization, saving and restoring a roll-back state at the last preliminary termination point, at the cost of additional overhead for storing the roll-back state at every preliminary termination point, and restoring it upon abortion.
5 Platform model

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.

5.1 Memory reservations

A memory reservation $R_{i,j}$ is specified by a tuple $(C_i, m_j)$, where $C_i$ is the component requesting the reservation, and $m_j$ is the size of the contiguous memory space requested. A component may request several reservations, and in this way distribute its memory requirements between several reservations.

6 Assumptions

Throughout this document we make the following assumptions:

Modes and mode changes

A1 The resource requirements of all components in their highest modes cannot be satisfied at the same time.

A2 System mode changes have hard deadlines. Transitions between modes need to have a predictable latency, satisfying timing constraints.

A3 A component mode change is limited to the reallocation of dynamic resources (e.g. memory allocated with malloc()). It does not include static resources (e.g. code, read-only data, static variables), because these will always be needed.

A4 There is a single system mode change at a time. Unlike traditional mode change protocols, a system mode change may be requested also during the transition between modes. An ongoing system mode change can be interrupted and its target mode can be changed, however, there can be no race condition between two mode changes. There is always a single target system mode.

A5 A component can be requested to change its mode at an arbitrary time during its execution.

A6 The time needed by the RMC to allocate and deallocate a budget is bounded. We assume, that the data stored within a memory budget is not cleared, allowing fast (bounded) memory deallocation.

A7 The system does not support arbitrary termination, meaning that a sub-task cannot be terminated at an arbitrary moment in time, but only at predefined preliminary termination points. Note that this does not influence preemption, where a task may be preempted (at an arbitrary moment in time under FPPS) and later resumed.
Platform

A8 The system is running on a single processor.

A9 Memory can be (re)allocated dynamically during runtime and allocation is not limited to the initialization phase.

A10 There is no memory management unit available to protect tasks from accessing memory outside their allocated memory space. There is also no means for segmentation and paging.

Components

A11 Resources are not locked across decision points.

A12 At a preliminary termination point the component can always change its mode to one with lowest resource requirements.

A13 There is no resource sharing between application and framework components.

A14 There is a one-to-one-to-one mapping between the decision, preemption and preliminary termination points.

A15 The system is closed, meaning that all components are known upfront. Similar to the closed-system assumption in offline scheduling, if a component enters or leaves the system, the application model has to be recomputed, requiring order of magnitude more effort, than a single system mode change.

7 Component modes

We investigate a system running a single application composed of \( n \) scalable components \( C_1, C_2, \ldots, C_n \), where each component \( C_i \) can operate in one of \( m_i \) predefined component modes \( M_{i,1}^c, M_{i,2}^c, \ldots, M_{i,m_i}^c \). Each mode specifies a tradeoff between the output quality of the component and its resource requirements. For the time being we will restrict the discussion to modes ranging over a single resource: the memory, and disregard other resources.

From the resource management perspective a mode specifies the reservations which are required by the component to operate in that mode, i.e. \( M_{i,j}^c = \{ R_{i,j1}, R_{i,j2}, \ldots \} \). Figure 9 shows an example of a component mode graph, which shows possible modes for a component requiring 100B, 250B or 600B of memory for three different quality settings.

In a component mode graph the nodes represent the component modes, and the edges represent the order between the modes, with the arrow head indicating the mode requiring more resources.

For the time being we consider only component mode chains, i.e. we assume a total order on the component modes. As future work, when we extend the resource management to other resource besides memory, and allow a component to specify requirements for several resources simultaneously, we will also extend the component mode graphs to proper graphs (see Section 15.10).
Each subjob requires a particular set of resources and therefore operates in a particular component mode, with the subtask-to-mode mapping shown in Figure 7.c.

When a job enters a decision point the component is assumed to be in the component mode mapped to the last executed subtask. However, while the component is resting at a preemption point (e.g. when it is preempted by a higher priority task at a preemption point in case of FPDS), its mode may change (e.g. when requested by the QMC). Therefore, while at a decision point, the component may be in any of the modes supported by the subtasks ending and starting at that point.

8 System modes

In a system comprised of several components we can define a system mode $M_s$ as the product of component modes, and represent it by a tuple ranging over $M_{c1} \times M_{c2} \times \ldots \times M_{cn}$, where $M_{ci}$ is the set of all possible modes of component $C_i$. Our system mode specifies the task parameters (memory requirements in particular), compared to the traditional mode definition found in literature (see Section 2.1), where a mode is defined as the task set (during a mode change old tasks may leave or new tasks may enter the system).

For example, let us consider a system consisting of three components $A$, $B$ and $C$, with their component mode graphs shown in Figure 10. The component modes are labeled with $A_1$, $A_2$, etc. for reference in later discussion.

An example of a system mode is the tuple $(A_2, B_1, C_2)$, representing the system with component $A$ in mode $A_2$, component $B$ in mode $B_1$ and component $C$ in mode $C_2$.

We define a system mode transition graph, where nodes represent system modes, and the edges represent possible system mode transitions. An edge can be traversed in both directions, while the arrow head represents the system mode requiring more resources. A system mode transition is mode change of a single component. Figure 11 shows an example of a system mode transition graph.

The edges in a system mode transition graph define a partial order on the
Figure 10: Example of a component mode graphs for components A, B and C.

Figure 11: Example of a system mode transition graph.

system modes, where an arrow from $x$ to $y$ represents $x \prec y$, meaning that $y$ requires more resources than $x$. Note that some system modes may be comparable in the sense of their total amount of memory requirements, but since the order is partial, they may be not comparable based on the defined order, e.g. modes $(A_2, B_1, C_1)$ and $(A_1, B_2, C_2)$ in Figure 11.

8.1 Components starting and stopping during runtime

To model systems where components can be started and stopped during runtime, we extend the component mode graph for each component with a bottom mode $\perp$. It stands for the resource requirements of a component when it is not running and not requiring any resources. This way we can still represent system modes as tuples ranging over all known components, since dynamic loading is not supported (see assumption A15). For the time being, however, to keep the discussion simple, we will assume that all components are running and cannot be stopped, and we will omit the $\perp$ modes.

8.2 Restrictions on the system modes

If our system could support all components in their highest modes, then the problem of resource management would disappear. Therefore, we assume that not all highest modes can be satisfied at the same time (see Assumption A1).

\footnote{Besides the resources needed by the system to administer the component in its dormant state.}
Returning to our previous example, let us assume that the system manages 600B of memory. In this case the mode \((A_2, B_2, C_2)\) cannot be satisfied. We indicate the system restrictions by removing the unfeasible system modes from the system mode transition graph. Figure 12 shows the restricted system mode transition graph for our example.

![Figure 12: Example of a system mode transition graph, restricted to feasible modes.](image)

Observe that every system mode transition graph contains one *bottom* node, with all components in their “smallest” modes. Also, the graph contains several “leaves”, i.e. modes with no outgoing edges to bigger modes. Since the system in a leaf mode always provides more resources to components than in one of the lower modes, the system will strive at all times to operate in a leaf mode\(^3\). It will only move to other modes in order to change from one leaf mode to another.

### 9 Mode changes

Given a system mode transition graph, we can use it to reallocate budgets between components during runtime. A system mode change is defined by a *mode change path* traversing mode transitions in the graph, starting in the current system mode and ending in the target mode. Figure 13 shows a mode change in the system from our continuing example.

![Figure 13: Example of a mode change.](image)

Initially the system is in mode \(X\) and we want to change to mode \(Y\). The

---

\(^3\)One may argue that this holds only as long as we ignore non-functional requirements, such as power consumption. However, these non-functional requirements can be included in the system modes by treating them as additional resources, as discussed in Section 15.10
system will first reduce the budget of one component and move to an interme-
diate mode, before increasing the budget of an other component to get to mode
Y.

Transitions downward in the graph require a reduction of resource provisions
to one component. The component has to be notified about the desired change
so that it can adapt its quality settings to the reduced resources. After it has
adapted, it will discard part of its current budget making it available to the
system for reallocation. Such a transaction will usually take some time. Note
that from the resource management’s perspective, we assume a monotonically
decreasing relation between modes in a system mode transition graph, meaning
that if the system can support mode $i$, then it can also support mode $j$, if there
is a directed path from $j$ to $i$ in the graph.

Transitions upwards in the graph are always possible from the component’s
perspective, since they simply increase the resource provisions to a component.
From the resource management’s perspective, depending on the current memory
reservation allocation, granting a new budget may not always be possible, in
spite of there being enough total memory space available (e.g. if there is no
single gap large enough to accommodate the new budget). In such cases the
mode change path has to be adjusted to include lower system modes, until
a mode is reached which allows the target budget reallocation, as shown in
Figure 14.

![Figure 14: Example of an adjustment to the mode change path.](image)

The dashed edge indicates a transition which is infeasible due to current re-
source allocation. Section 9.2 discusses the infeasible transitions in more detail.

9.1 Preempting a system mode change

At every step of the mode change from $X$ to $Y$ the system is in a stable state, i.e.
each component is executing in one of its predefined modes, even though some of
them may be operating at sub-optimal quality level. Therefore a system mode
change can be “preempted” before mode \( Y \) is reached, and the target mode can be changed from \( Y \) to a different mode.

This does not mean that the system may not request from several components to change their modes at the same time. After a successful component mode change, the component notifies the system of the change. This notification signifies a system mode transition.

### 9.2 Restrictions on the system mode transitions

As indicated earlier, depending on the current memory reservation allocation, allocating a new budget may not always be possible, even if there is enough total memory space available (see [Holenderski 2008] for the discussion of the budget fragmentation problem).

In this section we show an example of budget allocation which leads to infeasible system mode transitions. We continue with the example system from Section 8, with a system comprised of three components \( A \), \( B \) and \( C \), their component mode graphs shown in Figure 10 and a total of 600B of memory managed by the system. Figure 15.a shows the memory space (represented by an array of 6 blocks of 100B each) and a possible memory budget allocation for the root system mode \( (A_1, B_1, C_1) \). Letter \( A \) in a memory block indicates that this particular chunk of 100B is allotted to a budget owned by component \( A \).

![Figure 15: Example of an adjustment to the mode change path.](image)

Let us assume our target mode is \( (A_2, B_2, C_1) \), with component \( A \) in mode \( A_2 \) requiring two reservations: one of 100B and one of 200B. Looking at the system mode transition graph in Figure 12 we have two path choices to the target mode: one going through mode \( (A_2, B_1, C_1) \) and one going through \( (A_1, B_2, C_1) \). Let us pick the one going through \( (A_1, B_2, C_1) \). Now we can choose which available memory chunk to assign to component \( B \). If we pick the one shown in Figure 15.b then the transition from \( (A_1, B_2, C_1) \) to the target mode \( (A_2, B_2, C_1) \) becomes infeasible (indicated by the dashed arrow). However, if we allocate the other memory chunk to \( B \), as shown in Figure 15.c, then we can reach the target as shown in Figure 15.d. Note that we also had the choice of selecting the mode change path going through \( (A_2, B_1, C_1) \), in which case the allocation of memory budgets leading to the target mode was unambiguous.

This example shows that during mode changes, some mode transitions may become infeasible.
10 Quality Manager Component

To manage system modes we introduce the Quality Manager Component (QMC) responsible for governing the quality of service of the complete system. It can be regarded as a controller component, aware of the mode graphs of all components and the resource assignment strategies of the Resource Management Component (RMC). During the initialization phase it requests the mode graphs from all components and creates a system mode graph. During runtime the QMC consults the system mode transition graph when requesting mode changes from individual components.

Upon a mode change request the component will discard already owned reservations or request new ones. An example of interaction between the QMC, the Decoder and the RMC is shown in Figure 16.

![Figure 16: Interaction between the QMC, Decoder and the RMC.](image)

As indicated earlier, some mode transitions may not be feasible due to the current reservation allocation. The QMC therefore needs to communicate with the RMC, at least to be aware if the target system mode is feasible with the current reservation allocation, indicated by the dashed line between the QMC and the RMC.

The QMC is system specific, as it needs to respond to specific events triggering system mode changes. In this document we focus on the generic QMC interface implementing the mode changes, and we assume that the QMC somehow knows when to change the system mode and to which target mode, e.g. based on a user request [Hentschel and Bril ????].

11 Protocol between components and the QMC

Components and the QMC have to adhere to the following protocol, as depicted in Figure 17:

**Initialization**

1. Component registers its mode graph at the QMC.
2. QMC requests the component to start in a particular mode.
### Runtime

3. QMC requests the component to change its mode.

4. When the component has changed its mode it notifies the QMC.

![Figure 17: The protocol for negotiating mode changes between components and the QMC.](image)

Components comply with the protocol by implementing the `QMC_RegisterModeTransitionGraph()` and `QMC_SetMode()` methods, and calling the `QMC_SetModeDone()` method in QMC upon completing a requested mode change.

The mode change requests are asynchronous: a component may return from the `QMC_SetMode()` method before completing the mode change. It allows the QMC to request several component mode changes in parallel, without having to wait for each component to complete its mode change before requesting a mode change from another component. This may speed up a mode change which otherwise would take a long time, e.g. when decreasing a mode requires to free up data structures used by a video processing algorithm currently processing a video stream.

We assume that the QMC will not request a mode change from a component before the previous one has been handled. If it does, than the new request will be ignored (with the status indicated in the return value of `QMC_SetMode()`).

Call to `QMC_SetModeDone()` signifies a transition in the system mode transition graph, since at the moment of the call the QMC is certain that the component is indeed in its target mode.

### 11.1 Three phases to changing a system mode

A system mode change, depicted in Figure 14, from the current mode $M^*_c$ to the target mode $M^*_t$, is executed according to the following three phases:

1. The QMC determines which components need to reduce their mode in...
order to accommodate the increase of other component modes necessary for the target system mode. This determines the lowest mode $M_s^l$, which is the system mode on the mode change path requiring the least resources. Note that this mode is not necessarily the root mode.

2. The QMC then asks all the components on the path from $M_s^c$ to $M_s^l$ to reduce their mode (by calling their QMC_SetMode() method).

3. When all components have decreased their mode (and called the QMC’s QMC_SetModeDone() method), the QMC asks all the components on the path from $M_s^l$ to $M_s^t$ to increase their mode (by calling their QMC_SetMode() method). The mode change is complete when all components have increased their mode (and called the QMC’s QMC_SetModeDone() method).

The remainder of this section describes the interface methods in more detail.

### 11.2 Registration of the mode graph at the QMC

During the initialization phase, each component submits its mode graph to the QMC, by calling the QMC_RegisterModeTransitionGraph(handle, modeGraph) method with the following parameters:

- **handle**: Pointer to the component instance. It is used by the QMC to associate a mode graph with a particular component.
- **modeGraph**: Pointer to an array representing the mode graph, i.e. the memory requirements for each mode, with the element at index 0 indicating the smallest mode. Each element is an array of integers specifying the size of memory reservations (in bytes) required by the corresponding mode. The data structure pointed to by modeGraph must remain in tact throughout system execution. It is later used by the QMC to communicate target mode changes.

### 11.3 Request a mode change

During runtime the QMC may decide to increase the mode of a component, if there are enough resources available, or to decrease a component’s mode, to accommodate an increase in the mode of another component. It requests a mode change by calling the QMC_SetMode() method, implemented by the component. The component then proceeds to adapt its mode by requesting more budgets or discarding some budgets it already owns (see [Holenderski 2008] for the description of the protocol for negotiating memory budgets).

The QMC_SetMode(modeChangeHandle, targetMode) method takes the following parameters:

- **targetMode**: Pointer to the target mode inside the modeTransitionGraph supplied earlier with a call to QMC_RegisterModeTransitionGraph().
- **modeChangeHandle**: An identifier used by the QMC to match the later to QMC_SetModeDone() with the requested component mode change.

Note: Before asking the components to increase their modes, the QMC needs to tell the RMC where to allocate the reservations, in order to prevent infeasible system mode transitions.
11.4 Confirm a mode change

After the component has changed its mode it notifies the QMC. A mode change may take some time, e.g. when a video processing component is not able to pre-empt the processing of a video frame and free the memory space at an arbitrary moment in time, but has to wait until it completes the current frame. Also, a mode change may not always be possible, e.g. due to unfortunate budget allocation by the RMC (see Section 9.2). The $\text{QMC\_SetModeDone}(\text{modeChangeHandle}, \text{status})$ method takes the following parameters:

- **modeChangeHandle**: The mode change identifier passed with the corresponding $\text{QMC\_IncreaseMode()}$ call.
- **status**: An indication if the mode change was successful.

12 Analysis

A system mode change is defined by the source (or old) system mode and the target (or new) system mode. There may be different paths possible from the source to the target mode. In this section we bound the latency of a system mode change for a particular path through the system mode transition graph.

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, 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 strange and unreasonably expensive to first wait to complete the work, when reducing a component mode we will, and then throw away the result. However, since arbitrary termination is not supported (according to assumption A7), by coinciding the preliminary termination points with preemption points, this allows us to reduce the latency of mode changes.

12.1 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. According to assumption A6 the system 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. We distinguish two cases, depending on whether the component is requested to decrease or increase its mode.

Decreasing a mode will require the component to discard some of its resource budgets. As discussed above, assumption A7 implies that a component will respond to a mode change request within the duration of the longest subtask. The latency of reducing a component mode is therefore given by:

$$ld(\tau_i) = \max_{\tau_{ij} \subseteq \tau_i} (C_{\tau_{ij}}) + C_{sys}$$  \hspace{1cm} (5)
Increasing a mode is easy, from the perspective of the component, as no shared resources which are being currently used need to be discarded. We can therefore ignore the component overhead and bound the latency of increasing a mode by:

\[ li(\tau_i) = C_{sys} \]  \hspace{1cm} (6)

12.2 System mode change latency

Every mode change is defined by the current mode \( M^c_s \) and the target mode \( M^t_s \). However, the path through the system mode transition graph is defined also by the lowest node in the graph which is on the path, i.e. the node corresponding to the system mode with components requiring the least resources during the mode change. Let us therefore define a mode change path by three modes: \( M^c_s \rightarrow M^l_t \rightarrow M^t_s \), where \( x \rightarrow y \) means that there is a path in the system mode transition graph from mode \( x \) to mode \( y \), and \( M^l_t \) corresponds to system mode with the lowest resource requirements. Note that several different paths may be possible in a system mode transition graph which satisfy this constraint, as shown in Figure 18.

**Figure 18:** Different paths satisfying the constraint \((A_2, B_2, C_1) \rightarrow (A_1, B_1, C_1) \rightarrow (A_1, B_2, C_2)\)

**Lemma 12.1.** All mode change paths satisfying the constraint \( M^c_s \rightarrow M^l_t \rightarrow M^t_s \), with \( M^l_t \) representing the lowest system mode on the path, have equal latency.

**Proof.** Section 11.1 describes a three-phase protocol, where the QMC first computes the \( M^l_t \), then it asks all the affected components to reduce their modes to bring the system from \( M^c_s \) to \( M^l_t \), and when this step is completed, it asks all the affected components to increase their modes to bring the system into mode \( M^t_s \).

Let \( A_2 \) and \( B_2 \) be the modes of components \( A \) and \( B \) in system mode \( M^c_s \), and \( A_1 \) and \( B_1 \) the corresponding modes in system mode \( M^l_t \). Section 12.1 shows that the latency of a mode change of a single component does not depend
on other components. Therefore, following a path $M_i^c \rightarrow M_i^s$, it does not matter whether first component $A$ reduces its mode from $A_2$ to $A_1$ and then component $B$ reduces its mode from $B_2$ to $B_1$, or vice versa. The same holds for components increasing their modes on the path $M_i^s \rightarrow M_i^c$.

Since different paths satisfying the $M_i^c \rightarrow M_i^s \rightarrow M_i^c$ requirement differ only in the order of component mode changes, all these paths have the same latency.

\[ L(\gamma) = \sum_{\tau_i \in \gamma} ld(\tau_i) + \sum_{\tau_i \in \gamma_s} li(\tau_i) = \sum_{\tau_i \in \gamma_d} \max_{\tau_{ij} \in \tau_i} (C_{\tau_{ij}}) + |\gamma| \cdot C_{sys} \quad (7) \]

\[ L_d(\gamma_d) = \max_{\tau_i \in \gamma_d} (ld(\tau_i)) + |\gamma - 1| \cdot C_{sys} = \max_{\sigma \in \tau_i, \tau_i \in \gamma_d} (C_{\sigma}) + |\gamma| \cdot C_{sys} \quad (8) \]

\[ L_d(\gamma_d) : \text{latency of a system mode change given by path } \gamma_d, \] when the currently running component is requested to decrease its mode

Note that this bound is tight, but pessimistic, as it does not take into account
which component is currently active, but considers the longest subtasks of all components along the system mode change path.

Latency of the system mode change, when increasing the mode of the currently running component, is bounded by the sum of the system overheads for (de)allocating the reservations of all mode transitions along the path.

\[
\mathcal{L}_i(\gamma_i) = \max_{\tau_i \in \gamma_i} (li(\tau_i)) + |\gamma - 1|C_{sys} = |\gamma|C_{sys}
\]  

(9)

\[
\mathcal{L}_i(\gamma_i) : \text{ latency of a system mode change given by path } \gamma_i, \text{ when the currently running component is requested to increase its mode}
\]

12.3 Improving the bound for synchronous mode change protocols, without periodicity

As indicated in Section 2.1, 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

\[
\mathcal{L} = \sum_{\tau_i \in \Gamma_{delc}} C_i
\]

(10)

Their algorithm behaves similarly to the one we described for the FPPS case, in Section 12.2.1. 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 subsets: those that can be aborted upon a mode change request (\(\Gamma_a\)), and those which need to run till completion (\(\Gamma_c\)). It is the tasks that need to complete, which contribute to the mode change latency, i.e. \(\Gamma_{delc} \subseteq \Gamma_c\).

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

\[
\mathcal{L} = \max_{\tau_i \in \Gamma_{delc}} C_i
\]

(11)

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 FPNS.

13 Prototype

14 Conclusions

The system mode transition graph model presented in this document allows to manage the system mode changes during runtime in such a way, that the overall system utilization is maximized, by minimizing the disruption in the operation
of affected components, while reducing the utilization of selected components by the amount just necessary to perform the change.

Our approach assumes, however, that the swapping of data in the memory is expensive. Otherwise, instead of selecting a path through the the system mode transition graph, one could simply swap the memory reservations affected by the mode change with negligible overhead. Therefore our approach is best suited for systems where the cost of moving data in the memory is not negligible, e.g. when there is no memory management or DMA support, or in memory constrained systems where there is not enough spare memory space for swapping the data.

We have also considered the bound of the latency of a system mode change, and compared two approaches, based on FPPS and FPDS. We showed that FPDS 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.

15 Future work

15.1 Aborting a component at an arbitrary moment

In this document we have assumed that all subjobs are non-preemptive. In multimedia processing systems, however, we can think of examples where one may desire to abort the current subjob, when we know that its results will be discarded, or when we would like to provide an immediate mode change. We would like to extend the control with abort mechanisms, considering different roll back scenarios, where the state of the task is restored to a previous state or completely reset.

15.2 Relaxed mapping between subjobs and component modes

In this document we have assumed that at a preemption point the component can always change its mode to one with lowest resource requirements (see assumptions A7 and A 12 and Section 4).

We would like to relax this mapping and for example allow two consecutive subjobs to operate in the same component mode. Unfortunately, this will increase our bounds on the latency of a mode transition, as we cannot guarantee that the mode will change upon the next preemption point. Since we assume acyclic subjob graphs, the mode change latency remains bounded by the duration of the longest path in the graph. A tighter bound, however, will probably require the use of additional resource access protocols, to guarantee the longest chain of subjobs which may “lock” the current mode, before changing to the target mode. We would like to investigate how SRP could fill this role, and provide the corresponding analysis.

15.3 Extension to arbitrary levels of component modes

In this document we have considered only a two-level hierarchy of modes: system modes and component modes. We would like to investigate extending our results to a hierarchy of arbitrary depth, where components can be composed of other
subcomponents, each with its own mode graph, where all subcomponent mode graphs define the component mode graph.

15.4 Speeding up system mode transitions
Section 9 suggests that the QMC should change system modes one by one, following the mode change path in the system mode transition graph. The interface in Section 11, however, allows to specify an arbitrary target component mode, rather than only a neighboring mode in the component mode graph. Also, the QMC may request a mode change from several components simultaneously. This suggests that we might relax the assumption that a system mode transition spans a single edge in the system mode transition graph and extend it to an arbitrary path. It is to be investigated whether such an extension will speed up mode changes and how it will affect the resource allocation.

15.5 Apply FPDS to reduce the latency of a processor mode change
[Real and Crespo 2004] present an asynchronous mode change protocol with (weak) periodicity, and bound the delay of a mode change under FPPS by the sum of worst-case execution times of all tasks in the old mode. Their results hold under weaker assumptions than our, in particular, they do not assume the existence of preliminary termination points. We can extend their algorithm to work under FPDS and show that

- under our stronger assumptions the bound on the mode change latency can be reduced
- under their weaker assumptions the average mode change latency can be reduced

15.6 Pruning infeasible mode transitions
As mentioned in Section 9.2 the Resource Management Component (RMC) may not always be able to increase a budget of a component, due to the current budget allocation; the transitions upward in the graph may not always be possible. Techniques for budget allocation are to be investigated helping to avoid such situations.

If we would like to keep the semantics of edges in the system mode transition graph as possible mode transitions, then the edges have to be limited to possible transitions, e.g. by some sort of pruning technique, coinciding with an appropriate budget allocation strategy. However, this results in an inconvenient intertwining of functionality between the QMC and the RMC.

15.7 Interface between the QMC and the RMC
As mentioned in Sections 10, 9 and 15.6, the QMC needs to know how the RMC allocates the requested budgets, in order to chose appropriate paths in the system mode transition graph. With the current interface between components and the QMC, the QMC can request the desired mode change and the

Note: This is a good topic for a paper! Required: simulation results
component will notify if the mode change was possible or not, via the callback. However, this will result in increased mode change delay.

An interface and the associated protocol between the QMC and RMC are needed, for communicating the information about the current memory allocation, while keeping the functionalities and responsibilities well separated.

15.8 Extrapolation to processor mode changes

The literature on mode changes is limited to modes describing the processor requirements. In this document we have considered mode changes, where a mode specifies the memory space requirements of tasks. We have shown a method for bounding the latency of a mode change under weaker assumptions, than the assumptions made in the literature on processor mode changes. For example, we allow a mode change to be requested also during another mode change. In Section 3.3.1 we have shown the close relation between processor and memory reservations. It remains to be investigated whether and how the mechanisms presented here could be applied to processor mode changes.

15.9 Multi processors

In this document we have considered a uni-processor system. Under this assumption FPDS avoids the need for resource access protocols. In a multi-processor system FPDS may require additional mechanisms to guarantee the non-preemptivity of subjobs, necessary for the predictability of mode changes.

15.10 Extension of component modes to multiple resources

So far we considered only the memory resource. We would like to extend our model to other resource types, such as the processor or network. It is to be investigated whether the techniques for modeling and managing resource requirements in scalable components as described in this document can adapted to other resource types.

15.11 Exploiting spare memory capacity

[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.

15.12 Efficient representation of the system mode transition graph

During runtime the QMC component uses the component mode graphs and system mode transition graphs extensively. For every mode change it has to identify a path from the current system mode to the target mode. It can also use the system mode transition graph to chose between alternative system modes, when
more than one system mode satisfies a particular target component mode. To keep the runtime overhead small, efficient data structures are to be investigated for storing all mode and mode transition graphs and the relations between them.

References


[Hentschel and Bril ????] C. Hentschel, R. Bril. In ? ???


