Memory Management for Multimedia Quality of Service in Resource Constrained Embedded Systems*

Mike Holenderski, Chidiebere Okwudire, Reinder J. Bril, Johan J. Lukkien
Eindhoven University of Technology,
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands
m.holenderski@tue.nl, c.g.u.okwudire@student.tue.nl, r.j.bril@tue.nl, j.j.lukkien@tue.nl

Abstract

In this paper, we consider multimedia Quality-of-Service (QoS) in resource constrained embedded systems, where scalable applications are structured as directed acyclic graphs of tasks, which communicate via shared buffers. Scalable multimedia applications allow to trade quality for resource usage during run-time. We present two QoS problems: (i) temporal dependencies between subchains of tasks due to a common predecessor, and (ii) mode change latency in applications. These problems are addressed through advanced memory management techniques. For the first problem, it is shown how additional access to the buffer, in particular the support for dropping selected frames, allows to guarantee the Quality of Service of the application during overload conditions, preventing congestion in one buffer to propagate across the whole application. For the latter problem, the approach of in-buffer scaling is applied to reduce the mode change latency in scalable multimedia processing applications, which can adapt their memory requirements during run-time according to a set of predefined modes. The latter approach is validated with simulation results.

1 Introduction

In this paper we investigate memory management for real-time multimedia applications running on top of a resource constrained platform, and demonstrate an approach for dynamic storage allocation which can improve the Quality of Service (QoS) provided by the whole system.

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. Tasks use *components* to do the work, and can be regarded as the control-flow through the components, as shown in Figure 1.

We consider multimedia applications which are implemented as a directed acyclic graph (DAG) of data-driven tasks, with a time-driven (i.e. periodic) head and tail task. Tasks communicate via bounded buffers. Task execution is determined by priority, data availability, buffer sizes and time triggering at the boundaries of the system, however, we assume that the end-to-end latency of the complete chain is bounded. Task execution times may vary and depend on the data they process. We focus on applications which pose a QoS requirement in terms of a minimum and maximum bound on the time interval between consecutive different outputs generated by the last task in the chain.

Scalable applications are those which can adapt their workload to the available resources. In this paper we consider scalable applications which can operate in one of several predefined modes. A *application mode* specifies the maximum 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, giving rise to a tradeoff between the resource requirements and provided quality. 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, e.g. a mode change latency may not be longer than the time interval between two consecutive rendered frames.

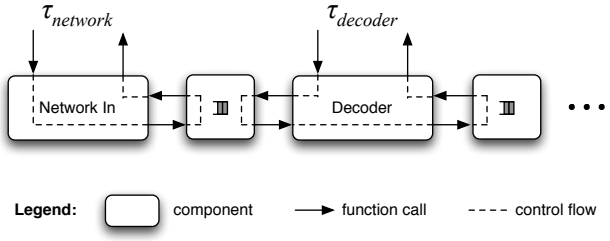

Figure 1: An example of a network task and a video decoding task communicating via a shared buffer. Tasks are represented by the paths traversing the components.

1.1 Problem statement and its motivation

In this paper we address two QoS related problems in resource constrained multimedia systems. We assume a single processor environment, where tasks are preemptive.

Hentschel et al. [1] present the theory and practice of video Quality-of-Service for Consumer Terminals. They describe a pipelined multimedia processing application, where a multiplexed audio and video input stream is demultiplexed into two buffers and processed independently by two processing subchains, as shown in Figure 2. They observe that fluctuation in the processing time for decoding different video frames may fill up the buffers along the video processing subchain, in particular the video output buffer of the demultiplexer. This may cause the demultiplexer to block and in effect starve the audio subchain, leading to the dropping of audio frames, manifested by sound artifacts. In this paper we present an approach which prevents the demultiplexer from blocking.

The second problem we address in this paper is bounding the mode change latency in scalable multimedia applications, distinguishing between the *application latency* and the *system latency*. From the application

*This work has been supported in part by Information Technology for European Advancement (ITEA2), via the Content Aware Networked Systems Towards Advanced and Tailored Assistance (CANTATA) project.

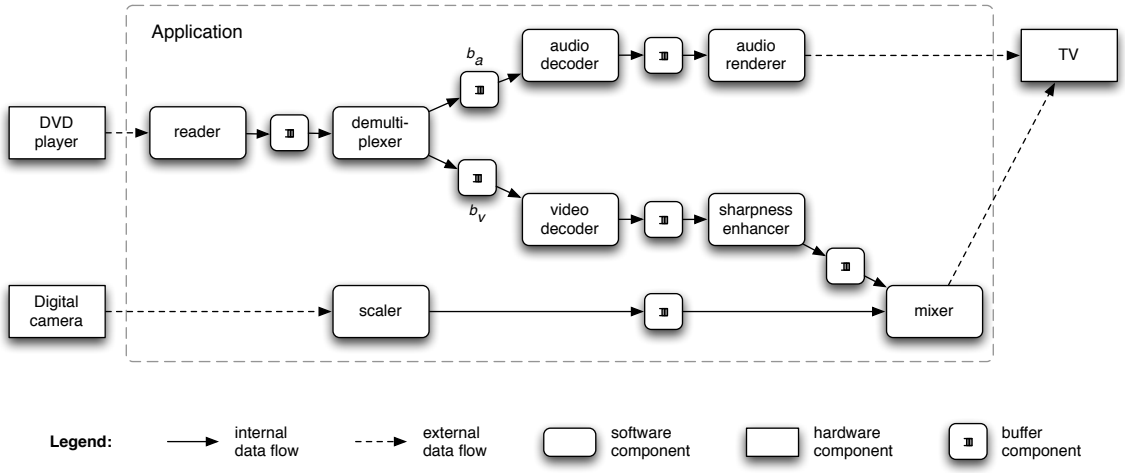

Figure 2: Block diagram of the system implementation on a multimedia processor presented in [1]. The demultiplexer component writes to two output buffers: the audio output buffer b_a and the video output buffer b_v.

perspective, a mode change may interfere with the processing of a video stream. A large mode change latency will delay the rendering of frames and lead to artifacts in the output video. From the system perspective, during a mode change, resources can be allocated to applications in the new mode only *after* they have been released by the applications in the old mode. Therefore the mode change latency will impact when the resources can be reallocated between applications and when these applications can resume operating in the new mode.

1.2 Contributions

The contribution of this paper is two fold. First, it is shown how additional access to the buffer, in particular the support for dropping selected frames, allows to guarantee the QoS of the application during overload conditions, preventing congestion in one buffer to propagate across the whole application.

Second, the approach of in-buffer scaling is applied to reduce the mode change latency bound in scalable multimedia processing applications, which can adapt their memory requirements during runtime according to a set of predefined modes.

This paper presents a general technique targeted at streaming, or data driven, systems, and uses example media processing applications for illustration and motivation.

1.3 Outline

Section 2 provides an overview of the related work, followed by a description of our application and platform models in Section 3. Section 4 describes the proposed architecture, which is refined in the following sections as new concepts are added to address the tackled problems. Our first contribution is described in Section 5, followed by our second contribution in Section 6. Section 7 concludes this paper.

2 Related work

In this section we summarize the related work on memory management in embedded systems, QoS during overload conditions, and mode changes in scalable applications.

2.1 Memory management in embedded systems

In Geelen [2], systems are investigated 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 2.3, where the total memory requirements are determined by the current application 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 static overlays, created during the initialization phase. An overlay groups together the memory requirements of several components belonging to a particular mode (e.g. dvd playback or tv recording). 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, and implement their approach in a particular DVD player product. They assume soft deadlines on the mode changes, aiming at providing QoS in terms of "smooth" transitions.

In this paper we present an approach for dynamic reallocation of memory between components during runtime, on a platform without a memory management unit. We organize the memory requirements of components into memory budgets, which components can request, resize and discard during runtime. We show how the components can resize their budgets during a mode change, before the memory can be reallocated to other components.

Our approach is well suited for platforms with explicitly managed local memory, such as *scratch-pad memory*. A scratch-pad provides low latency data storage, similar to on-chip caches, but under explicit software control. The simple design and predictable nature of scratchpad memories has seen them incorporated into a number of embedded and real-time system processors [3]. McIlroy et al. [4] present a dynamic run-time heap management algorithm for scratch-pad memory. Their approach is

based on a combination of fixed-sized macro blocks, which are later subdivided into variable sized allocations.

In this paper, while in Section 5 we do not assume any particular memory organization, in Section 6 we assume the memory is managed in terms of fixed sized blocks, allowing to easily reallocate memory between components. We reuse the notion of *memory reservations* described in [5, 6]. They are defined as containers for memory allocations, allowing components to request memory and guarantee granted requests during runtime, avoiding static memory allocation in the 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.

2.2 Quality of Service during overload conditions

Wüst et al. [7] define QoS constraints in terms of three elements which need to be balanced: (i) minimizing deadline misses, (ii) maximizing the processing quality, and (iii) minimizing the quality changes.

The mechanism for selectively dropping buffer elements for frames, as presented in this paper, aims at resolving the induced temporal dependencies. To the best of our knowledge this problem has not been addressed in literature in the context of memory constrained multimedia applications with QoS constraints.

2.3 Scalable applications and mode changes

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 [7], or *network bandwidth* requirements for quality [8]. In this paper we investigate trading *memory* requirements for quality. We assume there is no memory management unit available.

In [5, 6] we investigate a system running a single application using n *scalable components* $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$, where each component $\mathcal{C}_i$ can operate in one of several predefined *component modes*. Each component mode represents a tradeoff between the output quality of the component $\mathcal{C}_i$ and its *memory requirements*, expressed in terms of memory reservations. The component modes of all components at a particular moment in time define an *application mode* $\mathbb{A}$.

We assume the existence of a *Quality Manager* (QM) component, which is responsible for managing the individual components and their modes to provide a system wide Qo.

While the application is operating in mode $\mathbb{A}$ the QM may be requested (by an external event) to perform a *mode change*, defined by the *current mode* and 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 real-time system has to guarantee that all required modes are feasible, as well as all required *mode changes*. The *(system) 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.

Real and Crespo [9] present a comprehensive overview of mode change protocols, where modes express the application's processor requirements.

In [5, 6] we present a mode change protocol which bounds the mode change latency by

$$\mathcal{L}(\gamma) = \max_{\tau_j \in \Gamma}\Big(\max_{\tau_{j,k} \in \tau_j} C_{\tau_{j,k}}\Big) + \sum_{\mathcal{C}_i \in \gamma} (C_{i,p} + C_{sys}) \quad (1)$$

where $\mathcal{L}$ is the upper bound on the mode change latency, γ is the set of components involved in the mode change, Γ is the task set, $\tau_{j,k}$ is a subtask of task τ_j with worst-case computation time $C_{\tau_{j,k}}$, $C_{i,p}$ is the worst-case time required by $\mathcal{C}_i$ to do any pre- and post-processing during a mode change, and C_{sys} is the worst-case system overhead for reallocating the affected reservations.

Equation (1) divides the mode change latency into two parts: the $\max$ part and the $\sum$ part. The $\max$ part defines the time interval between the mode change request and the time when the mode change actually starts. The $\sum$ part defines the duration of the mode change itself (i.e. the duration of the resource reallocation). In [5, 6] we showed how to reduce the former, and in this paper we show how to reduce the latter.

3 The model

In this section we describe our application and platform models.

3.1 Application model

An application $\mathcal{A}$ is defined by a set of m tasks $\{\tau_1, \tau_2, \ldots, \tau_m\}$. We assume several applications in our system. The applications are independent: they do not share any logical resources and their interaction is limited to time-sharing the physical resources, such as the processor or memory.

A task τ_i is defined by its fixed and unique priority i, with 1 being the highest priority, and worst-case computation time C_i. A task may optionally be periodic, with T_i specifying the interarrival time of its jobs.

Tasks use components to do their work. 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 memory to store the encoded frame data which it reads from the input buffer and provides methods for decoding the frame and placing it in the output buffer.

A task can be regarded as the control flow through the components it uses to do the work, as shown in Figure 1. Tasks of different applications do not share components.

The components are responsible for the memory requirements of an application. While tasks represent the processor-time requirements, the components they use are the major contributors to their computation time parameters (besides the system overheads).

Multimedia application

A multimedia application $\mathcal{A}$ consists of a chain of $N_{\mathcal{A}}$ tasks communicating via $N_{\mathcal{A}} - 1$ shared buffers. The first and the last tasks in the chain are time driven with period $T_{\mathcal{A}}$. All other tasks in the chain are data driven. In the remainder of this paper we will focus on the behavior of a single application and omit the $\mathcal{A}$ index.

To accommodate different frame sizes at different stages in the processing pipeline, we assume frames may span across several buffer elements. To keep things simple, however, we assume that each buffer element contains at most one video frame, i.e. video frames do not share buffer elements. Therefore we assume an n-to-1 relationship between buffer elements and video frames.

3.2 Platform model

In this paper we assume that memory is managed in terms of fixed sized blocks. They avoid external fragmentation and simplify the reallocation of memory between components.

A component expresses its memory requirements in terms of *memory reservations* (or *memory budgets*), where each reservation guarantees access to a particular number of blocks during runtime. A memory reservation R_i is specified by its capacity s_i and manages access to s_i memory blocks. Once a reservation is granted the component may request memory blocks from it. A reservation R_i makes sure that it does not serve more blocks than s_i. A component may request several reservations, and in this way distribute its memory requirements among several reservations.

We assume a single processor environment, where tasks are preemptive.

4 The architecture

Before describing the architecture we first summarize the assumptions on the system:

- A1 A system-wide fixed memory allocation size: a block.
- A2 A variable budget size which is always a multiple of block size. A budget does not necessarily represent a contiguous chunk in physical memory.
- A3 An n-to-1 mapping between buffer elements and video frames.
- A4 Each buffer has a single writer and a single reader, where both writer and reader are aware of the mapping of frames to buffer elements.
- A5 The end-to-end latency for the complete chain is bounded by MT, where M is a natural number greater than 0 and T is the period of the head and tail task in the chain.
- A6 The application poses a QoS requirement in terms of a minimum and maximum bound on the interarrival time between consecutive outputs generated by the leaf task in the DAG.

The remainder of this section describes the basic architecture, introducing the necessary components and their interactions. In later sections we extend it with additional interfaces and components.

4.1 Resource Manager

The Resource Manager (RM) component 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 RM has the following interface:

`void RM_Init()` allocates the memory managed by the RM and initializes local data structures for managing it.

`budget RM_RequestMemoryBudget(size)` checks if a budget with the given `size` can be accommodated and if so allocates it and returns a handle to it.

`void RM_DiscardMemoryBudget(budget)` discards the `budget`, allowing other requests to reuse the discarded memory.

`block RM_AllocateMemory(budget)` checks if there is a free block still available in the `budget` and if so marks it as occupied and returns a pointer to it.

`void RM_FreeMemory(block)` discards the `block` making it available for later `RM_AllocateMemory()` requests.

`void RM_Finit()` frees the memory managed by the RM.

4.2 Buffer Component

The buffer components are responsible for most of the application's memory requirements. Traditionally a buffer component provides interfaces for creation, reading, writing and destruction of the underlying bounded FIFO queue:

`void Buffer_Init()` allocates the memory for the queue, i.e. requests a budget from the RM.

`element Buffer_Peek()` returns a pointer to the head of the queue, or blocks if the queue is empty.

`void Buffer_Pop()` pops the head of the queue, or does nothing if queue is empty.

`element Buffer_Pull()` returns a pointer to the first free element at the tail of the queue, or blocks if the queue is full.

`void Buffer_Push()` marks the element returned by the last call to `Buffer_Pull()` as occupied.

`void Buffer_Finit()` frees the memory allocated for the queue, i.e. discards the corresponding budget.

5 Handling overload conditions

Hentschel et al. [1] present a study of a multimedia processing application, shown in Figure 2. They observe that fluctuation in the processing time for decoding different video frames may fill up the buffers along the video processing subchain. The buffers may continue to fill up until the demultiplexer component becomes blocked, when it is not able to write both the audio and the video frames into its two output buffers. This is an example of how the decoding of the audio stream may become blocked due to insufficient buffer space along the video processing path. In the end this leads to the dropping of audio frames, manifested by sound artifacts.

In this section we will focus on the audio and video subchains following the demultiplexer.

The demultiplexer reads multiplexed frames from its input buffer and writes the audio part to its audio output buffer b_a and the video part to its video output buffer b_v. If either of these queues is full, then the demultiplexer blocks.

Let t be the time when b_v becomes full. If we do not address the congestion in the video subchain, then the demultiplexer will block and consequently stop feeding b_a, eventually leading to audio artifacts.

One option is to extend b_v with additional memory, to prevent the demultiplexer from blocking. However, since we assumed a memory constrained platform, we cannot simply allocate more memory to one application without other applications suffering. We therefore propose to free up some space for the demultiplexer by dropping some of the frames in b_v. Note that dropping frames from any other buffer along the video subchain will not alleviate the problem, since the demultiplexer will still be blocked on its output buffer.

We cannot simply pop the head element from the video output buffer. Since a single video frame is allowed to span across several buffer elements (A3), the head element could belong to a video frame which is currently being processed by the video decoder.

Waiting until the decoder has finished processing the current frame, and then discarding the head frame by simply popping it, is not an option either, since this is the very delay we want to avoid.

Instead, we propose to selectively drop those frames from b_v which can no longer be processed in time. Since we have assumed a bounded end-to-end latency (A5) and adjusted the buffer capacities accordingly, an overload condition means that a deadline was already missed. We want to prevent the deadline miss to propagate to other video *and* audio frames, leading to artifacts in the audio stream. We therefore propose to scan the video output buffer for the beginning of the next frame and drop all those elements which belong to that frame. A mechanism for dropping arbitrary frames also makes it possible to scan the buffer for the least significant video frame (e.g. a B or P frame in case of MPEG encoding [10]) and to drop that one instead, thus preserving the perceived quality of the output video as much as possible.

For this to work we extend the buffer component interface in Section 4.2 with additional methods providing limited access to arbitrary buffer elements:

`element Buffer_GetNextBufferElement()` allows to iterate through the buffer elements. Every following invocation returns the following buffer element, or a `NULL` pointer if the end of the queue was reached.

`void Buffer_ResetNextBufferElement()` resets the pointer returned by the next `Buffer_GetNextBufferElement()` call to the head of the queue.

`void Buffer_Drop(element)` deletes the buffer element pointed to by `element` from the queue.

We need to iterate through the buffer elements (using `Buffer_GetNextBufferElement()` and `Buffer_ResetNextBufferElement()`) and drop those belonging to the least significant frames (using `Buffer_Drop()`). This congestion control should be triggered by the demultiplexer whenever it notices a full output buffer. It should be executed by a component which is aware of the semantics of the frames in the congested buffer.

Only the access to the shared data structures (such as the administration data structures for buffers and budgets) is synchronized, according to the Stack Resource Policy [11]. Having the highest priority task do all the scaling and dropping of buffer elements guarantees that no other task will interfere.

6 Reducing mode change latency

In Section 5 we have dealt with overload conditions. In this section we investigate reallocating memory between applications. We describe a scalable multimedia application, which can operate in one of several modes (where a mode describes its memory requirements), and show how to reduce the mode change latency. We outline our implementation of the proposed method in the μC/OS-II real-time operating system, and present simulation results which validate our approach.

Let us consider a multimedia processing application shown in Figure 3. It is an instance of an application comprised of a chain of tasks described in Section 3.1. It consists of three communicating tasks. The network task τ_n receives incoming frames via the Network component and writes them to the buffer b_e containing encoded frames. The decoder task τ_d reads the encoded frames from b_e, decodes them using the Decoder component, and writes the result to b_d containing decoded frames. The renderer task τ_r reads the decoded frames from b_d and has the Renderer component render them. The hardware attached to the Renderer component buffers the last rendered frame, and in case a new frame has not arrived in time, the buffered frame is rendered.

We introduce scalability by having scalable buffer components. A scalable buffer component can adapt its size during runtime by requesting to resize its reservation. We add the following method to the RM interface in Section 4.1:

`void RM_ResizeBudget(budget, size)` resizes a particular `budget`, as long as the new `size` does not exceed the available memory space.

We also add an additional component to the system, the Quality Manager component (QM), which is responsible for coordinating the mode changes. There is one QM per application, which is responsible for distributing the resources between the components comprising the application. In a system comprised of several applications, there may be a global QM, which distributes the resources between the applications. Following the architecture description in [6] we add the following methods to the interface of each scalable component, in particular the buffer component in Section 4.2:

`void QM_BeforeModeChange(targetMode)` In case the `targetMode` is smaller than the current mode, this method allows the component to gracefully scale down its memory requirements and to reduce its budget.

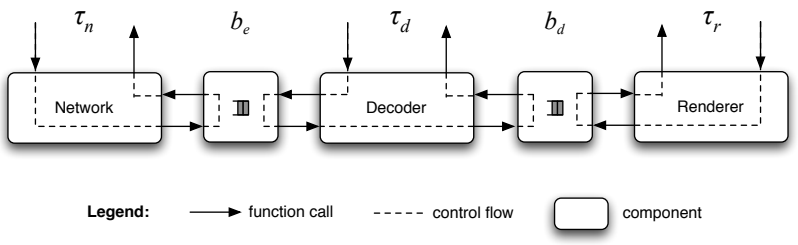

Figure 3: An example of a scalable multimedia application.

`void QM_AfterModeChange(oldMode)` The argument `oldMode` refers to the component's mode before the mode change. In case the target mode is larger than the current mode, this method allows the component to increase its budget and to do any post processing after a mode change, e.g. update its internal data structures to match the new mode.

A mode change request is represented by the arrival of the QM task τ_q with the highest priority among all tasks. It uses the QM, which performs the following steps:

1 Wait for the currently executing subtask to complete.
2 Identify the components involved in the mode change, by comparing the current application mode and the target application mode.
3 Call `QM_BeforeModeChange()` in all components involved in the mode change.
4 Change the corresponding modes in the data structure keeping track of all current component modes and the application mode.
5 Call `QM_AfterModeChange()` in all components involved in the mode change.

A mode change will affect each component in one of three ways with respect to resource provisions: reduce, increase or keep them the same. The interesting case is when the target mode allocates fewer resources than the current mode and the component has to adapt its resource requirements by giving up some of its resources. Reducing the mode of a buffer component (implemented by `QM_BeforeModeChange()`) can be divided into two steps: scaling down its frames followed by reducing its budget size. There are several approaches to scaling down the frames.

According to assumption A5, the end-to-end latency of the complete task chain is bounded by MT. If we reduce the application mode, the M parameter is likely to change as well, since processing frames across the chain is likely to take less time. Let M_c and M_t be the M parameters in the current and target modes, respectively. Note that a change of M, e.g. a decrease from M_c to M_t, will *always* cause artifacts in the output. Such artifacts are inherent (and unavoidable) in an application mode change. Others can be avoided by selecting an appropriate mode-change protocol, as we propose in Section 6.2.

We continue by describing two different approaches to scaling down the frames throughout the chain. The first approach drops all buffered frames upon a mode change request, while the second approach applies the method presented above and drops only selected frames. For each approach we will analyze its mode change latency bound from the application and the system perspective (referred to as the *application latency* and the *system latency*).

6.1 Approach 1: drop all buffered frames

Instead of waiting for the renderer to read all the frames from the last buffer, we can:

1 Immediately drop all frames from all buffers.
2 Suspend τ_r and wait until the pipe is filled again before resuming it.

The first step avoids the delay in the previous approach, however, dropping frames means loosing the work invested in processing the dropped frames. Note that $(M_c + 1)T$ frames will be dropped.

The second step again suffers a delay of $(M_t + 1)T$ time units. Moreover, due to interdependencies between video frames (e.g. in MPEG a B frame depends on a P or an I frame), several of the first frames which arrive in step 2 may be useless, thus increasing the delay even further.

This approach speeds up the mode change with respect to other applications compared to the previous approach, by restricting the delays and lost work to the application reducing its mode.

6.2 Approach 2: drop only selected frames

In our approach we scale the two buffers b_e and b_d differently. We reduce the mode of b_e by dropping only selected frames, and reduce the mode of b_d by reducing the resolution of the frames already in the buffer.

1 Drop selected frames from b_e (and optionally b_d), with preference for the least significant ones.
2 Reduce the quality of the remaining frames in both buffers.

We can accomplish the first step using the `Buffer_GetNextBufferElement()`, `Buffer_ResetNextBufferElement()` and `Buffer_Drop(element)` methods as described in Section 5. Note that we need to drop exactly $M_c - M_t$ frames. We start with dropping selected frames from b_e (in case of MPEG the B and P frames)[1]. However, if we there are not enough frames in b_e to drop, then we proceed with dropping selected frames from b_d, until we have dropped $M_c - M_t$ frames.

In the worst case, selectively dropping the desired frames will require scanning through $M_c + 1$ frames in all the buffers. If iterating through a single frame and optionally dropping it costs C_{drop}, then the worst-case

[1]We assume here temporal video scaling. See [12] for other scaling methods, such as spatial or SNR scaling.

	System latency	Application latency
Approach 1	0	$(M_t + 1)T$
Approach 2	$(M_c + 1)(C_{drop} + C_{scale})$	$(M_c + 1)(C_{drop} + C_{scale})$

Table 1: Impact of different scaling approaches on the mode change latency with respect to the complete system and the application itself.

total cost for selectively dropping the desired frames will be $(M_c + 1)C_{drop}$ time units.

In the second step, the mechanism for reducing the quality of frames is likely to be different for different buffers. For example, in b_e we can drop the enhancement layers [12], while we in b_d we can reduce the resolution. Note that dropping enhancement layers assumes a particular video encoding and in general may not always be possible.

We can accomplish the second step using the standard push and pop interface of a buffer described in Section 4.2. Let n_d be the number of frames in b_d upon the mode change request. We cycle through all the n_d frames by popping the head frame from the b_d, reducing its resolution and pushing it back on b_d, until we have processed all the n_d frames. We can do the same for the frames in b_e[2]. In the worst case we will need to scale $M_c + 1$ frames in all the buffers. If C_{scale} is the worst-case time required to scale a frame in any of the buffers[3], then the second step will take $(M_c + 1)C_{scale}$ time units.

Both buffers will be able to reduce their budgets, allowing the system to reallocate the reclaimed memory, after $(M_c + 1)(C_{drop} + C_{scale})$ time units.

From the application perspective, since the scaling is executed at the highest priority, the renderer may suffer a delay of $(M_c + 1)(C_{drop} + C_{scale})$ time units.

Note that since we drop exactly $M_c - M_t$ frames we end up with M_t frames in the chain, hence we do not need to suspend τ_r to fill the pipeline (as it is done in the first two approaches). Also note, that if $M_c = M_t$, then we can skip step 1 altogether, reducing the system and application latency to $(M_c + 1)C_{scale}$.

Table 1 summarizes the impact of the different approaches on the system and the application latency. Note that both latencies define a time interval starting with the arrival of the mode change request.

In practice, iterating through a frame and scaling down the resolution of a frame are simple operations (relative to decoding) and so $C_{drop} + C_{scale}$ is very likely to be smaller than the frame period T.

6.3 Simulation results

Throughout this paper we have assumed a system comprised of several applications. During runtime the resources may be reallocated between the applications, referred to as a mode change. We simulate the complete system from the perspective of a single application, and represent the resource contention between applications by varying the target modes in mode changes.

We have implemented the application depicted in Figure 3 in the μC/OS-II real-time operating system [13]. The goal of guaranteed memory allocation was realized by having the RM use the memory partitions provided by the uC/OS-II operating system, where each partition manages the pool of blocks of the same size.To get consistent results we have run the simulation within the cycle accurate OpenRISC simulator inside Linux (Ubuntu 9.04)[4].

We ran the simulation with the following parameters:

- period of the network task τ_n and the renderer task τ_r, $T = 50ms$,
- computation time of the network task $C_n = 2ms$,
- computation time of the decoder task $C_d = 3ms$,
- computation time of the renderer $C_r = 3ms$,
- worst-case time needed to scale down a single frame $C_{scale} = 2ms$,
- period of the QM task $T_q = 410ms$,
- $M_c = M_t = M$,
- two application modes: high and low,
- each encoded frame in b_e occupies one buffer element,
- each decoded frame in b_d occupies one buffer element in the low mode and two buffer elements in the high mode.

The QM task would periodically request a mode change alternating between the high and low modes. We varied the computation time of the complete task chain by executing the decoder task for C_d most of the time, and once in MT having it fill the remaining capacity by executing it for an additional $MT - M(C_n + C_d + C_r)$.

The particular task parameters were selected based on data collected in a study of video frame sizes in MPEG encoding by [15][5]. The processing time of the decoder was assumed to be proportional to the frame size. We determined the ratio between the minimum and maximum computation times for the decoder by averaging over several data sets.

The simulations were run for 3300ms. The first invocation of the renderer task was offset with $T(M + 0.5)$ ms to fill the pipeline (according to [14]), resulting in around 60 frames for approach 2.

Figure 4 compares approaches 1 and 2, illustrating their application latency. The graph visualizes the interruptions in the rendered frames. A peak means that during one or more consecutive iterations of the renderer the b_e buffer was empty, forcing the renderer to repeat the last rendered frame. Therefore a peak means a larger interval between consecutive *different* rendered frames, possibly leading to video artifacts. Note that during the time interval represented by a peak no new frames are rendered, meaning that fewer different frames are rendered, explaining why the peaks for approach 2 are slightly shifted.

Figure 4 confirms the behavior claimed in Section 6 for $M = 2$: the large peaks of about 150 ms for approach 1

[2]If the enhancement layers are stored in separate buffer elements, then we can optimize this step by using the buffer interface for selective dropping.

[3]Note that dropping enhancement layers is likely to take less time than reducing the resolution of a frame, hence C_{scale} is pessimistic.

[4]The application source code, together with the simulator setup, is available at `http://www.win.tue.nl/~mholende/`

[5]Their collected data is available at `http://www.tkn.tu-berlin.de/research/trace/trace.html`

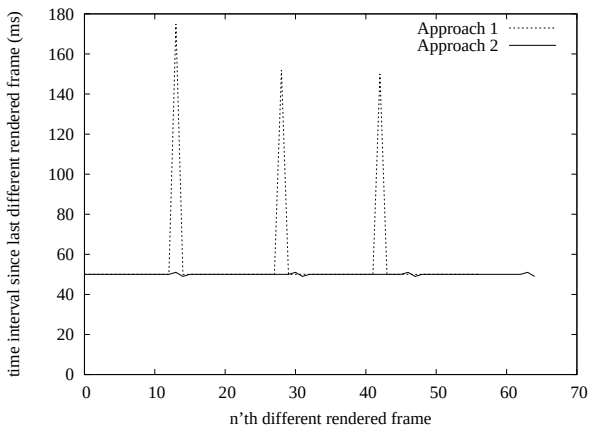

Figure 4: Simulation results, for $T = 50ms$, $C_n = 2ms$, $C_d = 3ms$, $C_r = 3ms$, $C_{scale} = 2ms$, $T_q = 410ms$, and $M = 2$

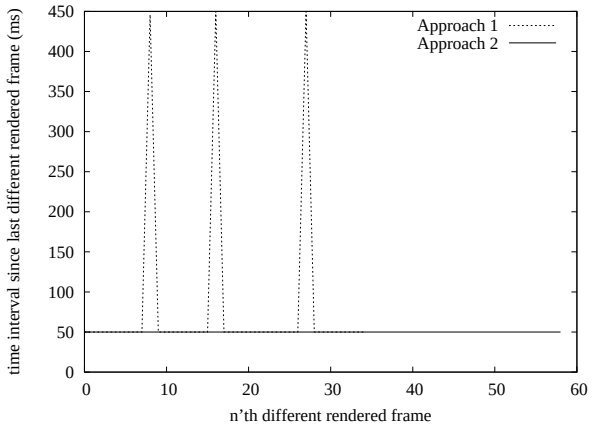

Figure 5: Simulation results, for $T = 50ms$, $C_n = 2ms$, $C_d = 3ms$, $C_r = 3ms$, $C_{scale} = 2ms$, $T_q = 410ms$, and $M = 8$

correspond to the waiting time of $(M_t + 1)T$. The small peaks for approach 2 correspond to the overhead for scaling the frames residing in b_d.

Figure 5 shows the simulation results for $M = 8$. As expected, the behavior is very similar, with larger application latencies for approach 1 due to the larger M_t resulting in a longer waiting time for filling the pipeline.

The lack of a bump in Figure 5 may be explained by most frames residing in the encoded buffer when a mode change occurred, due to $MT = T_q$. Since the overhead for dropping frames is negligible it does not show in the figure.

7 Conclusion

We have shown how additional access to the buffer, in particular the support for dropping arbitrary frames, can be used to guarantee the Quality of Service of the application during overload conditions,

We have also shown how a combination of two in-buffer scaling approaches: dropping selected frames in the buffer, and reducing the resolution of frames in the buffer by iterating through all its elements, can guarantee a low mode change latency from the system and application perspective. We have validated our analysis with simulation results.

In the future we would like to implement our approach on a real system to see how it impacts the *perceived quality* of the output video.

Acknowledgements

We would like to thank Ashu Gebreweld for the insightful discussions.

References

[1] C. Hentschel, R. Bril, Y. Chen, R. Braspenning, and T.-H. Lan, "Video quality-of-service for consumer terminals - a novel system for programmable components," *IEEE Transactions on Consumer Electronics*, vol. 49, no. 4, pp. 1367–1377, 2003.

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

[3] D. Brash, "The ARM architecture version 6 (ARMv6)," White Paper, 2002. [Online]. Available: www.arm.com/pdfs/ARMv6_Architecture.pdf

[4] R. McIlroy, P. Dickman, and J. Sventek, "Efficient dynamic heap allocation of scratch-pad memory," in *International Symposium on Memory Management (ISMM)*, 2008, pp. 31–40.

[5] M. Holenderski, R. J. Bril, and J. J. Lukkien, "Swift mode changes in memory constrained real-time systems," in *International Conference on Embedded and Ubiquitous Computing (EUC)*, 2009, pp. 262–269.

[6] ——, "Swift mode changes in memory constrained real-time systems," Eindhoven University of Technology, Tech. Rep. CS-09-08, 2009. [Online]. Available: http://w3.win.tue.nl/en/research/research_computer_science/

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

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

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

[10] D. Isovic and G. Fohler, "Quality aware mpeg-2 stream adaptation in resource constrained systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2004, pp. 23–32.

[11] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

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

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

[14] M. Weffers-Albu, "Behavioral analysis of real-time systems with interdependent tasks," Ph.D. dissertation, Technische Universiteit Eindhoven, April 2008.

[15] F. H. Fitzek and M. Reisslein, "MPEG4 and H.263 video traces for network performance evaluation," *IEEE Network*, vol. 15, no. 6, pp. 40–53, 2000.