Reducing Memory Requirements in a Multimedia Streaming Application

Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien, *Member*, IEEE

Abstract — *This paper investigates memory management for real-time multimedia applications running on resource-constrained electronic devices. The target applications are comprised of a data-driven task chain with a time-driven head and tail and a bounded end-to-end latency. The necessary buffer capacities along the task chain are derived. Subsequently it is shown how a shared memory pool can reduce the total memory requirements of the whole application. The impact of a shared memory pool is also evaluated in the context of scalable applications. The general technique targeted at memory-constrained streaming systems is demonstrated with a video encoding example, showing memory savings of about 19%.* [1]

Index Terms — memory management, multimedia systems, streaming applications, reducing memory requirements, buffer capacities, variable execution time.

I. Introduction

Multimedia applications are known to be data intensive. Many of these applications, especially in the consumer electronics domain, are implemented on resource-constrained embedded systems where the memory space is scarce [1], [2], [3]. Reducing the memory requirements of these applications is therefore very important.

We consider multimedia streaming applications which are implemented as a chain of data-driven tasks communicating via bounded buffers, with a time-driven (i.e. periodic) head and tail task. Fig. 1 shows an example of such an application.

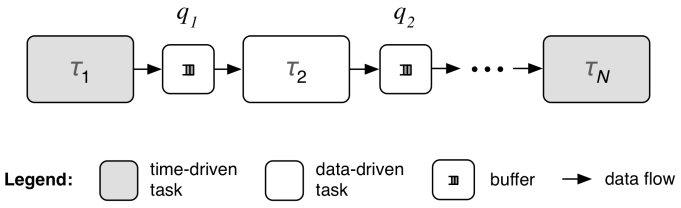

Fig. 1. A linear chain of media processing tasks communicating via shared buffers.

Task execution is determined by task priorities, data availability, buffer sizes and time triggering at the boundaries of the system. Let the first and last task in the chain be periodic with period T. The execution times of tasks may vary depending on the data they process, however, we assume that the end-to-end latency for processing a window of M consecutive frames is (strictly) bounded by MT.

In this paper we analyze the execution of a surveillance system, consisting of a video digitizer at the head, a video renderer at the tail, and a number of data-driven tasks. The data-driven tasks have the role of improving the video frames received from the video digitizer through additional processing.

We address the problem of how large the buffers must be and how we must choose the task priorities, in order to meet the real-time constraints.

Contributions

In Section IV we show that, in case of varying execution times, meeting the real-time constraints of the last task in the chain requires a particular priority assignment, and the first and last buffers in the chain to have capacity for M and $M+1$ frames, respectively, with all other buffers having capacity for 1 frame[2]. We also observe that in the above scenario the number of frames in transit never exceeds $M+1$.

In Section V we introduce the concept of a *shared memory pool*, which encapsulates the memory shared between all buffers in an application. We show how a shared memory pool in an application consisting of a chain of N tasks can save memory for storing $M+N-3$ frames, for $N \geq 3$. To be more precise, since at different stages of the task chain frames may have different sizes, we can save memory for storing $M+N-3$ *smallest* frames. Memory is managed in terms of fixed-sized blocks, which simplifies the reallocation of memory between buffers, allowing for an efficient implementation of a shared memory pool.

In Section VI we combine shared memory pools with our earlier work on scalable applications [4] and discuss how they affect the memory requirements of applications in different modes, and how they can reduce the mode change latency.

In Section VII we evaluate the memory savings in a real application and show experimental results for a H.264 video encoder.

II. Related Work

Optimizing memory usage in resource-constrained devices is ever so important with the increasing number of mobile and

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

Mike Holenderski, Reinder J. Bril, Johan J. Lukkien are with the Department of Mathematics and Computer Science, Eindhoven University of Technology, Den Dolech 2, 5600 AZ Eindhoven, The Netherlands (email: m.holenderski@tue.nl, r.j.bril@tue.nl, j.j.lukkien@tue.nl)

[2] In case of an MPEG-like video encoding, a video-processing task may need to store past frames for encoding or decoding a new frame. In this paper we ignore the internal memory requirements of each task, and focus on the memory required for communicating frames between tasks.

multimedia applications [1], [3], [5]. Yim et al. [1] present an approach for reducing the internal memory fragmentation in flash memory for mobile devices. Ahn et al. [3] consider memory-constrained portable media players and propose a memory allocation scheme for multimedia stream buffers, which allows reducing the number of page faults in heap and thus helps multimedia players perform with a consistent quality. Kim et al. [5] propose a region reuse technique, based on storing objects in upper local regions to the disk and reusing the reclaimed space for new object allocations, and hence reducing heap memory usage in mobile consumer devices with very limited memory. In contrast to [1], [3], [5] which try to manage the memory requirements dictated by the applications, in this paper we focus on actually reducing the memory requirements.

Albu [6] explores how the assignment of task priorities and buffer capacities impact the behavior of multimedia streaming applications comprised of a task chain. The author shows that a task chain with a time-driven tail exhibiting varying task execution times (where processing a window of M consecutive frames is bounded by MT) will meet its real-time constraints if the last buffer has capacity for M frames. They also show that a task chain with a time-driven head *and* tail, and $M = 2$, will meet its real-time constraints if all buffers have capacity 1. In this paper we derive the buffer capacities for a chain consisting of a time-driven head *and* tail and an *arbitrary* M.

Goddard 0 studies the real-time properties of PGM dataflow graphs, which closely resemble our media processing graphs. Given a periodic input and the dataflow attributes of the graph, exact node execution rates are determined for all nodes. The periodic tasks corresponding to each node are then scheduled using a preemptive Earliest Deadline First algorithm. For this implementation of the graph, the author shows how to bound the response time of the graph and the buffer requirements. However, it is limited to task sets with deadlines equal to the period, i.e. without self-interference. This approach provided valuable insights, but no rigorous support for analyzing and steering system behavior and associated resource needs has resulted.

Though we consider variations in execution time, we do not consider overload problems resulting from the inability to process the input in time. Approaches aiming at that situation can be used to prevent this from happening [7], [8], or to deal with it when it happens, through scaling or skipping the media content [9], [8], [10].

In [4] we investigated scalable applications, which can operate in one of several predefined modes, where each *mode* specifies the resource requirements in terms of M for all the buffers belonging to the application. We showed how to use in-buffer scaling to reduce the memory requirements of each buffer. Upon a mode change request for a mode with a smaller M and smaller memory requirements, we would drop the least significant frames from the chain and/or reduce the number of blocks for certain frames, and show how this could reduce the mode change latency. In this paper we concentrate on how a shared memory pool will affect the memory requirements of a scalable application in different modes.

III. System Model

Below we describe our application and platform models.

A. Application model

An application consists of a chain of N tasks $\tau_1,\ldots,\tau_N$ communicating via $N-1$ shared buffers. Each task τ_i is assigned a fixed and unique priority $P(\tau_i)$, with $P(\tau_i) < P(\tau_j)$ meaning that τ_i has a higher priority than τ_j. The first and the last tasks in the chain are time driven, with periods T_1 and T_N, phasings φ_1 and φ_N, and relative deadlines D_1 and D_N respectively. The time between activations and deadlines represent the times that the application may access the input frame buffer and the output frame buffer respectively. All other tasks in the chain are data driven.

Clearly, the periods of the head and tail tasks must be consistent with the frame counts since this system can only be expected to work if the inflow equals the outflow. In this paper we do not consider variations in the number of frames produced or consumed by a task nor any other data dependent behavior than varying computation times. Therefore we adopt

$$T_1 = T_N = T. \tag{1}$$

Let E_i^k be the execution time needed by task τ_i to process the k'th frame, and $E_i = \max_k E_i^k$ be the worst-case execution time of task τ_i on any frame. We use $\tau_{i..j}$ to denote a sub-chain of tasks τ_k, $i \le k \le j$. Let $E_{a..b}^k = \sum_{i=a}^{b} E_i^k$ be the execution time needed by chain $\tau_{i..j}$ to process the k'th frame.

Buffers represent FIFO queues, which are used to communicate data between tasks. They are responsible for the majority of memory requirements of an application. Each buffer q has a finite initial capacity $Cap(q)$, defining the maximum number of frames which can be stored in the buffer.

Each buffer q provides a read interface comprised of methods *read_acquire*(*q, frame*) and *read_release*(*q, frame*), and a write interface comprised of methods *write_acquire*(*q, frame*) and *write_release*(*q, frame*). These interfaces provide a hand-shake protocol allowing to do in-memory processing. A call of *read_acquire(q, frame)* will remove a full element from q and leave a reference to that element in *frame;* a call of *read_release(q, frame)* will add the element referred to by *frame* to q as an empty slot. The semantics of *write_release* and *write_acquire* is symmetric. The *acquire* operations are blocking if no full, resp. empty frames are available.

The pseudo code for a data-driven task is shown in Fig. 2. In each iteration, task τ_i reads a frame from q_{i-1} using *read_acquire*(q_{i-1}, *inFrame*) and retrieves a reference to a buffer slot inside q_i using *write_acquire*(q_i, *outFrame*). While processing the input frame from q_{i-1} it writes the

output frame to the slot it obtained from q_i. After it has finished processing the frame, it releases the input frame by calling *read_release*(q_{i-1}, *inFrame*) and marks the output frame as ready for reading by calling *write_release*(q_i, *outFrame*).

```
Task τ_i:
|[ var inFrame, outFrame: Frame reference;
   while (true) {
      write_acquire( q_i, outFrame).
      read_acquire( q_{i-1}, inFrame);
      process_i (inFrame, outFrame);
      read_release( q_{i-1}, inFrame);
      write_release( q_i, outFrame);
   }
]|
```

Fig. 2. Pseudo-code for a data-driven task.

The head task has no input buffer and the tail task has no output buffer**.** Moreover, the head and tail task are time driven, as outlined in Fig. 3 and Fig. 4. Note that a time-driven task can block on both communication and time.

```
global inputFrameBuffer: Frame reference;

Task τ_1:
|[ var outFrame: Frame reference;
        k: Integer;
   k := 0; delay_until( φ_1);
   while (true) {
      write_acquire(q_1, outFrame);
      process_1 (inputFrameBuffer, outFrame);
      write_release(q_1, outFrame);
      k := k + 1; delay_until( φ_1 + kT_1);
   }
]|
```

Fig. 3. Pseudo-code for a time-driven head task τ_1.

```
global displayBuffer: Frame reference;

Task τ_N:
|[ var inFrame: Frame reference;
        k: Integer;
   k := 0; delay_until( φ_N );
   while (true) {
      read_acquire(q_{N-1}, inFrame);
      process_N (inFrame, displayBuffer);
      read_release(q_{N-1}, inFrame);
      k := k + 1; delay_until( φ_N + kT_N );
   }
]|
```

Fig. 4. Pseudo-code for a time-driven tail task τ_N.

Note that *inputFrameBuffer* and *displayBuffer* represent references to memory which is external to the task chain. $process_1$ and $process_N$ will simply copy frames to and from the local buffers q_1 and q_{N-1}, respectively.

B. Real-time constraints

The application expresses its real-time requirements in terms of relative deadlines D_1 on task τ_1 and D_N on task τ_N, with $E_1 \le D_1 \le T$ and $E_N \le D_N \le T$. More precisely, we require that the k'th instance of tasks τ_1 and τ_N (processing the k'th frame) execute within time intervals $[\varphi_1 + kT, \varphi_1 + kT + D_1)$ and $[\varphi_N + kT, \varphi_N + kT + D_N)$, respectively.

C. Platform model

We assume a single processor. Furthermore, we assume that memory is managed in terms of fixed-sized blocks. Each frame may span across several memory blocks, but each block contains data belonging to at most one frame. At different stages of the task chain, frames may have different sizes (e.g. raw video frames are likely to be larger than the encoded frames).

A buffer expresses its memory requirements in terms of memory reservations (or memory budgets) [4], where each reservation guarantees access to the requested number of blocks. Memory reservations are granted only if there is enough space in the system-wide memory pool, and memory allocations are granted only if there is enough space within the corresponding reservation.

Several buffers may request memory from the same reservation, giving rise to a shared memory pool. The memory pool guarantees that the cumulative requirement of all buffers using it does not exceed its capacity.

IV. TIME-DRIVEN HEAD AND TAIL TASKS

Because of the equal periods of the head and tail tasks, and the assumption that each instance of a task consumes and produces exactly one frame (and the worst-case execution time for processing one frame is within T), there are a constant number of frames in transit within the system. Clearly, if the worst-case execution time for processing one frame is within T the system will satisfy the real-time constraints with just 1 frame in transit.

However, this strict restriction on the execution time of the complete chain is not realistic. We would like to allow the processing time for individual frames to take longer than T, as long as the processing of other frames will compensate for it.

A. Task priorities

When the computation time of a frame is larger than T, more that one frame will be in transit. Then, the priority assignment becomes important.

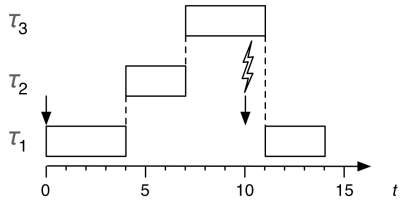

Fig. 5. Example of the τ_1 missing its deadline if it is assigned the lowest priority, for T = 10 and $D_1 = T$. After the head task has written the frame to the first buffer, the higher priority data-driven tasks preempt τ_1 which then has to wait until they complete.

Example 1: Consider the system of Fig. 5 with three tasks where the head task has the lowest priority. If the execution time of a frame exceeds $D_1 = T$ then the head task will miss the deadline of the next frame as the head task will not be scheduled before a computation has been completed. A similar remark holds for the tail task from which we conclude that the head and tail tasks should have a higher priority than the middle tasks.

We use S^k to denote the execution time needed to produce the k'th frame, after producing the $k-1$'th frame. We allow S^k to vary, but the duration of each size M window has to be smaller than MT, i.e.

$$\sum_{i=k-M+1}^{k} S^i < MT, \quad k \geq M. \tag{2}$$

Here M is a natural number, which can be regarded as a system parameter. The strict "smaller than" relation indicates that there will always be some idle time in the processing of any sequence of M frames.

Now, consider the system at time 0 when τ_1 starts and suppose τ_N does not execute. Then, according to (2), after MT time units, the last buffer will contain M frames. We start τ_N shortly after MT and choose $\varphi_N = MT + D_1$ (with $E_1 \leq D_1 \leq T - E_N$), making sure that, if we assign τ_N a lower priority than τ_1, τ_N will not be preempted by τ_1. From this point on the number of frames in transit within the chain is either M or $M+1$, as long as τ_1 does not block on output and τ_N does not block on input.

By the time $\varphi_N = MT + D_1$, the chain $\tau_{1..N-1}$ will have done work equal to

$$\sum_{i=1}^{M} E^i_{1..N-1} \quad . \tag{3}$$

During the time interval $[\varphi_N, \varphi_N + MT)$ the chain will produce M frames and will execute for

$$\sum_{i=1}^{M} S^i = \sum_{i=M+1}^{2M} E^i_{1..N-1} + \sum_{i=1}^{M} E^i_N \quad . \tag{4}$$

According to (2), the chain is guaranteed to reach idle state within MT time units. At that time all of the M or $M+1$ frames in transit will be residing in q_{N-1}. In general, processing a window of size M preceding the production of frame k, after frame $k-1$, will require

$$\begin{aligned} \sum_{i=k-M+1}^{k} S^i &= \sum_{i=k-2M+1}^{k-M} E^i_{1..N-1} + \sum_{i=k-M+1}^{k} E^k_N \\ &\leq M E_1 + \sum_{i=k-2M+1}^{k-M} E^i_{2..N-1} + M E_N \end{aligned} \quad . \tag{5}$$

Note that the contribution of $\tau_{1..N-1}$ and τ_N are of different iterations of the tasks (corresponding to different frames). Since the head and tail tasks in a multimedia streaming application will usually simply copy frames from or to an external buffer, we can bound their contribution from above and take their worst-case execution times E_N and E_N.

The previous example suggests that the head and tail tasks should be assigned priorities higher than any of the data-driven tasks in between. With these choices we can regard the system as consisting of three parts: high priority τ_1, low priority data-driven chain $\tau_{2..N-1}$ and high priority τ_N.

Let $P(\tau_1)$ be maximal, $P(\tau_2)$ be minimal, $P(\tau_N) = P(\tau_1) + 1$, and $P(\tau_{N-1}) = P(\tau_2) - 1$. Since tasks τ_1 and τ_N are time-driven and have priorities higher than any data-driven task in the chain, they can interrupt the execution of the chain at arbitrary places. However, since priorities $P(\tau_2)$ and $P(\tau_{N-1})$ are lower than the priorities of all other tasks in the chain $\tau_{2..N-1}$, tasks τ_1 and τ_N will not affect the execution order of actions in $\tau_{2..N-1}$.

B. Capacity of the first and last buffer

The following examples demonstrate how this particular priority assignment affects the required buffer capacities.

Example 2: Let us assume the above priority assignment and the following scenario. Task τ_1 writes a frame to q_1. Subsequently τ_2 reads it from q_1 and processes it. Let the frame be computationally intensive. Since $P(\tau_2)$ is minimal, the data-driven chain $\tau_{2..N-1}$ will process each new frame completely to q_{N-1} before τ_2 occurs again. According to (2) the data-driven chain may be busy with processing the frame for at most MT time units. During that time the time-driven τ_1, which has the highest priority, will have written M frames to q_1.

Example 3: Continuing with the last example, let us consider the capacity of the last buffer in the chain. Equation (2) implies that after MT time units the data-driven chain will process the frames in no time, since execution time for processing any window of size M is bounded by MT. Therefore, the data-driven chain $\tau_{2..N-1}$ will process the next M frames before the τ_1 writes another frame into q_1, essentially purging the first buffer. Since the head and tail task share the same period, these M frames will accumulate in q_{N-1}.

Assume that shortly after, the next frame which enters the chain is very easy to process, i.e. that $\tau_{2..N-1}$ will push this frame through to q_{N-1} before τ_N gets a chance to remove a frame from q_{N-1}. At this point q_{N-1} will contain $M+1$ frames. However, since tasks τ_1 and τ_N share the same period T, the following frame will not arrive before τ_N had the chance to remove a frame from q_{N-1}. The last buffer will therefore never exceed $M+1$ frames.

The previous two examples imply that the buffer capacities should be $Cap(q_1) = M$ and $Cap(q_{N-1}) = M+1$.

C. Capacity of the middle buffers

In the previous examples we made no assumption on the capacities the middle buffers, i.e. buffers q_i for $2 \le i \le N-2$. Now we show that all middle buffers can have capacity 1 by showing that $\tau_{2..N-1}$ will never become blocked on communication whenever there is work pending, or in other words, by showing that none of the tasks in $\tau_{2..N-1}$ will block on output.

Let us consider the situation at time $\varphi_N = MT + D_1$. During the initial MT time units, task τ_1 will have inserted M frames into the chain. According to (2), by the time φ_N the M frames have been pushed to the last buffer q_{N-1}. We need to show that none of the tasks in $\tau_{2..N-1}$ will block on a full buffer while there is still work pending. Assume that at time MT, i.e. before τ_N had a chance to remove a frame from q_{N-1}, a new frame arrives which is very easy to process. The chain $\tau_{2..N-1}$ will immediately push this frame through to q_{N-1}, which at this point will reach $M+1$ frames. However, since tasks τ_1 and τ_N share the same period T, the next frame will not arrive before τ_N had the chance to remove a frame from q_{N-1}. Since $Cap(q_{N-1}) = M+1$, task τ_{N-1} cannot become blocked on q_{N-1}. Moreover, since $P(\tau_2)$ is minimal, $\tau_{2..N-1}$ will process each new frame completely to q_{N-1} before τ_2 occurs again. Hence none of the buffers within the chain $\tau_{2..N-1}$ will exceed their capacity of 1, and consequently none of the tasks in $\tau_{2..N-1}$ will ever block when there is work pending.

D. Meeting real-time constraints

Since the head and tail tasks are activated periodically, if we show that τ_1 will never block on output and τ_N will never block on input, than we will have shown that the tasks τ_1 and τ_N will meet their real-time constraints. We demonstrate that this blocking of τ_1 or τ_N cannot occur, by showing that q_1 can never be full and q_{N-1} can never be empty. We show it by contraposition.

Assume that blocking of τ_1 or τ_N on communication does in fact occur and consider the first blocking of either of the two tasks at time *t*. Since we have assumed a single processor allowing only a single task executing at a time, the two tasks cannot block simultaneously.

This blocking task cannot be τ_1 since, as long as τ_N does not block, $\tau_{2..N}$ can either process or buffer the frames (given the buffer capacities derived in Sections IV.B and IV.C); hence, the contents of q_1 can vary up to M but will reach 0 within each MT time interval.

Then the blocking task must be τ_N, waiting for a new frame. However, as long as q_{N-1} contains less than M frames, the system is not idle as there are frames in transit; because of (2) the state with q_{N-1} containing M frames recurs within at most MT time units again and thus can never reach 0. It follows that no such first moment of blocking exists.

Now that we have shown that τ_1 does not block on output and τ_N does not block on input, we can state the following theorem.

Theorem 1: *Given a single application comprised of a task chain in* Fig. 1 *with a time-driven head and tail task, executing on a single processor, satisfying (2) and (5), and the following settings:*

- $P(\tau_1)$ *is maximal,* $P(\tau_2)$ *is minimal*
- $P(\tau_N) = P(\tau_1) + 1$, $P(\tau_{N-1}) = P(\tau_2) - 1$
- $\varphi_N = MT + D_1$
- $Cap(q_1) = M$, $Cap(q_{N-1}) = M+1$

the real-time constraints of tasks τ_1 *and* τ_N *will be satisfied, for* $3 \le N$, $E_1 \le D_1 \le T - E_N$, $E_N \le D_N \le T$, *and* $\sum_{i=k-M+1}^{k} S^i < MT$, $k \ge M$.

Note that since we made no assumptions on the sizes of buffers $q_i, 1 < i < N-1$, they can all have capacity 1.

In this section we have implicitly assumed a single application in the system. In case there are several applications running side by side, we have to increase the lower bound on deadlines D_1 and D_N, taking the interference of other applications into account.

V. Reducing Memory Requirements

From Theorem 1 we know that the total capacity of all buffers in an application consisting of a chain of $3 \le N$ tasks, as shown in Fig. 1, is equal to $M + (N-3) + (M+1)$ frames, where M represents the first buffer, $(N-3)$ represents the buffers with capacity 1 between the first and last buffer, and $(M+1)$ represents the last buffer. However, as mentioned in Section IV.A, the total number of frames in transit never exceeds $M+1$ frames.

Rather than allocating each buffer its required capacity, we can have them share a common memory pool, since all buffers together will never require more memory than for storing $M+1$ frames. In this way can save the memory for storing $M+N-3$ frames, for $3 \le N$.

At different stages of the task chain frames may have different sizes. Let s_i be the frame requirement for a single

frame at stage i, i.e the size (in terms of blocks) of the largest frame ever stored in the i'th buffer in the chain. A chain of N tasks defines a collection of frame requirements:

$$\{\underbrace{s_1,\ldots,s_1}_{M},\underbrace{s_2,s_3,\ldots,s_{n-2}}_{N-3},\underbrace{s_{n-1},\ldots,s_{n-1}}_{M+1}\}\ .$$

If we order the frame requirements in the application in ascending order of s_i, then using a shared memory pool can save the memory space required by the $M+N-3$ smallest frame requirements. More precisely, if we arrange the $M+(N-3)+(M+1)=2M+N-2$ frame requirements in ascending order in a sequence

$$\langle r_1, r_2, \ldots, r_{2M+N-2}\rangle\ , \tag{6}$$

such that

$$\forall i : 1 \le i \le 2M+N-2 : r_i \le r_{i+1}\ , \tag{7}$$

then the *absolute* memory savings are given by

$$\sum_{i=1}^{M+N-3} r_i\ , \tag{8}$$

and the *relative* memory savings are given by

$$\frac{\sum_{i=1}^{M+N-3} r_i}{\sum_{i=1}^{2M+N-2} r_i}\ . \tag{9}$$

The memory reservations based on fixed-sized blocks simplify the reallocation of memory between buffers, allowing for an efficient implementation of a shared memory pool.

VI. Mode Changes in Streaming Applications

In this section we investigate reallocating memory between applications. Let us continue with the multimedia processing application example shown in Fig. 1 and consider a system comprised of two such applications. In the new setting both applications are scalable, meaning that when they are provided more resources they can generate higher quality output. More specifically, given more processing time and more memory, each application will generate a higher quality video encoding. Higher quality video will require more memory for storing the buffered frames, thus increasing the s_i values. Also, longer processing time will result in greater fluctuations of the latency of the processing chain and consequently larger M, requiring larger buffers along the processing chain.

A scalable application can operate in one of several predefined modes, where each *mode* specifies the resource requirements in terms of M and s_i for all the buffers belonging to the application. In a system comprised of two such applications executing on a memory constrained platform, if we assume that the available memory cannot accommodate both applications at their highest quality at the same time, we may need to reallocate the memory between the applications during runtime to provide a system wide Quality of Service. We refer to such a resource reallocation as a *mode change*.

A mode change exhibits a *mode change latency*, defined as the length of the time interval between a *mode change request* and the time when all resources have been reallocated. In [4] we showed how to use in-buffer scaling to reduce the memory requirements of each buffer. In this section we concentrate on how a shared memory pool will affect the memory requirements of a scalable application in different modes.

Equation (9) indicates that the relative memory savings due to a shared memory pool increase with increasing M, as visualized in Fig. 7.

A higher quality mode (i.e. a mode requiring more resources, in particular more processor time) is likely to exhibit larger variation in execution time of the individual tasks, and hence also of the complete chain, thus increasing M (provided it is not assigned a larger processor share after the mode change).

Also, (8) and (9) indicate that the smaller the variation in s_i the larger the memory savings due to a shared memory pool. Many multimedia streaming applications can be classified as encoders or decoders. An encoder receives a fixed-sized input (e.g. raw video frames from a camera) and encodes it into an output, of which the size depends on the quality settings. Conversely, a decoder will receive a variable sized input and generate a fixed-sized output (e.g. decoded frames rendered to a screen). Given the fixed-sized output or input frames, an application operating in a lower quality mode is likely to exhibit large variation in the frame sizes at different stages of the chain.

In summary, scalable applications are likely to exhibit larger relative memory savings due to a shared memory pool when they execute in higher modes.

Note that if the memory reservations underlying the memory pools of different applications are managed in terms of fixed-sized blocks of the same size, then the reallocation of memory between memory pools belonging to different applications will be simpler and more efficient. Memory blocks can be simply added and removed from a list of available blocks, compared to a solution based on finding the best fit between memory requirements and available blocks. Moreover, rather than scaling the memory reservations for all buffers one by one, a shared memory pool allows to scale the memory requirements of the complete chain in a single step[3]. Thus a shared memory pool, next to reducing the memory requirements of a scalable application, can also reduce the mode change latency.

VII. Results

Fig. 6 shows an application example of a H.264 video encoder [11], which is commonly used in the consumer electronics domain. We used it to evaluate the memory savings in a real application.

[3] The data residing in the buffers still needs to be scaled for each buffer individually.

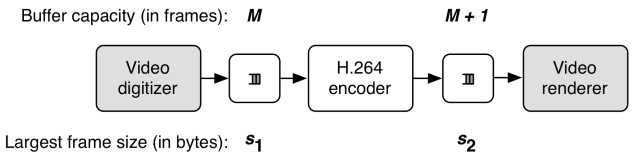

Fig. 6. A video encoding application.

The application consists of three tasks: the video digitizer provides raw frames in CIF format (with resolution 352x288 pixels) for the H.264 video encoder, which produces a 300kbs video stream with the same resolution for the video renderer. The s_1 parameter is equal to the size of a raw input frame, i.e $s_1 = 352*288 = 101376$ bytes. We have measured the largest frame ever produced by the H.264 encoder for a series of video sequences[4] to be $s_2 = 26002$ bytes. By using a shared memory pool, in our chain of $N = 3$ tasks we can save the memory for storing $M + N - 3 = M$ of the smaller frames, which in this case are the encoded frames of size s_2. The relative memory savings are therefore given by

$$\frac{M * s_2}{M * s_1 + (M+1)s_2} \quad . \tag{10}$$

Fig. 7 shows the relative memory savings of our approach as a function of M, by filling in the above values for s_1 and s_2 in (10).

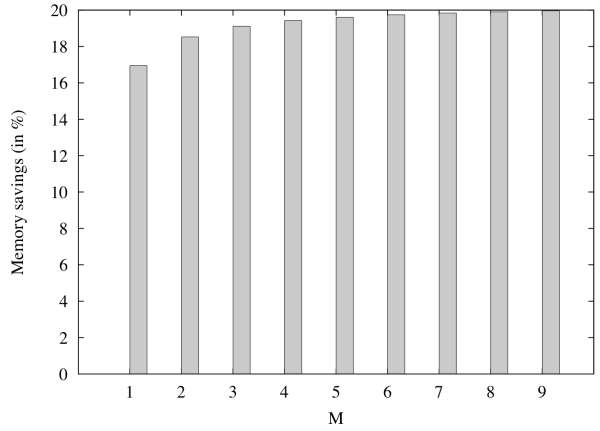

Fig. 7. Memory savings in our example application as a function of *M*.

Note that video scaling algorithms [4], [8], [10] can be applied to guarantee that the M parameter holds, i.e. that the processing of any sequence of M frames does not exceed MT.

In our video encoder application the raw frames were 4 times larger than the encoded frames. In general, the smaller the difference between the frame requirements at different stages, the larger the memory savings, as shown in Fig. 8.

[4] Available at http://media.xiph.org/video/derf/

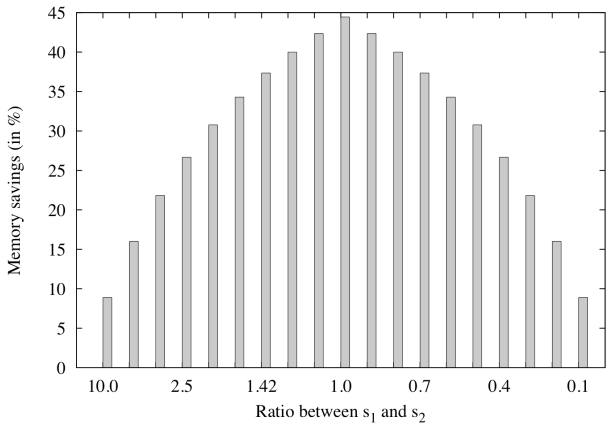

Fig. 8. Memory savings for different ratios between s_1 and s_2, assuming $N = 3$ and $M = 4$.

VIII. CONCLUSIONS

We have shown a general mechanism for reducing memory requirements in a streaming application comprised of a chain of tasks with periodic head and tail tasks communicating via shared buffers. The proposed method is based on having the buffers share a common memory pool. we have shown that the total capacity of all buffers in an application consisting of a chain of $3 \le N$ tasks is equal to $M + (N-3) + (M+1)$ frames We exploit the fact that in the above scenario the total number of frames in transit never exceeds $M+1$, and propose to share a memory pool with capacity for $M+1$ frames between all the buffers. As a result, in an application consisting of a chain of N tasks, we can save memory for storing $M+N-3$ frames. To be more precise, since at different stages of the task chain frames may have different sizes, we can save memory for storing $M+N-3$ *smallest* frames.

Managing the memory in terms of fixed-sized blocks will simplify the reallocation of memory between buffers, allowing for an efficient implementation of a shared memory pool. If applied to scalable applications, a shared memory pool will result in greater relative memory savings for applications operating in higher modes.

The results for an H.264 encoder show memory savings of around 19%. The approach is targeted at resource-constrained systems, such as those found in consumer electronics.

ACKNOWLEDGMENT

We would like to thank Martijn van den Heuvel for his insightful comments on an earlier version of this document.

REFERENCES

[1] K. S. Yim, H. Bahn, K. Koh. "A flash compression layer for smartmedia card systems," *IEEE Transactions on Consumer Electronics*, vol. 50(1), pp. 192–197, 2004.

[2] F. Menichelli and M. Olivieri, "Static minimization of total energy consumption in memory subsystem for scratchpad-based systems-on-chips," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 2, pp. 161–171, 2009.

[3] H. Ahn, S. Cho, H. Na, H. Han. "Access pattern based stream buffer management scheme for portable media players," *IEEE Transactions on Consumer Electronics*, vol. 55(3), pp. 1522–1529, 2009.

[4] M. Holenderski, C. G. Okwudire, R. J. Bril, and J. J. Lukkien, “Memory management for multimedia quality of service in resource constrained embedded systems,” in Proc. *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, 2010.

[5] S. Kim, T. Kim, E. G. Im, H. Han. "Efficient reuse of local regions in memory-limited mobile devices," *IEEE Transactions on Consumer Electronics*, vol. 56(3), pp. 1297–1303, 2010.

[6] M. Albu, “Behavioral analysis of real-time systems with interdependent tasks,” *Ph.D. dissertation*, Technische Universiteit Eindhoven, 2008.
S. Goddard, “Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application,” in *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS)*, pp. 60–71, 1997.

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

[8] 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.

[9] T.H. Lan, Y. Chen, and Z. Zhong, “MPEG-2 decoding complexity regulation for a media processor,” in *Proc. IEEE Workshop on Multimedia and Signal Processing (MMSP)*, pp. 193–198, 2001.

[10] D. Jarnikov, P. van der Stok, and C. Wüst, “Predictive control of video quality under fluctuating bandwidth conditions,” in *Proc. IEEE International Conference on Multimedia and Expo (ICME)*, Vol. 2, pp. 1051–1054, 2004.

[11] T. Wiegand, G. Sullivan, G. Bjontegaard, A. Luthra. “Overview of the h.264/avc video coding standard,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13(7): pp. 560–576, 2003.

BIOGRAPHIES

Mike Holenderski is a Ph.D. student at the Eindhoven University of Technology, the Netherlands. He received his B.Sc. in 2003 and M.Sc. (with honors) in 2007, both from the Eindhoven University of Technology. His main research interests are in the area of reservation-based multi-resource scheduling in embedded real-time systems.

Reinder J. Bril received a B.Sc. and an M.Sc. (both with honors) from the University of Twente, and a Ph.D. from the Technische Universiteit Eindhoven, the Netherlands. He started his professional career in January 1984 at the Delft University of Technology. From May 1985 till August 2004, he has been with Philips, and worked in both Philips Research as well as Philips’ Business Units. He worked on various topics, including fault tolerance, formal specifications, software architecture analysis, and dynamic resource management, and in different application domains, e.g. high-volume electronics consumer products and (low volume) professional systems. In September 2004, he made a transfer back to the academic world, i.e. to the System Architecture and Networking (SAN) group of the Mathematics and Computer Science department of the Technische Universiteit Eindhoven. His main research interests are currently in the area of reservation-based resource management for networked embedded systems with real- time constraints.

Johan Lukkien is head of the System Architecture and Networking Research group at Eindhoven University of Technology since 2002. He received M.Sc. and Ph.D. from Groningen University in the Netherlands. In 1991 he joined Eindhoven University after a two years leave at the California Institute of Technology. His research interests include the design and performance analysis of parallel and distributed systems. Until 2000 he was involved in large-scale simulations in physics and chemistry. Since 2000, his research focus has shifted to the application domain of networked resource-constrained embedded systems. Contributions of the SAN group are in the area of component-based middleware for resource-constrained devices, distributed coordination, Quality of Service in networked systems and schedulability analysis in real-time systems.