Optimal Resource Sharing Strategies for a Streaming System

Abhinandan Kedlaya
Ittiam Systems (P) Ltd.
Consulate 1,
Richmond road
Bangalore, India
Tel: +91-80-66601159
abhhinandan.kedlaya@ittiam.com

Ranjith KA
Ittiam Systems (P) Ltd.
Consulate -1
Richmond road,
Bangalore, India
Tel: +91-80-66601179
ranjith.ka@ittiam.com

Abstract: Embedded streaming applications are continually tending towards convergence with several functions integrated into each multimedia-enabled device. The shrinking sizes of the devices and the increasing demands from the user pose a complex resource management challenge to the designers of embedded applications. The key lies in leveraging the available processing power and resources to the fullest. The challenge is to achieve this without compromising on the viewing experience. Streaming applications such as set-top boxes today feature video conferencing, interactive gaming capabilities in addition to the traditional Video on Demand features. Network surveillance applications are designed to capture high-resolution images at regular intervals over and above the encode-stream functionality.

This paper primarily focuses on addressing the resource sharing challenges posed by the new genre of streaming applications on embedded platforms. It introduces the key challenges with respect to resource management imposed by each of these streaming applications. The typical solutions adopted to meet the challenges and their impact there of are discussed. Advanced solutions which are a combination of a few typical solutions but provide significant flexibility are also discussed. We conclude by summarizing the resource optimization techniques and thoughts for future work.

Index Terms: Multi-format, Multi-instance, Memory bandwidth, Overlay, Scheduling.

I. INTRODUCTION

Streaming applications on embedded platforms or otherwise are intensive with respect to their requirements of resources such as processor time, memory for code (.i.e program memory) and data, memory bandwidth, hardware accelerators to carry out commonly used signal processing computations. Embedded platforms come with a limited supply of these resources vis-à-vis a general-purpose platform such as PC. Although it helps keep the hardware costs low, it leaves behind the tough challenge of meeting the ever-increasing user demands by efficient management of the available resources.

Most of the embedded platforms employ lightweight Real Time Operating Systems (RTOS), which do not provide sophisticated resource management algorithms. Thus, the onus for the efficient usage of the resources lies with the appli cation operating on the embedded platform.

The resource usage is largely governed by the requirements of the embedded application. This paper focuses on multimedia streaming applications operating on embedded platforms, which can be broadly classified into the following categories

- Surveillance applications used in public and private establishments for security and monitoring purposes.
- Video Conferencing applications used in low latency video telephony over IP networks.
- Multimedia content distribution applications used in interactive streaming systems such as Video on Demand, in flight and in car entertainment, IPTV etc.

Surveillance applications usually consist of a server, which takes-in multiple camera

feeds, performs media compression on each of them and transmits it to a remote monitoring station. This is typically accompanied by remote PTZ control of individual cameras. This might require multiple instances of compression algorithms to be instantiated on the server side. Algorithms such as object tracking etc might be needed in high-end surveillance applications. The above requirements translate to a key challenge for embedded applications, which is, enabling multi-instance capability i.e. the application should support multiple instances of an algorithm to operate simultaneously. The server might have to vary the bitrate depending on the prevailing network conditions. In such a scenario, the server application will have to re-configure encoders to encode at the new bitrate and schedule the network transmission accordingly to transmit at a CBR without overflowing or under flowing the decoder buffers. This necessitates a dynamic scheduling algorithm, which efficiently schedules the various components/tasks.

Low latency video conferencing applications require simultaneous encoding and decoding to be performed at each individual device. The encoded media data is packetized and streamed across to the peers. Simultaneously, the packetized media data from the peers is aggregated and decoded. With multi-way conferencing capability, these applications enable us to connect to more than one peer seamlessly. The challenge in such applications is to enable components of different nature (encoder/decoder) to operate simultaneously in addition to the multi-instance capability.

Multimedia content distribution applications meant for personal entertainment are primarily designed to provide a rich multimedia experience to the viewers. Examples of such applications are In-Flight Entertainment solutions and IPTV. The feature set on offer includes VoD, VCR functionalities of pause, play, fast-forward, fast-rewind, seek to time. Additionally the user should be allowed to choose a desired program/channel at the client side. Advanced features such as Picture-in-Picture (PIP) allow the viewer to watch programs from two different channels on the same display screen. Due to the numerous formats of the media content, the media clients should be multi-format enabled. In order to provide a rich multimedia experience, the application might have to operate post-processing algorithms, which add to the memory bandwidth requirement and processing complexity. These algorithms could be activated/de-activated by the user based on the current viewing experience. Thus, an efficient management of the available memory bandwidth is a must.

Summarized below is the list of the challenges introduced above.

- Multi-Instance capability: Enabling simultaneous operation of multiple instances of a given component.
- Multi-format capability: Enabling simultaneous operation of multiple components each of different types.
- A dynamic scheduling algorithm.
- An algorithm for the efficient management of the available memory bandwidth.

II. Resource Challenges

This section translates the challenges listed in the introduction to a set of implementation challenges. Introduced below are a few key resources that would need efficient management:

- *On-Chip Memory* The memory available on the processor chip. Accessing this memory is faster than other types of memories. Some processors offer two levels of On-Chip memory along with flexibility of configuring a part of it as cache. The first level usually consists separate memories for program and data. Due to the negligible memory access time for this type of memory, it is desirable to have the components in this memory.
- *Off-Chip Memory* The memory provided off the processor chip as a part of the embedded platform. The performance of the components will be comprised if they are placed in off-chip memory. Caching, if enabled, prevents repeated Off-chip memory accesses

and helps improve the performance of the component.

- *DMA controller* Most processors designed for multi media applications come with a DMA controller to handle the huge memory bandwidth requirements involved in media processing algorithms.
- *Processor time* The processor is the primary resource in an embedded platform. Scheduling the various media processing algorithms on the fly based on the dynamic priorities of the tasks is a key care about in the design of embedded applications.

Following are the expectations from the components in order to optimize resource allocation.

- Components should be sufficiently super real-time so as to support the number of instances desired.
- Components should route all its resource requirements (memory/EDMA/accelerators) through the overlying application.
- The memory requests made by the components should
  - Indicate the priority of the memory requested. (Which would be used to map it to either On-chip or Off-chip memory).
  - Indicate the nature of usage of memory. (Either read-write scratch, read-write persistent, read-only etc)
- The component builds should segregate critical sections of the code into a common lump so that it can be loaded in On-Chip memory and export the code section to the application.

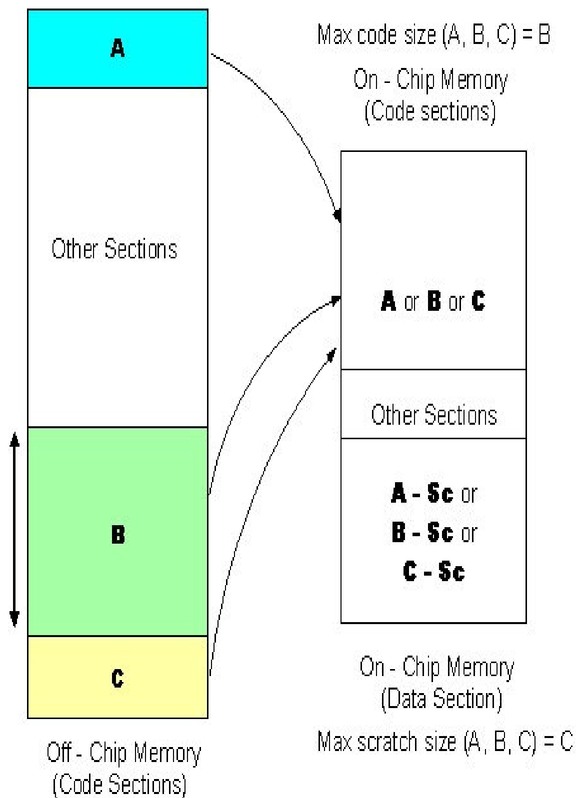
III. Managing Overlay

The cost implication of increasing the On-Chip memory is huge whereas, it is relatively easy to increase the Off-chip memory. To enable multi-instance and multi format capability not all components can be loaded simultaneously in the On-Chip memory, some of them will have to be placed in the off-chip memory. But this will have a significant impact on the performance. To prevent performance degradation of components arising out of this, a technique called overlay is adopted. This involves, dynamically downloading a component from off-chip to on-chip memory as and when required.

Overlay techniques enable us to share the scarce resources in a system by having different load time and run time placement of components. Component Memory requirements can be classified into three types, namely

- *Code or Program memory* refers to the memory occupied by the text section of the executable. This can be further classified into
  - *On-chip Code memory* contains the sections of code that get executed very frequently and thus need to be easily and quickly accessible to the processor.
  - *Off-chip Code memory* contains the sections of code that get executed not so frequently and thus can afford to be in the Off-chip memory.
- *Persistent Data memory* refers to the data memory buffers/handles that need to be retained right from the creation till the destruction of the component. Components might require both On-chip and Off-chip Persistent memories.
- *Scratch Data memory* refers to the data memory that the component requires only during the process function. The scratch buffers would be used for the temporary data buffers required during individual process calls and need not be maintained across calls to the component. Components might require both On-chip and Off-chip scratch memories.

Assuming that sufficient Off-chip memory is available to hold all the component code and persistent data, the On-chip memory can be multiplexed between the various components as shown below.

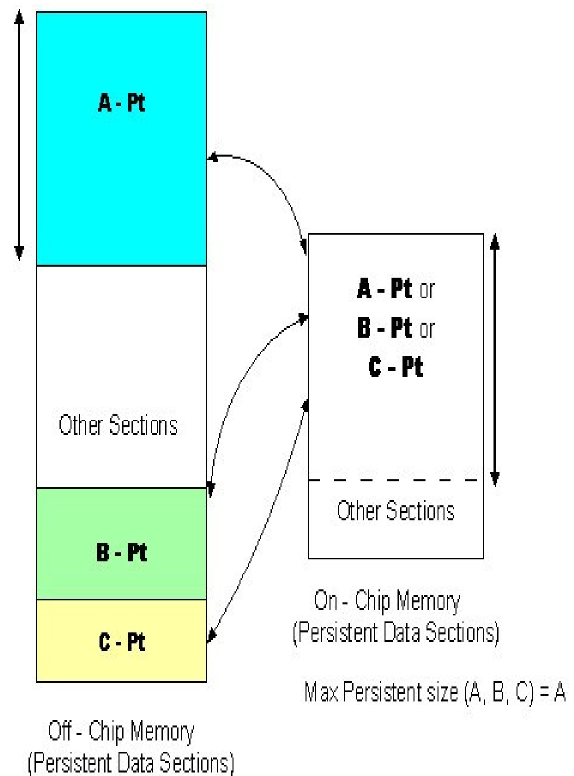


CODE AND SCRATCH OVERLAY

Figure 1 Illustration of Code and scratch Overlay

The On-chip code memory sections of three hypothetical components A, B and C are placed in Off-chip memory at load time. A single On-chip memory chunk with a size equal to the maximum of the three sections is created as shown above. Thus reducing the On-chip code memory requirement to MAX (A, B, C) vis-à-vis SUM (A+B+C) in a typical system. However, this necessitates an additional code memory download from Off-chip to On-chip memory for each of the components.

As the content of the scratch memory buffers of a component need not be retained across process function calls, it can be reused across the components. Thus, single chunk of On-Chip scratch memory with size equal to the maximum of the individual scratch memory requirements is allocated as shown in the figure above. This facilitates optimal On-chip memory usage without any additional overhead.



PERSISTENT OVERLAY

Figure 2 Illustration of State Overlay

Although similar in nature to code overlay in terms of memory re-use, and optimality, overlay of persistent memory requires that the persistent memory section in the On-chip memory be written back to its Off-chip counterpart for subsequent accesses. Unless the On-chip persistent memory requirements of media components are large, the write back overhead in this method is insignificant.

Code, scratch and persistent On-chip memory overlay can individually provide efficient usage of On-chip memory. However, it would be unfair to assume that the above memory requirements will be comparable across all components. Illustrated below is one such scenario where the code, scratch and state memory requirements for the components under consideration are different. The techniques described above applied individually would require an On-Chip memory size equal to

{MAX (A, B, C) + MAX (A-Sc, B-Sc, C-Sc) + MAX (A-Pt, B-Pt, C-Pt)}. As illustrated by (d) in figure 3

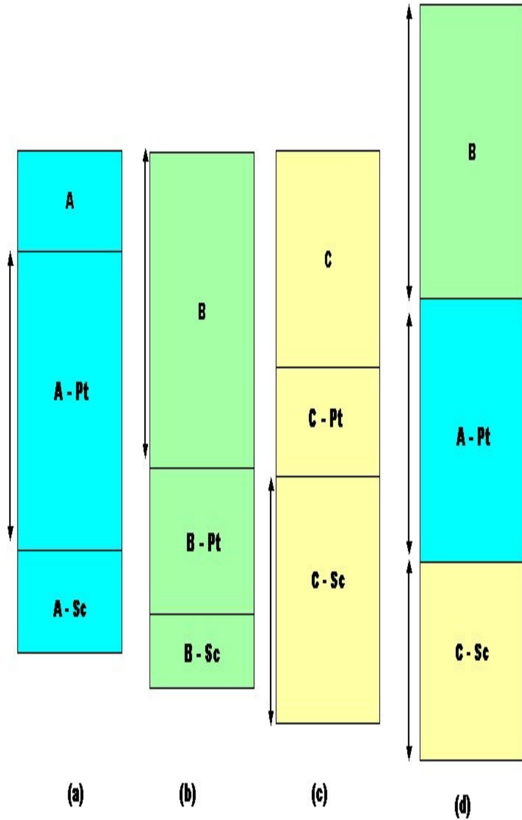


Figure 3 Illustration of Code-Scratch and Persistent Overlay.

However, on closer examination it can be found that by overlaying the On-chip memory requirements of the components taken as a set rather than individually, makes an efficient overlay design. The On-chip requirement now would be {MAX (SUM (A, A-Pt, A-Sc), SUM (B, B-Pt, B-Sc), SUM (C, C-Pt, C-Sc))}.

IV. Managing DMA

The techniques that a system designer can employ to share the available memory bandwidth across multiple algorithms is dependent on the offerings of the platform. If a platform does not support Direct Memory Access (DMA), the application designer has to depend on the transferring data using processor requests. The memory bandwidth utilization can be improved if individual algorithms integrate the memory sections that are required to be copied and use block memory transfers. This burst transfer is faster than transferring a few bytes at a time, taking into account the address decoding mechanism and other hardware dependencies involved in a memory access.

Most of the present day platforms support DMA and few of them support advanced or enhanced versions of the DMA. These advanced versions come with features such as multiple DMA channels for memory transfers that can be triggered wither by external events or by events raised by the CPU. Auto triggering mechanisms where the events are generated by completion of a different transfer are also supported. Maximum throughput can be achieved from a DMA process if the addresses of source and destination memory location is aligned/integral multiple of the data bus width of DMA in bytes. When a transfer request is placed with the DMA controller, CPU has to trigger the transfer process on the need basis. The advanced versions of the DMAs have auto triggering feature which triggers the transfer process once another pre-defined transfer is completed. This minimizes the interrupt servicing overheads which otherwise would have been inevitable in the CPU triggering mechanism. Multiple channels enable the servicing of multiple transfer requests in parallel. The transfers can also be prioritized thereby giving the application designer the flexibility to prioritize/schedule the time critical memory transfers over the other transfer requests. In real time streaming systems, it is common to come across few critical memory transfers like transferring the data from serial port registers into the off - chip memory. These transfers are time critical (have to be serviced before a pre-defined deadline expires) and have to be given a high priority.

V. Scheduling

Scheduling or assigning the CPU time for various tasks plays deciding role in any real time streaming systems. The scheduling of the CPU becomes even more critical when

the peak processing conditions occur. In such a condition, the scheduler algorithm should be intelligent to take a decision on scheduling the most sensitive task specific to the application.

In most of the embedded systems, various tasks are dependent on the input/output from the external/real world. This requires those specific tasks to be completed within the time deadline. The impact of few tasks not being scheduled within the time deadline might be more noticeable to the human perception than any other task. Few tasks are computationally intensive and consume much of the CPU time. The time critical tasks will be scheduled or assigned the CPU time even if CPU is involved in executing the computationally intensive or non - critical task. For an illustration, human perception is more sensitive to audio losses than the visual losses. Hence in any multimedia playback system or in the streaming system, the time critical audio processing task is prioritized over the computationally intensive video task.

The scheduling mechanism of the tasks mainly depends on the operating system environment around the application.

VI. Conclusion

In a nutshell, the streaming applications on embedded platforms or otherwise are intensive with respect to their requirements of resources such as processor time, memory, memory bandwidth, hardware accelerators to carry out commonly used signal processing computations. The key challenges involved in the resource management are the effective usage of available memory, optimizing the memory bandwidth and scheduling the various tasks on the processor.

By recognizing the pattern in the memory requirements of various components and depending on the component which has to be activated, the code, persistent and the scratch memory is overlayed. However, this requires individual components to adhere to certain interface standard through which they can notify the application about their memory requirements.

Optimizing the memory bandwidth requires scheduling the individual memory transfers according to their criticality with respect to time and the size of the memory to be transferred.

Scheduling of the tasks can be done statically or dynamically depending on the requirements of the application, In order to address the peak conditions, the scheduling algorithm has to be tuned to the application.

Copyrights: “Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific permission.

GSPx 2006. October 30-November 2, 2006. Santa Clara, CA. Copyright 2006 Global Technology Conferences, Inc.”