Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 74

ALEXANDER MAKSYAGIN

# Modeling Multimedia Workloads for Embedded System Design

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology (ETH) Zürich
for the degree of Doctor of Sciences

Diss. ETH No. 16285

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Petru Eles, co-examiner
Examination date: October 13, 2005

Diss. ETH No. 16285

# Modeling Multimedia Workloads for Embedded System Design

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZÜRICH

for the degree of

Doctor of Sciences

presented by

ALEXANDER MAKSYAGIN
Dipl. Radio-Eng. MTUCI, Russia

born 15.03.1973
citizen of Russia

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Petru Eles, co-examiner

2005

# Abstract

To design a successful computer system, designers need to know characteristics of the computational workload that this system is supposed to process. This knowledge forms the necessary basis for optimizations of the system. In order to use this knowledge in the design process, designers need to characterize the workload using a formal workload model. This model represents an abstraction of the concrete workload and serves as an input to a number of critical design tasks, such as system performance analysis. The quality of the workload model largely determines the quality of the design decisions made based on it.

Coming up with a proper workload model represents a difficult problem in many computer system design contexts. One such context, addressed in this thesis, is system-level design of embedded computers whose main functionality involves real-time processing of media streams (e.g. streams of audio-video data). Of late, there is a growing demand for such computers because they are increasingly being embedded into many electronic products, especially those found in consumer electronics domain, e.g., digital TVs, audio and video players, digital video cameras, advanced set-top boxes, multimedia-enabled mobile phones and a myriad of other electronic devices supporting multimedia applications. To meet high performance requirements and stringent constraints pertaining to cost, size and energy consumption, these embedded computers tend to have complex, heterogeneous multiprocessor architectures. This architectural complexity, coupled with the ever-growing complexity of the multimedia applications themselves, results in a very complex workload behavior and by that poses many challenges to the workload modeling.

In this thesis, we argue that the variability of various parameters of the multimedia workloads is the key property to be captured in a workload model for the embedded systems design. We show that conventional workload models fail to accurately characterize the dynamic nature of the multimedia workloads and, as a result, return overly pessimistic estimations of system performance (especially, if worst-case performance bounds are of interest). As a solution, we propose a novel workload model capable of accurately capturing the workload's dynamic nature. We demonstrate the advantages of the proposed workload model over conventional ways to characterize the workload and develop a number of system-level design methods which use this model. These methods include system-level performance analysis, automatic identification of representative workload scenarios for system simulation, design and optimization of resource management policies and a run-time processor rate adaptation strategy for energy-efficient processing of media streams on heterogeneous multiprocessor embedded architectures with stringent memory constraints. We demonstrate the utility of our workload model and evaluate it through a number of case studies involving comparisons to detailed simulation models.

# Zusammenfassung

Um ein erfolgreiches Computersystem zu entwerfen, müssen die Entwickler die Rechenanforderungen für das System kennen. Daher ist es notwendig diese Anforderungen mittels eines formalen Auslastungsmodells zu charakterisieren. Dieses Modell repräsentiert eine Abstraktion der konkreten Rechenauslastung und dient beim Entwurf als Eingabe für verschiedene kritische Entwurfsaufgaben. Die Qualität des Auslastungsmodells wirkt sich hierbei direkt auf die Qualität der hierauf basierenden Entwurfsentscheidungen aus.

Oftmals ist es schwierig, ein geeignetes Modell für die Auslastung von Computersystemen in verschiedenen Einsatzgebieten zu finden. Ein solches Gebiet, mit welchem sich auch diese Arbeit befasst, ist der Systementwurf von eingebetteten Computern, deren Hauptfunktion die Echtzeit-Verarbeitung von Media-Datenströmen beinhaltet (z.B. Datenströme von Audio- und Video-Daten). In letzter Zeit ist die Nachfrage nach solchen Computern stark gewachsen, da sie zunehmend in den meisten elektronischen Produkten verwendet werden. Besonders im Unterhaltungselektroniksbereich finden sich viele Beispiele wie digitale Fernseher, Audio- und Video-Recorder, digitale Videokameras, Digitalempfänger, Multimedia-Mobiltelefone und andere elektronische Geräte, die Multimedia-Anwendungen unterstützen. Um die hohen Ansprüche an die Leistung eines solchen Systems zu erfüllen, gleichzeitig aber die Budgets bezüglich Kosten, Grösse und Energieverbrauch nicht zu sprengen, werden diese eingebetteten Computer als komplexe, heterogene Multiprozessorsysteme entworfen. Die ständig wachsende Komplexität dieser Systeme und der darauf ausgeführten Multimedia-Anwendungen führen zu einem sehr komplexen Verhalten der Rechenauslastung, das die Modellierung erschwert.

In dieser Arbeit zeigen wir, dass die Variabilität verschiedener Kenngrössen von der Multimedia-Rechenauslastung die Haupteigenschaft ist, die ein geeignetes Auslastungsmodell umfassen sollte. Wir zeigen weiterhin, dass herkömmliche Auslastungsmodelle diese dynamischen Eigenschaften der Rechenauslastung nicht genau modellieren und demzufolge zu pessimistische Abschätzungen der Systemleistung liefern, besonders dann, wenn die Extremwerte der Leistung von Interesse sind. Als Lösung schlagen wir ein neuartiges Auslastungsmodell vor, welches die dynamischen Eigenschaften der Rechenauslastung gut charakterisieren kann. Wir zeigen die Vorteile des vorgeschlagenen Modells gegenüber herkömmlichen Auslastungsmodellen, und entwickeln einige Systementwurfsmethoden, welche auf diesem Modell beruhen. Diese Methoden umfassen die Leistunganalyse auf Systemebene, die automatische Identifizierung der charakteristischen Rechenauslastung für die System-Simulation, den Entwurf und die Optimierung der Strategien zum Management der Systemressourcen, und ein Verfahren für die Anpassung der Prozessortaktfrequenz zur Laufzeit für eine energieeffiziente Verarbeitung von Media-Datenströmen auf heterogenen eingebetteten Multiprozessorsystemen mit Speicherplatzeinschränkungen. Wir zeigen den Nutzen unseres Auslastungsmodells und evaluieren es durch eine Reihe von Fallstudien, unterstützt durch detaillierte Simulationen.

I would like to thank

- Prof. Dr. Lothar Thiele for advising my research work and providing an excellent research environment,
- Prof. Dr. Petru Eles, for his willingness to be the co-examiner of my thesis,
- Prof. Dr. Samarjit Chakraborty for a very fruitful research cooperation,
- Dr. Jens Benndorf and Alexander Zhvania for their great encouragement and support, and
- my family for their love and understanding.

*To my wife, Natalia, and*
*to my daughter, Ekaterina.*

# Contents

# 1

# Introduction

Design of virtually any computer system starts from defining the system's intended range of *applications*. Subsequently, designers try to architect the computer system such that it supports its target applications in a most efficient and economical way. The designers optimize the system architecture based on such criteria as system's cost, size, performance and energy consumption. In this process, knowing the characteristics of the *workload* which the target applications will impose on the architecture is essential for arriving at an optimal architectural solution.

Those workload characteristics that are important in a given design context can be captured in a *workload model*. Using such a model is an established practice in computer system design and performance evaluation. A workload model serves to formally characterize the workload and on the basis of this characterization to distinguish between different workload scenarios. Such a characterization represents an important input to the system architecture design and optimization process. Further, a workload model is indispensable during design of various resource management policies and run-time adaptation strategies for the architecture. Typically, it is also an integral part of a *performance model* used for performance analysis. How good (e.g. accurate, reliable and efficiently analyzable) a workload model is largely determines the quality of the design solutions and the accuracy of the performance estimations based on it. In many computer design contexts, finding an appropriate workload model represents a difficult problem.

In this thesis, we address the problem of modeling *multimedia workloads* for system-level design of embedded systems whose functionality involves real-time processing of media streams, e.g., streams containing audio and video data.

Driven by application requirements and fuelled by technological advances, the architectures of such embedded systems are increasingly being designed to contain a composition of diverse parallel processing elements integrated on a single chip. Such *heterogeneous multiprocessor system-on-chip* (MpSoC) architectures have a potential to provide high performance and flexibility in a cost- and energy-efficient manner. However, in many cases, this potential is difficult to realize as there is still a lack of methods and tools that could streamline the design process of MpSoC architectures while producing high-quality results. This problem to a large extent stems from the inability of the models traditionally used for the system-level design to accurately capture important characteristics of the multimedia workloads imposed on the MpSoC architectures as a result of processing media streams.

This chapter first introduces the workload modeling problem arising in the system-level design context of heterogeneous multiprocessor embedded computers for media processing, such as multimedia MpSoCs. After that, it summarizes contributions and gives an outline of this thesis.

## 1.1 Embedded Computers for Media Processing

The number of various consumer electronics products supporting multimedia applications rapidly grows. Digital TVs, DVD players, digital video cameras, advanced set-top boxes, media adapters, game consoles and multimedia-enabled cell phones are just a few examples of such products. The vast majority of these products have special-purpose computers embedded in them. The workloads imposed on these *embedded computers* are dominated by applications involving digital processing of media streams, such as audio, video, graphics, as well as other kinds of streaming data (e.g. web or voice-over-IP traffic). A typical multimedia application includes receiving data streams from the environment (e.g. from a microphone or a broadband communication network), processing these streams using various algorithms — mainly falling into four categories: compression-decompression algorithms, digital signal processing, content analysis and network packet processing [38, 173] — and sending the processed streams back to the environment (e.g. to display devices).

Embedded systems like those just described have to process media streams under stringent timing constraints determined by the environment. For example, a digital video camera has to process video frames at the rate with which they arrive at its input. Furthermore, strict delay and jitter constraints may be associated with the processing of each individual frame. If these timing constraints are not met, the quality of the processed video stream may seriously degrade. Therefore, during design of such *real-time embedded systems*, ensuring *temporal correctness* of their behavior is equally important as ensuring its functional correctness.

The need to execute complex media processing algorithms under tight timing constraints implies that the embedded computers have to be designed to sustain *high computational loads*. On the other hand, to be suitable for the deployment in the consumer electronics products, these embedded computers must be aggressively optimized to have *low energy consumption and cost*. In addition, continuous evolution of multimedia standards and emergence of new media formats, coupled with ever increasing complexity of multimedia applications, motivate *flexible* architectures. This combination of requirements calls for application-specific, *heterogeneous architectures* containing multiple computational components with different degrees of programmability, ranging from fully programmable processors to dedicated function blocks.

### 1.1.1 Multiprocessor systems-on-chips

Rapid advances of the integrated circuit technology make it possible to design and implement embedded systems as *multiprocessor systems-on-chips* (MpSoCs) [172]. According to the MpSoC paradigm, multiple coarse-grain components of an embedded architecture (e.g. multiple processors, busses, memories, peripheral devices, etc.) are integrated on a single chip. This enables creation of flexible heterogeneous architectures that can satisfy high performance requirements of the multimedia applications in a cost- and energy-efficient way. Hence, increasingly, embedded computers for media processing are being implemented as MpSoCs.

There are many examples of multimedia MpSoCs available from the industry and academy [38, 62, 63, 151, 155]. Most of them follow the design pattern shown in Fig. 1: A typical multimedia MpSoC contains a number of software programmable processors (CPUs, DSPs, media processors, etc.), weakly programmable co-processors, fixed-function hardware modules, and peripheral devices (e.g. video and audio I/O blocks). These coarse-grain computational components are interconnected by an on-chip communication network which may encompass various types of busses, bridges, direct memory access (DMA) controllers, distributed memories, and other communication components. Following the emerging *network-on-chip* (NoC) paradigm [31, 79], the on-chip communication infrastructure may resemble a large-scale computer network, involving such concepts as routers, switches, protocols, communication queues, etc.

The on-chip communication network may have a complex architecture consisting of several subnetworks interconnected by bridges, as shown in Fig. 1. A subnetwork (low-level network) combines intensively communicating computational components into a tight cluster (subsystem). In this way, the local communication traffic between the components within a subsystem is isolated from the system-wide data exchange taking place via a high-level on-chip network. Besides supporting the system-wide communication, this high-level network provides an arbitrated access to a relatively large amount of inexpensive *off-chip*

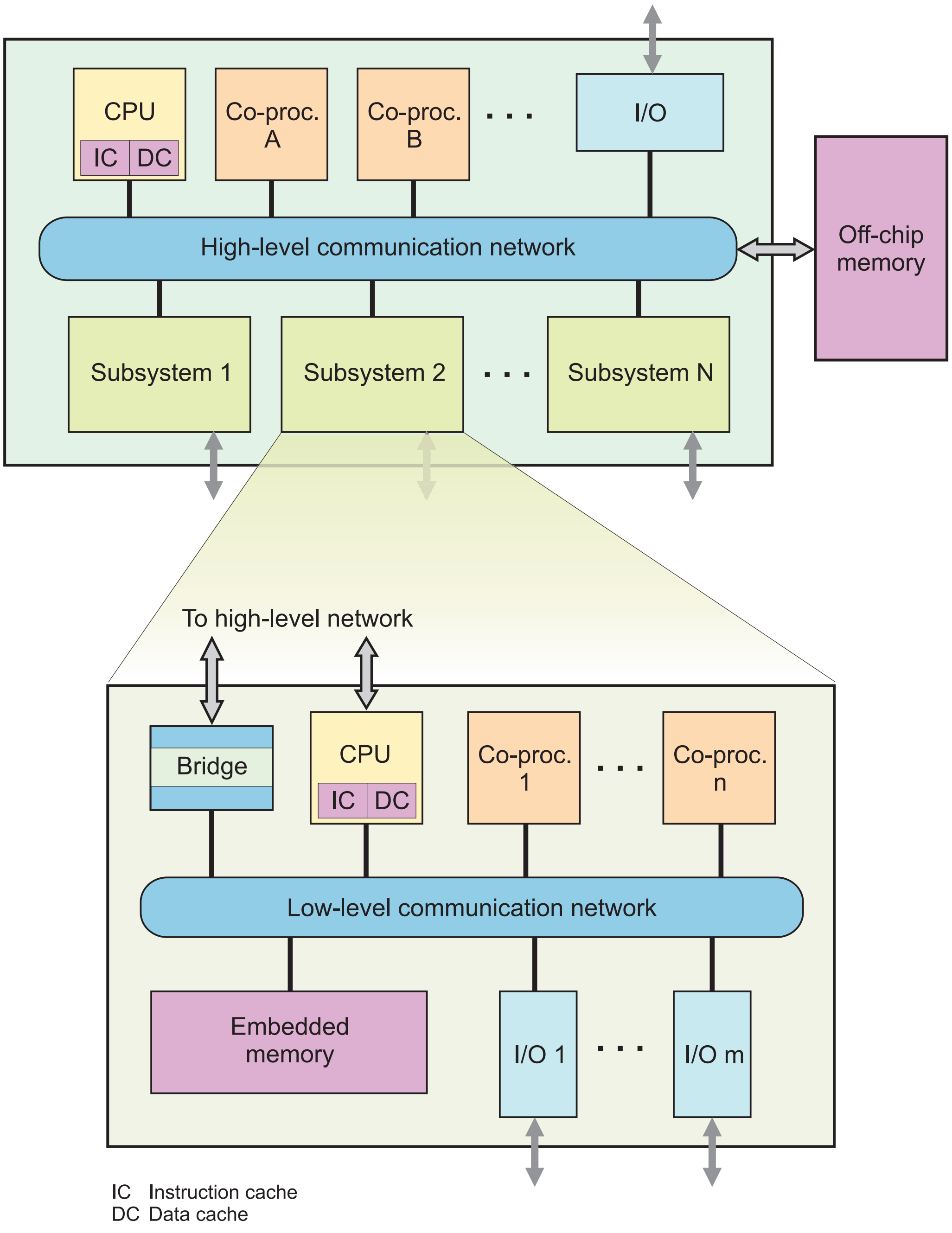


**Fig. 1:** A multimedia MpSoC template architecture.

*memory*. This memory is primarily devoted for storing global data structures as well as large data sets (e.g. full video frames) not fitting into smaller *embedded memories* located on the chip. The on-chip memory is typically more expensive but faster than the off-chip memory. Distributed around the architecture, the embedded memories store frequently accessed program code and data structures. They are also used to implement performance-critical data exchange between the on-chip computational components. To communicate certain data types, the components may need to bypass the bridges interconnecting different subnetworks. For this, the components may be connected to more than one subnetwork or directly to each other, thereby resulting in an irregular application-specific communication architecture.

### 1.1.2 System-level view of media processing

At the system level, a multimedia application executing on a heterogeneous multiprocessor architecture, such as a multimedia MpSoC, can be viewed as a set of tasks (or processes) concurrently running on different execution resources of the architecture. These tasks communicate with each other solely through *unidirectional data streams* [134, 135]. Each stream is sent from a producer task to the corresponding consumer task through a first-in-first-out (FIFO) buffer. The buffer allows for *asynchronous communication* between the tasks, thereby leading to reduced communication overheads and increased utilization of the execution resources [95]. Within the architecture, the buffers are allocated in shared memories or instantiated as dedicated hardware FIFO memory blocks.

## 1.2 System-Level Design Issues

Although the integrated circuit technology provides great opportunities for manufacturing increasingly complex MpSoCs, optimally designing such systems under high time-to-market pressures becomes more and more difficult. The growing complexity of MpSoCs poses many challenges to their system-level design. Currently, there is a lack of methods and tools that could help system designers to effectively tackle this complexity. A comprehensive discussion of the system-level design issues can be found elsewhere [72]. This section concentrates only on few of them relevant to the workload modeling problem addressed in this thesis.

The goal of embedded system designers is to construct system's architecture out of a set of hardware and software components. Due to a large number of such components and their heterogeneity, integrating them into a consistent working whole (such as an MpSoC) represents an excessive design effort. Well-defined (and standardized) component interfaces and protocols can significantly reduce this effort [72]. Although they may easy the task of building a functionally correct system, they cannot help in *verifying* whether the resulting system architecture meets performance requirements of the target applications.

*Platform-based design* further reduces the system design complexity by providing a generic, domain-specific template architecture that only needs to be customized for the target application range. Examples of such platforms in the multimedia domain include OMAP from Texas Instruments [53], Nomadik from STMicroelectronics [6] and Nexperia [38] from Philips. The platform customization involves tuning various parameters of the template architecture, such as bus widths, memory sizes, cache configurations, clock rates of processor cores, *etc.*; selecting and configuring resource management policies; and, possibly, adding to the basic architecture some application-specific components (e.g. co-processors), with the aim of obtaining an architecture that represents a desirable tradeoff between performance, energy consumption and cost.

Already for systems of moderate complexity, the resulting design space formed by all possible platform configurations may be huge and highly irregular. To efficiently explore this space, system architects must be able to quickly evaluate the performance of candidate architectures. The performance evaluation has to predict with a sufficient accuracy such characteristics of the prospective system as throughput, memory requirements, utilization of execution resources, processing delays, *etc.* It should also help system designers to identify performance bottlenecks within the system.

Nowadays the mainstream in system-level performance evaluation of complex real-time embedded systems relies on simulation. Although simulation may return very accurate performance estimations, its coverage is limited only to those workload instances that have been simulated. Hence, achieving a good coverage necessitates multiple simulation runs using carefully chosen *representative workload scenarios*. In addition, accurate simulators oftentimes exhibit high running times making them poorly suitable for a fast design space exploration cycle.

Irrespective of how many and which workload scenarios have been simulated, the simulation can never achieve, in a reasonable time, the full coverage required for the *performance verification*. Because of this, it may not be used, for example, to verify whether an embedded system satisfies the imposed on it timing constraints in all possible workload scenarios. Such a verification is possible using formal analytic approaches that perform *worst-case performance analysis*, i.e. return worst-case performance bounds.

The worst-case performance analysis uses a *performance model* which represents an abstraction encompassing all system's states and behaviors and all possible workload scenarios. The performance analysis can therefore provide the full coverage needed for the performance verification. Moreover, it is typically faster than the simulation. However, due to the complexity of both the analyzed architectures and their workloads, it is extremely difficult to find proper system abstractions that would lead to accurate (i.e. tight) worst-case performance bounds. This explains why in many design contexts, embedded system engineers prefer to use simulation for the performance evaluation, in spite of its drawbacks.

### 1.2.1 Issues in design of multimedia MpSoCs

The performance evaluation issues discussed so far arise in various system-level design contexts and are not pertinent exclusively to the domain of multimedia MpSoCs. The discussion in this subsection concentrates on the design issues that are more specific to this domain.

Users expect from the media devices a high-quality and stable delivery of a multimedia content, and these expectations are growing fast. A central concern in the design of multimedia MpSoCs is therefore to ensure a specified *quality of service* (QoS) to the processed media streams. If a device has committed to provide a certain QoS level, it has to guarantee this level under any circumstances. This requirement is especially difficult to fulfill because many multimedia applications impose *highly variable and unpredictable workloads* on the underlying architectures [14, 57, 142, 165]. The designers of multimedia MpSoCs therefore face a challenging problem of designing architectures capable of providing a *predictable performance* under uncertain workload conditions and stringent cost and energy constraints.

The quality of media streams processed on an MpSoC depends on two factors:

- First, as mentioned in Section 1.1, a violation of the *timing constraints* associated with the stream processing may seriously impair the quality of media streams.
- Second, the quality may also degrade if the FIFO buffers between the application tasks executing on the MpSoC experience *overflows or underflows*. This leads to the concept of *buffer constraints*: An overflow (underflow) buffer constraint requires that the corresponding buffer never overflows (underflows).

Hence, the QoS guarantees are specified in terms of the timing and buffer constraints.

To provide the QoS guarantees under uncertain workload conditions, the resources of an embedded architecture for media processing have to be dimensioned for the *worst-case workload*. However, since the worst-case workload occurs rarely, the resources may remain underutilized most of the time. A way to improve the utilization is to share the resources among several independent concurrent applications (or application tasks). Such a sharing has to respect the QoS guarantees associated with the processed media streams. This necessitates deployment of sophisticated *resource management policies*. These policies must be able to satisfy timing and buffer constraints associated with several concurrent streams imposing varying resource demands on the shared communication and computational components of the architecture. Whereas the current practice relies on computationally expensive dynamic schemes [135, 137], the goal is to design low-overhead resource management policies.

A major design effort is directed towards making embedded systems energy-efficient. This issue is crucial in the design of battery-operated multimedia devices, such as portable media players or cell phones. Achieving the energy efficiency requires multimedia MpSoCs to be adaptable to changing workload conditions. For this, the architectures provide various energy-saving mechanisms, e.g., support a variety of power modes. Intelligent *run-time adaptation strategies* are needed to control these mechanisms. For instance, to reduce the energy dissipated on an MpSoC component, the operating frequency and voltage of this component can be dynamically adjusted in response to workload fluctuations experienced by it. Such a run-time energy management must be performed without jeopardizing the QoS guarantees associated with the media streams processed by this component. This implies that the run-time adaptation strategies must be able to handle the worst-case workload, which may occur sporadically.

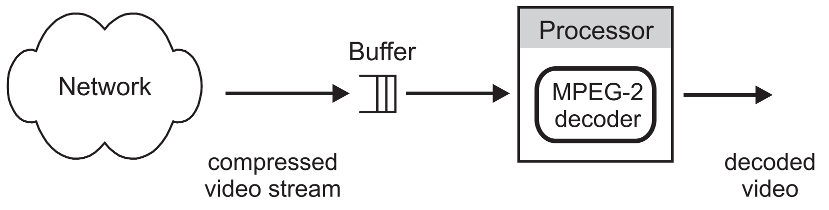
## 1.3 The Workload Modeling Problem

Effectively addressing the system-level design issues outlined in the preceding section requires a proper workload model:

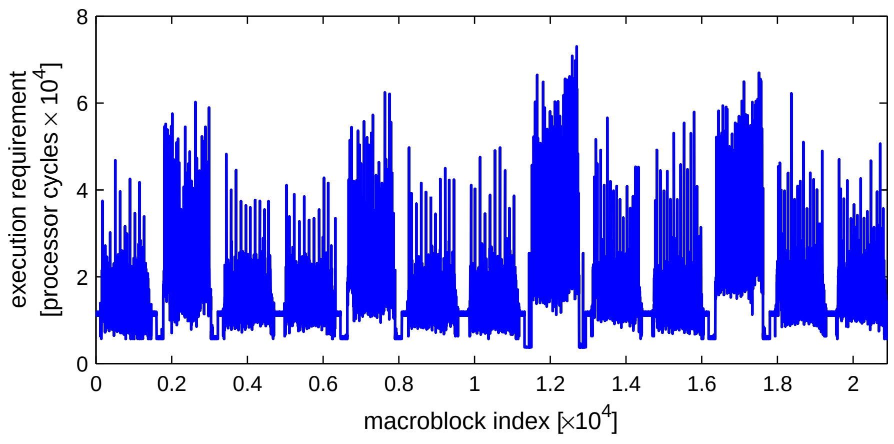
- The selection of representative workload for an effective simulation-based performance evaluation necessitates a comparison of different workload scenarios. For the comparison, the workload scenarios have to be characterized based on a model which captures interesting for the performance evaluation workload properties. For example, if designers intend to determine by the simulation the required FIFO buffer sizes, they may want to identify a diverse set of workload scenarios which produce maximum backlogs in different FIFO buffers of the architecture. Thus, the model they use for the workload characterization may include such a property as burstiness of the communication patterns between application tasks.

- A workload model also forms the basis of any analytic performance model. It is therefore responsible for tightness of the worst-case bounds returned by the performance analysis. Tighter bounds imply less pessimism in the resource dimensioning, thereby leading to lower system cost and energy consumption. Hence, having a workload model which provides a pessimistic but *accurate* workload characterization is essential in this context. Additionally, a successful workload model should allow for an efficient analysis.

- Finally, a workload model is necessary in design of the resource management policies and the energy-saving run-time adaptation strategies. These techniques have to be aware of the workload dynamics. This implies that these dynamics



(a)



(b)

**Fig. 2:** Motivating example: A processor executing an MPEG-2 decoder application (a); and a trace of execution requirements imposed on the processor by this application (b). The plot shows the number of processor cycles required to decode a sequence of macroblocks within an MPEG-2 video stream.

must be reflected in the workload model. As for the worst-case performance analysis, such a model should describe the workload with a sufficient accuracy, and, at the same time, if hard QoS guarantees are required, it has to represent a wort-case characterization.

Clearly, to be effective in the above roles, a workload model should provide a proper abstraction of the actual workload. However, due to the complex, highly variable nature of many multimedia workloads, coming up with such an abstraction represents a difficult problem. Existing system-level design methods and tools rely on workload models which are unable to accurately capture the workload variability. As a consequence, they may produce unsatisfactory results. Worst-case performance analysis methods are a salient manifestation of this fact: Overly pessimistic bounds that they return sometimes are not useful at all for an economical design.

Fig. 2 shows a simple example illustrating the above concern: The processor depicted in Fig. 2(a) executes an MPEG-2 decoding algorithm on a video stream arriving from a network. The complex, highly variable nature of the workload imposed by the video stream on this processor is apparent from the plot in Fig. 2(b), which shows a trace of processor cycles required to decode a sequence of macroblocks[1] within the video stream.

While analyzing the performance of a system such as the one shown in Fig. 2, the existing methods typically assume that *each* macroblock in the stream requires for its processing the largest possible number of cycles, i.e. imposes on the processor the *worst-case execution demand* (WCED). Such an assumption would be necessary, for example, to *guarantee* that the buffer at the input of the processor in Fig. 2(a) never overflows. However, this assumption would be too pessimistic and therefore may lead to unnecessary costly designs: The ratio of the worst-case to the average load on a processor due to a multimedia application can easily be as high as a factor of 10 [134]. In this case, the assumption that *each* macroblock in the video stream requests from the processor the WCED represents a very inaccurate workload abstraction. Hence, a better abstraction capable of capturing the workload variability and thereby resulting in more accurate performance estimates is needed.

Fig. 2 demonstrates only one aspect of the workload variability. In reality, a typical multimedia task can be characterized by the variability of several parameters. For example, in the scenario shown in Fig. 2, the arrivals of the media stream from the network may be characterized by bursts which depend on the network congestion levels. Another source of the variability may be a nonconstant rate with which the MPEG-2 decoding task consumes the data from the input buffer. Similarly, a task may produce the data at its output at a variable rate. In addition, a combination of a particular application with a given hardware architecture may result in many other sources of the variability. For instance, advanced microarchitectural features, such as caches and branch prediction, may result in variable task execution times, etc.

Finally, at the system level, a composition of multiple tasks, executing concurrently on distributed resources of an MpSoC and each characterized by variability of several parameters, results in complex non-functional interactions and interdependencies between the architectural components [133]. Accurately estimating performance of such a composition represents a challenging task. Even more challenging task is to *verify* that the composition meets certain performance requirements or to decide which resource management policies and run-time adaptation strategies should be used to orchestrate it. This gives rise to the following research question:

*What kind of workload model (and the associated with it analysis methods) can help to effectively address these system-level design problems?*

[1] An MPEG-2 video stream encodes a sequence of video frames. Every frame in the sequence is composed out of *macroblocks*, with each macroblock representing a certain $16 \times 16$ block of pixels within the frame [118].

## 1.4 Thesis Contributions

In this thesis, we propose a new model for characterization of multimedia workloads in the system-level design of heterogeneous multiprocessor embedded computers. This workload model allows to effectively address many of the design issues described in the preceding sections. In particular, this thesis makes the following main contributions:

- We introduce the concept of *Variability Characterization Curves* (VCCs) as a means to characterize entire classes of increasing functions or sequences based on their worst-case and best-case variability. We then define several VCC types for the multimedia workload characterization (collectively referred to as *multimedia VCCs*).

- We extend the modeling capabilities of the *Modular Performance Analysis* framework [159, 160] and its mathematical foundation, the *Real-Time Calculus* [24, 121, 157, 158], with the multimedia VCCs. Towards this, we introduce the concept of *workload transformations*, which enable an *accurate and efficient* performance analysis of heterogeneous multiprocessor embedded systems under variable multimedia workloads. The extended analysis framework can return significantly tighter performance bounds than those achievable without the workload transformations.

- We formulate the problem of *scheduling bursty media streams under strict buffer constraints* and propose methods to address this problem. In particular, we present *a framework for design of resource management policies* for multimedia MpSoCs. The framework provides methods to quickly evaluate the quality and check the feasibility of various resource management policies to be deployed in an MpSoC. It fully relies on the VCC-based characterization of the media streams.

- We show how the VCC-based workload model can be used for *energy-efficient media stream processing*. Towards this, we develop a run-time processor rate adaptation strategy which can be used in conjunction with the dynamic voltage scaling to achieve considerable energy savings while processing bursty multimedia workloads under strict buffer constraints. In comparison to other methods addressing similar problems, our scheme handles multimedia workloads characterized by both, the data-dependent variability in the execution time of multimedia tasks and the burstiness in the on-chip traffic arising out of multimedia processing, and at the same time it provides hard QoS guarantees.

- We introduce the problem of *selecting representative workload* for system-level performance evaluation of MpSoCs and propose a solution to this problem for the case of multimedia workloads. Our method employs VCCs for the workload characterization and supports *automatic identification of the representative workload*.

- Finally, we demonstrate the utility and experimentally assess the quality of the VCC-based workload model through several case studies involving realistic application scenarios. In the experiments, we compare our model with the existing analytic approaches and with a detailed (transaction-level) system simulator.

## 1.5 Thesis Overview

- The main purpose of Chapter 2 is to introduce the MPA framework whose modeling capabilities we extend in Chapter 3.
- Chapter 3 introduces the concepts of VCCs and workload transformations, defines the multimedia VCC types and proposes several workload transformations based on them. This chapter also discusses possible ways to obtain VCCs and presents results of an experimental evaluation of the VCC-based workload model.
- In Chapter 4, we address the problem of selecting representative workload for system-level performance evaluation of MpSoCs. We show how the VCC-based workload characterization model can be used for quantitative comparison and classification of media streams and present results of an empirical validation of the proposed method.
- Chapter 5 introduces the problem of stream scheduling under buffer constraints and presents the framework for design of resource management policies for multimedia MpSoCs. It focuses mainly on the methods for quick feasibility tests of stream schedulers and estimation of buffer memory requirements resulting from deploying these schedulers on the processing elements of an MpSoC.
- Chapter 6 presents the VCC-based run-time processor rate adaptation technique for energy-efficient media stream processing under buffer constraints.
- Finally, Chapter 7 summarizes main results of this work.

# 2

# System-Level Performance Analysis

System-level performance analysis plays a key role in the design of complex embedded systems. It is used early in the design cycle to estimate characteristics of the prospective embedded system and based on this estimation make critical design decisions. The quality of these decisions therefore largely depends on the quality of the estimates obtained from the performance analysis. This explains why a significant research effort is being invested in devising efficient performance analysis methods capable of producing accurate and reliable estimates of the system performance.

This chapter introduces the problem of system-level performance analysis of heterogeneous multiprocessor embedded systems. It briefly outlines existing approaches to solving this problem and treats in detail one of them — the Modular Performance Analysis (MPA) framework based on the Real-Time Calculus (RTC). This framework provides powerful abstractions and mathematical support for a compositional performance analysis of distributed embedded systems. However, the basic abstractions it offers are not sufficient for an accurate performance modeling of heterogeneous multiprocessor embedded computers for media processing. We will address this problem in the next chapter by extending the modeling capabilities of the MPA framework.

## 2.1 Introduction

### 2.1.1 Requirements

Early in the design cycle, embedded system designers face the problem of evaluating many candidate hardware-software architectures with respect to various performance indexes. These indexes may include system's throughput, response times, end-to-end delays, resource utilization, memory requirements, etc. In most cases, building a prototype for each design alternative to directly measure these performance characteristics is infeasible because of high implementation costs and stringent time-to-market constraints. On the other hand, due to the increasing complexity of modern embedded systems, back-of-the-envelope estimations cannot be used without taking the risk of being totally incorrect. Hence, the only option left for the designers is to carry out the performance analysis based on some kind of a *performance model* of the system. This can be a simulator or a mathematical model. In any case, it should return sufficiently accurate estimates of the system performance. Furthermore, to allow for a fast design space exploration, the performance model should also be efficiently analyzable and easily constructible. The latter property is especially important for supporting automated design space exploration.

Designing embedded systems that must satisfy real-time constraints faces additional challenges associated with the need to *verify* timing correctness of their behavior. For instance, it might be necessary to verify whether the time elapsed between two specified events within the system *ever* exceeds a given value. Such a verification can only be accomplished using a formal system model supporting worst-case analysis, which implies a complete coverage of all possible states of the system and of its environment. Neither system's prototype nor its simulator can be employed for the performance verification purposes as (due to the high system complexity) it is hardly possible to check all system states within a reasonable time frame.

### 2.1.2 Input specification

A starting point for the system-level performance analysis is a specification which typically describes the following aspects of an embedded system:

- **Application task structure**
  The application task structure is typically modeled by a *task graph* (or a set of task graphs) that captures a partitioning of the target application into individual tasks, and models data and control dependencies between them. Interactions between the tasks in a task graph may be governed by a specific *model of computation*. For example, multimedia applications are often modeled using the formalism of *Kahn Process Networks* [70, 134, 135], which assumes that the tasks communicate via FIFO channels.

- **Task assignment to processing elements**
  The application tasks performing data transformations are assigned for execution to *computational resources* such as CPUs, DSPs and co-processors, while the tasks responsible for data transfers are assigned to *communication resources* such as busses, DMA controllers, bridges, etc. Throughout this thesis we refer to both resource types as *processing elements* (PEs) because in principle for the performance analysis it is irrelevant whether an architectural resource executes computation or communication tasks.

- **Resource management policies**
  As a result of the task assignment, multiple tasks may be mapped on to one PE. In this case, a scheduling (or arbitration) policy is deployed to manage tasks' access to this PE. In general, several different scheduling and arbitration policies may be deployed within the architecture.

- **Storage resource allocation**
  Data arrays manipulated by the tasks, for example, the buffers implementing the FIFO communication channels, are assigned to the off- and on-chip memories.

- **Characteristics of processing elements**
  For the performance analysis we need to specify capabilities of processing elements. Therefore, such parameters as clock rates of processors and effective communication bandwidths of busses typically form a part of the input specification.

- **Task properties**
  These include a variety of relevant to the performance analysis task characteristics, for example, the number of processor cycles needed to complete a task on a given PE and the size of data items to be exchanged between the tasks.

- **Characteristics of the environment**
  The input specification should also capture characteristics of the event streams to be processed by the embedded system. These characteristics may include timing properties of the event streams (e.g. their arrival rates) as well as their possible contents, for example, different event types that may appear in a given event flow.

**Ex. 1:** *Fig. 3 shows a mapping of an example application onto a hypothetical architecture. The application is specified by two task graphs describing the processing of two independent event flows. The nodes in a task graph correspond to the application tasks, while the edges model the data dependencies between these tasks. $T1$ and $T4$ are communication tasks mapped on to a bus, which is shared between these tasks. $T1$, $T3$, and $T5$ are computational tasks. $T1$ is assigned for execution to a DSP, whereas $T3$ and $T5$ share a CPU.*

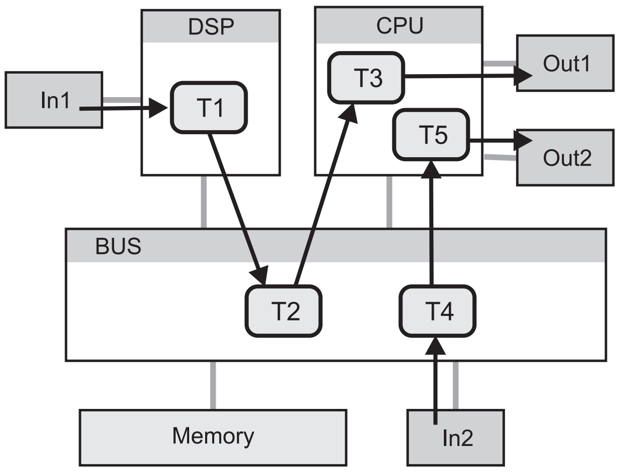


**Fig. 3:** An example application-to-architecture mapping.

### 2.1.3 Existing approaches to performance analysis

Based on the kind of specification described in the previous subsection, designers need to build a performance model of the system. They can do this in several ways. For example, they could construct a system simulator [15], use a trace-based performance evaluation technique [83, 125], create a stochastic model of the system [93] or employ a worst-case performance analysis method. Their choice depends on the analysis goals (and on the available expertise and tools). A comparative overview of various approaches to the performance analysis of embedded systems can be found elsewhere (see, e.g. [159]). In this chapter, we concentrate on techniques suitable for the *performance verification*, i.e. on the worst-case performance analysis methods. Furthermore, here we limit the discussion only to those methods that can be applied to distributed (multiprocessor) embedded systems having heterogeneous hardware-software architectures. By the *heterogeneity* we mean not only the diversity of processing elements making up the architecture but also the variety of scheduling and arbitration policies that might be deployed on those processing elements.

The need to ensure the timing correctness of *distributed* real-time embedded systems has led to the development of methods that can analyze worst-case end-to-end response times of entire task chains mapped onto multiple processor nodes communicating via a shared bus. Such methods have been termed *holistic scheduling analysis* because they tightly integrate the schedulability analysis of individual processing elements (i.e. processors and communication channels) into an overall piece of analysis [163]. The first holistic method proposed in Tindell *et al.* [163] addressed systems with fixed priority scheduling policy deployed on processor nodes communicating via a bus using a time division multiple access (TDMA) protocol. Later many extensions and generalizations of this method appeared in the literature (see, e.g. [48, 128, 130] and references therein).

These methods can be very effective in modeling complex timing relations (e.g. phasing) between the tasks. However, they are often attributed a lack of scalability and modularity [68, 159], which are needed for modeling large heterogeneous systems (perhaps, with hundreds of nodes) and for quick modifications of these performance models during a design space exploration cycle. In other words, these techniques might need to be redesigned for each new system configuration.

The above problem has been partially addressed in Ernst *et al.* [68, 133]. Their approach advocates a *compositional performance analysis* methodology which uses propagation of abstract event streams between various scheduling analysis techniques locally applied to the processing elements (components). The basic idea is to reuse existing (standard) scheduling techniques for the local analysis. This entails using standard event models (e.g. sporadic, periodic, periodic with jitter, periodic with bursts) and adapting them between the components which use incompatible event models. These adaptations as well as the standard event models themselves may be overly pessimistic, leading to a loss in accuracy. Furthermore, since the method heavily relies on the existing scheduling analysis techniques, supporting any new (not yet existing) scheduling policy necessitates devising an analysis for it; i.e. essentially the method suffers from the same problem as the holistic scheduling analysis discussed above.

In the next section, we describe the MPA framework [159, 160], which tries to overcome the drawbacks of other scheduling analysis methods by following a completely different approach to the performance analysis, which does not rely neither on the standard event models nor on the traditional scheduling analysis methods, while offering a high degree of generality and modularity.

## 2.2 Modular Performance Analysis

### 2.2.1 Basic idea

In essence, any performance analysis involves two basic concepts — the *service requested* by an application (task) and the *service offered* by the architecture to this application (task). Temporal interactions between the requested and the offered service determine performance characteristics of the system. The ultimate goal of any performance analysis method is therefore to properly capture these interactions. In this subsection, we describe how this is achieved in the MPA framework.

The basic idea behind the MPA framework is to model the interactions between the requested and the offered service using the concept of *scheduling network*. In a scheduling network, the requested and the offered service are modeled by *event and resource streams*. These streams flow through the network nodes,

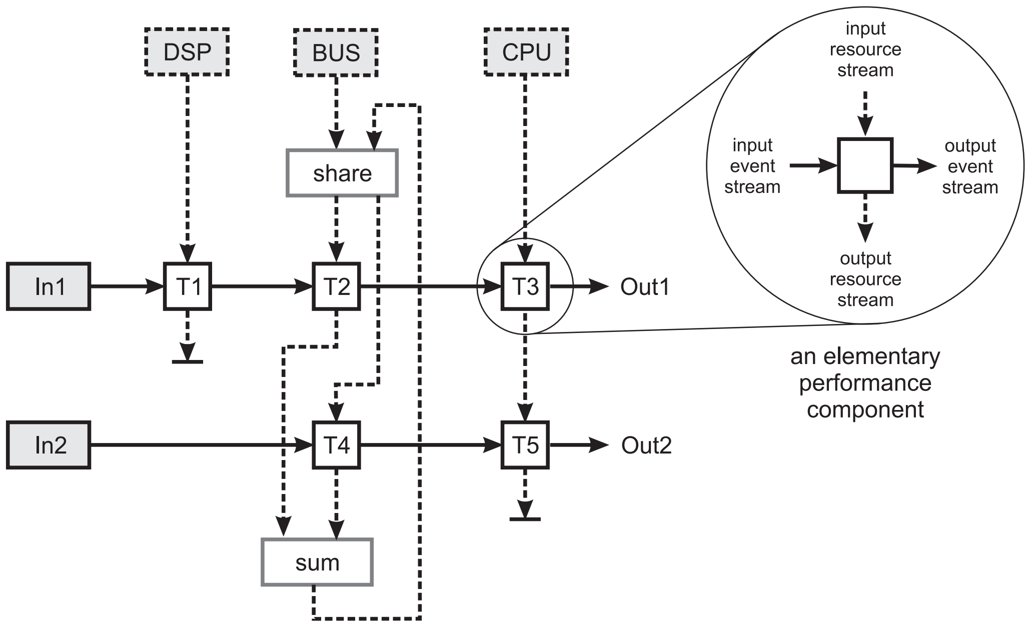


**Fig. 4:** A scheduling network modeling the application-to-architecture mapping shown in Fig. 3.

called *performance components*, that model the interactions between the streams. Fig. 4 shows an example scheduling network corresponding to the application-to-architecture mapping discussed in Ex. 1. Solid and dashed arrows correspond to the event and resource streams, respectively.

An *elementary* performance component receives one event and one resource stream as its input (see Fig. 4). The input event stream abstracts arrivals of a certain request type, while the input resource stream models availability of a given resource for processing of this request type. Abstractly seen, the input event stream triggers the performance component, which in response proceeds by consuming resources provided by the input resource stream. This represents execution of a task on a PE.

An elementary performance component typically also produces one event and one resource stream as its output. An event within the output event stream signifies a completed processing of a corresponding input event. The output resource stream represents the *remaining service*, i.e. the service which has not been consumed by the performance component. This remaining service can then be used to process another event stream, i.e. it may serve as an input to another performance component. Likewise, the output event stream may represent requests for another resource, i.e. it also may serve as an input to a different performance component. In this way, a scheduling network representing a performance model of the entire system (with a multitude of event streams and processing resources) can be constructed out of multiple independent performance components.

Besides the elementary performance components, a scheduling network may contain other types of nodes:

- **Resource modules** model processing capabilities of PEs within the architecture. A resource module produces a stream corresponding to the *unloaded* resource that it models. In Fig. 4, resource modules are marked with dashed boxes. They represent the bus, DSP and CPU resources from Ex. 1.
- **Input modules** inject into the scheduling network event streams generated by the system's environment. In Fig. 4, these are $In1$ and $In2$ modules.
- **Scheduling modules** distribute resource streams between different performance components in accordance with a given resource management policy. A scheduling module receives and produces only resource streams (originated by the same resource). Using scheduling modules we can model different scheduling and arbitration policies deployed on the PEs of the architecture. In Fig. 4, for example, we have $share$ and $sum$ scheduling modules.
- **Hierarchical modules** are complex performance components containing sub-networks of other components.

For the performance analysis, in addition to the *structural* performance view of the system provided by the scheduling network, we need also to characterize *behavior* of the event and resource streams, and of the associated performance components. That is we need to characterize timing properties of the streams and determine how these properties change when the streams pass through the performance components in the scheduling network. This can be done in many different ways. For example, we could simply simulate the scheduling network using appropriate event traces. However, our objective is a method which can be used for the worst-case performance analysis. To achieve this objective, we can rely on the mathematical foundation provided by the Real-Time Calculus, which is briefly introduced in the next section.

### 2.2.2 Real-Time Calculus

The Real-Time Calculus [24, 121, 157, 158] provides powerful abstractions of the event and resource streams and uses these abstractions to mathematically model the behavior of an elementary performance component. This basic model can then be used for a component-wise evaluation of a whole scheduling network. In addition, the Real-Time Calculus allows to compute various performance indexes of the system, such as upper bounds on the delay and backlog experienced by the events while being processed in the system.

#### Characterization of event and resource streams

Timing properties of event and resource streams are captured using *arrival and service curves*.

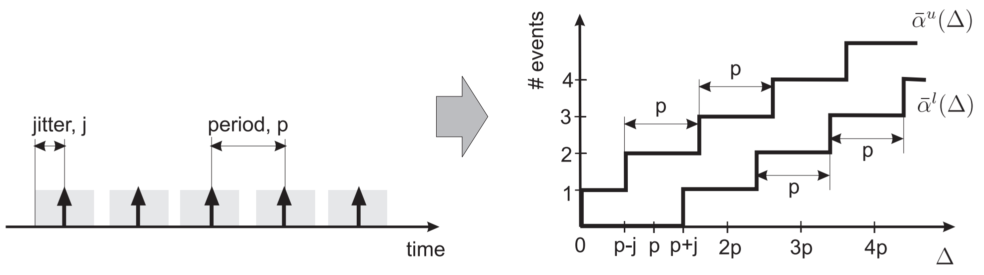


**Fig. 5:** Modeling periodic event streams with jitter using arrival curves.

An event stream is abstracted by a pair of arrival curves, $\bar{\alpha}^u(\Delta)$ and $\bar{\alpha}^l(\Delta)$, which give respectively upper and lower bounds on the number of events seen in the event stream within any time interval of length $\Delta$.

A resource stream is modeled by a pair of service curves, $\bar{\beta}^u(\Delta)$ and $\bar{\beta}^l(\Delta)$, which give respectively upper and lower bounds on the resource amount (e.g. number of processor cycles) offered within any time interval of length $\Delta$.

The arrival and service curves can accurately describe streams with arbitrary complex timing behavior. On the other hand, a single pair of upper and lower curves can capture an entire class of streams with similar timing properties. For example, many standard event models (e.g. sporadic, periodic, periodic with jitter, periodic with bursts) can be represented by the arrival curves [24]. Fig. 5 illustrates this fact by showing how the arrival curves model a class of periodic event streams with jitter.

### Scheduling network evaluation

The performance analysis using the MPA approach entails a scheduling network evaluation. The evaluation can be accomplished component-wise, by propagating the event and resource streams through the network. Doing this requires a model describing how the timing properties of the event and resource streams get changed as a result of passing through the performance components. Since event and resource streams are abstracted by the arrival and service curves, we need a mathematical model describing how an elementary performance component transforms the shapes of these curves. Such a model, provided by the Real-Time Calculus, is given by the following set of equations [24]:

$$\bar{\alpha}_O^u = [(\bar{\alpha}_I^u \underline{\otimes} \bar{\beta}_I^u) \overline{\oslash} \bar{\beta}_I^l] \wedge \bar{\beta}_I^u \tag{2.1}$$
$$\bar{\alpha}_O^l = [(\bar{\alpha}_I^l \overline{\oslash} \bar{\beta}_I^u) \underline{\otimes} \bar{\beta}_I^l] \wedge \bar{\beta}_I^l \tag{2.2}$$
$$\bar{\beta}_O^u = (\bar{\beta}_I^u - \bar{\alpha}_I^l) \underline{\oslash} 0 \tag{2.3}$$
$$\bar{\beta}_O^l = (\bar{\beta}_I^l - \bar{\alpha}_I^u) \overline{\otimes} 0 \tag{2.4}$$

$\bar{\alpha}_I^u$, $\bar{\alpha}_I^l$, $\bar{\beta}_I^u$ and $\bar{\beta}_I^l$ denote the arrival and the service curves characterizing the event and the resource streams at the input of an elementary performance component, respectively. $\bar{\alpha}_O^u$, $\bar{\alpha}_O^l$, $\bar{\beta}_O^u$ and $\bar{\beta}_O^l$ provide the corresponding characterization of the streams at the output of the component. The model assumes that the events belonging to the same stream are processed in their arrival order and that they are stored in a FIFO buffer while waiting to be served.

Equations (2.1)–(2.4) use $(max, +)$- and $(min, +)$-algebra operators defined as follows [10].

$$(f \underline{\otimes} g)(t) = \inf_{0 \le u \le t} \{f(t-u) + g(u)\} \tag{2.5}$$
$$(f \overline{\otimes} g)(t) = \sup_{0 \le u \le t} \{f(t-u) + g(u)\} \tag{2.6}$$

$$(f \overline{\oslash} g)(t) = \sup_{u \ge 0} \{f(t+u) - g(u)\} \tag{2.7}$$
$$(f \underline{\oslash} g)(t) = \inf_{u \ge 0} \{f(t+u) - g(u)\} \tag{2.8}$$

(2.5) and (2.6) denote $(min, +)$ and $(max, +)$ convolutions, respectively, and (2.7) and (2.8) are corresponding deconvolutions. $f(t)$ and $g(t)$ denote non-decreasing functions.

**Modeling scheduling polices**

In comparison to other performance analysis methods, the MPA framework models the service offered to an event stream *explicitly*, using the concept of resource streams [159]. This approach has a number of advantages: First, it allows to model arbitrary complex resource availability patterns which may be experienced by individual event streams (or tasks) as a result of applying a certain scheduling or arbitration policy. Second, it supports the modularity of the performance analysis. Third, using the concept of resource streams it is easier to model hierarchical scheduling schemes and various resource reservation mechanisms.

A variety of scheduling and arbitration policies can be modeled by a proper calculation (or definition) of the service curves within a scheduling network. For example, a fixed priority scheduling can be modeled by directly connecting the output resource stream (i.e. the remaining service) of a higher priority component to the resource input of the next (in terms of priority) component. For example, in the scheduling network in Fig. 4, tasks $T3$ and $T5$ are scheduled on the CPU resource using the fixed priority scheme. $T3$ has the highest priority.

To model proportional share schemes and their derivatives, we need to introduce into the scheduling network the corresponding scheduling modules that distribute the resource streams according to specified shares, and after that collect the remaining service. Fig. 4 depicts an example of such an arrangement for tasks $T2$ and $T4$.

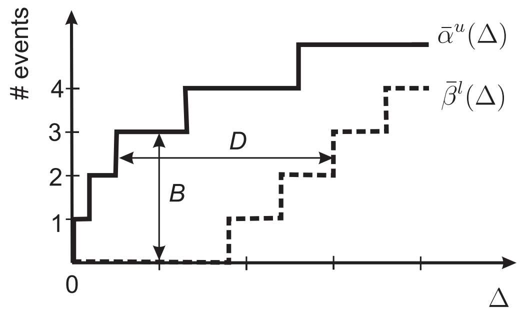


**Fig. 6:** Computing upper bounds on the delay, $D$, and the backlog, $B$.

**Calculating upper bounds on delay and backlog**
Given an upper arrival curve $\bar{\alpha}^u$ and a lower service curve $\bar{\beta}^l$ at the input of an elementary performance component, we can compute upper bounds on the delay and on the backlog experienced by the event stream as a result of passing through this component [85]:

$$delay \leq \sup_{\Delta \in \mathbb{R}_{\geq 0}} \left\{ \inf\{\tau \geq 0 : \bar{\alpha}^u(\Delta) \leq \bar{\beta}^l(\Delta + \tau)\} \right\} \tag{2.9}$$

$$backlog \leq \sup_{\Delta \in \mathbb{R}_{\geq 0}} \{\bar{\alpha}^u(\Delta) - \bar{\beta}^l(\Delta)\} \tag{2.10}$$

Fig. 6 illustrates these formulas.

In a similar way, we can find upper bounds on the total delay and on the total backlog which an event stream may suffer as a result of passing through a chain of performance components. How this can be done is described in [85, 159]. This allows to estimate such performance indexes of an embedded system as the worst-case end-to-end delay and memory requirements.

# 3

# Modeling Variable Workload

This chapter introduces two central to this thesis concepts:

- *Variability Characterization Curves (VCCs)* ; and
- *Workload Transformations*.

VCCs allow to capture variability of different workload characteristics. In this chapter, we define several VCC types for modeling multimedia workloads in the system-level design context of MpSoC architectures and describe various ways to obtain VCCs.

Tightly coupled with the concept of VCCs are the workload transformations. They extend the modeling capabilities of the RTC-based Modular Performance Analysis (MPA) framework introduced in the previous chapter. This extension potentially leads to considerably tighter analytic performance bounds than those obtained using traditional workload models. This chapter provides a discussion on how the workload transformations can be optimally placed in an RTC scheduling network.

Towards the end of this chapter we present the results of an experimental study comparing the proposed VCC-based workload model with a conventional model. We also assess the quality of the VCC-based model using a system simulator. As a basis for the experimental study we consider two design problems from the area of media processors and show how they can be solved using the VCC-based model. We thereby demonstrate first applications of VCCs in the system-level design context of multimedia MpSoC architectures. Other applications of VCCs in this context will be presented in the following chapters.

**Contributions of this chapter**

- We introduce the concept of Variability Characterization Curves—a general model for compact representation of whole classes of increasing functions or sequences based on their worst-case and best-case variability.
- We propose and define VCC types for workload modeling of multimedia applications mapped onto multiprocessor heterogeneous architectures.
- We extend the existing RTC-based MPA framework with workload transformation operations which enable system-level performance analysis of heterogeneous multiprocessor architectures under workloads characterized by variability of several parameters, such as task's execution demands and I/O rates. We show how such workload transformations can be optimally used in the analysis.
- Through experiments we evaluate the VCC-based workload model. We quantify the gain from using this model by comparing it to a traditional task model widely used in the literature. We also demonstrate utility and assess the accuracy of the VCC-based model using measurements obtained from a detailed system simulator. This experimental study gives important insights on the nature of MPEG-2 video workloads and their characterization with VCCs.

**Organization of this chapter**

- Section 3.1 gives an overview of the related work
- Section 3.2 introduces the concept of VCCs and develops the necessary theoretical background.
- Section 3.3 reviews key properties of multimedia workloads and on this basis defines VCC types for multimedia workload modeling (multimedia VCCs).
- Section 3.4 introduces the concept of workload transformations and proposes several such transformations based on the multimedia VCCs defined in Section 3.3. Furthermore, this section discusses optimal placement of workload transformations in a RTC scheduling network.
- Section 3.5 elaborates on the ways to obtain VCCs.
- Section 3.6 presents results of the experimental evaluation of the VCC-based model.
- Finally, Section 3.7 concludes the chapter.

## 3.1 Related Work

This section outlines the existing approaches to model the workload for real-time scheduling and performance analysis of embedded systems.

Many results in the classical real-time scheduling theory [41, 150] are based on the task model introduced in [100] by Liu and Layland. In this model, tasks are characterized by tuples $(C_i, T_i)$, where $C_i$ is execution time of task $\tau_i$ and $T_i$ is the period with which $\tau_i$ arrives into the system; tasks are assumed to be independent and have deadlines equal to their periods. Subsequent research work mainly aimed at relaxing the assumptions about strict periodicity of task arrivals and deadlines. For instance, [97] considers tasks with arbitrary deadlines, less than their periods, whereas in the model of [92] the deadlines are greater than the task periods. The model in [96] allows periodic tasks to arrive with fixed offsets in time. In [162] tasks can have arbitrary deadlines, release jitter and bursty arrivals. Sporadic tasks are often modeled by constraining their minimum inter-arrival time [101].

To provide hard real-time guarantees workload models used in the classical real-time scheduling theory assume that every task instance requires WCET to complete. This assumption, although safe, is too pessimistic for a large class of applications characterized by high execution time variability; it may lead to poor processor utilization and, as a consequence, to designs with unreasonably high cost or power consumption or both. Different approaches addressing this problem have been reported in the literature [140]. In the sequel we focus the discussion on those approaches that further generalize the workload model.

One important direction in modeling tasks characterized by variable execution demands and irregular arrivals is to use *stochastic models*. For example, task models in [7, 71, 109, 161] specify task execution demands using probability distributions and assume periodic arrivals. Methods in [7, 161] handle sets of independent tasks, while [71, 109] consider task sets with precedence relations. *Real-Time Queuing Theory*, first introduced in [93], uses stochastic characterization for inter-arrival times, execution demands and deadlines, and relies on queuing theoretic methods for performance evaluation. These and other stochastic workload models can result in tighter analytic bounds and hence in more economical designs, but at the expense of some (usually controlled) fraction of missed deadlines. Because of this their application area is limited to *soft real-time* systems only.

Another line of research work aims at reducing the pessimism of the classical real-time task models by developing more expressive "deterministic" task models suitable for the analysis of *hard real-time* systems. Mok and Chen [117] proposes a *multiframe task model*. As its basis this model has the classical periodic task model of Liu and Layland [100]; however, it permits tasks whose WCETs may vary from one instance to another. Such a task can be represented by a set of subtasks, each characterized by its own WCET. The subtasks in the set are cyclically triggered in a predetermined order and with a time separation equal to the period of the task they represent. In [13] this multiframe model has been extended to allow for the time separation between subtask activations to be also variable (i.e. to cycle through a fixed pattern).

Baruah [11, 12] presents a *recurring real-time task model* (RRT)—a further generalization of the multiframe models. In the RRT model, a task is modeled by a set of subtasks arranged in a directed acyclic graph representing the conditional, non-deterministic behavior of the task. Each subtask is characterized by its WCET, a relative deadline and a minimum triggering separation from its direct predecessors. The whole task graph is triggered sporadically with a specified minimum time separation between the triggering of the last subtask in the graph and the triggering of the next task instance. Another workload model, also using conditional directed acyclic graphs to model tasks, is reported by Pop et al. in [127]. Instead of associating a deadline to *each* subtask in a task graph, the model in [127] associates a single deadline with the whole graph. Furthermore, it exposes the parallelism within a task for mapping on a multiprocessor architecture.

In comparison to classical task models, the RRT model offers a great flexibility in modeling variability of the execution demand and irregular inter-arrival times. This flexibility is, however, limited to *recurring* patterns. If workload bursts (characterized by periods with dense arrivals of tasks or increased execution demand or both) occur relatively seldom, then avoiding overly pessimistic results under the RRT model necessitates to consider very large task graphs, leading to inefficiency of the analysis. In other words, designers have to trade off the accuracy of the analysis for the analysis time, which for the RRT model increases exponentially with the problem size [12].

Inspired by traffic characterization models in the domain of communication networks [85], an alternative workload model generalizing many previous results, including the RRT model, has been proposed by Thiele et al. [121, 158]. The workload imposed by a task on a processor or a communication resource is abstracted by an *arrival curve* giving the maximum amount of *resources* which can be requested by the task within any time interval of a given length. In addition, this model also captures the variability of the service offered to a task: a *service curve* gives the minimum amount of resources offered to a given task on a resource within any time interval of a given length. The resulting workload model can be efficiently analyzed using the mathematical framework of *Real-Time Calculus* (RTC) [158], having its roots in the min-max algebra [10]. As the RRT model, the arrival curves allow to capture arbitrary complex patterns of inter-arrival times and execution demands; however, in contrast to the RRT model, an accurate characterization of both short-term and long-term behavior of the workload is achieved in a relatively compact form. Furthermore, the complexity of the analysis in general is not dependent on the accuracy of the workload model, and efficient approximations can be done if needed [156].

Another important feature of the workload model presented in [121, 158] is that unlike previous lines of work this model explicitly characterizes the service variability, thereby allowing to effectively abstract arbitrary complex scheduling and arbitration policies deployed on communication and computational resources, as well as such architectural features as caches, pipelines, write buffers, protocols etc. Continuing this line of work, Chakraborty and Thiele [26] proposed a new task model for streaming applications combining the concept of arrival curves with the RRT model, which may help to reduce the size of task graphs of the RRT model while modeling complex event streams.

Most of the approaches discussed so far are not concerned with modelling tasks which *asynchronously* interact while processing event streams; meaning that these approaches assume that a task producing an event (or a stream object) is never activated again before the dependent task consumes this event (and finishes its processing). Hence, these approaches are not interested in the properties of output event streams (activating the consumer tasks) and in variations of input and output rates of tasks (i.e. in the number of events consumed or produced by a task per activation). However, these properties of the workload become important in context of distributed execution platforms for stream processing applications. In this context different event streams may interact on shared resources, leading to *scheduling anomalies*: when a best-case load on one architectural component may cause a worst-case load scenario for another component [133]. In this situation it becomes important to capture in the model not only worst-case but also best-case behavior of the workload.

The importance of modeling both the worst-case and the best-case workload behavior in design context of embedded systems has been recognized in such modeling frameworks as SPI (System Property Intervals) [171, 182, 183]. In contrast to the research work on real-time scheduling mentioned above, the SPI framework has a different focus: its prime goal is modeling of heterogeneous embedded systems for their *global* performance analysis, design space exploration, optimization and synthesis. The SPI model represents a system as a network of communicating processes which allows (besides other communication modes) the asynchronous communication via unidirectional FIFO channels. In the SPI model each process is characterized by a set of *behavioral intervals* capturing worst-case and best-case values of various process properties such as execution time and the number of tokens consumed from input and produced to output channels. The SPI framework allows for refinement of this workload model through the concept of *process modes*. Following this concept, each process is associated with a set of modes, each of which is characterized by its own set of the behavioral intervals. When the model is evaluated (e.g. executed), the process may change its modes depending on, for example, input values.

Through the concepts of behavioral intervals and process modes the SPI framework can model the workload with a high accuracy. However, this requires an *explicit* specification of conditions upon which the modes are changed, and therefore significantly complicates the workload modeling process and may preclude an efficient analyzability of the model. In fact, the SPI model offers a flexible tradeoff between the accuracy of the workload model and the modeling overhead: depending on the scenario a designer may decide how many different modes to associate with a process. In the simplest case, a process may have only one mode, as it is the case, for example, in [67] where the input (output) rate of a process is specified with a single behavioral interval. This approach, however, results in overly pessimistic bounds and does not accurately capture the long-term behavior of the workload, which is important, for instance, in multimedia applications.

Another framework for analysis of system properties proposed in [23, 24, 157] is based on RTC developed in [121, 158]. In comparison to the model used in [121, 158], the workload model in [23, 24, 157] has a concept of *lower and upper* arrival and service curves which capture the best- and worst-case behavior of the workload. In addition, [23, 24, 157] enhance the analytical framework in [121, 158] with mechanisms to determine properties of the output event streams. These developments pave the way to a *modular* approach to the performance analysis [159, 160]. However, the workload model in [23, 24, 157] can model only tasks that consume and produce only one event per activation. Furthermore, for computing the output event streams it becomes necessary to convert the arrival curves expressed in terms of event-based units into equivalents expressed in resource-based units and backwards. Since this conversion is performed by scaling the curves with a constant factor corresponding to WCED for processing of one event, the execution time variability is not accounted for, resulting in overly pessimistic analytic bounds for workloads with large variations in execution demand of tasks. These limitations of the framework have been addressed in [113, 115]. The results of [113, 115] are included in this chapter. Further refinements of this workload model can be found in [166, 168, 169].

## 3.2 Variability Characterization Curves

This section defines Variability Characterization Curves (VCCs) in a generic way, i.e. without regard to any concrete system property that they characterize. It also states some properties which are common to all VCCs and which will be useful in the course of this thesis.

### 3.2.1 Definitions

Let $\mathcal{A}$ denote a set of increasing functions $A_i,\ i = 1, 2, \ldots$ and $\mathbb{T}$ denote the domain of these functions, such that $A_i : \mathbb{T} \to \mathbb{R}_{\geq 0}$. $\mathbb{T}$ can be either the set of nonnegative real numbers ($\mathbb{T} = \mathbb{R}_{\geq 0}$) or the set of nonnegative integers ($\mathbb{T} = \mathbb{Z}_{\geq 0}$).[1] In the latter case, the functions in $\mathcal{A}$ represent sequences, i.e. $A_i = \langle A_i(0), A_i(1), \ldots \rangle$. Formally, a function $f$ is *increasing* if $f(x_1) \leq f(x_2)$ for any $x_1 < x_2$. In contrast, $f$ is *strictly increasing* if $f(x_1) < f(x_2)$ whenever $x_1 < x_2$. Using this notation we define VCCs as follows.

**Def. 1:** **(Upper VCC)** *An upper VCC for the set of increasing functions $\mathcal{A}$ is an increasing function $\mathcal{V}^u_{\mathcal{A}}$ satisfying the condition*

$$A_i(t+s) - A_i(t) \leq \mathcal{V}^u_{\mathcal{A}}(s) \;\; \forall t, s \in \mathbb{T},\; \forall A_i \in \mathcal{A} \quad \textit{and} \quad \mathcal{V}^u_{\mathcal{A}}(0) = 0$$

**Def. 2:** **(Lower VCC)** *A lower VCC for the set of increasing functions $\mathcal{A}$ is an increasing function $\mathcal{V}^l_{\mathcal{A}}$ satisfying the condition*

$$A_i(t+s) - A_i(t) \geq \mathcal{V}^l_{\mathcal{A}}(s) \;\; \forall t, s \in \mathbb{T},\; \forall A_i \in \mathcal{A} \quad \textit{and} \quad \mathcal{V}^l_{\mathcal{A}}(0) = 0$$

In case $\mathbb{T} = \mathbb{R}_{\geq 0}$ we adopt the convention that any upper VCC is a left-continuous function and any lower VCC is a right-continuous function. Note that functions $A_i$ are not required to conform to this convention. They can be either left-continuous or right-continuous without restriction.

In case $\mathbb{T} = \mathbb{Z}_{\geq 0}$ the corresponding VCC is a sequence. If necessary a VCC which is a sequence can be converted into an equivalent VCC defined on $\mathbb{R}_{\geq 0}$. The conversion can be done by a continuation in $\mathbb{T}$ which respects the above convention about continuity of VCCs. More specifically, let $\mathcal{V}^u_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}$ and $\mathcal{V}^l_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}$ denote upper and lower VCCs defined on $\mathbb{Z}_{\geq 0}$. They can be converted into their continuous equivalents $\mathcal{V}^u_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}$ and $\mathcal{V}^l_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}$ as follows

$$\begin{aligned} \mathcal{V}^u_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}(s) &= \mathcal{V}^u_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}(\lceil s/\tau \rceil) && (3.1)\\ \mathcal{V}^l_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}(s) &= \mathcal{V}^l_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}(\lfloor s/\tau \rfloor) && (3.2) \end{aligned}$$

where $\tau$ denotes the desired spacing between samples of $\mathcal{V}^u_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}$ (or $\mathcal{V}^l_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}$) in $\mathbb{R}_{\geq 0}$.

A conversion in the opposite direction, i.e. $\mathcal{V}_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}} \rightarrow \mathcal{V}_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}$, is also possible:

$$\begin{aligned} \mathcal{V}^u_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}(k) &= \mathcal{V}^u_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}(k\tau) \quad k \in \mathbb{Z}_{\geq 0} && (3.3)\\ \mathcal{V}^l_{\mathcal{A}_{\mathbb{Z}_{\geq 0}}}(k) &= \mathcal{V}^l_{\mathcal{A}_{\mathbb{R}_{\geq 0}}}(k\tau) \quad k \in \mathbb{Z}_{\geq 0} && (3.4) \end{aligned}$$

**Remark on notation:** To simplify the notation, in some contexts we will skip subscript $\mathcal{A}$ in $\mathcal{V}^u_{\mathcal{A}}$ and in $\mathcal{V}^l_{\mathcal{A}}$ if it is irrelevant which function set is characterized by the VCCs, i.e. we will simply write $\mathcal{V}^u$ and $\mathcal{V}^l$ instead of $\mathcal{V}^u_{\mathcal{A}}$ and $\mathcal{V}^l_{\mathcal{A}}$.

[1] Throughout this thesis, whenever we use $\mathbb{T}$ without specifying whether $\mathbb{T} = \mathbb{R}_{\geq 0}$ or $\mathbb{T} = \mathbb{Z}_{\geq 0}$ we mean that both cases are possible, i.e. $\mathbb{T}$ is a placeholder for $\mathbb{R}_{\geq 0}$ and $\mathbb{Z}_{\geq 0}$.

From Defs. 1 and 2 it follows that a VCC is a mapping from $\mathbb{T}$ to $\mathbb{R}_{\geq 0}$. Often, however, while modeling systems with VCCs, there is a need for the inverse transformation. Such a transformation could be possible via inverse functions of VCCs. Unfortunately, since VCCs in general are not strictly increasing functions, they are not invertible in the conventional sense, i.e. for a given VCC there may not exist a function $\mathcal{V}^{-1}$ such that $\mathcal{V}^{-1}(\mathcal{V}(t)) = t$ for all $t$. Therefore, we need to introduce a notion of *pseudo-inverse functions*.

**Def. 3:** **(Pseudo-inverse of Upper VCC)** *The pseudo-inverse of an upper VCC $\mathcal{V}^u$ is the function*

$$\mathcal{V}^{u^{-1}}(v) = \sup\{t : \mathcal{V}^u(t) \leq v; t \in \mathbb{T}\} \tag{3.5}$$

**Def. 4:** **(Pseudo-inverse of Lower VCC)** *The pseudo-inverse of a lower VCC $\mathcal{V}^l$ is the function*

$$\mathcal{V}^{l^{-1}}(v) = \inf\{t : \mathcal{V}^l(t) \geq v; t \in \mathbb{T}\} \tag{3.6}$$

### 3.2.2 Properties

Several useful properties of VCCs follow from the definitions given in Section 3.2.1.

**Prop. 1:** **(Pseudo-inversion of Upper VCC)** *For any upper VCC $\mathcal{V}^u$ and its pseudo-inverse $\mathcal{V}^{u^{-1}}$ the following always holds*

$$\mathcal{V}^u(t) \leq v \Leftrightarrow \mathcal{V}^{u^{-1}}(v) \geq t \tag{3.7}$$

**Prop. 2:** **(Pseudo-inversion of Lower VCC)** *For any lower VCC $\mathcal{V}^l$ and its pseudo-inverse $\mathcal{V}^{l^{-1}}$ the following always holds*

$$\mathcal{V}^l(t) \geq v \Leftrightarrow \mathcal{V}^{l^{-1}}(v) \leq t \tag{3.8}$$

**Prop. 3:** **(Duality of Upper VCC)** *If $\mathcal{V}^u_{\mathcal{A}}$ is an upper VCC for a set of functions $\mathcal{A}$, then $\mathcal{V}^{u^{-1}}_{\mathcal{A}'}$ is a lower VCC for the set of functions $\mathcal{A}'$, where*

$$\mathcal{A}' = \{A' : A'(v) = \sup\{t : A(t) \leq v; t \in \mathbb{T}\}; A \in \mathcal{A}\}$$

**Prop. 4:** **(Duality of Lower VCC)** *If $\mathcal{V}^l_{\mathcal{A}}$ is a lower VCC for a set of functions $\mathcal{A}$, then $\mathcal{V}^{l^{-1}}_{\mathcal{A}'}$ is an upper VCC for the set of functions $\mathcal{A}'$, where*

$$\mathcal{A}' = \{A' : A'(v) = \inf\{t : A(t) \geq v; t \in \mathbb{T}\}; A \in \mathcal{A}\}$$

**Prop. 5:** **(Lower and Upper VCCs for the Same Set)** *If $\mathcal{V}^u$ and $\mathcal{V}^l$ are upper and lower VCCs, respectively, characterizing the same set of functions $\mathcal{A}$ then the following relation always holds:*

$$\mathcal{V}^u(t) \geq \mathcal{V}^l(t) \quad \forall t \in \mathbb{T} \tag{3.9}$$

**Proof.**

**Prop. 1** Define subset $S_v = \{t : \mathcal{V}^u(t) \leq v\} \subseteq \mathbb{T}$. Note that by Def. 3 $\mathcal{V}^{u^{-1}}(v) = \sup S_v$. Suppose that $\mathcal{V}^u(t) \leq v$, then $t \in S_v$. This implies that $t \leq \sup S_v$, that is $t \leq \mathcal{V}^{u^{-1}}(v)$. Thus, it follows that $\mathcal{V}^u(t) \leq v \Rightarrow \mathcal{V}^{u^{-1}}(v) \geq t$. Now suppose that $\mathcal{V}^{u^{-1}}(v) \geq t$, this is equivalent to saying that $\sup S_v \geq t$. This implies that $t \in S_v$, i.e. $t$ satisfies $\mathcal{V}^u(t) \leq v$. Thus, we have shown that $\mathcal{V}^{u^{-1}}(v) \geq t \Rightarrow \mathcal{V}^u(t) \leq v$. This proves (3.7).

**Prop. 2** Define subset $\tilde{S}_v = \{t : \mathcal{V}^l(t) \geq v\} \subseteq \mathbb{T}$. Note that by Def. 4 $\mathcal{V}^{l^{-1}}(v) = \inf \tilde{S}_v$. Suppose that $\mathcal{V}^l(t) \geq v$. This implies that $t \in \tilde{S}_v$. Hence, $t \geq \inf \tilde{S}_v$ which is equivalent to $t \geq \mathcal{V}^{l^{-1}}(v)$. Thus $\mathcal{V}^l(t) \geq v \Rightarrow \mathcal{V}^{l^{-1}}(v) \leq t$. Now suppose that $\mathcal{V}^{l^{-1}}(v) \leq t$. Then $t \geq \inf \tilde{S}_v$, which implies that $t \in \tilde{S}_v$. It follows that $t$ satisfies $\mathcal{V}^l(t) \geq v$. Thus, we have shown that $\mathcal{V}^{l^{-1}}(v) \leq t \Rightarrow \mathcal{V}^l(t) \geq v$. This proves (3.8).

□

Two important relations directly follow from Prop. 1 and Prop. 2. Namely, by letting $\mathcal{V}^u(t) = v$ in Prop. 1 we get

$$\mathcal{V}^{u^{-1}}(\mathcal{V}^u(t)) \geq t \tag{3.10}$$

Similarly, Prop. 2 implies that

$$\mathcal{V}^{l^{-1}}(\mathcal{V}^l(t)) \leq t \tag{3.11}$$

If $\mathcal{V}^u_{\mathcal{A}}$ and $\mathcal{V}^l_{\mathcal{A}}$ characterize the same function set $\mathcal{A}$, from Props. 3–5 we can conclude that

$$\mathcal{V}^{l^{-1}}_{\mathcal{A}}(t) \geq \mathcal{V}^{u^{-1}}_{\mathcal{A}}(t) \quad \forall t \in \mathbb{T} \tag{3.12}$$

Furthermore, if we combine (3.10) and (3.11) with (3.12) we obtain

$$\begin{aligned} \mathcal{V}^{l^{-1}}_{\mathcal{A}}(\mathcal{V}^u_{\mathcal{A}}(t)) &\geq t \quad \forall t \in \mathbb{T} &(3.13)\\ \mathcal{V}^{u^{-1}}_{\mathcal{A}}(\mathcal{V}^l_{\mathcal{A}}(t)) &\leq t \quad \forall t \in \mathbb{T} &(3.14) \end{aligned}$$

**Remark on notation:** Sometimes we will denote a pair of a lower and an upper VCCs characterizing the same function set $\mathcal{A}$ by tuple $\mathcal{V}_{\mathcal{A}} = (\mathcal{V}^l_{\mathcal{A}}, \mathcal{V}^u_{\mathcal{A}})$.

### 3.2.3 Discussion

Suppose $\mathcal{A}$ is a set of increasing functions and $\mathcal{V}_{\mathcal{A}}$ is a VCC corresponding to it, then we say that $\mathcal{V}_{\mathcal{A}}$ *characterizes* $\mathcal{A}$. More precisely, $\mathcal{V}_{\mathcal{A}}$ bounds the variability of functions in $\mathcal{A}$. In particular, an upper VCC represents an upper bound on the maximum change in function value that can occur on any interval of a given length for any function in $\mathcal{A}$. Correspondingly, a lower VCC is a lower bound on the minimum change in function value that can occur on any interval of a given length. Thus, VCCs characterize *worst- and best-case variability* of the functions or sequences.

Le Boudec and Thiran [85] presents the concept of *arrival and service curves* used for characterization of traffic and nodes (e.g. routers) in communication networks. These arrival and service curves respectively characterize timing properties of data flows and the service offered to these flows on the network nodes (*service flows*). Such arrival and service curves represent concrete VCC types. In other words, in this thesis we generalize the concept of arrival and service curves to characterize any other properties of abstract sequences, and not only the timing of the data and service flows. However, the theory developed in [85] is applicable to a large extent to the generic VCCs defined in this section. More basic theory which may also be useful for system analysis with VCCs can be found in [10]. For this reason, we limited the discussion in this section to those theoretical concepts that are either new or differ in some way from those described in [10, 85]. This means that in the course of this thesis we may rely on some mathematical background which is not covered in this section but can be found in [10, 85].

Finally, before defining concrete VCC types for modeling of multimedia workloads (which we will do in the next section), we would like to point out an important property of any VCC, irrespective of its particular type: A single VCC can serve as a compact *abstraction* of a whole *class* of sequences or functions with similar worst-case or best-case variability. Therefore, VCCs represent an attractive means for capturing various aspects of system behavior in the design context of heterogeneous embedded systems where such abstractions are needed to tame the complexity of the design problems.

## 3.3 Variability Characterization Curves for Modeling Multimedia Workload

While the previous section presents the VCC concept in a generic way, this section introduces several concrete VCC types which enable effective modeling of multimedia workloads [113, 115]. These VCC types are useful in system-level design of embedded systems whose functionality involves distributed real-time processing of digital media streams. Later in this thesis, we demonstrate several applications of these VCC types to the performance evaluation and scheduling of multimedia MpSoC architectures, which represent such embedded systems.

In workload modeling there are two generally conflicting goals [12]:

- First, a workload model should be as general as possible such that it could accurately capture all relevant properties of the workload.

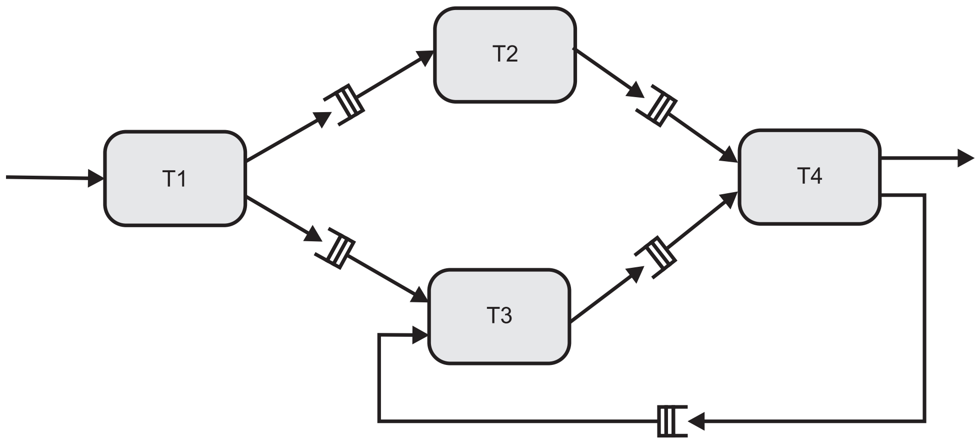


**Fig. 7:** Example task graph of a multimedia application.

- Second, the model should be efficiently analyzable to be useful in the design process.

The former concern is addressed in this section. To address the latter concern, we rely on already existing theoretical framework of Real-Time Calculus [120, 157, 158], which has been shown to be efficient in system-level performance analysis of network processors [23, 25]. In Section 3.4, using the VCC types introduced in this section, we extend the RTC framework with the concept of *workload transformations*, which are simple operations, having no significant impact on the efficiency of the whole analytical framework.

### 3.3.1 Execution model

In Section 2.2.1, we have mentioned that performance characteristics of a system are determined by the temporal interactions between the *requested service* and the corresponding *offered service* (i.e. between event and resource streams in the terminology of the MPA framework). The temporal characteristics of the requested and offered service and their interactions are tightly coupled with the *execution model* used to implement an application on the architecture. Hence, to understand the properties of multimedia workloads that might influence these temporal characteristics, we first need to define the execution model we assume for the multimedia applications mapped on to the multiprocessor architectures.

A multimedia application is partitioned into a number of concurrent (computation and communication) tasks which are assigned for execution to different PEs of the target execution platform. As mentioned in Chapters 1 and 2, we assume that the application tasks communicate solely via unidirectional data streams, which we can see as FIFO channels. A multimedia task may read and write data from (to) several such channels. To illustrate this, Fig. 7 shows a task graph of a hypothetical multimedia application.

We refer to the elementary data unit that a particular task can read from (write to) a particular channel as *stream object*. A stream object might be a bit belonging to a compressed bitstream representing a coded video clip, or a macroblock, or a video frame, or an audio sample — depending on where in the task graph the corresponding stream exists.

A task becomes active whenever there is a predetermined amount of stream objects available in the incoming channels. There may be some activation rules associated with the task [181]. These rules define specific conditions on the incoming channels upon which this task is activated (e.g. availability of a specified number of stream objects in *all* or only in *some* incoming channels).

Whenever the task is activated it requests from the PE on which it executes a certain amount of processor cycles. The task also consumes from the incoming channels a number of stream objects. We assume that this number equals to the amount of stream objects that were necessary to trigger the task. While executing (i.e. consuming the processor cycles) the task may write data into outgoing channels. We assume non-blocking writing.

Finally, a sequence of task activations can be abstractly seen as a *request stream*; and the availability of the PE for processing this stream can be seen as a *resource stream*.

## 3.3.2 Definitions of multimedia VCC types

The key workload characteristics to be captured in the workload model are those characteristics that affect the timing properties of the request and resource streams and their interactions. In this subsection, we pinpoint these key characteristics and define VCC types to characterize them. Some of the VCC types that will be introduced below are indicated in Fig. 8, which shows an abstract view of a multimedia task.

**Event-based arrival curves**

The timing of task's requests for service is influenced by the timing of arrivals of stream objects into the incoming channels of this task. These arrivals depend on the tasks writing into these incoming channels (or on the system's environment, if the stream is received from outside of the system). Each arrival into a channel can be modeled as an event. Hence, a sequence of arrivals into a given channel can be represented as an event stream. Similar to the characterization model described in Section 2.2.2, we model the timing properties of such event streams with VCCs called *event-based arrival curves* [85, 158].

**Def. 5:** **(Event-Based Arrival Curves $\bar{\alpha} = (\bar{\alpha}^l, \bar{\alpha}^u)$)** *A lower event-based arrival curve $\bar{\alpha}^l$ and an upper event-based arrival curve $\bar{\alpha}^u$ are VCCs characterizing timing properties of a given event stream. $\bar{\alpha}^l(\Delta)$ bounds from below and $\bar{\alpha}^u(\Delta)$ from above the number of events that can occur in the stream within any time interval of length $\Delta$.*

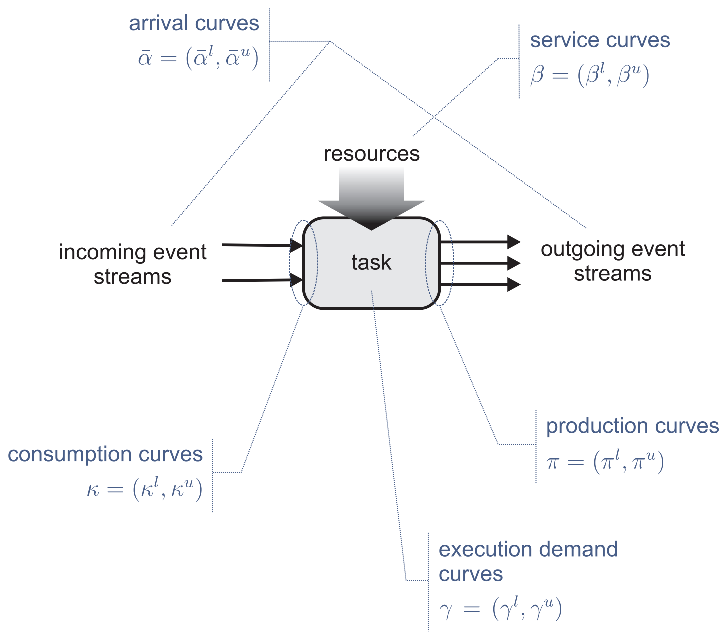


**Fig. 8:** Overview: VCCs for modeling multimedia workloads.

**Consumption curves**
Another factor influencing the timing of task's requests for service is the amount of stream objects that must be available in a given incoming channel for activation of the task. This amount may vary from activation to activation, e.g. due to data-dependent behavior of the task. Successive activations of the task result in a sequence of numbers representing the amounts of stream objects consumed by the task from the given incoming channel at each activation. As an abstraction of such sequences we use VCCs called *consumption curves* [115].

**Def. 6:** **(Consumption Curves $\kappa = (\kappa^l, \kappa^u)$)** *A lower consumption curve $\kappa^l$ and an upper consumption curve $\kappa^u$ are VCCs characterizing the relation between task's activations and the number of stream objects consumed by the task from a given input channel. $\kappa^l(k)$ bounds from below and $\kappa^u(k)$ from above the number of task activations needed to consume $k$ consecutive stream objects from the input channel.*

Besides the amount of stream objects that must be present in each individual channel to activate the task, the activation rules may impose some conditions on the set of incoming channels as a whole. The implications of such rules can also be reflected in the consumption curves. Alternatively, such rules can be accounted for at the higher modeling level, i.e. in the performance model itself. [68] reports some work in this direction.

**Execution demand curves**
The amount of execution resources requested by a task at each activation may vary depending, for example, on the task's current state and the values of processed stream objects, or on such architectural features as caches, branch predictors and pipelines. A sequence of task activations results in a sequence of execution demands imposed by the task on the PE. For modeling such sequences we use *execution demand curves* [113, 115].

**Def. 7:** **(Execution Demand Curves $\gamma = (\gamma^l, \gamma^u)$)** *A lower execution demand curve $\gamma^l$ and an upper execution demand curve $\gamma^u$ are VCCs characterizing execution demand of a given task. $\gamma^u(k)$ bounds from above and $\gamma^l(k)$ from below the amount of resource units (such as processor cycles) needed to complete any $k$ consecutive task executions.*

**Production curves**
The timing properties of event streams generated by a task at its outputs depend on the amount of stream objects produced by the task into the outgoing channels at each activation. Again, from activation to activation this amount may vary. To characterize this aspect of task's activation sequences, we employ VCCs called *production curves* [115].

**Def. 8:** **(Production Curves $\pi = (\pi^l, \pi^u)$)** *A lower production curve $\pi^l$ and an upper production curve $\pi^u$ are VCCs characterizing the relation between task's executions and the number of stream objects produced by the task into a given output channel. $\pi^l(k)$ bounds from below and $\pi^u(k)$ from above the number of stream objects produced by the task into the output channel as a result of $k$ consecutive executions of this task.*

**Resource-based service curves**
An important factor largely influencing the performance characteristics is availability of execution resources for a given task over the time. Several tasks mapped onto a single PE may compete for the execution resource of this PE. A scheduling or arbitration policy then determines in which order these tasks are to be executed on the PE. As a consequence, the supply of resources to the tasks is not anymore uniformly distributed in time. This variability of the PE's processing capacity as seen by an individual task is characterized using *resource-based service curves* [113, 115].

**Def. 9:** **(Resource-Based Service Curves $\beta = (\beta^l, \beta^u)$)** *A lower resource-based service curve $\beta^l$ and an upper resource-based service curve $\beta^u$ are VCCs characterizing the service offered to a given task on a given execution resource. $\beta^l(\Delta)$ bounds from below the number of resource units (such as processor cycles) that are* guaranteed *to be provided to the task within any time interval of length $\Delta$. $\beta^u(\Delta)$*

*bounds from above the number of resource units that* can *be provided to the task within any time interval of length* $\Delta$*.*

**Type rate curves**
In some cases, for the performance analysis it is useful to distinguish within an event stream (or any other sequence) different event types. For instance, this might be useful in case if different event types impose different execution demands on a PE. The different event types may follow in various patterns within the stream. For the characterization of these patterns we employ so called *type rate curves* [166].

**Def. 10:** **(Type Rate Curves** $\vartheta = (\vartheta^l, \vartheta^u)$**)** *A lower type rate curve* $\vartheta^l$ *and an upper type rate curve* $\vartheta^u$ *are VCCs characterizing the containment of a given event type in a given event stream.* $\vartheta^l(k)$ *bounds from below and* $\vartheta^u(k)$ *from above the number of events of the given type in any subsequence of* $k$ *consecutive events within the event stream.*

**Event-based service curves and resource-based arrival curves**
In contrast to the resource-based service curves, the event-based service curves express the availability of a PE for a given task in terms of the *task executions* instead of processor cycles. Similarly, in contrast to the event-based arrival curves, the resource-based arrival curves express the amount of *resources* (e.g. processor cycles) requested by a task.

**Def. 11:** **(Event-Based Service Curves** $\bar{\beta} = (\bar{\beta}^l, \bar{\beta}^u)$**)** *A lower event-based service curve* $\bar{\beta}^l$ *and an upper event-based service curve* $\bar{\beta}^u$ *are VCCs characterizing the service offered to a given task on a given execution resource.* $\bar{\beta}^l(\Delta)$ *bounds from below the number of task executions that are* guaranteed *to be completed within any time interval of length* $\Delta$*.* $\bar{\beta}^u(\Delta)$ *bounds from above the number of task executions that* can *be completed within any time interval of length* $\Delta$*.*

**Def. 12:** **(Resource-Based Arrival Curves** $\alpha = (\alpha^l, \alpha^u)$**)** *A lower resource-based arrival curve* $\alpha^l(\Delta)$ *and an upper resource-based arrival curve* $\alpha^u(\Delta)$ *are VCCs characterizing the service requested by a given task from a given execution resource.* $\alpha^l(\Delta)$ *bounds from below and* $\alpha^u(\Delta)$ *from above the number of* resource units *(such as processor cycles) that may be requested by the task within any time interval of length* $\Delta$*.*

## 3.4 Workload Transformations

Being able to analytically compute different performance figures of a system requires both the requested and the offered service being expressed in common units. This can be achieved through a mapping of one service representation to another. In simple cases, such a mapping can be done in a straightforward way via scaling with a constant factor [24]. In more complex cases, where the workload variability cannot be neglected, this approach does not work well: it often results in overly pessimistic performance bounds. In this thesis, we propose a more sophisticated mapping mechanism which we call *workload transformations*. It is based on the concept of VCCs. In contrast to the scaling with a constant factor, workload transformations in general are non-linear operations. This non-linearity stems from the need to properly account for the workload variability while performing the mapping between different workload and service representations. As a result, the tightness of the bounds returned by the performance analysis can be improved.

In this section, we define workload transformations and formulate general rules for applying them. We then discuss workload transformations which are useful for performance analysis of media processors. They involve the VCC types defined in Section 3.3. Toward the end of this section, we show how the modeling capabilities of the existing MPA framework can be enhanced through the use of the workload transformations.

### 3.4.1 The workload transformation operation

**Def. 13:** **(Workload Transformation)** *A workload transformation is an operation $\mathcal{V}_y(\mathcal{V}_x(t))$, where $\mathcal{V}_y$ and $\mathcal{V}_x$ are VCCs of different types, with $\mathcal{V}_y$ having as its domain the codomain of $\mathcal{V}_x$ and $t \in \mathbb{T}$. We say that $\mathcal{V}_y$* transforms *$\mathcal{V}_x$.*

For convenience, we introduce the workload transformation operator $\odot$ which we define as

$$(f \odot g)(t) = f(g(t)) \tag{3.15}$$

The operator $\odot$ has the highest precedence level and is applied from right to left, i.e.

$$(f \odot g \odot h)(t) = f(g(h(t))) \tag{3.16}$$

#### Conservative workload transformations

Not all workload transformations are suitable for worst case analysis. Such analysis has to guarantee correctness of computed upper and lower bounds. Therefore, only *conservative* workload transformations are allowed in it. This means that a workload transformation of a lower bound has to result in a valid lower bound. Similarly, a workload transformation of an upper bound has to return a valid upper bound. This can be achieved if the workload transformations satisfy either of the two following rules:

- *An upper VCC can only be transformed with another* upper *VCC.*
- *A lower VCC can only be transformed with another* lower *VCC.*

| $f \odot g$ | | $g =$ | | | |
|---|---|---|---|---|---|
| | | $\mathcal{V}^u$ | $\mathcal{V}^l$ | $\mathcal{V}^{u^{-1}}$ | $\mathcal{V}^{l^{-1}}$ |
| $f =$ | $\mathcal{V}^u$ | $\surd$ | | | $\surd$ |
| | $\mathcal{V}^l$ | | $\surd$ | $\surd$ | |
| | $\mathcal{V}^{u^{-1}}$ | | $\surd$ | $\surd$ | |
| | $\mathcal{V}^{l^{-1}}$ | $\surd$ | | | $\surd$ |

**Tab. 1:** Conservative workload transformations (marked with $\surd$).

Tab. 1 lists all possible workload transformations which satisfy the above rules.[2]

### 3.4.2 Workload transformations for multimedia VCCs

This subsection describes a set of conservative workload transformations which are defined on the multimedia VCC types proposed in Section 3.3. These transformations are needed to extend the existing RTC-based performance analysis framework introduced in Chapter 2. They significantly improve the modeling capabilities of the framework and thereby enable an accurate performance analysis of distributed execution platforms under multimedia workloads characterized by the variability of several parameters. This subsection explains the role of each individual transformation within the RTC-based performance analysis framework.

Suppose that different properties of the system whose performance we wish to analyze are specified using a subset of VCCs which were defined in Section 3.3. For the performance analysis, we need to build a performance model out of these VCCs. The first step in constructing such a model is to quantify the requested and the offered service in some common units. The choice of units depends on particular goals of the analysis and on the input specification. As we will see later in this chapter (in Section 3.4.3), within a single performance model we may need to carry out some computations using resource-based units and other computations using event-based units. Hence, a variety of workload transformations may be required during constructing a performance model. In what follows, we list and explain different sorts of workload transformations which can be useful in the performance analysis.

#### Input stream arrivals $\longrightarrow$ task activations

This workload transformation maps the arrival process of a stream to the activation process of the task which processes this stream. The timing of the activation process depends on the timing of the stream arrivals. This is because, according

[2]To correctly interpret Tab. 1, recall Props. 3 and 4 shown in Section 3.2, stating that the pseudo-inverse of an upper VCC is a lower VCC, and the pseudo-inverse of a lower VCC is an upper VCC.

to our model of computation, a task is activated only if the number of stream objects present at its input is larger or equal to the amount of stream objects that the task will consume during its next execution. The stream's arrival process is specified by event-based arrival curves $(\bar{\alpha}_I^l, \bar{\alpha}_I^u)$. The goal of the workload transformation is to obtain arrival curves $(\bar{\alpha}_{ta}^l, \bar{\alpha}_{ta}^u)(\Delta)$ that represent upper and lower bounds on the number of tasks activations which may occur in any time interval $\Delta$. The workload transformation can be performed using consumption curves $(\kappa^l, \kappa^u)$ as follows.

$$\bar{\alpha}_{ta}^u(\Delta) = (\kappa^u \odot \bar{\alpha}_I^u)(\Delta) \tag{3.17}$$
$$\bar{\alpha}_{ta}^l(\Delta) = (\kappa^l \odot \bar{\alpha}_I^l)(\Delta) \tag{3.18}$$

**Task activations $\longrightarrow$ execution demand**
This workload transformation maps the activation process of a task to the execution demand imposed by this task on the resource on which it executes. The amount of the imposed execution demand, measured in resource-based units per time interval, depends on the timing of the task activations and on the execution demand imposed by each individual activation. The goal of this workload transformation is to obtain arrival curves $(\alpha_I^l, \alpha_I^u)(\Delta)$ which represent upper and lower bounds on the number of resource units that can be requested by the task within any time interval $\Delta$. Suppose that the timing of task activations is given by arrival curves $(\bar{\alpha}_{ta}^l, \bar{\alpha}_{ta}^u)$ as defined by (3.17) and (3.18). Then using execution demand curves $(\gamma^l, \gamma^u)$ we can transform $(\bar{\alpha}_{ta}^l, \bar{\alpha}_{ta}^u)$ into $(\alpha_I^l, \alpha_I^u)$ as follows.

$$\alpha_I^u(\Delta) = (\gamma^u \odot \bar{\alpha}_{ta}^u)(\Delta) \tag{3.19}$$
$$\alpha_I^l(\Delta) = (\gamma^l \odot \bar{\alpha}_{ta}^l)(\Delta) \tag{3.20}$$

**Input stream arrivals $\longrightarrow$ execution demand**
Combining (3.17) with (3.19) and (3.18) with (3.20) gives a mapping of stream arrivals expressed in number of stream objects (i.e. in event-based quantities) per time interval on to the execution demand which is imposed by this stream on the execution resource. The execution demand is then expressed in resource-based quantities per time interval.

$$\alpha_I^u(\Delta) = (\gamma^u \odot \kappa^u \odot \bar{\alpha}_I^u)(\Delta) \tag{3.21}$$
$$\alpha_I^l(\Delta) = (\gamma^l \odot \kappa^l \odot \bar{\alpha}_I^l)(\Delta) \tag{3.22}$$

The resulting resource-based arrival curves $(\alpha_I^l, \alpha_I^u)$ can be combined in RTC computations with resource-based service curves $(\beta^l, \beta^u)$. Note that if a task always consumes only one stream object per one execution, i.e. if $\kappa^u(k) = \kappa^l(k) = k$, then (3.21) and (3.22) can be reduced to

$$\alpha_I^u(\Delta) = (\gamma^u \odot \bar{\alpha}_I^u)(\Delta) \tag{3.23}$$
$$\alpha_I^l(\Delta) = (\gamma^l \odot \bar{\alpha}_I^l)(\Delta) \tag{3.24}$$

**Consumed resources $\longrightarrow$ task executions**

Suppose we know resource-based arrival curves $(\alpha^l_O, \alpha^u_O)(\Delta)$ which represent the upper and lower bounds on the amount of resource units that a task may consume within any time interval $\Delta$. The goal of this workload transformation is to convert these arrival curves into arrival curves $(\alpha^l_{te}, \alpha^u_{te})$ that represent the upper and lower bounds on the number of task executions that could be performed for the given amount of resources modeled by $(\alpha^l_O, \alpha^u_O)(\Delta)$. This workload transformation can be performed through pseudo-inverses of the execution demand curves as follows.

$$\bar{\alpha}^u_{te}(\Delta) = (\gamma^{l^{-1}} \odot \alpha^u_O)(\Delta) \tag{3.25}$$
$$\bar{\alpha}^l_{te}(\Delta) = (\gamma^{u^{-1}} \odot \alpha^l_O)(\Delta) \tag{3.26}$$

**Task executions $\longrightarrow$ output stream arrivals**

This workload transformation maps the execution process of a task to the arrival process of a stream produced by this task. The timing of the stream arrivals depends on the timing of task executions (since the task can produce stream objects only as a result of its execution). In addition, the amount of produced stream objects may vary from execution to execution. The goal of this workload transformation is to obtain event-based arrival curves of the stream at the output of the task, i.e. $(\bar{\alpha}^l_O, \bar{\alpha}^u_O)$. Let $(\bar{\alpha}^l_{te}, \bar{\alpha}^u_{te})(\Delta)$ denote arrival curves that represent upper and lower bounds on the number of the task executions that can occur in any time interval $\Delta$. Then, using production curves $(\pi^l, \pi^u)$, we can perform the required workload transformation as follows.

$$\bar{\alpha}^u_O(\Delta) = (\pi^u \odot \bar{\alpha}^u_{te})(\Delta) \tag{3.27}$$
$$\bar{\alpha}^l_O(\Delta) = (\pi^l \odot \bar{\alpha}^l_{te})(\Delta) \tag{3.28}$$

**Consumed resources $\longrightarrow$ output stream arrivals**

Combining (3.25) with (3.27) and (3.26) with (3.28) gives a mapping of the amount of resources consumed by a task per time interval to the number of stream objects produced by this task per time interval. This workload transformation is performed as follows.

$$\bar{\alpha}^u_O(\Delta) = (\pi^u \odot \gamma^{l^{-1}} \odot \alpha^u_O)(\Delta) \tag{3.29}$$
$$\bar{\alpha}^l_O(\Delta) = (\pi^l \odot \gamma^{u^{-1}} \odot \alpha^l_O)(\Delta) \tag{3.30}$$

Note that if a task always produces only one stream object per one execution, i.e. if $\pi^u(k) = \pi^l(k) = k$, then (3.29) and (3.30) can be reduced to

$$\bar{\alpha}^u_O(\Delta) = (\gamma^{l^{-1}} \odot \alpha^u_O)(\Delta) \tag{3.31}$$
$$\bar{\alpha}^l_O(\Delta) = (\gamma^{u^{-1}} \odot \alpha^l_O)(\Delta) \tag{3.32}$$

**Offered resources $\longrightarrow$ task executions**

The goal of this workload transformation is to map the amount of resources which a task is offered per time interval to the number of executions that this task can perform per time interval if it uses the offered amount of resources. The offered amount of resources is modeled by resource-based service curves $(\beta^l, \beta^u)$. The number of executions which can be performed for the offered number of resources per time interval is modeled by event-based service curves $(\bar{\beta}^l, \bar{\beta}^u)$. The workload transformation can be achieved using pseudo-inverses of the execution demand curves as follows.

$$\bar{\beta}^u(\Delta) = (\gamma^{l^{-1}} \odot \beta^u)(\Delta) \tag{3.33}$$

$$\bar{\beta}^l(\Delta) = (\gamma^{u^{-1}} \odot \beta^l)(\Delta) \tag{3.34}$$

**Task executions $\longrightarrow$ offered resources**

The goal of this workload transformation is to map the service expressed in number of task executions (i.e. in event-based quantities) which can be performed per time interval on to the amount of resources which are necessary per time interval in order to offer this service. The number of task executions *offered* per time interval is modeled by event-based service curves $(\bar{\beta}^l, \bar{\beta}^u)$, and the amount of resource units offered per time interval is modeled by resource-based service curves $(\beta^l, \beta^u)$. The workload transformation is performed via execution demand curves $(\gamma^l, \gamma^u)$ as follows.

$$\beta^u(\Delta) = (\gamma^u \odot \bar{\beta}^u)(\Delta) \tag{3.35}$$

$$\beta^l(\Delta) = (\gamma^l \odot \bar{\beta}^l)(\Delta) \tag{3.36}$$

### 3.4.3 Extended Modular Performance Analysis Framework

This subsection explains how the MPA framework described in Section 2.2 can be extended with the workload transformations introduced in the previous subsection.

The workload transformations can be applied at different points within a scheduling network. Since the transformations must be conservative, each transformation is typically associated with some loss of accuracy of the bounds. Hence, as a rule of thumb, the total number of workload transformations within a scheduling network should be minimized. There is, however, a more specific guideline for the optimal placement of the workload transformations within a scheduling network. We elaborate on it now.

If we use Real-Time Calculus to analyze a scheduling network then all event and resource flows in the network are modeled with arrival and service curves. In general, we are free to choose whether an arrival (service) curve existing between any two performance components is expressed in event-based or resource-based

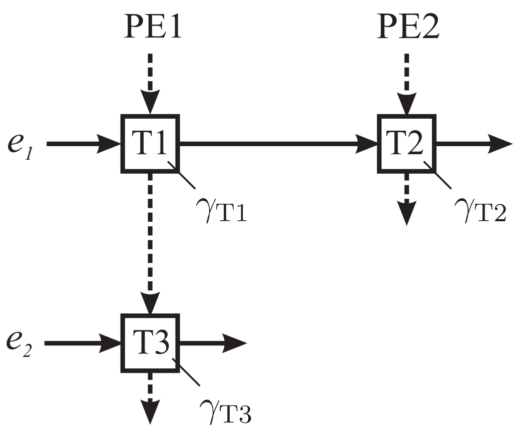


**Fig. 9:** Scheduling network used in Ex. 2.

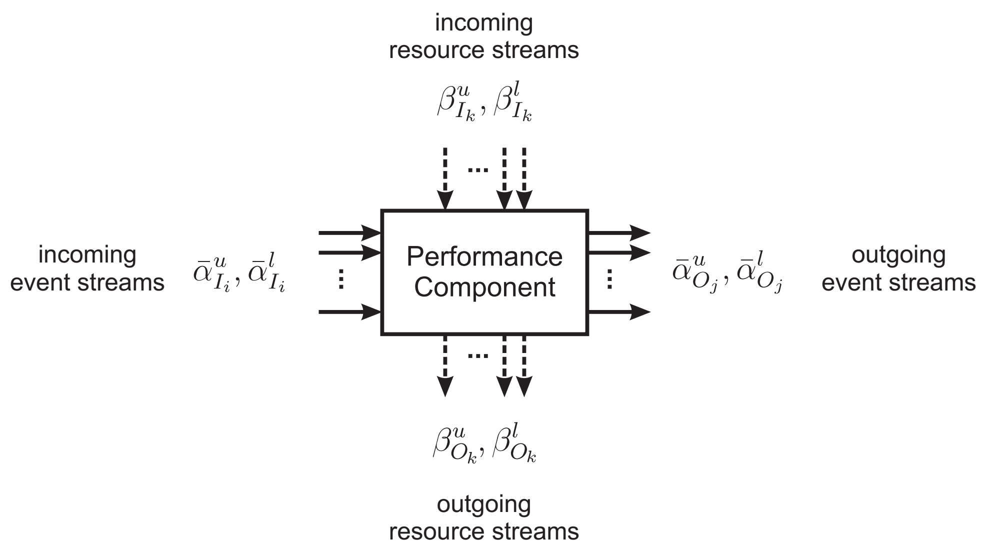
units. Both representations are possible. We can use this freedom of choosing the representation for minimization of the total number of workload transformations. Ex. 2 illustrates situations in which the workload transformations are needed and in which they can be avoided.

**Ex. 2:** *Consider a scheduling network shown in Fig. 9 It consists of three performance components that correspond to three tasks* $\mathrm{T1}$, $\mathrm{T2}$ *and* $\mathrm{T3}$. $\mathrm{T1}$ *and* $\mathrm{T2}$ *process event flow* $e_1$ *with* $\mathrm{T1}$ *being the first in the processing chain.* $\mathrm{T3}$ *processes event flow* $e_2$. $\mathrm{T1}$ *and* $\mathrm{T3}$ *are mapped to processing element* $\mathrm{PE1}$, *and* $\mathrm{T2}$ *is mapped to* $\mathrm{PE2}$. *Execution demand curves* $\gamma_{\mathrm{T1}}$, $\gamma_{\mathrm{T2}}$ *and* $\gamma_{\mathrm{T3}}$ *model the execution demand imposed by* $e_1$ *and* $e_2$ *on* $\mathrm{PE1}$ *and* $\mathrm{PE2}$. *Assume that all tasks per one execution consume and produce only one stream object (i.e. consumption and production curves can be omitted from the analysis). In this setup, consider the following cases:*

1. *Suppose that at* $\mathrm{T1}$*'s input* $e_1$ *is specified by an event based arrival curve and* $\mathrm{PE1}$*'s resource flow is specified by a resource based service curve. Clearly, in this case in order to analyze* $\mathrm{T1}$ *a workload transformation is unavoidable. Either the arrival curve of* $e_1$ *has to be transformed into its resource-based equivalent or the service curve of* $\mathrm{PE1}$ *must be transformed into an event-based representation.*

2. *Consider the* event *stream at the* output *of* $\mathrm{T1}$. *Its arrival curve can be expressed either in resource-based or in event-based units, depending on the way it was obtained during analysis of* $\mathrm{T1}$. *Assume that this is a resource-based arrival curve. Only if* $\gamma_{\mathrm{T1}} = \gamma_{\mathrm{T2}}$ *and* $\mathrm{PE1}$ *and* $\mathrm{PE2}$ *provide the same amount of resources per time unit, no transformation is needed on the resource-based arrival curve existing between* $\mathrm{T1}$ *and* $\mathrm{T2}$.

3. *Consider the* resource *stream at the* output *of* $\mathrm{T1}$. *Its service curve also can be expressed either in resource-based or in event-based units, depending on the way*



incoming resource streams
$\beta^u_{I_k}, \beta^l_{I_k}$
incoming event streams
$\bar{\alpha}^u_{I_i}, \bar{\alpha}^l_{I_i}$
Performance Component
$\bar{\alpha}^u_{O_j}, \bar{\alpha}^l_{O_j}$
outgoing event streams
$\beta^u_{O_k}, \beta^l_{O_k}$
outgoing resource streams

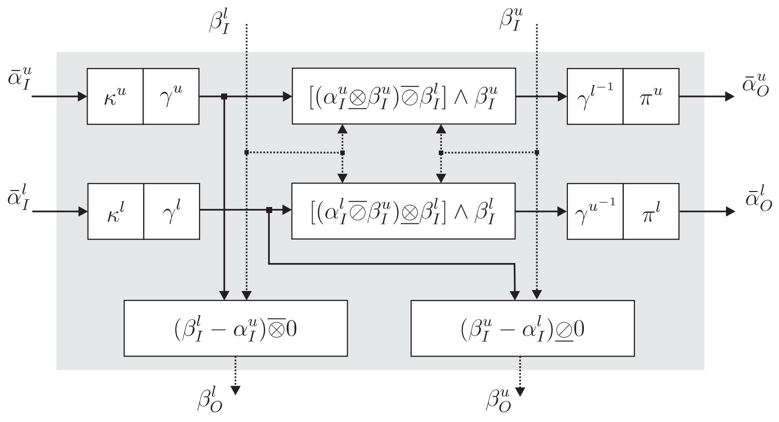
**Fig. 10:** High-level view of a performance component with the RTC interface. All incoming and outgoing event streams are modeled by *event-based* arrival curves, while all incoming and outgoing resource streams are modeled by *resource-based* service curves.

*it was obtained during analysis of* $\mathrm{T1}$. *Assume that this is an event-based service curve. Only if* $\gamma_{\mathrm{T1}} = \gamma_{\mathrm{T3}}$, *no transformation is needed on the event-based service curve existing between* $\mathrm{T1}$ *and* $\mathrm{T3}$.

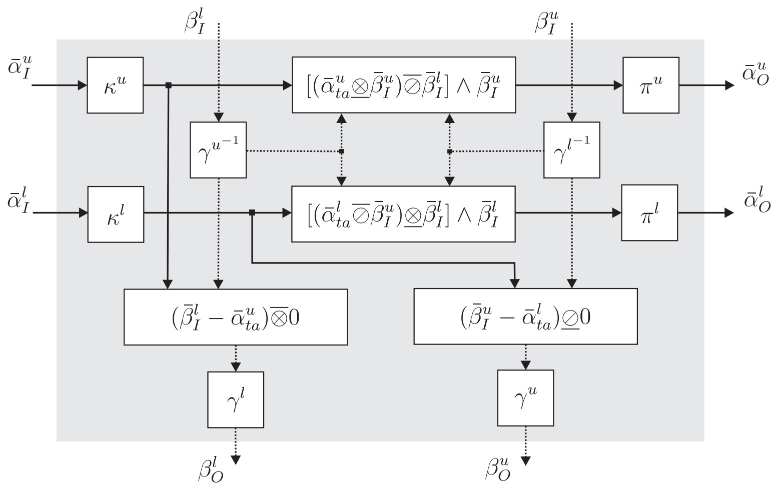
In Ex. 2, in cases 2) and 3) we were able to avoid some workload transformations. This was only possible under special conditions. Unfortunately, in practice embedded systems can rarely satisfy such conditions because typically their architectures are heterogeneous and execution demands imposed on these architectures by tasks and event streams are divergent. This implies that typically in an RTC scheduling network all event streams between performance components will be modeled by *event-based arrival curves* and all resource streams will be modeled by *resource-based service curves*. Fig. 10 shows a high-level view of a performance component that has the corresponding RTC interface.

Having defined the interface of a typical performance component in an RTC scheduling network, we can now look inside the component to figure out how the interface can be optimally implemented. The optimal implementation is one that results in the least loss in accuracy of arrival and service curves at the output of the component after its evaluation.

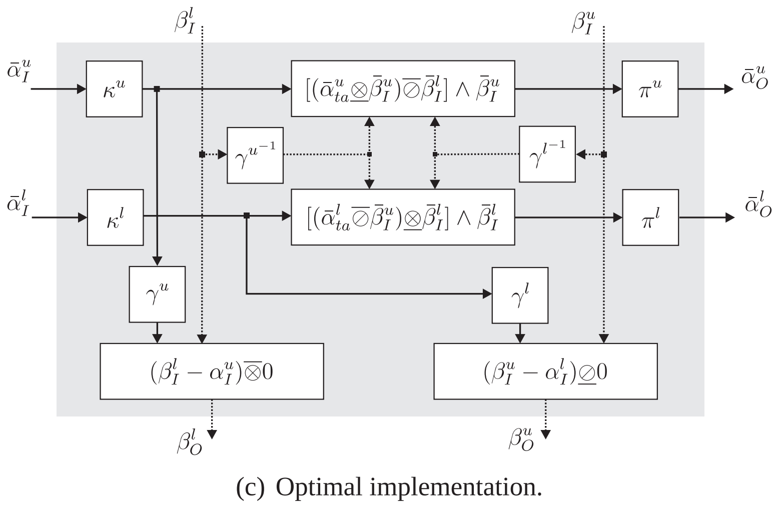
Fig. 11 shows three possible implementations of an RTC performance component which involves workload transformations. The implementation in Fig. 11(a) performs all RTC computations on resource-based curves. For this, the input event-based arrival curves $\bar{\alpha}^u_I$ and $\bar{\alpha}^l_I$ are first converted into the corresponding resource-based arrival curves $\alpha^u_I$ and $\alpha^l_I$ using (3.21) and (3.22). Then the RTC computations are carried out. After computing the output resource-based arrival


$\beta_I^l$
$\beta_I^u$
$\bar{\alpha}_I^u$
$\kappa^u$
$\gamma^u$
$[(\alpha_I^u \underline{\otimes} \beta_I^u)\overline{\oslash}\beta_I^l] \wedge \beta_I^u$
$\gamma^{l^{-1}}$
$\pi^u$
$\bar{\alpha}_O^u$
$\bar{\alpha}_I^l$
$\kappa^l$
$\gamma^l$
$[(\alpha_I^l \overline{\oslash} \beta_I^u)\underline{\otimes}\beta_I^l] \wedge \beta_I^l$
$\gamma^{u^{-1}}$
$\pi^l$
$\bar{\alpha}_O^l$
$(\beta_I^l - \alpha_I^u)\overline{\otimes}0$
$(\beta_I^u - \alpha_I^l)\underline{\oslash}0$
$\beta_O^l$
$\beta_O^u$

(a) Double transformation of arrival curves.


$\beta_I^l$
$\beta_I^u$
$\bar{\alpha}_I^u$
$\kappa^u$
$[(\bar{\alpha}_{ta}^u \underline{\otimes} \bar{\beta}_I^u)\overline{\oslash}\bar{\beta}_I^l] \wedge \bar{\beta}_I^u$
$\pi^u$
$\bar{\alpha}_O^u$
$\gamma^{u^{-1}}$
$\gamma^{l^{-1}}$
$\bar{\alpha}_I^l$
$\kappa^l$
$[(\bar{\alpha}_{ta}^l \overline{\oslash} \bar{\beta}_I^u)\underline{\otimes}\bar{\beta}_I^l] \wedge \bar{\beta}_I^l$
$\pi^l$
$\bar{\alpha}_O^l$
$(\bar{\beta}_I^l - \bar{\alpha}_{ta}^u)\overline{\otimes}0$
$(\bar{\beta}_I^u - \bar{\alpha}_{ta}^l)\underline{\oslash}0$
$\gamma^l$
$\gamma^u$
$\beta_O^l$
$\beta_O^u$

(b) Double transformation of service curves.


$\beta_I^l$
$\beta_I^u$
$\bar{\alpha}_I^u$
$\kappa^u$
$[(\bar{\alpha}_{ta}^u \underline{\otimes} \bar{\beta}_I^u)\overline{\oslash}\bar{\beta}_I^l] \wedge \bar{\beta}_I^u$
$\pi^u$
$\bar{\alpha}_O^u$
$\gamma^{u^{-1}}$
$\gamma^{l^{-1}}$
$\bar{\alpha}_I^l$
$\kappa^l$
$[(\bar{\alpha}_{ta}^l \overline{\oslash} \bar{\beta}_I^u)\underline{\otimes}\bar{\beta}_I^l] \wedge \bar{\beta}_I^l$
$\pi^l$
$\bar{\alpha}_O^l$
$\gamma^u$
$\gamma^l$
$(\beta_I^l - \alpha_I^u)\overline{\otimes}0$
$(\beta_I^u - \alpha_I^l)\underline{\oslash}0$
$\beta_O^l$
$\beta_O^u$

(c) Optimal implementation.

**Fig. 11:** Three possible ways to implement workload transformations within an elementary RTC performance component.

curves $\alpha_O^u$ and $\alpha_O^l$, we need to transform them back to their event-based representations ($\bar{\alpha}_O^u$ and $\bar{\alpha}_O^l$) for exposing them on the component's interface. We accomplish this with (3.29) and (3.30). Thus, in the scheme in Fig. 11(a) we apply the workload transformations twice to the event flow.

The implementation in Fig. 11(b) performs all RTC computations on event-based curves. This necessitates to apply the workload transformations twice to the resource flow, i.e. to the service curves. (The required workload transformations are performed using (3.33)–(3.36) as indicated in Fig. 11(b).)

Both implementations, the one depicted in Fig. 11(a) and the one depicted in Fig. 11(b), have a common problem—a loss of accuracy caused by the *double workload transformation* applied to the arrival (or service) curves through execution demand curves $\gamma^u$ and $\gamma^l$.

To understand the problem, consider the workload transformations described by (3.21) and (3.29). For the sake of simplicity, assume that $\kappa^u(k) = k$ and $\pi^u(p) = p$ and that there is no backlogged events at the component's input. Suppose that on some interval $\Delta$ all $\bar{\alpha}_I^u$ $(\Delta)$ events arrive at the component's input and that they all impose the worst-case execution demand of $(\gamma^u \odot \bar{\alpha}_I^u)(\Delta)$ resource units. Now, assume that all the arrived events get completely processed in the same interval $\Delta$. This means that there were at least $(\gamma^u \odot \bar{\alpha}_I^u)(\Delta)$ resource units available in $\Delta$ for processing the events. This also means that $\alpha_O^u(\Delta) \geq (\gamma^u \odot \bar{\alpha}_I^u)(\Delta)$. If instead of $\alpha_O^u(\Delta)$ we plug $(\gamma^u \odot \bar{\alpha}_I^u)(\Delta)$ into (3.29) in order to find how many events can be produced at most within $\Delta$ at the component's output for $(\gamma^u \odot \bar{\alpha}_I^u)(\Delta)$ resource units, we obtain $(\gamma^{l^{-1}} \odot \gamma^u \odot \bar{\alpha}_I^u)(\Delta)$. From the general property of VCCs (3.13) it directly follows that

$$(\gamma^{l^{-1}} \odot \gamma^u \odot \bar{\alpha}_I^u)(\Delta) \geq \bar{\alpha}_I^u(\Delta) \tag{3.37}$$

$\gamma^u$ and $\gamma^l$ of any event stream with *variable* execution demand never coincide and thus the inequality (3.37) is typically strict. Furthermore, the higher the variability of the execution demand the larger the difference between left and right sides of (3.37) can be (since, for some $k$, $\gamma^u(k)$ can be considerably larger than $\gamma^l(k)$). This results in an overly pessimistic workload conversion.

Inequality (3.37) leads us to the understanding of the basic problem with the double workload transformations. It can be summarized as follows. The resource-based arrival curves carry almost no information about timing of event arrivals in the corresponding event stream. As a result, an attempt to recover this information under safe assumptions ends up at overly pessimistic bounds. Similarly, the event-based service curves carry almost no information about the amount of offered resource units. Hence, recovering this information under safe assumptions also results in overly pessimistic bounds. Therefore, the double workload transformations must be avoided.

An implementation of the performance component which is free of any double workload transformations is shown in Fig. 11(c). This implementation is optimal in a sense that it minimizes the inaccuracy introduced into the resulting bounds as a result of workload transformations.

**Summary**
In this subsection, we addressed the problem of extending the existing MPA framework with workload transformations. We showed that in order to avoid a considerable loss of accuracy while introducing the workload transformations into an RTC scheduling network, the following two principles have to be followed:

- the total number of workload transformations in a network should be minimized;
- the double workload transformations should be avoided.

Based on these principles we developed a mathematical model of an RTC performance component which can be used in the extended MPA framework. The model represents the main result of this subsection. It is captured by the following set of equations:

$$\begin{aligned}
\bar{\alpha}_O^u &= \pi^u \odot ([(\kappa^u \odot \bar{\alpha}_I^u \,\underline{\otimes}\, \gamma^{l^{-1}} \odot \beta_I^u) \,\overline{\oslash}\, \gamma^{u^{-1}} \odot \beta_I^l] \wedge \gamma^{l^{-1}} \odot \beta_I^u) && (3.38)\\
\bar{\alpha}_O^l &= \pi^l \odot ([(\kappa^l \odot \bar{\alpha}_I^l \,\overline{\oslash}\, \gamma^{l^{-1}} \odot \beta_I^u) \,\underline{\otimes}\, \gamma^{u^{-1}} \odot \beta_I^l] \wedge \gamma^{u^{-1}} \odot \beta_I^l) && (3.39)\\
\beta_O^u &= (\beta_I^u - \gamma^l \odot \kappa^l \odot \bar{\alpha}_I^l) \,\underline{\oslash}\, 0 && (3.40)\\
\beta_O^l &= (\beta_I^l - \gamma^u \odot \kappa^u \odot \bar{\alpha}_I^u) \,\overline{\otimes}\, 0 && (3.41)
\end{aligned}$$

## 3.5 Obtaining Variability Characterization Curves

The quality of results delivered by performance evaluation and scheduling methods using VCCs largely depends on the quality of VCCs supplied to them as an input for the analysis. The quality of the VCCs, in turn, depends on the way in which they have been obtained. This section discusses relevant issues and outlines several approaches to obtaining VCCs.

### 3.5.1 Objectives and limitations

Recall from Section 3.2 that a VCC is essentially an upper or a lower *bound* on the worst-case or, respectively, best-case variability of functions belonging to a given set. Hence, any considerations that can be made about worst-case bounds in general are also applicable to VCCs. Nevertheless, to provide a framework for a discussion of different ways for obtaining VCCs, in this subsection we revisit some of the relevant general issues by putting them in the VCC context.

At least two objectives have to be pursued while obtaining a VCC:

- **Guarantee:** A VCC has to guarantee that it *fully* captures the worst-case (or best-case) variability of a given function set, i.e. it has to be a proper bound. This guarantee is a necessary condition for obtaining reliable results from any analysis based on the given VCC.

- **Tightness:** A VCC has to be as tight as possible. The tightness of the VCC can influence the accuracy of the analytical results. The tighter the VCC the more accurate variability characterization it provides and, hence, the more accurate analytical results can be achieved through its use.

In order to see the whole range of possibilities for obtaining VCCs we have to make a clear distinction between a given set of increasing functions to be characterized by a VCC (refer to Defs. 1 and 2) and the information based on which the VCC characterizing this set is constructed. We refer to such information as the *source information*. In the simplest case the source information represents the function set itself. The corresponding VCC can then be derived directly from this set. In other cases the function set might not be explicitly available or it may be incomplete, however, some knowledge about it still can be at our disposal. This knowledge, which can serve as the source information in such cases, may be in the form of assumptions, formal specifications, various bounds and parameter estimates pertaining to the system at hand or to its environment or to both. Sections 3.5.2, 3.5.3 and 3.5.4 provide some concrete examples of possible forms of the source information.

Having drawn the distinction between the function set to be characterized and the source information based on which a VCC is constructed, we make the following assertions.

- The guarantee, which a VCC is required to provide, depends on the particular modeling and analysis goals. These goals, therefore, determine the function set to be characterized by the VCC.

- Whether or not a VCC provides the required guarantee depends on how guaranteed (i.e. reliable and sufficient) is the source information based on which the VCC has been constructed and does not depend on a particular method used to construct it.[3]

- The tightness of a VCC depends on the accuracy of the source information as well as on the method which has been used for constructing the VCC.

In summary, we identify the following principle limitations in achieving the objectives set forth in the beginning of this subsection. The guarantee provided

[3]If the source information is reliable and sufficient for providing the required guarantee but nevertheless the resulting VCC does not provide this guarantee, then the method is wrong (e.g. contains some faults). We exclude such cases from the consideration by assuming that any method for obtaining VCCs is correct.

by a VCC is limited by the reliability and sufficiency of its source information, whereas the tightness of the VCC is limited by the accuracy of both the source information and the method by which the VCC has been obtained.

### 3.5.2 Obtaining VCCs from traces

As Section 3.5.1 mentions, one of the possible ways to obtain a VCC is to construct it directly from the function set which this VCC has to characterize (i.e. in this case the source information for the VCC is the function set itself). Obviously, being able to obtain a guaranteed VCC in this way requires all the functions in the set to be precisely defined. Furthermore, by constructing a VCC directly from the function set we can obtain the tightest possible VCC for this set. Such *tightest upper and lower VCCs* can be computed as defined below.

**Def. 14: (Tightest Upper VCC)** *The tightest upper VCC $\mathcal{V}_{\mathcal{A}}^{u*}$ for a given set of increasing functions $\mathcal{A}$ is computed as*

$$\mathcal{V}_{\mathcal{A}}^{u*}(s) = \sup_{\forall A_i \in \mathcal{A},\ \forall t \in \mathbb{T}} \{A_i(t+s) - A_i(t)\} \ \ \forall s \in \mathbb{T} \tag{3.42}$$

**Def. 15: (Tightest Lower VCC)** *The tightest lower VCC $\mathcal{V}_{\mathcal{A}}^{l\,*}$ for a given set of increasing functions $\mathcal{A}$ is computed as*

$$\mathcal{V}_{\mathcal{A}}^{l\,*}(s) = \inf_{\forall A_i \in \mathcal{A},\ \forall t \in \mathbb{T}} \{A_i(t+s) - A_i(t)\} \ \ \forall s \in \mathbb{T} \tag{3.43}$$

Showing that $\mathcal{V}_{\mathcal{A}}^{u*}$ computed with (3.42) is the *tightest* upper VCC for $\mathcal{A}$ is straightforward: any other function $V$, such that $V(s) < \mathcal{V}_{\mathcal{A}}^{u*}(s)$ for some $s$, is not an upper VCC for $\mathcal{A}$ by Def. 1. By following a similar reasoning, $\mathcal{V}_{\mathcal{A}}^{l\,*}$ computed with (3.43) can be shown to be the tightest lower VCC.

In practice, the increasing functions $A_i(t)$ from which a VCC is constructed can be computed from *traces*. A trace is a sequence of events which have occurred as a result of a system execution at a given point in the system or its environment (e.g. at a specified input or output or processing element). Let $v_j \in \mathbb{R}_{\geq 0}$ and $t_j \in \mathbb{T}$ denote, respectively, the value and the occurrence time of $j$th event recorded in a trace. Then the increasing function $A$ corresponding to this trace can be computed as follows.

$$A(t) = \sum_{j:\ t_j \leq t} v_j \quad \text{and} \quad A(0) = 0$$

Traces are collected by measuring the parameters of interest in a real system or its simulator. The event values $v_j$ can represent different parameters of the system's behavior. For example, they can indicate arrivals of events at an input of the system, or they can be execution times of a task which are recorded into the trace each time the task finishes its execution, etc. The occurrence time $t_j$ does not need to be an absolute time. It can also be a sequence number of an event in the trace.

Note that in order to obtain the VCCs it is not necessary to first convert the traces into the increasing functions. The VCCs can be constructed directly from the traces. For this purpose, the traces are analyzed by sliding along them windows of different sizes. The window size corresponds to the argument value for which a VCC is computed. The value of upper (lower) VCC for a given argument value is then the maximum (the minimum) sum of trace values $v_j$ that fall into a window of the corresponding size.

**Discussion**

Although VCCs obtained from traces can accurately capture worst-case and best-case variability in the event streams, they have a restricted applicability. In particular, such VCCs cannot be used for the timing analysis of hard real-time systems. The hard real-time analysis has to fully cover all possible system states and interactions with the environment. Hence, if such an analysis uses VCCs that have been obtained from a set of traces, then this set must contain *all* possible traces. For most of the realistic systems, however, generating the exhaustive set of traces is infeasible.

Despite their limitation mentioned above, the VCCs obtained from traces have a wide range of applications in the analysis and design of soft real-time systems. For this class of systems the full coverage of the analysis is not so essential as for the hard real-time systems. Some rare or exceptional system behaviors can be omitted from the consideration especially in the context of a system-level design space exploration. Therefore, a set of traces representing typical cases may be sufficient for obtaining a VCC for analysis of soft real-time systems.

Since most of the multimedia systems belong to the class of soft real-time systems, in this thesis for experimental case studies we use VCCs that have been obtained from traces. Appendix A gives details on the simulation framework used to collect the traces and obtain from them the VCCs.

### 3.5.3 Obtaining VCCs from constraints

Sometimes no complete information is available about functions $A_i$ belonging to a given set $\mathcal{A}$ to be characterized by VCCs. The set of functions $\mathcal{A}$ might not be explicitly defined or it might be defined only partially. For instance, exact values of functions $A_i$ may not be known because of some uncertainty in parameters of a modeled system or its environment. In such situations the tightest upper and lower VCCs cannot be obtained using (3.42) and (3.43). Then VCCs can be constructed based on some other kind of source information than the function set itself. This subsection and Section 3.5.4 are devoted to a discussion of such ap-

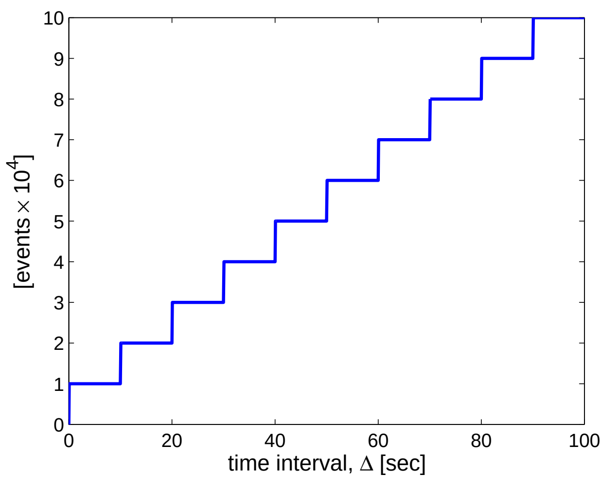


**Fig. 12:** An upper arrival curve for an event stream at the output of a PE which can at maximum process 10000 events in any interval of 10 sec.

proches. More specifically, this subsection illustrates how VCCs can be obtained from various kinds of constraints and assumptions about the system and its environment, while Section 3.5.4 gives an example of how VCCs can be obtained from formal system specifications.

Before a design of an embedded system is started, there is almost always a certain amount of pre-specified information about characteristics of the future system and its environment. In the design process this information becomes more and more refined. Such information may include various timing requirements and constraints, estimated or available from data sheets performance figures and other characteristics that pertain to the system, its individual components and tasks, its target execution platform and the environment. For example, if the embedded system is to process a media stream, the stream's parameters such as the arrival rate and the maximum jitter can be known and fixed quite early in the design cycle.

The various information available at the design time about the system and its environment can be used to reason about the worst-case (or best-case) variability of certain system characteristics. I.e. based on this information the VCCs can be defined for some, not necessarily all, argument values. For the rest of the argument values the VCCs can be approximated by making some worst-case assumptions.

For example, suppose that the only knowledge we have about a PE is that at maximum it can process 10000 events within any interval of 10 sec. Using only this constraint we can already construct a proper upper arrival curve for the processed event stream at the output of this PE, as it is shown in Fig. 12.

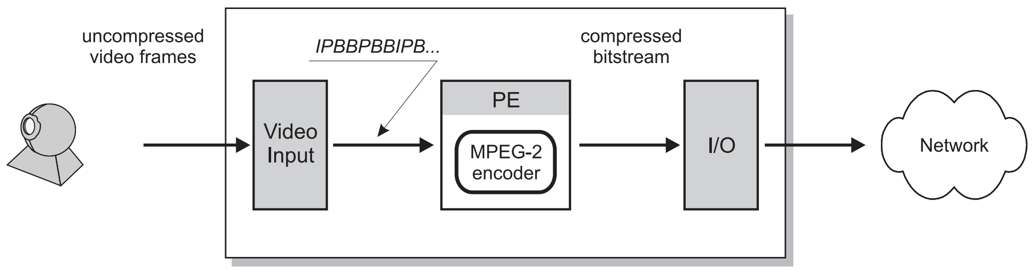


**Fig. 13:** MPEG-2 encoding system considered in Ex. 3.

### 3.5.4 Obtaining VCCs from formal system specifications

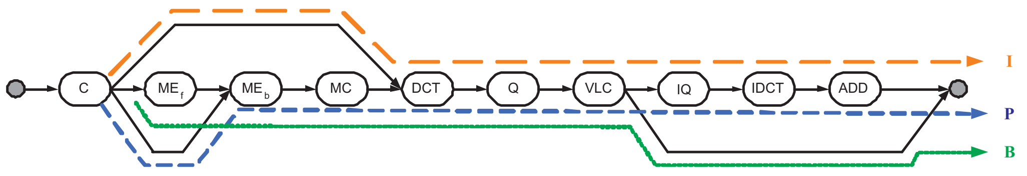
In some cases VCCs can be obtained from system specifications using formal methods, as opposed to somewhat ad hoc approaches illustrated in Section 3.5.3. This subsection exemplifies one such formal method for obtaining execution demand curves. This method is based on the formalism of finite state machines (FSM), which is used to specify system functionality and properties of event streams. A theoretical background and other relevant details and ramifications of this method can be found in [166, 168].

**Ex. 3:** *Consider an MPEG-2 encoding system shown in Fig. 13. Uncompressed video frames captured by a video camera arrive at the input of a media processor. The media processor compresses them using an MPEG-2 encoding algorithm and sends the compressed bitstream, for instance, to a network. We assume that the MPEG-2 algorithm is entirely implemented on one processing element (PE) within the media processor. Our goal is to construct an upper execution demand curve for the workload imposed by the MPEG-2 encoder task on the PE.*

*In order to formally specify the system we need to know some details about the behavior of the MPEG-2 encoder. They are presented below.*

*The MPEG-2 compression scheme exploits three frame types to encode video information. These are I-, P- and B-frames. Before the compression, a pre-processing stage indicated in Fig. 13 as* Video Input *assigns to each input video frame one of these types. After that, depending on the frame type the encoder executes different subtasks to compress the frame, as shown in Fig. 14. As a result, different frame types will impose different execution demands on the processing element, as specified in Tab. 2.*

*The MPEG-2 standard [118] does not specify any particular implementation of the encoder algorithm. System designers, therefore, can choose between many different implementation options. In particular, the standard does not specify specific frame patterns that can be used by the encoder to compress a video sequence. Advanced encoders use several predefined patterns and can arbitrary switch between them based on the characteristics of the input video sequence. Likewise our encoder can generate three commonly used patterns: IPB, IPBB and IPBBPBB. Furthermore, if a scene change has been detected in the input*



| | | | |
|---|---|---|---|
| $ME_{b,f:}$ | : Motion Estimation (backwards/forwards) | VLC | : Variable-length Coding |
| MC | : Motion Compensation | IQ | : Inverse Quantization |
| DCT | : Discrete Cosine Transform | IDCT | : Inverse DCT |
| Q | : Quantization | ADD | : Addition |

**Fig. 14:** Task graph of the MPEG-2 encoder algorithm indicating processing paths for different video frame types.

| Frame type | Worst-case execution time [cycles] |
|---|---|
| I | $2 \cdot 10^6$ |
| P | $8 \cdot 10^6$ |
| B | $20 \cdot 10^6$ |

**Tab. 2:** Execution demand of the MPEG-2 encoder for each frame type.

*video stream, the encoder can interrupt the currently generated pattern at any place and start generation of a new pattern. This behavior, however, is subject to a constraint: whenever a scene change has been detected and a new pattern has been started, at least three consecutive frames must be encoded* without *pattern interruption. An FSM describing the frame pattern generation behavior is shown in Fig. 15.*

*Given the specification in the form of FSM in Fig. 15 and using the worst-case execution demands of different frame types from Tab. 2, we can derive an upper execution demand curve $\gamma^u_{MPEG\text{-}2enc}$ for the MPEG-2 encoder task. For this we annotate all transitions of the FSM with the worst-case execution demands of the corresponding frame types. As a result of this annotation we obtain a weighted directed graph $G_{MPEG\text{-}2enc}$. The value of $\gamma^u_{MPEG\text{-}2enc}(e)$ then is the weight of the maximum-weight path of length $e$ in $G_{MPEG\text{-}2enc}$ [168]. Fig. 16 shows the resulting execution demand curve.*

## 3.6 Experimental Evaluation

This section presents results of an experimental evaluation of the VCC-based workload model [113, 167]. The VCC-based model is compared to a traditional task model widely used for scheduling and performance analysis of real-time embedded systems. Performance figures obtained from VCC-based analysis are

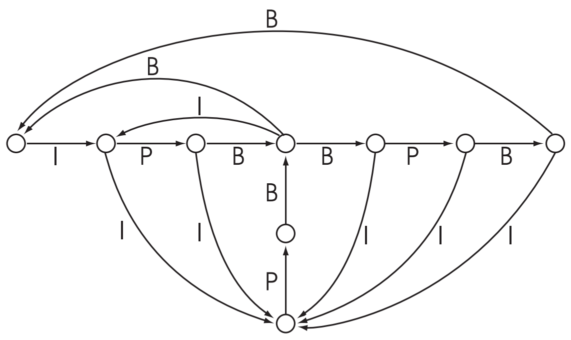


**Fig. 15:** FSM specifying frame patterns generated by the MPEG-2 encoder.

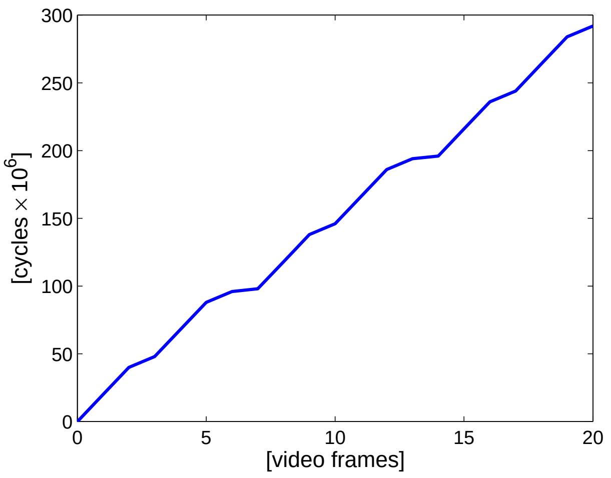


**Fig. 16:** Upper execution demand curve $\gamma^{u}_{\text{MPEG-2enc}}$ for the MPEG-2 encoder task obtained from FSM in Fig. 15 using worst-case execution demands given in Tab. 2.

also compared to measurements collected from a system simulator. The experimental results show that the VCC-based model returns considerably tighter performance bounds than those computed using the traditional task model. Furthermore, the comparison of these bounds with the simulation measurements indicates that they provide useful estimates of system properties and therefore can serve as a sound basis for making design decisions.

For the evaluation of the VCC-based model, we apply it to two optimization problems that may arise in the design context of multimedia MpSoC architectures. Both problems concern clock rate minimization of a PE processing media streams. Having the processor clock speed optimized is important in many design scenarios, as even a modest reduction in the clock rate may lead to considerable savings in the system cost and energy consumption.

The design scenarios which we consider in this experimental study a relatively simple. This simplicity facilitates the comparison of the VCC-based model to a traditional workload model. That is, to achieve a fair comparison, we se-

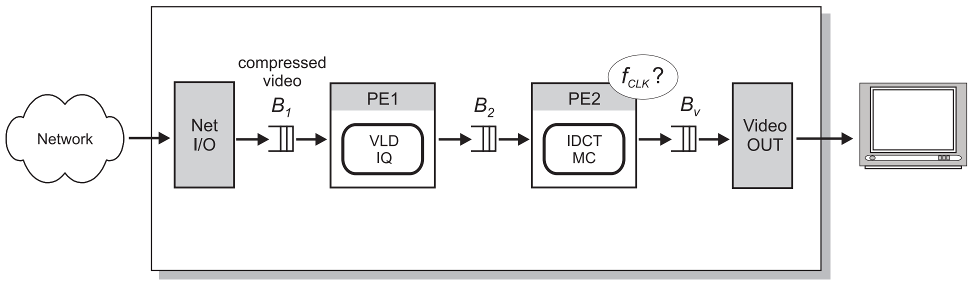


**Fig. 17:** MPEG-2 video decoder design scenario: The MPEG-2 decoder algorithm is mapped onto two PEs of an MpSoC platform. $\mathrm{PE1}$ executes VLD and IQ functions, while $\mathrm{PE2}$ executes IDCT and MC functions of the decoding algorithm. The goal is to determine minimum clock rate of $\mathrm{PE2}$ ensuring that the FIFO buffer at the $\mathrm{PE2}$'s input, $B_2$, never overflows.

lected the optimization problems such that they can be solved by using either of the models. Applications of VCCs to more complex scheduling and performance analysis problems are demonstrated later in this thesis, in Chapter 5.

The traditional workload model to which we compare the VCC-based model in this experimental study characterizes execution demand of a task by a single value—its WCED. As an alternative to this characterization we propose the execution demand curves. To clearly see the gain from using only this VCC type, we characterize other properties of the tasks, such as timing of arrivals, in both models in the same way. In this sense, our "traditional" model corresponds to that proposed in [24, 25].

Section 3.6.1 addresses the processor rate minimization problem under a *buffer constraint*, while Section 3.6.2 addresses the processor rate minimization problem under a *delay constraint*. In the former case, for calculations of the minimum processor rate we use VCCs obtained from traces. In the latter case, the calculations are performed on analytically obtained VCCs. In both cases, the evaluation criteria is the tightness of the computed lower bound on the processor rate.

### 3.6.1 Processor clock rate optimization under buffer constraint

#### Design scenario

Fig. 17 shows a mapping of an MPEG-2 video decoding application onto an MpSoC platform consisting of two PEs, $\mathrm{PE1}$ and $\mathrm{PE2}$. One part of the MPEG-2 decoding algorithm, including VLD and IQ functions, is implemented on $\mathrm{PE1}$, while the rest of the algorithm, i.e. IDCT and MC functions, is implemented on $\mathrm{PE2}$. Given this mapping, the video decoding occurs on the *macroblock level* and proceeds as follows. First, an MPEG-2 bitstream arrives through a networking interface at the input of $\mathrm{PE1}$. After applying to it VLD and IQ functions, the video stream emerges at the $\mathrm{PE1}$'s output as a sequence of *partially decoded* macroblocks. Each such partially decoded macroblock represents a data structure containing decompressed IDCT coefficients and, if applicable, motion vectors. $\mathrm{PE1}$ writes this stream of partially decoded macroblocks into FIFO buffer $B_2$. $\mathrm{PE2}$ reads from $B_2$ one macroblock at a time and completes its decoding by processing it with IDCT and, if necessary, MC functions. After that, $\mathrm{PE2}$ writes the fully decoded macroblock into buffer $B_v$ which is read by a video output interface. Finally, after some additional post-processing in the video output interface, the resulting video signal appears at the system output for rendering on a display device.

The design goal in the scenario described above is to determine an optimal clock rate of $\mathrm{PE2}$. Suppose that one of the criterion for the selection of the clock rate is to ensure that buffer $B_2$ at $\mathrm{PE2}$'s input never overflows. That is, we are interested in finding a *lower bound* on the clock rate of $\mathrm{PE2}$ ensuring that $B_2$ never overflows. For simplicity, we assume that both PEs in Fig. 17 execute no tasks other than the tasks performing the MPEG-2 video decoding functions. This assumption means that processing capacity of $\mathrm{PE1}$ and $\mathrm{PE2}$ is fully devoted to execution of the MPEG-2 decoding tasks.

Intuitively, avoiding infinite growth of the backlog at $\mathrm{PE2}$'s input necessitates $\mathrm{PE2}$ to run at a rate which is high enough to fully process in a *long term* the workload imposed on it by the video stream. This condition is necessary but not sufficient for preventing overflows of $B_2$. For this condition to be sufficient, buffer $B_2$ has to be large enough to completely absorb transient overloads of $\mathrm{PE2}$ due to workload bursts. However, in this design scenario, we assume that the size of $B_2$ is insufficient to completely absorb such transient overloads. This means that to avoid overflows of $B_2$ the clock rate of $\mathrm{PE2}$ may need to be substantially higher than the rate determined by the long-term average workload imposed on this PE. In other words, $B_2$ imposes a constraint on the minimum clock rate at which $\mathrm{PE2}$ is allowed to run. In this situation, we say that there is a *buffer constraint* on the clock rate of the PE. Again, for simplicity, we will assume that in Fig. 17 no buffers are constrained besides $B_2$.

**Modeling workload with VCCs**

The minimum clock rate at which $\mathrm{PE2}$ in Fig. 17 has to run in order to guarantee that buffer $B_2$ at its input never overflows depends on how bursty is the workload imposed on $\mathrm{PE2}$. For a given size of $B_2$, the higher the workload variability is, the higher the clock rate must be. Hence, for the analysis we need somehow to capture this workload variability on $\mathrm{PE2}$. For this we employ VCCs.

Workload variability on $\mathrm{PE2}$ originates from two sources: (i) execution time of VLD and IQ functions on $\mathrm{PE1}$ is highly variable resulting in bursty arrivals into buffer $B_2$; and (ii) execution time of IDCT and MC functions on $\mathrm{PE2}$ itself is also variable. To characterize the burstiness of the stream at the $\mathrm{PE1}$'s output, we use upper event-based arrival curve $\bar{\alpha}^u$, while for the characterization of the execution demand of IDCT and MC functions on $\mathrm{PE2}$ we employ upper execution demand curve $\gamma^u$. Furthermore, since at each execution the task performing IDCT and MC functions always consumes from $B_2$ exactly one macroblock, its consumption curves $\kappa^l(k) = \kappa^u(k) = k$.

**Lower bound on processor rate under buffer constraint**
Let $L$ denote the size of buffer $B_2$ measured in number of stream objects (i.e. macroblocks). The upper bound on the backlog experienced by a stream at the input of a processing element can be computed using (2.10). Based on (2.10) we can formulate a constraint which requires that this upper bound never exceeds $L$:

$$L \geq (\bar{\alpha}^u - \bar{\beta}^l)(\Delta) \quad \forall \Delta \geq 0$$

Using this constraint, we can easily compute the required lower *event-based* service curve $\bar{\beta}^l$ which ensures no buffer overflows for the stream constrained by $\bar{\alpha}^u$. However, instead of $\bar{\beta}^l$ we are interested in finding the required *resource-based* service curve $\beta^l$ which is expressed in number of processor cycles per time unit. From $\beta^l$ we will be able to compute the required *clock* rate. Thus, we need to reformulate the above constraint for $\beta^l$. We can do this using workload transformations via $\gamma^u$ in two ways:

(i) $\beta^l(\Delta) \geq \gamma^u \odot ((\bar{\alpha}^u(\Delta) - L) \vee 0)$; and

(ii) $\beta^l(\Delta) \geq ((\gamma^u \odot \bar{\alpha}^u)(\Delta) - \gamma^l(L)) \vee 0$.

The latter constraint is more intuitive and more conservative than the former and hence there may be a temptation to consider it as the only correct one. We show that the former constraint also leads to a correct bound on $\beta^l$ and therefore, to achieve a tighter bound on $\beta^l$, the former constrained has to be used. Since this basic result will be used in other parts of the thesis we formulate it as a theorem:

**Thm. 1:** **(Buffer-constrained service)** *The backlog in front of a processing element caused by an event stream characterized by upper event-based arrival curve $\bar{\alpha}^u$ and upper execution demand curve $\gamma^u$ never exceeds $L$ events if the lower resource-based service curve $\beta^l$ offered to the event stream on this processing element satisfies*

$$\beta^l(\Delta) \geq \gamma^u \odot ((\bar{\alpha}^u(\Delta) - L) \vee 0) \quad \forall \Delta \geq 0 \tag{3.44}$$

*under the assumption that the consumption curves $\kappa^l(k) = \kappa^u(k) = k \quad \forall k \geq 0$.*

**Proof.** We have to show that if $\beta^l$ satisfies (3.44) then $L$ represents a valid upper bound for the maximum backlog. For this, it is sufficient to show that $L$ is larger or equal to the known upper bound on the backlog determined by (2.10) [85], i.e. we need to show that

$$L \geq (\bar{\alpha}^u - \bar{\beta}^l)(\Delta) \quad \forall \Delta \geq 0 \tag{3.45}$$

Using Prop. 1 of upper VCCs, we can rewrite (3.44) as follows

$$(\gamma^{u^{-1}} \odot \beta^l)(\Delta) \geq (\bar{\alpha}^u(\Delta) - L) \vee 0 \quad \forall \Delta \geq 0 \tag{3.46}$$

Assume that $\bar{\alpha}^u(\Delta) - L > 0$, then (3.46) takes the form

$$L \geq (\bar{\alpha}^u - \gamma^{u^{-1}} \odot \beta^l)(\Delta) \quad \forall \Delta \geq 0 : \; \bar{\alpha}^u(\Delta) - L > 0$$

From (3.34) we know that $\bar{\beta}^l(\Delta) = (\gamma^{u^{-1}} \odot \beta^l)(\Delta)$, and hence we get:

$$L \geq (\bar{\alpha}^u - \bar{\beta}^l)(\Delta) \quad \forall \Delta \geq 0 : \; \bar{\alpha}^u(\Delta) - L > 0$$

Now, if we assume that $\bar{\alpha}^u(\Delta) - L \leq 0$, then (3.46) takes the form

$$(\gamma^{u^{-1}} \odot \beta^l)(\Delta) \geq 0 \quad \forall \Delta \geq 0 : \; \bar{\alpha}^u(\Delta) - L \leq 0$$

i.e.

$$\bar{\beta}^l(\Delta) \geq 0 \quad \forall \Delta \geq 0 : \; \bar{\alpha}^u(\Delta) - L \leq 0$$

The same hods true for (3.45) if $\bar{\alpha}^u(\Delta) - L \leq 0$.

Thus we have shown that in both cases, when $\bar{\alpha}^u(\Delta) - L > 0$ and when $\bar{\alpha}^u(\Delta) - L \leq 0$, (3.44) implies (3.45), i.e. $L$ represents a valid upper bound for the maximum backlog. $\square$

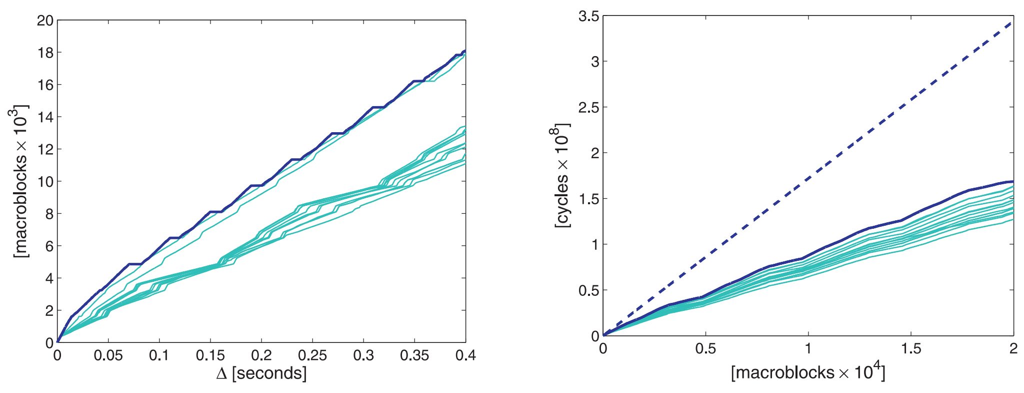
Since the full processing capacity of $\mathrm{PE2}$ is devoted to the video decoding task, the shape of the lower resource-based service curve is determined by $\beta^l(\Delta) = f \cdot \Delta$, where $f$ denotes the clock rate of $\mathrm{PE2}$. Hence, to find the lower bound on the clock rate of $\mathrm{PE2}$, we can rewrite (3.44) as follows

$$f \geq \frac{\gamma^u \odot ((\bar{\alpha}^u(\Delta) - L) \vee 0)}{\Delta} \quad \forall \Delta > 0$$

Finally, we obtain

$$f_{min} = \sup_{\forall \Delta > 0} \left\{ \frac{\gamma^u \odot ((\bar{\alpha}^u(\Delta) - L) \vee 0)}{\Delta} \right\} \tag{3.47}$$

(3.47) gives the analytical lower bound on the clock rate of $\mathrm{PE2}$ which guarantees that the buffer constraint $L$ is satisfied.



(a) Event-based arrival curve $\bar{\alpha}^u$ (b) Execution demand curve $\gamma^u$

**Fig. 18:** Workload characterization on $\mathrm{PE2}$. Thin lines correspond to VCCs of individual video sequences. Solid thick lines show VCCs representing the whole set of video sequences. The dashed line on $\gamma^u$ plot (b) represents the execution demand curve corresponding to the traditional task characterization with only one value—task's WCED.

**Obtaining $\bar{\alpha}^u$ and $\gamma^u$ from traces**

We conducted experiments with a set of 14 different MPEG-2 video sequences encoded with the following parameters: 9.78 Mbit/s constant bit rate (CBR), main profile at main level (MP@ML), frame rate of 25 fps, and resolution of $720 \times 576$ pixels. By simulating decoding of these sequences on the SimpleScalar instruction set simulator (ISS) [8], we collected execution demand traces for tasks mapped onto $\mathrm{PE1}$ and $\mathrm{PE2}$ in Fig. 17. From these traces, using the analysis technique described in Section 3.5.2, for each video sequence in the set we obtained event-based arrival curve $\bar{\alpha}_i^u$ and execution demand curve $\gamma_i^u$, $i = 1, 2, \ldots 14$. These VCCs are shown with thin lines on the plots in Fig. 18.

From all VCCs $\bar{\alpha}_i^u$ and $\gamma_i^u$ obtained for individual video sequences, we calculated event-based arrival curve $\bar{\alpha}_\Sigma^u$ and execution demand curve $\gamma_\Sigma^u$ representing the *whole* set of video sequences:

$$\bar{\alpha}_\Sigma^u(\Delta) = \max_{\forall i}\{\bar{\alpha}_i^u(\Delta)\}$$

$$\gamma_\Sigma^u(k) = \max_{\forall i}\{\gamma_i^u(k)\}$$

The resulting $\bar{\alpha}_\Sigma^u$ and $\gamma_\Sigma^u$ are indicated in Fig. 18 with solid thick lines.

**Comparison to traditional task characterization model**

Two workload models can be compared based on the tightness of the lower bound on the processor rate $f_{min}$ computed using these models for a given buffer constraint $L$.

As a basis for the comparison, we took a widely used task model in which task's execution demand is characterized by only one value — its WCED[4] [100].

[4]In the literature, WCED is more often referred to as WCET.

Unlike execution demand curves, this model does not exploit any knowledge of task execution sequences to characterize the task execution demand. Instead, it considers only individual task instances, assuming that every such instance imposes WCED on the processor. As for modeling arrivals of a task, this task model specifies either the period with which the task arrives into the system or the minimum time interval between its two consecutive arrivals (the minimum inter-arrival time).

We were interested in quantifying the gain resulting from using only one VCC type—the execution demand curve. For this we computed the lower bound on the processor rate by evaluating (3.47) for the VCC-based model *and* for the traditional task model. For the VCC-based model, we directly used in (3.47) $\bar{\alpha}^u_\Sigma$ and $\gamma^u_\Sigma$ obtained above. For the traditional task model, we evaluated (3.47) as follows:

- Instead of using the minimum inter-arrival time (or the period) for the characterization of the arrivals of the macroblock stream at the input of buffer $B_2$ in Fig. 17, we used less pessimistic characterization with the measured $\bar{\alpha}^u_\Sigma$. (In this respect, we did not fully respect the the traditional workload model.)

- For the characterization of the execution demand imposed by the video decoding task on $\mathrm{PE2}$, we used WCED of this task measured over all video sequences. We note that such WCED equals to $\gamma^u_\Sigma(1)$ determined above. Hence, the execution demand curve corresponding to the traditional characterization is a straight line with the slope $\gamma^u_\Sigma(1)$, i.e. $\gamma^u_{WCED}(k) = k \cdot \gamma^u_\Sigma(1)$, $k = 0, 1, 2, \ldots$. Consequently, we used $\gamma^u_{WCED}$ for the evaluation of (3.47). Fig. 18(b) shows $\gamma^u_{WCED}$ with a dashed line.

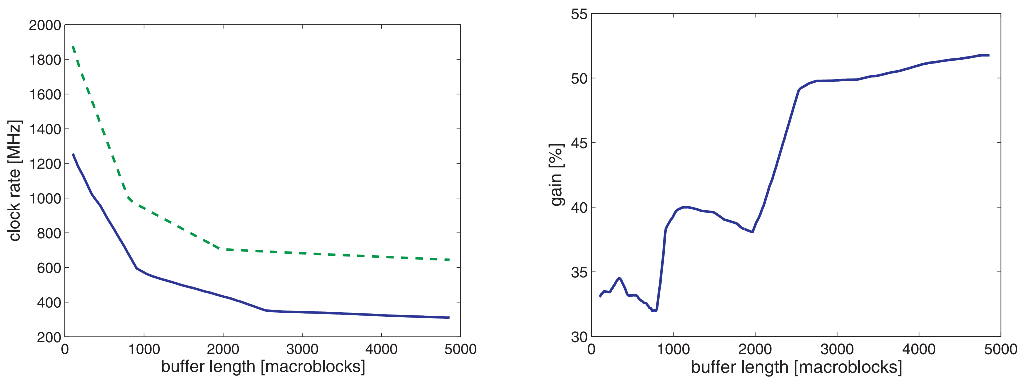
Note that the existing RTC MPA framework reported in [24, 25] uses exactly this model (with $\gamma^u_{WCED}$ in place of $\gamma^u$).

Fig. 19(a) shows the lower bound on the clock rate of $\mathrm{PE2}$ versus the length of buffer $B_2$ computed using (3.47) for the VCC-based model (solid line) and for the traditional workload model (dashed line). By inspecting the plots in Fig. 19(a) we can make the following observations:

1. With increasing size $L$ of buffer $B_2$ the minimum clock rate at which $\mathrm{PE2}$ can run without causing buffer overflows decreases. Both workload models expose this trend.

2. The clock rate bound calculated using VCC-based model is significantly tighter than that calculated using the traditional workload model with a single WCED value.

Fig. 19(b) shows how much exactly we can gain by using the VCC-based model in place of the traditional model. For small buffer sizes ($L < 2000$) the gain is



(a) Lower bound on the clock rate of $\mathrm{PE2}$

(b) The gain from using the execution demand curve

**Fig. 19:** Experimental results. On the left plot, the solid line corresponds to the lower bound on the clock rate of $\mathrm{PE2}$ calculated using execution demand curve $\gamma^u_\Sigma$; the dashed line corresponds to the bound calculated using the traditional task characterization with a single WCED value.

about 30–40%, while for larger buffer sizes the gain may be as large as 50%. This means that the VCC-based workload model may result in up to two times less pessimistic estimations of system parameters than the traditional model, leading to considerable savings in system cost and power consumption.

## Comparison to simulation

To estimate the degree of pessimism incurred by the VCC-based model, we simulated the video decoding system shown in Fig. 17 with the clock rate of $\mathrm{PE2}$ set to the values calculated using (3.47). More specifically, using (3.47) we computed $f_{min}$ for a given buffer constraint $L$ and then simulated a transaction-level model[5] of the system shown in Fig. 17 with the clock rate of $\mathrm{PE2}$ set to $f_{min}$. For each simulated video sequence, we measured maximum backlog in buffer $B_2$ and compared this measured number to the given buffer constraint $L$. The closer the measured maximum backlog to the given buffer constraint $L$ was, the less pessimism the VCC-based workload model incurred.

For investigation of the pessimism incurred by the VCC-based model, we conducted a number of experiments with different values of $\bar{\alpha}^u$ and $\gamma^u$ plugged into (3.47). For the experiments, we used the same set of 14 MPEG-2 video sequences as described in the preceding paragraphs of this subsection. Following is a summary of the conducted simulation experiments.

- First, we experimented with VCCs $\bar{\alpha}^u_\Sigma$ and $\gamma^u_\Sigma$ characterizing the *whole* set of video sequences (as defined above). That is, in (3.47) we used $\bar{\alpha}^u = \bar{\alpha}^u_\Sigma$ and $\gamma^u = \gamma^u_\Sigma$. Fig. 20 shows the maximum backlog registered in buffer $B_2$ while de-

[5]Appendix A gives details about the simulation environment.

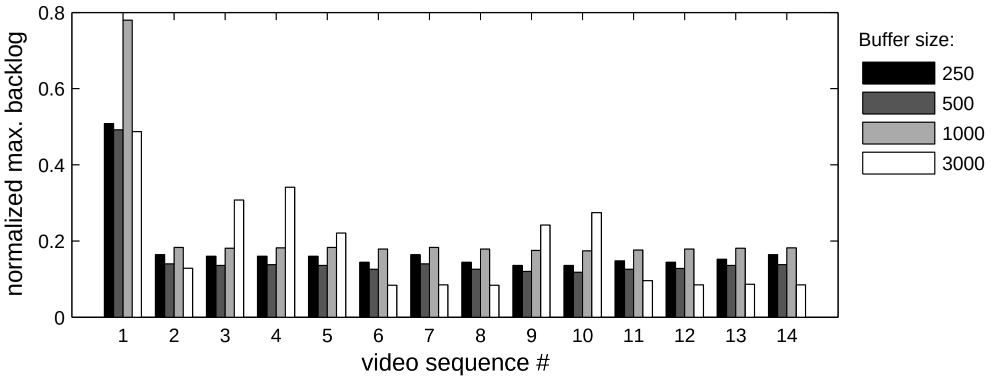


**Fig. 20:** Normalized maximum backlogs registered in buffer $B_2$ in Fig. 17 when the clock rate of $\mathrm{PE2}$ was set to the values obtained from the VCC-based model characterizing the *whole* set of the MPEG-2 video sequences.

coding different video sequences in the set. The maximum backlog shown in Fig. 20 is normalized to buffer constraint $L$ ("buffer size") for which the corresponding $f_{min}$ was computed. The bar plot in Fig. 20 shows that for all but one video sequence (#1) the maximum backlog in $B_2$ was less than half of buffer size $L$. For video sequence #1 and buffer sizes $L = \{250, 1000, 3000\}$, the maximum backlog was about half of $L$. However, for buffer size $L = 500$, video sequence #1 caused $B_2$ to be filled up to 80%. Although for the rest of video sequences the computed bound on the clock rate is seemingly pessimistic, given the fact that we performed worst-case analysis by abstracting the *whole* set of video sequences with only one pair of VCCs, $\bar{\alpha}^u_\Sigma$ and $\gamma^u_\Sigma$, we can conclude that the pessimism incurred by the VCC-based model is relatively low. The simulation results presented in Fig. 20 indicate that sequence #1, probably, imposes a highly bursty workload on $\mathrm{PE2}$, and represents for the system the most "adverse" (in terms of the imposed workload) video sequence among all other sequences in the set. VCCs could accurately capture this worst-case.

- To estimate how pessimistic a VCC abstraction of a *single* video sequence can be, we conducted a series of experiments in which for computation of the lower bound on the clock rate of $\mathrm{PE2}$, we plugged into (3.47) VCCs characterizing individual video sequences. That is, in (3.47) we used $\bar{\alpha}^u = \bar{\alpha}^u_i$ and $\gamma^u = \gamma^u_i$, where $i = 1, 2, \ldots, 14$. Thus, for a given buffer size $L$ we computed values of $f_{min}$ for each video sequence in the set and simulated decoding of each video sequence with the corresponding to it value of $f_{min}$. The resulting normalized maximum backlogs registered in buffer $B_2$ are depicted in Fig. 21. The simulation results in Fig. 21 show that for all video sequences and all buffer sizes $L$ the maximum backlogs in $B_2$ were larger than 50%. Moreover, in Fig. 21 we can see that in many configurations the buffer was almost full. This indicates that

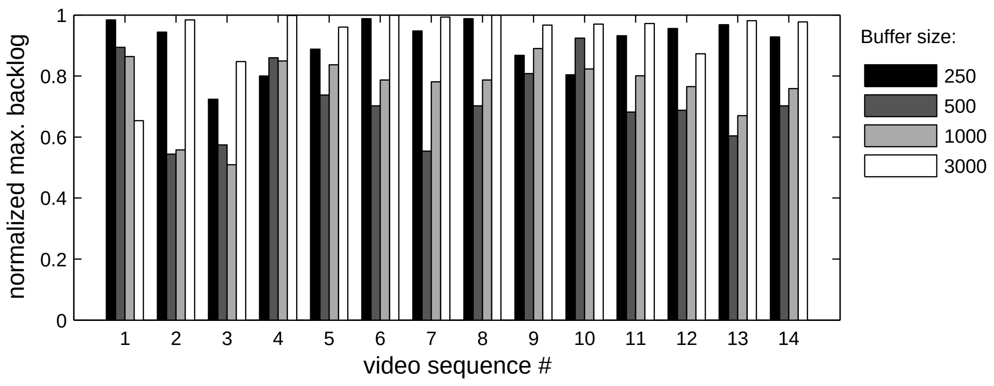


**Fig. 21:** Normalized maximum backlogs registered in buffer $B_2$ in Fig. 17 when the clock rate of $\mathrm{PE2}$ was set to the values obtained from the VCC-based model characterizing each MPEG-2 video sequence individually.

VCCs obtained for individual traces (in our case video sequences) can provide highly accurate abstractions for these traces. This suggests that instead of time consuming simulations, system-level performance evaluation could be carried out analytically using these VCC-based abstractions without a considerable loss of accuracy of the resulting performance numbers. Although in this case VCC-based model would not be able to guarantee coverage of the worst case, at least it could speedup the performance evaluation process, provided that the VCC-based analytical model can be efficiently evaluated. In fact, other comparative studies [25, 115] have demonstrated that the VCC-based models can be few orders of magnitude faster than the corresponding system-level simulators.

- Finally, based on the experimental results obtained above, we hypothesized that VCCs could provide a relatively accurate characterization for *groups* of traces (video sequences) with *similar* properties. To check this hypothesis, we conducted simulations with a subset of the video sequences used in the experiments above. We formed this subset by inspecting VCCs $\bar{\alpha}_i^u$ and $\gamma_i^u$ of individual video sequences (shown with thin lines in Fig. 18). Such ”outliers” as, for example, video sequence #1, which mainly determined shapes of $\bar{\alpha}_\Sigma^u$ and $\gamma_\Sigma^u$ (shown in Fig. 18 with thick lines), were not included into the subset. As a result, the subset included video sequences whose VCCs had similar shapes— sequences #4 through #14. For this subset we then calculated VCCs $\bar{\alpha}_{\Sigma^*}^u$ and $\gamma_{\Sigma^*}^u$ by following the same procedure as we used above for the calculation of $\bar{\alpha}_\Sigma^u$ and $\gamma_\Sigma^u$. Using (3.47), we computed $f_{min}$ for different values of buffer size $L$ and with $\bar{\alpha}^u = \bar{\alpha}_{\Sigma^*}^u$ and $\gamma^u = \gamma_{\Sigma^*}^u$, and performed corresponding system simulations. The resulting normalized maximum backlogs registered in buffer $B_2$ are shown in Fig. 22. In this figure, we can see that for most configurations buffer $B_2$ was occupied approximately for 50–90%. This is an indicator that VCCs can provide an accurate

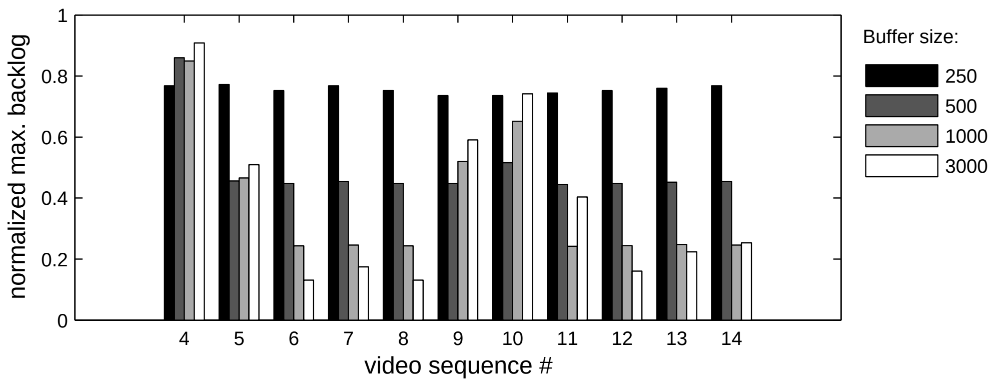


**Fig. 22:** Normalized maximum backlogs registered in buffer $B_2$ in Fig. 17 when the clock rate of $\mathrm{PE2}$ was set to the values obtained from the VCC-based model characterizing a *subset* of the MPEG-2 video sequences with similar VCC shapes.

abstraction for groups of traces (video sequences) with similar properties. Hence, if the emphasis of a performance analysis is on the accuracy of the results rather than on the speed with which these results are obtained, instead of individually characterizing every trace with VCCs (as described in the preceding paragraph), without significant loss in the accuracy we could use VCCs to characterize distinctive groups (or *classes*) of traces with similar workload properties. This idea is further developed in Chapter 4 of this thesis.

### 3.6.2 Processor clock rate optimization under delay constraint

#### Design scenario

Fig. 23 shows a system-level view of a networked multimedia device implemented on a media processor. The processor receives at its inputs real-time audio and video streams, compresses and sends them to a network. The compression is performed by tasks running on a PE within the media processor. Audio and video frames periodically arrive into FIFO buffers $B_a$ and $B_v$ at the PE's input. The PE reads from a buffer one frame at a time and processes it by executing the corresponding compression task: Audio frames get processed by an MP3 encoding task, while the video stream get processed by an MPEG-2 encoding task. After this processing, the compressed audio and video streams are sent to the FIFO buffers at the PE's output.

The video and audio streams must be processed in real time. Since the audio stream has lower priority than the video stream, audio frames may experience a processing delay which depends on interference from the video encoding task. To ensure quality of the audio stream, we impose a constraint on the delay. We require the delay to be not larger than some value.

Our design problem is to determine a lower bound on the clock rate of the

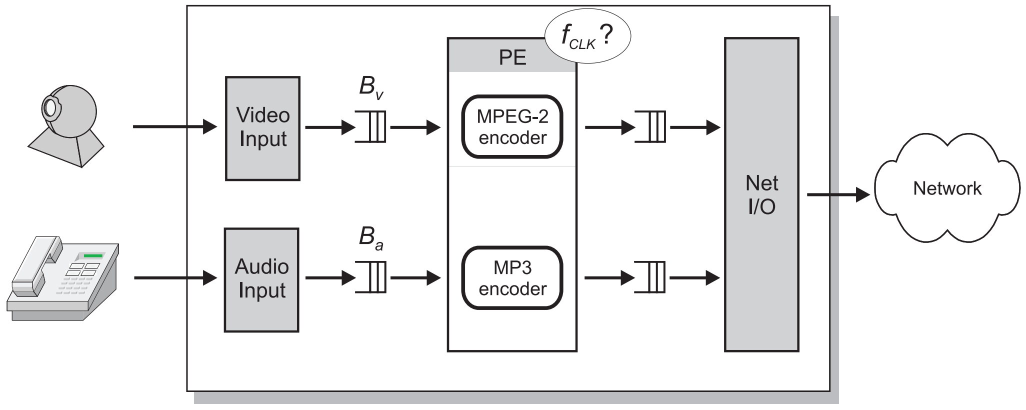


**Fig. 23:** Design scenario of a networked multimedia embedded system. The PE concurrently executes two tasks: one for MPEG-2 encoding of a video stream and the other for MP3 encoding of an audio stream. The MPEG-2 task has a higher priority than the MP3 task. The goal is to determine the minimum clock rate of the PE which satisfies a delay constraint associated with the audio stream.

PE which guarantees that the *delay constraint* for the audio stream is satisfied.

First, we analytically derive the lower bound on the clock rate, and after that, we explain how it can be computed in a practical setting.

**Lower bound on processor rate under delay constraint**

Let $D$ denote the delay constraint that we wish to satisfy for the audio stream. From Section 2.2.2 we know that the upper bound on the delay can be computed using (2.9). Hence, we require that this bound is smaller than our delay constraint $D$, i.e.

$$D \geq \sup_{\Delta \in \mathbb{R}_{\geq 0}} \left\{ \inf \left\{ \tau \geq 0 : \bar{\alpha}_a^u(\Delta) \leq \bar{\beta}_a^l(\Delta + \tau) \right\} \right\} \tag{3.48}$$

where $\bar{\alpha}_a^u$ and $\bar{\beta}_a^l$ denote event-based arrival and service curves of the audio stream, which have $\mathbb{Z}_{\geq 0}$ as their codomain. (We consider only integer quantities of stream objects.)

Let $e \in \mathbb{Z}_{>0}$ denote a number of stream objects. Then the delay constraint (3.48) can be expressed via pseudo-inverse functions of the arrival and service curves

$$\bar{\beta}_a^{l^{-1}}(e) \leq D + \bar{\alpha}_a^{u^{-1}}(e-1), \quad \forall e \in \mathbb{Z}_{>0} \tag{3.49}$$

Using Def. 4 we can restate (3.49) as follows

$$\bar{\beta}_a^l(D + \bar{\alpha}_a^{u^{-1}}(e-1)) \geq e, \quad \forall e \in \mathbb{Z}_{>0} \tag{3.50}$$

The constraint (3.50) is expressed in terms of event-based quantities. It says that in order to satisfy the delay constraint $D$ for the audio stream, we have to ensure that at least $e$ audio frames are completely processed within any time interval of length $D + \bar{\alpha}^{u^{-1}}(e-1)$. Since we are interested in finding the minimum rate of the PE expressed in terms of *clock cycles*, we have to apply a workload transformation to (3.50). This will give us a constraint on the amount of clock cycles required to satisfy (3.50). For the workload transformation we employ the execution demand curves of the audio task $(\gamma_a^l, \gamma_a^u)$. Under pessimistic assumptions we get

$$(\gamma_a^l \odot \bar{\beta}_a^l)(D + \bar{\alpha}_a^{u^{-1}}(e-1)) \geq \gamma_a^u \odot e, \quad \forall e \in \mathbb{Z}_{>0}$$

and using (3.36) we obtain

$$\beta_a^l(D + \bar{\alpha}_a^{u^{-1}}(e-1)) \geq \gamma_a^u(e), \quad \forall e \in \mathbb{Z}_{>0} \tag{3.51}$$

The left hand side of (3.51) denotes the required resource-based service curve for the audio stream, while the right-hand side denotes the upper bound on the number of processor cycles that may be required to completely process any number $e$ of consecutive audio frames.

From the problem definition we know that the video encoding task has a higher priority than the audio task. Hence, the video stream can acquire the full processor capacity, while the audio stream can get only the service which has been left after the processing of the video stream. To find the remaining service for the audio stream, we can use (2.4) as follows.

$$\beta_a^l(\tau) = \sup_{\forall\Delta\in[0,\tau]} \left\{\beta_v^l(\Delta) - \alpha_v^u(\Delta)\right\} \vee 0 \tag{3.52}$$

where $\beta_v^l$ and $\alpha_v^u$ denote the resource-based service and arrival curves of the video stream.

Using (3.52) and by noting that the service curve corresponding to the full processor capacity is determined as $f \cdot \Delta$, where $f$ denotes the processor clock rate, we can restate constraint (3.51) as follows.

$$\sup_{\forall\Delta\in[0,D+\bar{\alpha}_a^{u^{-1}}(e-1)]} \{f \cdot \Delta - \alpha_v^u(\Delta)\} \geq \gamma_a^u(e), \quad \forall e \in \mathbb{Z}_{>0} \tag{3.53}$$

To satisfy (3.53) for a given value of $e$, it is sufficient to select $f$ large enough such that there exists $\Delta' \in [0, D + \bar{\alpha}_a^{u^{-1}}(e-1)]$ for which

$$f(e) \cdot \Delta' \geq \alpha_v^u(\Delta') + \gamma_a^u(e)$$

holds true. Since we are looking for the lower bound on the clock rate we require that

$$f(e) \geq \inf_{\forall\Delta\in[0,D+\bar{\alpha}_a^{u^{-1}}(e-1)]} \left\{\frac{\alpha_v^u(\Delta) + \gamma_a^u(e)}{\Delta}\right\}, \quad \forall e \in \mathbb{Z}_{>0} \tag{3.54}$$

To ensure that for any $e \in \mathbb{Z}_{>0}$ the above constraint is satisfied, we take the maximum value of all possible values of $f(e)$.

$$f_{min} = \sup_{\forall e\in\mathbb{Z}_{>0}} \left\{ \inf_{\forall\Delta\in[0,D+\bar{\alpha}_a^{u^{-1}}(e-1)]} \left\{\frac{\alpha_v^u(\Delta) + \gamma_a^u(e)}{\Delta}\right\}\right\} \tag{3.55}$$

or, equivalently, if we use (3.23) we obtain

$$f_{min} = \sup_{\forall e \in \mathbb{Z}_{>0}} \left\{ \inf_{\forall \Delta \in [0, D + \bar{\alpha}_a^{u^{-1}}(e-1)]} \left\{ \frac{(\gamma_v^u \odot \bar{\alpha}_v^u)(\Delta) + \gamma_a^u(e)}{\Delta} \right\} \right\} \tag{3.56}$$

(3.56) gives the analytical lower bound on the PE's clock rate which guarantees that the delay constraint $D$ for the audio stream is satisfied.

**Computing the lower bound**

To see the gain from the application of VCC-based workload model, we compute the lower bound on the clock rate for two different cases. In one case, we use upper execution demand curve $\gamma_v^u$ that accounts for the per-frame variability of the execution demand of the MPEG-2 encoding task. Such a curve is analytically obtained in [167] from *type rate curves* (which represent another VCC type). In the other case, we use a conventional worst-case analysis approach: we get $\gamma_v^u$ under the pessimistic assumption that all frames within the video stream require the same, largest possible, number of cycles for their processing. I.e. we model the MPEG-2 encoding task by a single value—its worst-case execution demand. Other VCCs involved in the computation of the lower bound are identical in both cases.

As mentioned above, $\gamma_v^u$ which captures the execution demand variability of the MPEG-2 encoding task can be computed from the type rate curves. How this can be accomplished is out of scope of this thesis. The corresponding method is described in detail in [167]. For the purpose of this experimental study, we use $\gamma_v^u$ which was computed in [167] from a formal specification of the MPEG-2 encoding task. The formal specification was identical to one described in Ex. 3 (including the values of the worst-case execution times for different video frame types given in Tab. 2). Fig. 24 shows $\gamma_v^u$ obtained in [167] that we use in this experimental study for the calculation of the lower bound on the processor clock rate.

For comparison, in Fig. 24 we also plot $\gamma_v^u$ computed *without* using the type rate curves. In this case, the upper execution demand curve of the encoding task is determined as

$$\gamma_v^u(e) = e \cdot \max\{wced_I, wced_P, wced_B\}$$

where $wced_I$, $wced_P$ and $wced_B$ denote worst-case execution demands for the I-, P- and B-frame types, respectively. The values of the worst-case execution demands used in this experimental study are given in Tab. 2.

Besides the upper execution demand curve of the video encoding task $\gamma_v^u$, calculating the lower bound on the processor clock rate also requires knowing upper event-based arrival curves of the video stream $\bar{\alpha}_v^u$ and of the audio stream $\bar{\alpha}_a^u$ as well as upper execution demand curve of the audio MP3 task $\gamma_a^u$.

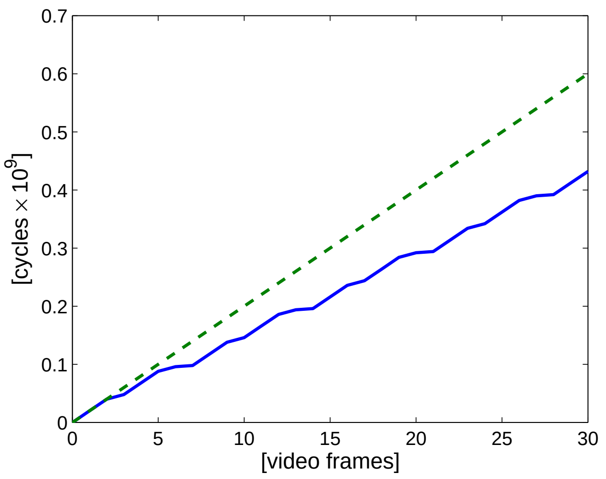


**Fig. 24:** Upper execution demand curves of the video encoding task which were used in the experimental study. The solid line corresponds to the execution demand curve obtained in [167] from the type rate curves. It captures the per-frame execution demand variability of the video encoding task. The dashed line corresponds to the execution demand curve which does not capture this variability.

We obtain $\gamma_a^u$ under a pessimistic assumption that all audio frames, in the worst case, require the same amount of cycles $wced_a$ to be processed, i.e.

$$\gamma_a^u(e) = e \cdot wced_a$$

According to our problem statement, the video and audio frames arrive at constant rates into the input FIFO buffers. Let $r_v$ and $r_a$ denote the video and audio input frame rates, respectively. Then, the corresponding arrival curves can be determined as $\bar{\alpha}_v^u(\Delta) = \lceil r_v \Delta \rceil$ and $\bar{\alpha}_a^u(\Delta) = \lceil r_a \Delta \rceil$. Note that in general the shapes of the arrival curves may be more complex.

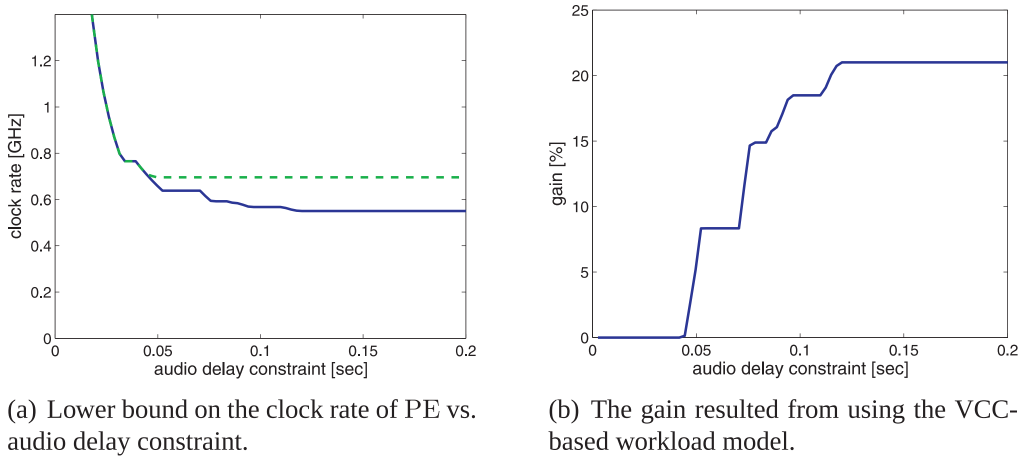
Finally, using (3.56) we can compute the lower bound on the $\mathrm{PE}$'s clock rate which guarantees that the audio delay constraint $D$ is satisfied. Note that in theory (3.56) has to be evaluated up to $e \to \infty$. This is to ensure that the value of the clock rate is high enough to sustain the overall average load imposed by the streams on $\mathrm{PE}$. However, computing up to $e \to \infty$ is impractical. We can overcome this problem by trading off the accuracy of the computation to the computational time. We can evaluate (3.56) up to any given number of stream objects $e_{max}$, but we have to put an additional constraint on the clock rate:

$$f_{min} \geq \frac{\gamma_v^u \odot \bar{\alpha}_v^u(\bar{\alpha}_a^{u^{-1}}(e_{max})) + \gamma_a^u(e_{max})}{\bar{\alpha}_a^{u^{-1}}(e_{max})} \tag{3.57}$$

With (3.57) we simply require that, on some time interval of length $\bar{\alpha}_a^{u^{-1}}(e_{max})$, the PE must provide the amount of service needed to fully process all video and

| Parameter | Value |
|---|---|
| Audio stream | |
| $r_a$ | $44100/1152$ fps |
| $wced_a$ | $5 \cdot 10^6$ cycles |
| Video stream | |
| $r_v$ | 25 fps |

**Tab. 3:** System parameters used in the experimental study of the design scenario shown in Fig. 23. The worst-case execution demand of the MPEG-2 encoding task for different video frame types is given in Tab. 2



(a) Lower bound on the clock rate of PE vs. audio delay constraint.

(b) The gain resulted from using the VCC-based workload model.

**Fig. 25:** Experimental results. On the left plot, the solid line corresponds to the clock rate bound computed with VCCs which capture the execution demand variability of the video encoding task; the dashed line corresponds to the bound computed using the traditional task characterization model.

audio frames that may arrive within that interval. If $e_{max}$ is too small, then the constraint (3.57) may be very pessimistic but the evaluation time of (3.56) can be short. If $e_{max}$ is large enough, then the right-hand side of (3.57) approaches the average case from above but the evaluation time may correspondingly increase.

### Experimental results and discussion

Using equation (3.56) in conjunction with the constraint (3.57) we computed the lower bound on the PE's clock rate for the system parameters specified in Tab. 3 and for a range of values of the delay constraint $D$. Fig. 25(a) shows the results of the computations for the case when the execution demand variability of the MPEG-2 encoding task is captured by $\gamma_v^u$ and for the case when it is not captured by $\gamma_v^u$. The latter case represents the traditional task characterization model in which the execution demand of a task is modeled by a single value.

In Fig. 25(b), we plot the gain resulted from the usage of the VCC-based workload model. For different values of the audio delay constraint, the plot shows the percentage by which the lower bound on the clock rate computed using $\gamma_v^u$ that captures the execution demand variability of the video task is smaller than the corresponding bound computed using $\gamma_v^u$ that does not capture this variability (i.e. which was obtained under the assumption that all video frames have the same execution requirement in the worst case). By inspecting the plot in Fig. 25(b), we can see that for large values of the delay constraint we can gain up to 20% of savings in the clock rate by capturing the workload variability using VCCs.

## 3.7 Summary

In this chapter, we presented two central to this thesis concepts — *Variability Characterization Curves* and *Workload Transformations*. We discussed different ways to obtaining VCCs and defined several VCC types for modeling multimedia workloads. We also proposed and justified the optimal placement of the workload transformations in an MPA scheduling network. Finally, we presented results of an experimental evaluation of the VCC-based workload model. Our experiments showed that the VCC-based model results in significantly tighter analytic bounds than a traditional model. The simulation study suggested that these bounds represent useful estimates of system properties and therefore can serve as a sound basis for making design decisions. In this simulation study we also investigated the effect of grouping the traces for obtaining VCCs on the tightness of the analytic bounds. We came to the conclusion that in certain design contexts it might be of advantage to distinguish between different workload classes.

In the rest of this thesis, we demonstrate the utility of the VCC-based workload model and of the workload transformations in different design contexts of multimedia MpSoC architectures. In particular, in the next chapter we further explore the idea of distinguishing between different workload classes: we employ VCCs for automatic exploration of the *workload* design space.

# 4

# Workload Design

Chapter 3 introduced the concept of Variability Characterization Curves (VCCs) and demonstrated how they can capture different properties of multimedia workloads. This chapter shows how VCCs can help to address an emerging and challenging problem of *workload design* for system-level performance evaluation of multimedia MpSoC architectures.

"Workload design" refers to a process of selecting representative workload for performance evaluation and comparison of computer architectures [39]. It is a well recognized problem in the domain of microprocessor design. In this domain, different program characteristics that influence the selection of a representative workload include microarchitecture-centric properties such as cache miss rates, instruction mix and accuracy of branch prediction. However, workload properties that are pertinent to the context of system-level design of multiprocessor SoC architectures are very different. To date the problem of workload design, in this specific context, has not been sufficiently addressed. This chapter presents results appeared in [114], which suggests how this problem can be approached in the specific case of media processor design. This chapter has the following structure:

- Section 4.1 introduces and motivates the problem of workload design for system-level performance evaluation of multiprocessor SoC platform architectures.
- Section 4.2 summarizes relevant research work.
- Section 4.3 outlines the proposed approach.
- Section 4.4 proposes VCCs as a basis for workload characterization necessary for quantitative comparison and classification of media streams.

- Section 4.5 describes how various media streams can be classified based on VCCs characterizing them.
- Section 4.6 presents results of an empirical validation of the proposed workload classification method.
- Finally, Section 4.7 concludes this chapter.

## 4.1 Introduction

A typical design process of a complex embedded system such as a multimedia MpSoC platform involves a thorough exploration of the available design space. Starting from some template architecture — the platform, system designers iteratively evolve this platform with the goal to arrive at an architecture which would be optimal for the target application range. Searching for the optimal architecture necessitates evaluating and comparing to each other many alternative platform configurations. Traditionally, performance evaluation of architectures heavily relies on simulations. For system designers to be sure in representativeness of performance numbers obtained from such simulations, ideally, each candidate architecture needs to be simulated for a large and diverse set of possible inputs (or application scenarios). However, in most cases this is impractical. This is because simulating a single design point may be prohibitively expensive in terms of the simulation time. For example, simulation of only a few minutes of video, for a video decoding application, may take tens of hours [165]. This significantly limits the number of different inputs for which simulations can be performed within an allotted design time. Therefore, from the large set of possible inputs, the system designers have to choose a small subset which would be *representative*[1] of the workload that the system would experience in reality. Simulations can then be restricted to this subset only.

Obtaining a representative input set is, of course, not a new concern—workload design and other relevant problems, such as workload characterization, benchmark construction, synthetic workload generation, etc., are well recognized problems in different areas of computer performance evaluation. However, issues involved in solving these problems are almost always domain-specific. This is because they depend on the nature of applications and architectures, and on the abstraction level at which systems are evaluated. For example, the main issues

[1]Many sources in the literature refer to "representative workload" as the workload (e.g. a collection of traces) which represents a *realistic* general case, and is not biased towards a particular architecture, environment, etc. This term is often used in conjunction with a *synthetic workload*, to indicate that the synthetic workload closely resembles the *real* workload. In context of this thesis, we use the term "representative workload" in a slightly different way. By "representative workload" we mean a workload which was selected from a larger collection of real workloads, such that it best represents (*covers*) all important classes of the workload within this collection.

in the domain of microprocessor design are microarchitecture-centric, where a designer is mostly concerned with program characteristics like instruction mix, data and instruction cache miss rates and branch prediction accuracy. On the other hand, the concerns in the case of system-level design of SoC platform architectures are very different and these are not suitably reflected in a benchmark suite designed for microarchitecture evaluation.

In this chapter, we attempt to address this issue of workload design in the specific context of system-level design of SoC platform architectures for multimedia processing (i.e. media processors). Although simulation-oriented design and evaluation are widespread in the domain of system-level SoC design, to the best of our knowledge the issue of methodically selecting representative inputs for architecture evaluation has not received any attention so far. Most of the work reported in the Embedded Systems literature, on novel system models or simulation schemes, shirk off this problem and leave the responsibility of choosing a representative input or stimuli to the architecture on system designers (see, for example, [83]).

There are many reasons why this problem is interesting in the specific case of multimedia processing on multiprocessor SoC platforms. First, media streams may impose very complex and diverse workloads on such platforms. Many multimedia applications exhibit a large degree of data-dependent variability that complicates the problem of choosing a representative input set. Second, in contrast to general-purpose architectures, MpSoC platforms, which are optimized for stream processing, have heterogeneous distributed architectures. This fact further complicates the problem. Third, multimedia processing is in general computationally intensive, requiring for performance evaluation to simulate a relatively large number of events. This makes selection of the representative workload for design of media processors an important problem.

Arbitrarily selecting inputs to form the “representative” input set is certainly not a good idea. The goal of “representative” workload design should be to select inputs such that they cover, as much as possible, the whole space of possible workloads, including those that represent *corner cases* for the target architecture. Such corner cases are represented by inputs which impose worst- and best-case loads on different parts of the architecture. Determining what constitutes a “corner case” is, however, not a trivial undertaking due to the complex nature of most multimedia workloads. Attempts towards using some *qualitative* (i.e. subjective) technique to judge the properties of media streams based on their content (for example, by simply viewing video clips to be processed by the architecture and classifying them based on experience or intuition) might easily fail. Hence, a quantitative methodology is necessary, using which it should be possible to objectively assess and compare the properties of different media streams. Based on such a comparison, a small *representative* subset can then be chosen from a large collection of samples.

We propose a methodology to classify media streams which can be used to identify a small representative set meant for architecture evaluation. Towards this, we first hypothesize that key characteristics of media streams that influence the performance of an MpSoC platform architecture, are related to their "variability". This variability stem from the fact that execution time requirements of multimedia tasks and the amount of data consumed (produced) by these tasks at their inputs (outputs) depend on the properties of particular audio/video samples being processed. Now, given a collection of media streams, we classify two streams from this collection as *similar* if both of them exhibit the same kind of variability with respect to the execution time requirements and the task input/output rates, as mentioned above. Therefore, given a set of video streams which are *similar*, it would be sufficient to simulate an architecture with only one video stream from this set, as all the other streams would impose similar load on the architecture. To quantitatively characterize the variability associated with a stream, with respect to a given architecture, we use the concept of VCCs, introduced in Chapter 3. As an illustration of our methodology, throughout this chapter we use a case study of an MPEG-2 decoder system whose system-level architecture, including the mapping of MPEG-2 tasks onto it, is shown in Fig. 17 and described in Section 3.6.1.

We would like to point out here that the kinds of variabilities that should be considered in a media stream for an effective classification would depend on the platform architecture and the application at hand. This is mainly due to the fact that SoC architectures are often highly specialized for a narrow application spectrum. For this reason, defining a set of workload attributes which would result in an effective stream classification in *any* design scenario is difficult, if not impossible. Also, we note that defining a common benchmark for multimedia MpSoC platforms is out of scope of this chapter. The contribution of this chapter is to point out that the properties of media streams which should be considered for representative workload identification in the context of performance evaluation of multimedia MpSoC platforms can be expressed in the form of VCCs, and to propose the corresponding stream classification method.

Finally, we note that, in a system-level design framework, the selection of the representative workload can be carried out outside of the time-critical design space exploration loop (namely, prior to the exploration) and does not require time-consuming system simulations. Furthermore, the workload characterization and classification procedures presented in this chapter can be fully automated, reducing to minimum designers' participation in the workload selection process.

## 4.2 Related Work

The construction of representative workloads for performance evaluation of computer systems has always been an area of active research since early 70s (see

[148] and references therein). Since then the term *workload* has been widely understood as a mix of programs (or jobs, or applications) for which the performance of a computer system was evaluated. Domain-specific collections of such programs, called *benchmarks*, have been designed and widely used as a standard means to evaluate and compare computer architectures. Examples of these, in the multimedia domain, are MediaBench [86] and the Berkeley multimedia workload [146]. Design of such representative workloads was mainly concentrated on proper selection of the *programs* to be included in the workload. The selection of corresponding input data sets was limited to the definition of their size (e.g. sampling rate, resolution etc.) The dependency of program behavior on the values of the input data sets did not receive enough consideration in the process of forming such representative workloads.

Recently Eeckhout et al. [39] have shown that the *workload design space* may be very complex and therefore should be systematically explored during the construction of representative workloads. Their workload design space consists of *program-input pairs* that capture both, the variety of programs as well as various input data sets to those programs. They use techniques such as principle component analysis and cluster analysis to efficiently explore the space of possible workloads and select representative program-input pairs from it.

The problem of reducing simulation time has been addressed using *trace sampling techniques* (see [82] and references therein). The goal of such techniques is to identify representative fragments in the program execution and simulate only those fragments, thereby eliminating the need for simulating the entire program. Trace sampling techniques heavily rely on the characterization and classification of the workload imposed on the architecture by the different fragments in the program execution trace. However, it should be noted that all the above mentioned research efforts were primarily targeted towards characterization and composition of representative workloads in the domain of microprocessor design.

## 4.3 Overview

We assume that system designers have at their disposal a large collection of media streams that fully represents streams which the designed system may have to process in reality. Then, forming a representative workload from this large collection involves several steps:

1. **Identifying key workload properties:** The first step in the workload design is to decide which workload properties are important for the given design context. Based on these properties the workload classification will be carried out. In the case of performance evaluation, we have to select those properties that have largest influence on the performance of the architecture.

2. **Characterizing media streams:** The next step is to characterize each stream in the collection. It is accomplished by measuring the properties upon which the workload classification will be performed. We refer to this step, together with the preceding step, as *workload characterization*.

3. **Defining the dissimilarity metric:** Based on the workload characterization, we need to define how the dissimilarity between two workloads (media streams) will be measured. That is, we need to define a *metric* that would represent the dissimilarity between a pair of media streams as a value. After all media streams in the collection have been characterized (i.e. their relevant properties have been measured), we compute this metric for each pair of streams in the collection.

4. **Classifying media streams:** Having computed the pairwise dissimilarity between the streams in the collection, we can identify groups of streams that may impose similar workload on the architecture. Such a group would consists of streams that have similar properties. From these groups, we can then select representatives. They will form the representative workload for our architecture. We refer to this step and the preceding step as *workload classification*.

The next two sections describe particular considerations that we made while realizing the above steps.

## 4.4 Workload Characterization

Workload characterization should be based on *key properties* that are important in a particular design context. These are properties that have a strong impact on the performance of the architecture being designed. For instance, in microarchitectural design such properties would be instruction mix, branch prediction accuracy and cache miss rates [39]. As mentioned in Section 4.1, our hypothesis is that *on the system level* the performance of multimedia MpSoC architectures is largely influenced by various kinds of *data-dependent variability* associated with the processing of media streams. This hypothesis rests on the observation that such variability is the major source of the burstiness of on-chip traffic in such multimedia MpSoC platforms [165]. The burstiness of the on-chip traffic necessitates insertion of additional buffers between architectural entities processing the media streams, and deployment of sophisticated scheduling policies across the platform. Both of these inevitably translate into increased design costs and power consumption [57].

Individual media streams with *identical* parameters, such as bit rate, frame rate and resolution, may impose significantly *different* workload on the architecture; in particular, the streams may exhibit different kinds of the variability. This phenomenon can be explained by the fact that, although these streams have identical parameters, they contain diverse multimedia information and may have different structure (e.g., in MPEG video streams, different frame types may be arranged in various patterns). For MPEG-2 video streams, this fact is supported by our experiments reported in Section 4.6. [57] demonstrates the variability in streams of several other multimedia formats. Therefore, in the system-level design context of multimedia MpSoC architectures, it is certainly meaningful to characterize and classify multimedia workloads with respect to their variability properties.

In a typical MpSoC architecture, consisting of a heterogeneous collection of interconnected PEs, often there are several sources of variability that depends on properties of the processed media streams. Consider, as an example, the MPEG-2 decoding system shown in Fig. 17 in Chapter 3. The system consists of two programmable processors, $\mathrm{PE1}$ and $\mathrm{PE2}$, and input and output interfaces. $\mathrm{PE1}$ executes a task performing VLD and IQ functions, whereas $\mathrm{PE2}$ executes a task performing IDCT and MC functions of the MPEG-2 decoding algorithm. For brevity, we will refer to these tasks as VLD and IDCT, respectively. In Fig. 17, stream objects belonging to the input stream emerging from the network interface are single bits. Stream objects sent from $\mathrm{PE1}$ to $\mathrm{PE2}$ are partially decoded macroblocks, whereas stream objects entering the video interface are fully processed macroblocks. What are the sources of variability associated with media streams processed on such an MpSoC platform?

- First, arrival patterns of media streams at the input of the system may have a bursty nature, i.e. stream objects may arrive at the system's input in highly irregular intervals. A typical example of this is a multimedia device receiving streams from a congested network.

- Second, each activation of a task may consume and produce a variable number of stream objects from the associated streams. For example, each activation of VLD in Fig. 17 consumes a variable number of bits from the network interface, although it always produces one macroblock at its output.

- Third, the execution demand of a task may vary from activation to activation due to data-dependent program flow. Both the tasks in our running example of the MPEG-2 decoder—VLD and IDCT—possess this property.

- Finally, stream objects belonging to the same stream may require different amounts of memory to store them in communication channels between PEs. Again, in the example architecture in Fig. 17, we note that the partially decoded macroblocks stored in buffer $B_2$, depending on their type, may or may not include motion vectors.

All these types of variability must be carefully considered and characterized during the workload design process. In this chapter, we will be concerned with the variability of the execution demand and the consumption and production rates of tasks. As mentioned before, depending on the architecture and the application at hand, it might be meaningful to consider other types of variabilities as well.[2] However, we show that the two variability types we consider here already lead to meaningful results.

We propose to use VCCs introduced in Section 3.2 as a model for the workload characterization that can capture different kinds of variability in media streams. To each stream $i$ in a stream collection, we associate a set of tuples $\mathcal{S}_i = \{(\mathcal{V}^l_{\mathcal{P}}, \mathcal{V}^u_{\mathcal{P}})\}$, where $\mathcal{V}^l_{\mathcal{P}}$ and $\mathcal{V}^u_{\mathcal{P}}$ denote lower and upper VCCs characterizing variability of property $\mathcal{P}$ in stream $i$. For any $\mathcal{P}$, $\mathcal{V}^l_{\mathcal{P}}$ and $\mathcal{V}^u_{\mathcal{P}}$ must represent the *tightest upper and lower VCCs*, as defined by Defs. 14 and 15. To obtain such tightest $\mathcal{V}^l_{\mathcal{P}}$ and $\mathcal{V}^u_{\mathcal{P}}$, we use (3.42) and (3.43) under the condition that the *function set* for which the VCCs are calculated contains only one function—the function which corresponds to property $\mathcal{P}$ of the stream being characterized. Thus, the resulting VCCs, $\mathcal{V}^l_{\mathcal{P}}$ and $\mathcal{V}^u_{\mathcal{P}}$, represent *tightest* bounds on the worst- and best-case variability of property $\mathcal{P}$ in a *single* stream. The set of all such VCCs, $\mathcal{S}_i$, represents a complete characterization of stream $i$. As a result, in form of $\mathcal{S}_i$, we have an accurate, compact and easy-to-obtain abstraction of stream $i$.

In Section 3.3, we defined some VCC types useful for multimedia workload characterization. Out of them, for the workload design in this chapter, we use the execution demand curves and the consumption and production curves, denoted by tuples $(\gamma^l, \gamma^u)$, $(\kappa^l, \kappa^u)$, and $(\pi^l, \pi^u)$, respectively. Each task in a multimedia stream-processing application is characterized by these VCC types.

We note that $(\gamma^l, \gamma^u)$ depends on the PE type on which the corresponding application task is to be executed. For example, if a PE has an application specific instruction set which may significantly alter the execution demand of tasks, then this will be reflected in $(\gamma^l, \gamma^u)$. In contrast, $(\kappa^l, \kappa^u)$ and $(\pi^l, \pi^u)$ are not dependent on the architecture, but on the dataflow properties within the application. Hence, the assumption here is that, prior to the workload characterization, we need to know the partitioning of the application into tasks and the mapping of those tasks onto PE types. We believe that this assumption is not too restrictive. Similar assumptions are common in the Embedded Systems design community, where applications are modeled by task graphs in which the (worst-case) execution demands of tasks are known. In fact, the *same* assumption is made in a number of trace-based performance evaluation techniques recently reported in the literature [74, 75, 83, 116]. These trace-based performance evaluation techniques rely on pre-collected execution traces of tasks on PEs of the MpSoC platform being evaluated. Our method can be useful especially in this context, by providing such techniques with the representative set of traces.

[2]It might be also necessary to account for correlations existing between different variability types. However, this question goes beyond the scope of this chapter.

## 4.5 Workload Classification

In the previous section, we described how media streams can be quantitatively characterized to enable their comparison and classification. In this section, we explain *how* such a comparison and classification can be accomplished based on this characterization.

We propose to classify streams based on the *shapes* of the VCCs associated with them. If two streams are characterized by VCCs having similar shapes, then their behavior, in the worst/best-case, will also be similar. Each stream might be associated with several VCC types, characterizing different aspects of variability within the stream. Therefore, if two streams have similarly shaped VCCs of respective types, then these streams will impose similar workload on the architecture (in the worst- and best-case). For example, the maximum backlogs that such streams will create in the buffers of the architecture as a result of their processing will almost be the same.

### 4.5.1 Dissimilarity based on a single VCC type

Let us first define a metric that would allow to compare two streams based on only one VCC type. This metric should be a measure of *dissimilarity* between shapes of two VCCs of the same type.

In general, any measure of dissimilarity between two objects depends on the specific problem at hand [45]. Each property, based on which two objects are to be compared, is associated with a variable. That is, if there are $n$ properties upon which two objects have to be compared, then there will be $n$ variables describing each object. Any valuation of these variables constitutes a representation of an object. The dissimilarity between two objects is then found by computing some metric defined over these $n$ variables. In our case, a VCC, which is defined for a set of points $k = 1, 2, .., n$, can be seen as an object described by $n$ variables.

Intuitively, to see how dissimilar the shapes of two VCCs are, we need to compare their values for each of the points $k = 1, 2, .., n$. By noting that all $n$ variables represent a VCC along essentially *separable* dimensions, we can quantitatively measure the dissimilarity between two VCCs using the City Block metric [45]. We decided to use this metric as a measure of dissimilarity, because, in comparison to other known metrics (e.g. Euclidean Distance), it is more "sensitive" to differences in each of the dimensions (variables). I.e., in our case, the metric is more "sensitive" to the differences in the shapes of two VCCs. A formal definition of the dissimilarity between two VCCs based on the City Block metric is given below.

Let $\mathcal{V}_i^r(k)$, $k = 1, 2, .., n$, denote a VCC of type $r$ associated with stream $i$. The measure of the pairwise dissimilarity between two streams $i$ and $j$, with respect to VCC $\mathcal{V}^r$, is then defined as

$$d_{rij} = \sum_{k=1}^{n} \omega_r(k)\ |\mathcal{V}_i^r(k) - \mathcal{V}_j^r(k)| \tag{4.1}$$

where $\omega_r(k) = 1/k$ are weights that are necessary to normalize the differences $|\mathcal{V}_i^r(k) - \mathcal{V}_j^r(k)|$ with respect to the length $k$ of the analysis interval. The longer the analysis interval $k$ is, the less *critical* the difference in the values of the two VCCs becomes. For example, suppose that we want to compare upper execution demand curves of two streams. Assume from the execution demand curves we know that any two consecutive stream objects (i.e. $k = 2$) in the first stream may cause a maximum execution demand of 100 units, while for the second stream this value is 150. Suppose that we also know that any 10 consecutive stream objects ($k = 10$) in the first stream may cause a maximum execution demand of 1000 units, and it is 1150 units for the second stream. Although the absolute difference between the curves for $k = 2$ is smaller than that for $k = 10$, the difference in the execution demand computed *per stream object* for $k = 2$ is larger than for $k = 10$ ($\frac{|100-150|}{2} > \frac{|1000-1150|}{10}$). For $k = 10$ the absolute difference is *distributed* over a larger number of stream objects than in the case of $k = 2$, and therefore this difference becomes less critical. The weights $\omega_r$ allow to correctly compare the streams for large $k$, e.g. to properly account for long-term average execution demands.

### 4.5.2 Dissimilarity based on several VCC types

Media streams may be characterized by more than one VCC type. How can the dissimilarity between the streams be quantified then?

First, we propose to compute the dissimilarities between VCCs of identical types as defined by (4.1). Then, the computed “single-type” dissimilarities can be combined in a variety of ways. One possibility is to simply sum up all of them. Our experiments showed that this simple approach works relatively good. Hence, we define the pairwise dissimilarity between two streams $i$ and $j$ with respect to VCCs of types $r \in \mathcal{R}$ as [114]

$$d_{ij} = \sum_{\forall r \in \mathcal{R}} d_{rij} \tag{4.2}$$

An improved version of (4.2) sums up *normalized* dissimilarities $\frac{d_{rij}}{\max_{\forall i,j} d_{rij}}$, i.e.

$$d_{ij} = \sum_{\forall r \in \mathcal{R}} \frac{d_{rij}}{\max_{\forall i,j} d_{rij}} \tag{4.3}$$

### 4.5.3 Clustering

Finally, to *classify* streams using the dissimilarity measures described above, we employ a conventional hierarchical clustering algorithm which uses the *complete* *linkage* algorithm [45] to compute distances between clusters. The choice of the complete linkage algorithm was motivated by the need to keep the clusters as dense as possible.

## 4.6 Empirical Validation

To see how the workload classification method described in previous sections performs on real data samples, we conducted a number of experiments with MPEG-2 video streams. MPEG-2 streams represented an interesting target for our experiments because they have a complex nature and a rich set of characteristics [78].

**Workload design scenario**

Consider the following design scenario. Suppose our goal is to study the impact of different MPEG-2 streams on the MpSoC platform shown in Fig. 17 in Chapter 3. At our disposal we have a large library of video clips that our architecture should be able to support. However, due to design time constraints we cannot afford to simulate the platform architecture for each clip in the library. Furthermore, since simulation of an entire clip takes a prohibitively long time, we are constrained to simulating only *short fragments* extracted from *selected* video clips in the library.

We assume that any video clip in the library contains only one *scene*. In a visual sense, a scene is "*a portion of the movie without sudden changes in view, but with some panning and zooming*" [78]. Distinguishing between different scenes is necessary, because even within a single MPEG-2 stream different scenes might have substantially different characteristics. For example, characteristics of MPEG-2 streams (such as bit rate) may *significantly* vary at a large time scale, i.e. across different scenes, while at a short time scale (i.e. within a scene) the variations are more moderate [78, 84]. Since VCCs represent worst/best-case bounds, if different scenes are not treated separately while deriving their VCCs, then details about variability in some scenes may be "overshadowed" by other scenes. Finally, we note that in practice it is always possible to split a long movie into a series of individual scenes (see [78] for the relevant references).

For our experiments, we used a library of MPEG-2 video clips summarized in Tab. 4. Each clip in the library is an 8 Mbps constant-bit-rate stream, containing one scene with resolution $704 \times 576$ pixels and frame rate 25 fps. We believe that the variety of scenes represented in this library is sufficient for the demonstration of our workload design method.

To select representative streams, for performance evaluation of the platform architecture shown in Fig. 17, we classified the streams in the library based on (i) the variability in the execution demand, and (ii) the variability in the production

| video | file name | video | file name |
|---|---|---|---|
| 1 | 100b_080.m2v | 7 | pulb_080.m2v |
| 2 | bbc3_080.m2v | 8 | susi_080.m2v |
| 3 | cact_080.m2v | 9 | tens_080.m2v |
| 4 | flwr_080.m2v | 10 | time_080.m2v |
| 5 | mobl_080.m2v | 11 | v700_080.m2v |
| 6 | mulb_080.m2v | | |
| **Source**: `ftp.tek.com/tv/test/streams/Element/MPEG-Video/` | | | |

**Tab. 4:** MPEG-2 video clips used in the experiments

and consumption rates of the MPEG-2 tasks to be executed on the PEs of the platform. VLD task can be characterized by both variability types. Its execution demand varies and, per execution, it consumes a variable number of bits from its input. Hence, we characterized VLD task using execution demand curves $(\gamma^l_{VLD}, \gamma^u_{VLD})$ and consumption curves $(\kappa^l_{VLD}, \kappa^u_{VLD})$. In contrast, IDCT task was characterized by execution demand curves $(\gamma^l_{IDCT}, \gamma^u_{IDCT})$ only. This is because its execution demand can vary, but the consumption and production rates remain constant.

**Experimental setup**

Our simulation environment consisted of the SimpleScalar instruction set simulator [8], a system simulator and an MPEG-2 decoder program [119]. The MPEG-2 decoder program was used as an executable for both simulators and as a means to obtain traces of bit allocation to macroblocks. The system simulator served for validation of our workload design method. It consisted of a SystemC [153] transaction-level model of the architecture in Fig. 17. This model was based on the simulation environment described in Appendix A.

The SimpleScalar simulator served for modeling $\mathrm{PE1}$ and $\mathrm{PE2}$ of the platform architecture in Fig. 17. It was used to obtain traces of execution times for VLD and IDCT tasks. Both tasks worked at the macroblock granularity. In our experimental setup, the SimpleScalar simulator was used in `sim-profile` configuration and with the PISA instruction set [8]. Although this configuration does not model advanced microarchitectural features of a processor, such as caches, branch predictors, etc., it requires less time to simulate, and therefore was a suitable choice for our purposes. This choice was also justified by the fact that advanced features in the microarchitecture of general purpose processors do not significantly impact the variability of multimedia workloads [57].

VCCs, $(\gamma^l_{VLD}, \gamma^u_{VLD})$, $(\kappa^l_{VLD}, \kappa^u_{VLD})$ and $(\gamma^l_{IDCT}, \gamma^u_{IDCT})$, were obtained from the execution traces using the trace analysis method described in Section 3.5.2. We set the maximum analysis interval to 12 video frames. This corresponded to the most frequently occurring length of group of pictures (GOP) [118] in the

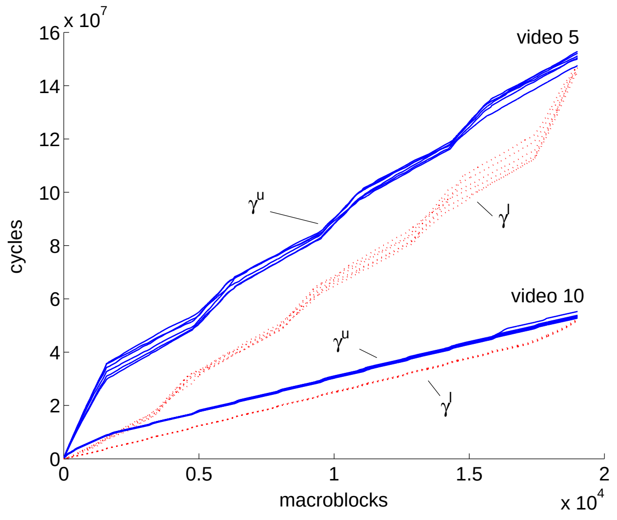


**Fig. 26:** $(\gamma^l_{VLD}, \gamma^u_{VLD})$ for different fragments of video 5 and video 10

MPEG-2 bitstreams. Obtaining the VCCs relied only on the instruction set simulation and a simple trace-analysis algorithm, both of which can be orders of magnitude faster compared to a full system simulation

### Results and discussion

Our first step was to compute the maximum dissimilarity between VCCs obtained from different fragments of the *same* scene (i.e. clip). The goal was to check whether this dissimilarity would be sufficiently small to allow for using a *randomly* selected short fragment as a representative of the whole video clip. If the dissimilarity were too large, then randomly selecting fragments from the clips would not be a good strategy, and we would need to look for other approaches for selecting fragments from the clips. For example, all fragments of a scene could be classified first using the method presented in this chapter, and then *several* fragments with diverse charcateristics could be chosen to represent that scene.

From each clip in the library, we extracted 10 unique fragments of the same length (30 frames) and measured their VCCs. Fig. 26 shows measurement results for $(\gamma^l_{VLD}, \gamma^u_{VLD})$ for two video clips—*video 5* and *video 10* from Table 4. *Video 5* contains a natural full-motion scene, whereas *video 10* is a video test pattern with a small running timer on a still background. By inspecting the plots in Figure 26, we can see that the dissimilarity between fragments of *video 5* is larger than the dissimilarity between fragments of *video 10*. This can be explained by a higher degree of motion present in the scene of *video 5*. Nevertheless, we can see that the curves for different fragments of *video 5* exhibit a similar behavior. For other videos in the library, we observed same trends.

Using (4.1), for each VCC type and each video clip in the library, we com-

| VCC | max.dissim | video | VCC | max.dissim | video |
|---|---|---|---|---|---|
| $\gamma^u_{VLD}$ | 57151356 | 4 | $\gamma^l_{IDCT}$ | 37220944 | 3 |
| $\gamma^l_{VLD}$ | 23548299 | 4 | $\kappa^u_{VLD}$ | 2146073 | 4 |
| $\gamma^u_{IDCT}$ | 22903156 | 9 | $\kappa^l_{VLD}$ | 752238 | 4 |

**Tab. 5:** Maximum dissimilarities between fragments of the same clip for each VCC type.

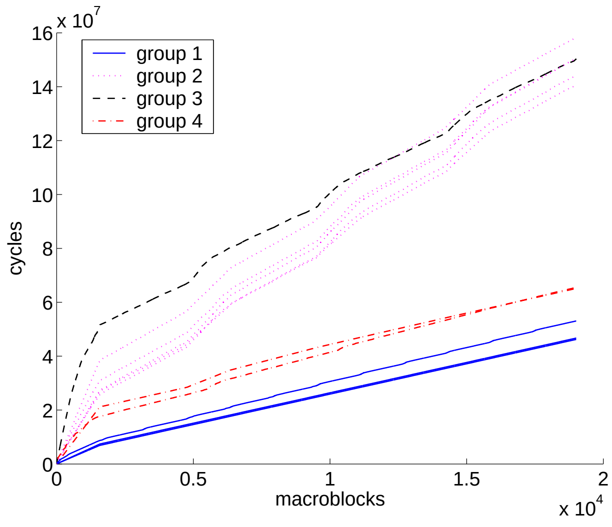


**Fig. 27:** Classification based on $\gamma^u_{VLD}$ only

puted pairwise dissimilarities between fragments of the same clip. Tab. 5 summarizes results of this experiment. It shows the *maximum* dissimilarities for each VCC type over the whole set of video clips. From this table, we can conclude that *video 4* probably contains a very complex and changing scene, because for almost all VCC types its fragments exhibit larger dissimilarity between each other than the fragments of other clips.

Based on the above results, for the classification of video clips in the library, we decided to randomly pick one fragment from each clip and then perform the classification based only on these selected fragments.

For the purpose of illustration, we first performed the classification based on only *one* VCC type, $\gamma^u_{VLD}$. The results of the $\gamma^u_{VLD}$-based classification into four groups are presented in Fig. 27. As we can see in the figure, our method could correctly identify groups of curves having similar shapes. This indicates that the measure of dissimilarity defined by (4.1) and the chosen clustering algorithm lead to a meaningful classification.

Fig. 28 shows a *dendrogram* of the hierarchical cluster tree obtained as a result of the classification based on all six VCC types and by using (4.2). In this dendrogram, we can clearly distinguish between two major groups of clips:

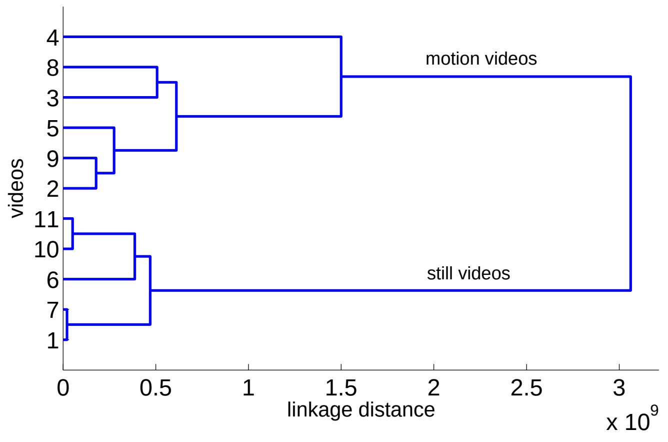


**Fig. 28:** Cluster tree

| video | $B_2$ | $B_v$ | video | $B_2$ | $B_v$ |
|---|---|---|---|---|---|
| 1 | 8282 | 9433 | 4 | 4443 | 8732 |
| 2 | 5128 | 9027 | 7 | 8390 | 9593 |
| 3 | 7953 | 8867 | 9 | 3018 | 9272 |

**Tab. 6:** Measured maximum buffer backlogs

still and motion videos.[3] This kind of a coarse-grained division into two groups would have been possible to obtain just by viewing the videos on the screen. However, a more refined classification would be difficult to achieve using such a subjective technique. For example, before performing the experiments, by simply viewing the clips we could not predict that *video 4* would have such different properties in comparison to the other motion videos. However, we can easily see this in the dendrogram: all other motion videos except *video 4*, form a tight cluster with the maximum linkage distance almost three times smaller than the maximum linkage distance when *video 4* is included into the cluster.

Finally, to see how the results of the stream classification correlate with the actual impact of the streams on the architecture, we performed simulations of the system in Fig. 17. We simulated the decoding of several *full-length* video clips from the library. As a measure of the architectural impact we decided to use maximum backlogs occurring in buffers $B_2$ and $B_v$ in Fig. 17. The backlog in buffer $B_1$ at the $\mathrm{PE1}$'s input was not taken into account because of its relatively small size.

Tab. 6 summarizes results of the system simulations. Our buffer measurements show that, for example, *video 1* and *video 7* produce very similar maximum backlogs in both buffers. The maximum backlogs produced by *video 9* and *video 2* are less similar than the backlogs produced by *video 1* and *video 7*. For

[3]Since *video 10* is mostly still, it was assigned to the group of still videos by our method.

*video 9* and *video 2*, the differences in the backlogs in $B_2$ and $B_v$ are 2110 and 245 macroblocks, respectively. We can also see that *video 9* is more similar to *video 2* than to *video 3*. The maximum backlogs for *video 3* and *video 9* differ in 4935 and 405 macroblocks for $B_2$ and $B_v$, respectively. Hence, we can conclude that the simulation results exhibit the same tendency as that shown by the classification in Fig. 28.

## 4.7 Summary

In this chapter, we presented an approach for workload design in the specific context of system-level performance evaluation of multimedia MpSoC architectures. The two main contributions of this chapter were: (i) establishing the utility of VCCs as a model for multimedia workload characterization, and (ii) a workload classification method based on VCCs which allows to identify groups of media streams that impose similar workload on a platform architecture. System designers can use this classification method for constructing small representative workload sets for performance evaluation of MpSoC platforms for multimedia processing. We presented experimental results that validate and show usefulness of this approach. However, there is a considerable scope for further research in this direction. For example, a more systematic study needs to be done to identify "variability types" beyond the ones considered in this chapter.

# 5

# Designing Stream Scheduling Policies

In Chapter 4, we demonstrated one possible application of Variability Characterization Curves: we proposed a method to automate the selection of representative workload for performance evaluation of the multimedia MpSoC architectures. In this chapter, we demonstrate another application of VCCs. The focus of this chapter is on design of *platform management policies* for such MpSoC architectures.

A platform management policy specifies how the computational and communication resources of an execution platform should be shared among application tasks, i.e. it defines the scheduling and arbitration[1] policies implemented on these resources. These policies are the knobs which designers can use to tune up the system such that a desired tradeoff is achieved between system's cost, performance and power consumption. That is why a proper selection and optimization of platform management policies play an important role in the system-level design of multimedia MpSoC platforms.

There is a large body of research work on real-time scheduling, covering a broad spectrum of applications—from control-dominated to signal processing systems. However, the scheduling problems arising in the domain of media processing execution platforms (such as multimedia MpSoC architectures) involve a number of specific issues that are not effectively addressed by the existing scheduling methods: Due to the streaming nature of multimedia applications, the scheduling of processing elements in a multimedia MpSoC architecture more resembles scheduling of packet flows in a communication network than the traditional real-time task scheduling. Many existing real-time task scheduling

[1]Hereafter, we use the terms *scheduling* and *arbitration* interchangeably.

techniques are *deadline-driven*; in contrast, streaming multimedia applications have *quality of service requirements* which do not directly translate into task deadlines. Additionally, multimedia workloads are highly dynamic and variable, making it difficult to identify optimal scheduling strategies. As a result, platform management policies for multimedia MpSoC architectures often have very large and irregular design spaces.

All these factors call for new approaches to designing schedulers for multimedia MpSoC architectures. We explore this direction in this chapter. The VCC-based workload model and the extended Modular Performance Analysis framework developed in Chapter 3 serve in this chapter as a basis for an efficient framework for design space exploration of the platform management policies. In this framework we mainly concentrate on fast and accurate performance evaluation methods and on schedulability tests which can effectively speedup and guide the design space exploration process. To demonstrate the utility of the framework, we describe two case studies involving Time-Division Multiplex Access (TDMA) scheduling policy.

**Contributions of this chapter**

- We formulate the problem of scheduling media streams under strict QoS constraints imposed by available buffer space. This problem is motivated by the tight on-chip memory constraints associated with the current multimedia MpSoC architectures.

- We propose a framework for fast system-level design space exploration and optimization of platform management policies for the media processing execution platforms, such as multimedia MpSoC architectures. The framework features a combination of an initial one-time simulation of individual architectural components to obtain relevant workload characteristics, and a subsequent fast *analytic* performance evaluation, iteratively performed in the time-critical design space exploration loop.

- Based on the extended Modular Performance Analysis framework, we propose a method for computation of the buffer space requirements under different platform management policies. In comparison to the previously published methods addressing similar problems, the main novelty of our method is in its ability to account for specific QoS requirements associated with processing media streams on buffer-constrained architectures. In addition, in this method we demonstrate how the VCC types defined in Section 3.3 can be used for the *modular performance analysis* of distributed heterogeneous architectures. By applying our method to a case study of an MpSoC architecture, we demonstrate the complexity of the design space of the seemingly simple, widely used TDMA scheduling policy.

- We propose a method for a fast feasibility test of *stream scheduling* policies under given QoS requirements. Towards this, we introduce the concept of *service bounds* and show how they can be computed and used for the feasibility test. Through a detailed case study, we demonstrate how the service bounds can guide the optimization of stream scheduling policies.

**Organization of this chapter**

- Section 5.1 introduces the problem of stream scheduling on buffer-constrained architectures through a motivating example of a *set-top box* application scenario. This scenario will be used in the course of this chapter for the case studies.
- Section 5.2 outlines the related work.
- Section 5.3 presents the framework for design space exploration and optimization of platform management policies for the media processing execution platforms.
- Section 5.4 addresses the problem of estimating buffer memory requirements resulting from deploying different scheduling policies on processing elements of an MpSoC architecture.
- Section 5.5 introduces the concept of *service bounds* used for quick feasibility tests of stream schedulers for the buffer-constrained architectures.
- Section 5.6 concludes the chapter.

## 5.1 Stream Scheduling under Buffer Constraints

This section introduces the stream scheduling problem arising on the system level in multimedia MpSoC architectures. We first describe a *set-top box* application scenario; and then, using this scenario as an example, we introduce the scheduling problem and point out its specifics.

### 5.1.1 Set-top box application scenario

Fig. 29 shows a simple system-level model of a set-top box device implementing an audio-video decoder. Audio and video streams enter the device, which includes two PEs: $\mathrm{PE1}$ and $\mathrm{PE2}$. The MPEG-2 video decoding algorithm is partitioned into two tasks—one mapped onto $\mathrm{PE1}$, the other onto $\mathrm{PE2}$. The MP3 audio decoder includes a single task, mapped onto $\mathrm{PE2}$. On-chip buffers $B_1$, $B_2$, and $B_3$ store the partially processed streams, whereas $B_v$ and $B_a$, so called *play-out buffers*, store the fully decoded streams. These buffers are read by the video

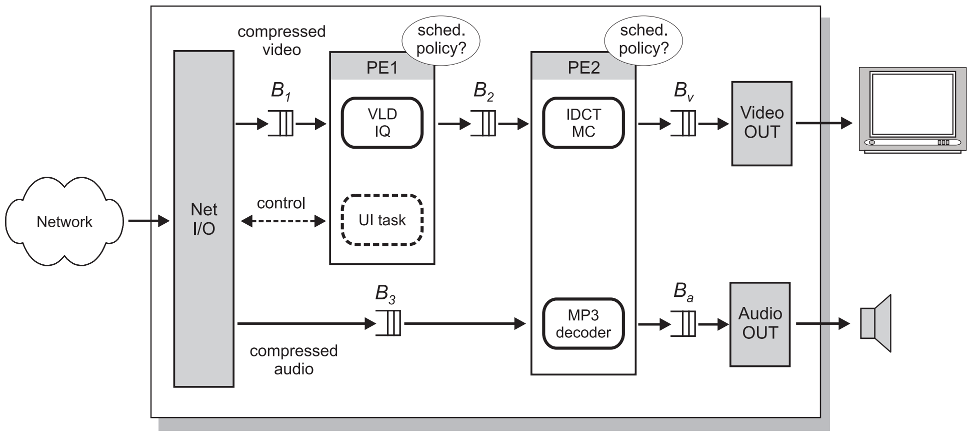


**Fig. 29:** System model of a *set-top box* device, processing an audio and a video stream.

and audio output devices through corresponding interfaces. Besides processing media streams, $\mathrm{PE1}$ and $\mathrm{PE2}$ can execute other tasks. For example, in Fig. 29, $\mathrm{PE1}$ executes a task responsible for interaction with the set-top box's user. This task handles control commands initiated through a user interface (UI).

In the setup described above, the MPEG-2 video decoding proceeds in the following way. A compressed video stream arrives through the network interface into $\mathrm{PE1}$'s input buffer, $B_1$. $\mathrm{PE1}$ reads the stream from this buffer and performs on it the *variable length decoding* (VLD) and *inverse quantization* (IQ) functions [118]. As a result of this processing, the video data appears at $\mathrm{PE1}$'s output as a stream of partially decoded *macroblocks*. $\mathrm{PE1}$ writes this stream into buffer $B_2$. $\mathrm{PE2}$ reads this buffer and completes the video decoding by computing the *inverse discrete cosine transform* (IDCT) and, if necessary, performing the *motion compensation* (MC) for each macroblock in the stream. $\mathrm{PE2}$ writes the fully decoded stream into playout buffer $B_v$, which is *periodically* read by the video output port. The output port finalizes the video processing and sends the video to a display device.

### 5.1.2 The stream scheduling problem

Different streams entering a PE (for example, $\mathrm{PE2}$ in Fig. 29) could be associated with different input rates. The respective output devices (such as *Video OUT* and *Audio OUT* in Fig. 29) could also consume the outgoing processed streams at different predetermined rates. Additionally, the processing requirements associated with the streams can vary widely. To satisfy the real-time constraints imposed by such I/O rates, it is necessary to suitably schedule the multiple streams entering a PE. Furthermore, to preserve the quality of the processed streams, any scheduling policy on a PE belonging to a setup similar to that in Fig. 29 must typically satisfy the following constraints: None of the buffers should overflow, and the playout buffers read by the real-time output devices should never underflow.

The output devices read the playout buffers at specified rates, depending on the required output quality. Therefore, the constraint on such a buffers underflow is to ensure that this required output quality is guaranteed. Guaranteeing that none of the buffers overflows is needed because, in many cases, using blocking writes to prevent the overflows is infeasible. Efficiently implementing such blocking mechanisms requires either a multithreaded processor architecture or substantial run-time operating-system support for context switching, and necessitates platform-wide flow-control mechanisms [134, 151] which might be difficult to implement in a distributed architecture.

The motivation for this *buffer-centric* design of schedulers is that buffers are available only at a very high premium because of their large on-chip area requirements [165]. On the other hand, the off-chip memory quickly becomes a bottleneck in high bandwidth applications such as multimedia; and its bandwidth cannot be easily increased due to a number of technology constraints (e.g. a limited number of I/O pins in a chip) [52]. Hence, the buffers play a central role in the design of any scheduling or MpSoC platform management policy.

**Specifics of the scheduling problem**

The scheduling problem just described has a number of characteristics that make it notably different from other real-time scheduling problems. These characteristics are summarized below.

- **Implicit deadlines:** The real-time scheduling traditionally relies on the notion of (hard and soft) *deadlines*: each task instance has an *explicit* deadline. This way, the deadlines specify the *real-time constraints* to be satisfied by the scheduler. In contrast to this, in the stream scheduling problem the real-time constraints are not naturally expressed through the deadlines. The main goal of a stream scheduler is to satisfy the *QoS requirements* associated with the processed streams. Hence, the real-time constraints are determined by these QoS requirements. The QoS requirements may, for example, include the required throughput, end-to-end processing delay[2] and the underflow/overflow buffer constraints. In general, these QoS requirements do not *directly* translate into the deadlines.

  In fact, the deadlines are not considered at all in the proposed method since we are not interested in individual task instances meeting their deadlines but in the whole stream satisfying given QoS constraints. Although the deadlines are not specified in the scheduling problem addressed in this chapter, the *real-time constraints* still exist and if needed can be *hard*. Finally, we note that due to the high variability of several workload characteristics and due to the buffering involved into the stream processing, it would be difficult to translate the QoS requirements into the deadlines for each task instance without introducing a significant amount of pessimism and runtime overhead.

[2]The delay through the whole stream processing chain.

- **“Too much service is as bad as too little service”:** The goal of the conventional real-time scheduling is to improve task response times. Therefore, the more service a task gets, the better it is (provided that schedulability of other tasks is not jeopardized). In contrast, in scheduling tasks processing streams under the buffer constraints, providing too much service to a task may be as dangerous as providing too little service. A higher service rate offered to a task increases the burstiness of the stream at the task’s output, and therefore may cause overflows of downstream buffers. Hence, in stream scheduling we are seeking for a *balanced service*.

- **Asynchronous communication style:** The majority of task models used in real-time scheduling assume that the tasks having data dependencies communicate synchronously[3]: a next instance of a producer task is not allowed to start execution before the output from the previous instance has been read by the consumer task. (The communication channel in this case represents a channel with destructive write, i.e. a *register*.) The underlying computation model of a stream processing application is different. In this model, tasks communicate *asynchronously* through the buffered channels. This communication style relaxes the timing coupling between dependent tasks, thereby creating a larger decision (or design) space for a scheduler.

- **Workload variability:** As discussed in Section 3.1, there are not many task models (and therefore scheduling techniques) that can effectively and efficiently handle the variable workloads. On the other hand, the major complication in scheduling of streaming multimedia applications on distributed execution platforms stems from the high workload variability, which leads to bursty and complex communication traffic between the multimedia tasks.

  The communication traffic on an MpSoC platform, such as the one shown in Fig. 29, tends to be highly complex and bursty for three main reasons [165]. First, the execution time of many media processing tasks highly depends on the properties of the particular audio-video sample being processed [14, 57, 134]. Second, the quantity of the I/O data consumed and produced by a task can also vary widely. An example of this is the VLD task in Fig. 29. Third, the input media streams already tend to be highly bursty when they enter a processing device. For example, in Fig. 29, the arrival pattern of the input streams entering the set-top box would depend on the networks congestion levels. In addition to these effects, the burstiness in the streams arrival pattern could increase as they pass from one PE to the next, depending on these PEs’ congestion levels and scheduling policies [133].

[3]See the discussion in Section 3.1.

## 5.2 Related work

In real-time scheduling, the QoS requirements of tasks are typically specified with deadlines (which are explicit and fixed for a given task). Thus, many existing scheduling methods proposed in this area are “deadline-driven” [41, 150].[4] However, there is a class of real-time tasks which rather require a guaranteed *rate of progress* (average throughput) than the satisfaction of the fixed deadlines [140]. Tasks that process continuous media streams fall into this category. There are scheduling techniques that can guarantee the required average throughput to such tasks in presence of other tasks by effectively *reserving* a portion of the processor bandwidth for execution of these tasks. Examples of such techniques include various kinds of aperiodic task *servers* [1, 22, 94, 147, 152] and the *rate-based execution* model [65]. Scheduling techniques based on the bandwidth reservation principle are also used to guarantee QoS to packet flows in communication networks (see e.g. [179]). The same principle can also be applied to schedule streams on an MpSoC execution platform. The scheduling techniques mentioned above may serve as a basis for designing platform management policies for the media processors. However, their direct application in this domain is limited because most of them are concerned with scheduling of a single PE (e.g. a processor or a communication link) without considering system-level issues such as the restricted buffer space.

There is a large body of research on scheduling of directed acyclic task graphs on multiprocessor architectures [40, 76, 80]. A task graph specifies precedence relations between tasks. These relations model data dependencies between the tasks. An application is typically modeled by a *set* of independent task graphs; each task graph is associated with a deadline and an activation period. This set is then scheduled on heterogeneous architectures [98, 132]. Several hardware/software co-synthesis frameworks use this model [32, 33, 35]. The method in [32] schedules task graphs with periodic and aperiodic activations. To guarantee QoS to aperiodic task graphs it employs resource reservations. [127] proposes an algorithm to schedule task graphs which model both data and control dependencies. Only a few approaches in this category consider memory constraints while scheduling the tasks [108, 131, 154].

All scheduling methods just mentioned assume *single-rate data dependencies* between tasks, i.e. the execution rate of any consumer task exactly matches the execution rate of the corresponding producer task. This task model is too restrictive for a large class of multimedia applications in which tasks have *multi-rate data dependencies* [67]. A more adequate task model for multimedia applications relies on the concept of dataflow process networks [89], which permits the multi-rate data dependencies as well as cycles in the task graph. Scheduling of dataflow process networks on multiprocessor architectures received a lot of attention in the digital signal processing (DSP) domain [50, 88, 149]. The basic

[4]Some of these methods have been mentioned in Section 3.1.

model used to represent DSP applications, called Synchronous Dataflow (SDF), assumes tasks with constant I/O rates [88]. A generalization of this model, the *cyclo-static dataflow* [19], allows for the I/O rates to cyclically change. Scheduling of the dataflow graphs is mainly concerned with minimization of data and program memory needed to execute a graph [18]. Although different tasks may execute at different rates, all these rates are tightly related to each other and fixed.

The existing dataflow scheduling algorithms focus on scheduling of only one graph, ignoring the fact that several independent graphs, each having its own QoS requirements, may need to be concurrently executed on the target architecture. Combining these algorithms with the results known from real-time scheduling (e.g. with the bandwidth reservation techniques discussed above) may help to address this problem. An interesting approach going in this direction is presented in [43] where the rate-based execution model [65] based on *earliest deadline first* policy is used to schedule an SDF graph on a single processor architecture. For this setup, [65] provides an analysis of buffer requirements and the latency.

A number of *system-level performance analysis* frameworks have been proposed in the literature for evaluation and optimization of platform management policies. [71] proposes a framework for evaluation of run-time schedulers in embedded multimedia systems. Given a system architecture, a set of periodic task graphs with execution times characterized by probability distributions, and a scheduling policy implemented on PEs of the architecture, for each task graph the framework in [71] computes a probability distribution of the processing delay. The framework in [129] performs a system-level schedulability analysis of distributed real-time systems which rely on the time-triggered protocol (TTP) [76] as the communication infrastructure. Along with the schedulability tests, [129] reports techniques to optimize the parameters of the TTP-based communication infrastructure and select suitable message passing strategies for it. The work in [51] uses evolutionary multi-objective optimization techniques and the SymTA/S performance evaluation framework [68, 133] to find suitable period and time slot lengths of the TDMA scheduling policy. [25, 156] address the problem of performance evaluation and design space exploration of network processors. Similar to the framework presented in this chapter, the approach in [25, 156] relies on the Real-Time Calculus [121, 158]. It estimates various performance metrics, such as required buffer sizes and packet delays, resulting from implementing different scheduling policies on PEs of a network processor. However, unlike the framework presented in this chapter, the work in [25, 156] does not account for the buffer constraints (especially those related to the *playout buffers*) and the variability of the task I/O rates, which are natural for media stream processing on distributed execution platforms, such as multimedia MpSoC architectures.

Finally, we note that designing schedulers for on-chip PEs involves significantly different constraints from those for scheduling and buffer management of multimedia applications in operating systems and communication networks

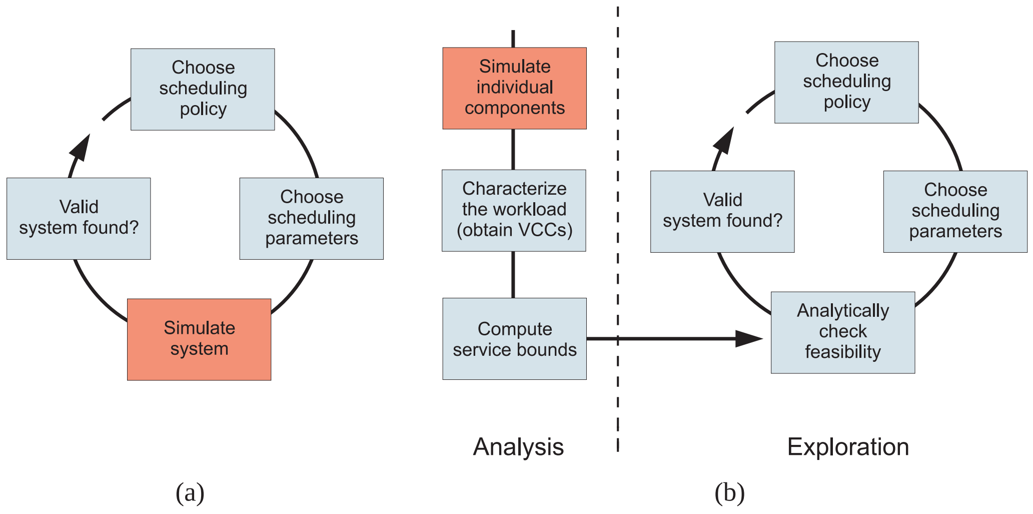

Choose scheduling policy
Valid system found?
Choose scheduling parameters
Simulate system
(a)
Simulate individual components
Characterize the workload (obtain VCCs)
Compute service bounds
Analysis
Choose scheduling policy
Valid system found?
Choose scheduling parameters
Analytically check feasibility
Exploration
(b)

**Fig. 30:** Evaluating multiple scheduling policies (or their associated parameters): (a) Traditional design cycle, purely based on simulation, (b) Proposed design cycle, which resorts to simulation only once and subsequent iterations are based on the analytical framework.

(see [91] and the references therein). In the latter domain, the scheduling overhead is often negligible in comparison to the execution times of the tasks. This allows for complicated, online scheduling algorithms. However, in our resource-constrained setup, implementing such algorithms might be infeasible: Often, on-chip PEs have only lightweight or even no operating system support. Furthermore, in the communication networks domain, buffer-use restrictions are not as acute, and its possible to recover from data loss due to buffer overflows. However, such mechanisms are too complicated for an on-chip setup.

## 5.3 Design Framework

In this section, we present an overview of our framework for design space exploration and optimization of platform management policies [111, 112]. Most of the state of the art in this area rely on simulation-oriented techniques to evaluate a platform management policy (see e.g. [126]), and follow the design cycle in Fig. 30(a). Our technique follows the design cycle in Fig. 30(b); we resort to simulation *only once*, to derive certain system bounds. Subsequently, in our framework the evaluation of scheduling policies and their parameters (such as suitable weights for a TDMA scheduler) relies solely on analytical methods. Hence, when the parameter space associated with designing a scheduler is relatively large, our framework can be a few orders of magnitude faster than purely simulation-based approaches.

As shown in Fig. 30(b), in our framework we distinguish two phases: analysis and exploration. During the analysis phase a simulation of individual architectural components is performed. For example, execution of application tasks is simulated on an instruction set simulator, and traces of task execution times and other relevant task characteristics are collected. Using analysis techniques described in Section 3.5, these traces are then abstracted by a set of VCCs. We refer to this step as *workload characterization*. After the workload characterization, the obtained VCCs are used either for the analytical estimation of the memory requirements associated with a platform management policy (Section 5.4) or for the computation of the service bounds (Section 5.5) that are then used for fast feasibility tests of stream schedulers. In both cases, the evaluation of schedulers is performed analytically in the exploration phase. In this way, the time-consuming system-level simulation is pushed out of the design space exploration loop.

This basic scheme of using an initial simulation to generate traces is not new and is also followed in [125, 126]. However, simulation-oriented methods such as [99, 125, 126, 184] then rely on a symbolic simulation of these traces (see also [74, 75, 83] for work on performance analysis of bus-based SoC communication architectures), whereas we rely on purely analytical methods which are specific to multimedia processing.

## 5.4 Applying Modular Performance Analysis

This section shows how the MPA framework introduced in Section 2.2 and then extended with multimedia-specific workload transformations in Section 3.4 can be used for designing scheduling policies for a multimedia MpSoC execution platform. In particular, we describe how the buffer memory requirements resulting from deploying different scheduling policies on processing elements of an MpSoC architecture can be analytically estimated using this extended MPA framework. For illustration of the method, we use the set-top box application scenario described in Section 5.1.1 and depicted in Fig. 29. Towards the end of this section, we use this application scenario as a case study in which the performance of the Time Division Multiple Access (TDMA) scheduling policy is evaluated. The results of this case study indicate that in a multiprocessor environment even a simple scheduling policy such as TDMA may have a large and irregular design space. The extended MPA framework can help system designers to quickly discover these irregularities and make informed design decisions.

### 5.4.1 Problem formulation

As already mentioned above, the burstiness of processed event streams largely determines the buffer requirements in a typical MpSoC architecture. It stems from several factors. Some of the factors are application specific, while the others are platform specific. The application specific factors are related to data-dependent variability of stream parameters, such as variability of the execution

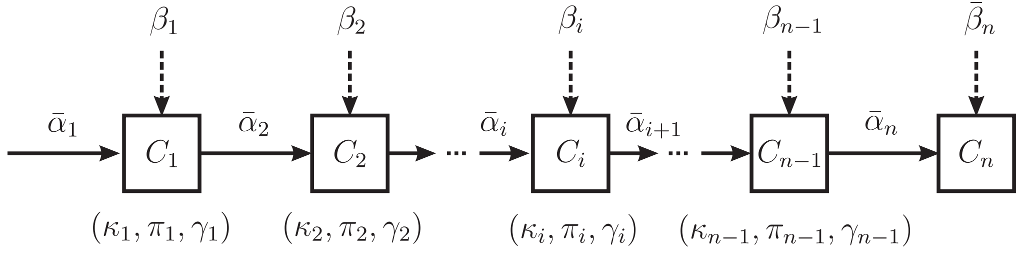


**Fig. 31:** A chain of performance components induced in a scheduling network by a media stream.

demand. The burstiness caused by the platform specific factors is associated with contention of event streams on shared communication and computational resources (PEs) of the architecture. The scheduling and arbitration policies used to manage the access to these resources influence the degree of the burstiness induced in the event streams and, therefore, largely determine the amount of buffer space required to process these streams at a certain QoS level.

Scheduling policies can be designed such that the buffer space requirements are minimized. However, in many cases, besides the minimization of the buffer space, there are also other design criteria. These criteria are often in conflict with the buffer space minimization goal. For example, minimizing the buffer requirements necessitates scheduling streams at a finer granularity. However, this may increase scheduling overhead and, therefore, result in wasting precious system resources (e.g. energy). While designing scheduling strategies for media processors, system designers are often concerned with identifying such *tradeoffs*. In many cases, these tradeoffs are difficult to identify since they represent complex relationships between the application at hand and different system parameters, including those pertaining to the resource sharing policies. Identifying these tradeoffs, therefore, necessitates evaluation of many alternative points in the design space. One of the major problems in this context is how to quickly evaluate a large number of the design points. The method proposed in this section employs the MPA framework to address this problem.

In order to use the MPA framework, we need to formulate our problem as a *scheduling network*. Such a scheduling network represents a performance model of a given application-to-architecture mapping. It specifies how event streams flow between different PEs of the architecture and how these PEs are shared between those streams. The scheduling network, therefore, consists of performance components interconnected through resource and event flows (see Section 2.2 for details). Concrete instances of event and resource flows are abstracted by arrival and service curves, $\bar{\alpha} = (\bar{\alpha}^l, \bar{\alpha}^u)$ and $\beta = (\beta^l, \beta^u)$. Additionally, in the MPA framework with multimedia extensions, performance components are characterized by consumption, production and execution demand curves, i.e. each component is associated with VCCs $\kappa = (\kappa^l, \kappa^u)$, $\pi = (\pi^l, \pi^u)$, and $\gamma = (\gamma^l, \gamma^u)$.

Consider a chain of performance components $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_i \rightarrow$

$\ldots \rightarrow C_{n-1} \rightarrow C_n$ induced in a scheduling network by a media stream, as shown in Fig. 31. This chain models a sequence of tasks mapped onto different PEs of the architecture, which process the media stream in a pipelined fashion. In Fig. 31, performance component $C_i$ ($i = 1, 2, \ldots$) has as its input event-based arrival curve $\bar{\alpha}_i$. This curve is transformed by $C_i$ into event-based arrival curve $\bar{\alpha}_{i+1}$. $\bar{\alpha}_{i+1}$, appearing at $C_i$'s output, serves as an input to the next performance component in the chain, $C_{i+1}$. The way in which $C_i$ transforms $\bar{\alpha}_i$ into $\bar{\alpha}_{i+1}$ depends on the workload variability at $C_i$, specified by VCCs $(\kappa_i, \pi_i, \gamma_i)$, and on the scheduling policy implemented on the PE with which $C_i$ is associated. The scheduling policy determines the amount of resources (e.g. processor cycles) received by $C_i$ from the PE in any given time interval. This amount is modeled by resource-based service curve $\beta_i$. In other words, $\beta_i$ characterizes the service offered to the media stream on the PE.

At the input of each performance component, there is an implicit buffer (not shown in Fig. 31). As explained above, to maintain the quality of the processed media stream at an acceptable level, we require that none of the buffers in the processing chain ever overflows. Furthermore, there may be some performance components in the processing chain whose input has to be continuous. This means that they cannot tolerate waiting on the empty input buffer for new events to arrive. Whenever such a component reads from the buffer, the data must be available for it. We refer to such buffers as *playout buffers*. Typically, a playout buffer is associated with the last performance component in a processing chain. This is because the last component often represents an output interface in a multimedia MpSoC architecture, e.g. a video or audio output, at which event streams have to satisfy strict real-time constraints imposed by external devices (such as digital-to-analog converters). Again, to ensure an acceptable quality of the processed stream, we require that any playout buffer neither overflows nor underflows.

Assume that, in Fig. 31, $C_n$ represents the performance component that needs to have at its input a playout buffer. To model the timing with which $C_n$ reads its playout buffer, we use *event-based* service curves $\bar{\beta}_n^l$ and $\bar{\beta}_n^u$. $\bar{\beta}_n^l(\Delta)$ and $\bar{\beta}_n^u(\Delta)$ specify, respectively, the minimum and the maximum number of events (stream objects) that $C_n$ reads from the playout buffer within any time interval of length $\Delta$.

Given a processing chain such as one shown in Fig. 31, our goal is to compute the maximal backlog which may occur in each of the buffers in this processing chain as a result of applying a given scheduling policy. After finding the maximal backlog in each of the buffers, we will be able to compute the maximum memory requirements associated with this scheduling policy.

## 5.4.2 Computing the required buffer space

**Upper bound on the backlog in a ”regular” buffer**

Consider a performance component $C_i$ which has at its input a buffer that must never overflow, but is allowed to underflow. (We refer to such a buffer as a ”regular” buffer as opposed to a playout buffer that is allowed neither to overflow nor to underflow.) To compute the upper bound on the backlog in the buffer at the input of $C_i$, we use results of the Real-Time Calculus [157] presented in Section 2.2.2. Namely, we rewrite (2.10) such that it includes new multimedia workload transformations developed in Section 3.4. We obtain

$$b_i = \sup_{\Delta \in \mathbb{R}_{\geq 0}} \{\bar{\alpha}_i^u(\Delta) - (\kappa_i^{u^{-1}} \odot \gamma_i^{u^{-1}} \odot \beta_i^l)(\Delta)\} \tag{5.1}$$

where $b_i$ is the upper bound on the backlog in the input buffer of $C_i$. In (5.1), $\beta_i^l$, $\kappa_i^u$ and $\gamma_i^u$ are known from the problem specification. In contrast, $\bar{\alpha}_i^u$ is, in general, unknown. Normally, we know only $\bar{\alpha}_1^u$, which characterizes the event stream at the input of the whole processing chain (i.e. at the input of the first performance component $C_1$). Starting from $\bar{\alpha}_1^u$, the value of $\bar{\alpha}_i^u$ can be iteratively computed using (3.38) as follows

$$\bar{\alpha}_{i+1}^u = \pi_i^u \odot ([((\kappa_i^u \odot \bar{\alpha}_i^u) \underline{\otimes} (\gamma_i^{l^{-1}} \odot \beta_i^u)) \overline{\oslash} (\gamma_i^{u^{-1}} \odot \beta_i^l)] \wedge (\gamma_i^{l^{-1}} \odot \beta_i^u)) \quad i = 1, 2, \ldots$$

Using the above formula in conjunction with (5.1), the upper bounds on the backlogs in all ”regular” buffers of the processing chain can be calculated.

**Upper bound on the backlog in a playout buffer**

For performance components that have playout buffers at their inputs, the computation of the upper bound on the backlog is in principle the same as for components with ”regular” buffers, however, it involves an additional step.

As stated above, a playout buffer must neither overflow nor underflow. Ensuring satisfaction of the underflow condition necessitates introducing a *playout delay*. This is a time period at the start of the operation of the whole processing chain during which the performance component associated with a playout buffer does not read this buffer. This playout delay is necessary to produce an *initial backlog* in the playout buffer. The initial backlog must be sufficient to ensure that even in the case when the processed stream experiences the worst-case delay (e.g. due to processing by upstream components) the playout buffer never gets completely empty. Hence, to compute the upper bound on the backlog in a playout buffer, we first need to compute the initial backlog in this buffer. Towards this we propose the following theorem.

**Thm. 2: (Initial backlog in playout buffer)** *The initial backlog, $b^0$, ensuring that the associated playout buffer never underflows is given by*

$$b^0 = \sup_{\Delta \in \mathbb{R}_{\geq 0}} \{\bar{\beta}^u(\Delta) - \bar{\alpha}^l(\Delta)\} \tag{5.2}$$

*where $\bar{\alpha}^l$ denotes the lower event-based arrival curve of the stream at the input of the playout buffer, and $\bar{\beta}^u$ denotes the upper event-based service curve offered to the stream by the performance component reading the playout buffer.*

**Proof.** Let $x(t)$ denote the total number of events that have arrived in the playout buffer within time interval $[0, t]$. Similarly, let $y(t)$ denote the total number of events that have been read out of the playout buffer within time interval $[0, t]$. Assume that $x(0) = y(0) = 0$. From the definitions of the arrival and service curves, we have $x(t + s) - x(t) \geq \bar{\alpha}^l(s)$ and $y(t + s) - y(t) \leq \bar{\beta}^u(s)$ for all $s, t \in \mathbb{R}_{\geq 0}$ [85]. The playout buffer never underflows if $x(t) > y(t)$ for all $t \in \mathbb{R}_{>0}$.

Now, consider some $t$ up to which the condition $x(t) > y(t)$ holds. To ensure that this condition also holds for some $t' = t + s$ the following relation has to be satisfied

$$x(t) + \bar{\alpha}^l(s) \geq y(t) + \bar{\beta}^u(s) \quad \forall t, s \in \mathbb{R}_{>0}$$

Here, $\bar{\alpha}^l$ (s) is the minimum number of events that may arrive in the interval $[t, t + s]$, whereas $\bar{\beta}^u$ (s) is the maximum number of events that can be read out of the playout buffer in the same interval. Let $b(t) = x(t) - y(t)$ denote the backlog in the buffer at time $t$. Then we have

$$b(t) \geq \bar{\beta}^u(s) - \bar{\alpha}^l(s) \quad \forall t, s \in \mathbb{R}_{>0}$$

By putting $t = 0$ in the above condition, we obtain a constraint for the initial backlog $b(0) \geq \bar{\beta}^u(s) - \bar{\alpha}^l(s)$ for all $s \in \mathbb{R}_{>0}$. This is equivalent to requiring that $b(0) \geq b^0$, where $b^0 = \sup_{s \in \mathbb{R}_{\geq 0}} \{\bar{\beta}^u(s) - \bar{\alpha}^l(s)\}$
□

Whenever even-based service curve $\bar{\beta}_i^u$ is specified, we can directly use (5.2) to compute initial backlog $b_i^0$ in the playout buffer of $C_i$. Otherwise, we need to apply corresponding workload transformations to resource-based service curve $\beta_i^u$. After applying these transformations, (5.2) takes the following form.

$$b_i^0 = \sup_{\Delta \in \mathbb{R}_{\geq 0}} \{(\kappa_i^{l^{-1}} \odot \gamma_i^{l^{-1}} \odot \beta_i^u)(\Delta) - \bar{\alpha}_i^l(\Delta)\} \tag{5.3}$$

In (5.3), $\bar{\alpha}_i^l$ can be computed in a similar way as $\bar{\alpha}_i^u$, as explained above, but using (3.39):

$$\bar{\alpha}_{i+1}^l = \pi_i^l \odot ([(\kappa_i^l \odot \bar{\alpha}_i^l \overline{\oslash} \gamma_i^{l^{-1}} \odot \beta_i^u) \underline{\otimes} \gamma_i^{u^{-1}} \odot \beta_i^l] \wedge \gamma_i^{u^{-1}} \odot \beta_i^l)$$

The computation of the above formula starts from $\bar{\alpha}_1^l$ (which is known from the problem specification) and iteratively proceeds up to the required index $i$.

Taking into account the required initial backlog $b_i^0$, the upper bound on the backlog in the playout buffer at the input of the performance component $C_i$ can be calculated as

$$b_i^* = b_i^0 + b_i \tag{5.4}$$

where $b_i$ is determined by (5.1).

**Maximum memory requirements**
Having computed the upper bounds on the backlogs in the buffers within a stream processing chain, we can calculate the maximum memory requirement associated with this chain. For this, let $S_i$ denote the maximum size of a stream object in the input buffer of performance component $C_i$. Then the maximum memory requirement, $\mathcal{M}$, of the whole processing chain can be computed as follows.

$$\mathcal{M} = \sum_{i=1}^{n} b_i \times S_i + \sum_{\forall i \in \mathcal{P}} b_i^0 \times S_i \tag{5.5}$$

where $\mathcal{P} = \{i : C_i$ has at its input a playout buffer$\}$.

### 5.4.3 Illustrative case study

This subsection presents results of a case study published in [115], where the concepts of the modular performance analysis and the new VCC types developed in the previous sections of the thesis are applied to evaluate performance of the TDMA scheduling policy in a multiprocessor environment. This multiprocessor environment is represented by the system architecture shown in Fig. 29. In this architecture, the TDMA scheduling policy is used for scheduling processing elements $\mathrm{PE1}$ and $\mathrm{PE2}$, with each PE having its own TDMA scheduler.

There are three main reasons for choosing TDMA for this case study. First, it is simple enough to implement in a SoC setup, and it has low scheduling overhead, therefore it is widely used for scheduling on-chip PEs and communication resources [44, 76, 129]. Second, since TDMA is fully predictable in terms of the worst-case delay and bandwidth provided to individual event flows, it is especially suitable for scheduling media streaming applications associated with real-time guarantees. Third, TDMA is also relatively easy to characterize in terms of service curves; hence, it provides a simple illustration of the theoretical concepts presented above. However, it may be noted here that our evaluation framework is not restricted to analyzing only TDMA schedulers. System designers can use it to analyze any static- or dynamic-priority scheduling algorithm, including preemptive and nonpreemptive versions. These include scheduling policies such as fixed-priority, weighted round-robin, and earliest-deadline first. In fact, our framework can evaluate any scheduling policy that is characterizable using service curves. Moreover, it can evaluate a platform in which different scheduling policies are used on the different PEs [24].

For this case study, we consider the set-top box application scenario described in Section 5.1.1 and also shown in Fig. 29. In this scenario, each PE processes two independent concurrent event flows. $\mathrm{PE1}$ performs partial decoding of the MPEG-2 video stream (by applying to it VLD and IQ functions) and handles control events (e.g. user's commands). $\mathrm{PE2}$ finalizes the decoding of the MPEG-2 stream (with IDCT and MC functions) and decodes an MP3 audio stream. The TDMA schedulers, therefore, regulate sharing of $\mathrm{PE1}$ and $\mathrm{PE2}$ by

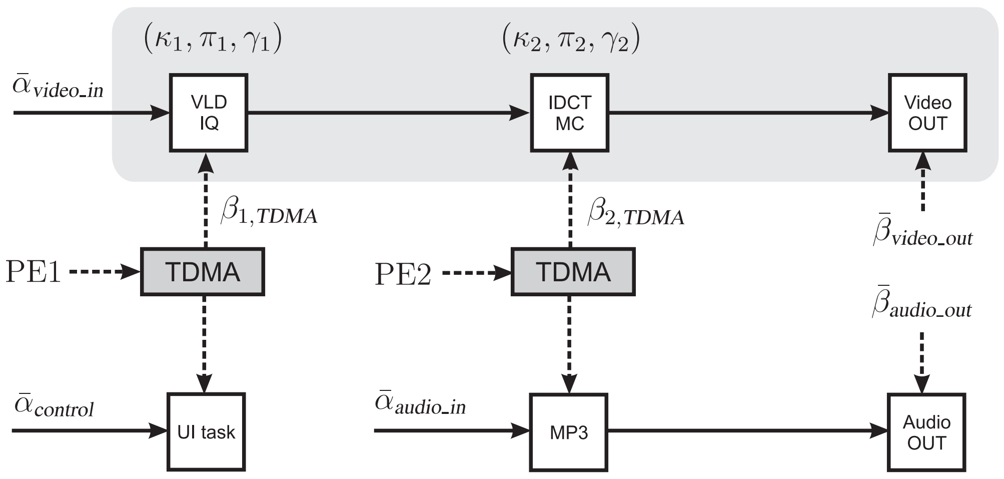

$(\kappa_1, \pi_1, \gamma_1)$
$(\kappa_2, \pi_2, \gamma_2)$
$\bar{\alpha}_{video\_in}$
VLD IQ
IDCT MC
Video OUT
$\beta_{1,TDMA}$
$\beta_{2,TDMA}$
$\bar{\beta}_{video\_out}$
PE1
TDMA
PE2
TDMA
$\bar{\beta}_{audio\_out}$
$\bar{\alpha}_{control}$
UI task
$\bar{\alpha}_{audio\_in}$
MP3
Audio OUT

**Fig. 32:** Scheduling network of the set-top box system shown in Fig. 29 using TDMA scheduling policy on $\mathrm{PE1}$ and $\mathrm{PE2}$.

these event flows. The scheduling network modeling this set-top box application scenario is shown in Fig. 32. As an example, the processing chain induced by the video stream in the scheduling network in Fig. 32 is indicated with a grey background area.

Let the TDMA schedulers on $\mathrm{PE1}$ and $\mathrm{PE2}$ have periods equal to $p_1$ and $p_2$, respectively. The smaller the lengths of the periods are, smaller the buffer requirements for processing the streams will be, but at the cost of higher scheduling overheads. The goal is to analytically compute a *tradeoff curve* showing how the on-chip buffer requirements change with different periods of the TDMA schedulers. For the demonstration of the method, in this case study, we restrict ourselves to computing such a tradeoff curve for the video stream only. The dependency of buffer requirements on the TDMA periods for the audio stream can be calculated in a similar way.

We now characterize a TDMA scheduler in terms of the service it provides to any particular stream when that scheduler is scheduling multiple streams. Consider two streams, $x$ and $y$, scheduled on a PE by a TDMA scheduler with period $p$. Assume that no other streams are processed by the PE. The weights associated with streams $x$ and $y$ are $w_x$ and $w_y$, where $w_x + w_y \leq 1$. The scheduler divides time into periods of length $p$. Within any period, the scheduler allocates $w_x p$ consecutive units of the PEs time to stream $x$, and $w_y p$ consecutive units to stream $y$. If a stream cannot exhaust the processor share allocated to it, the unused processor cycles are wasted.

Assuming $p$ is infinitesimally small, we can neglect the effects of a finite sampling of the processor cycles. Then, we can calculate the service offered to the two streams in terms of processor cycles as follows. If $f$ is the number of processor cycles available from the PE per unit time (that is, $f$ is the PEs clock

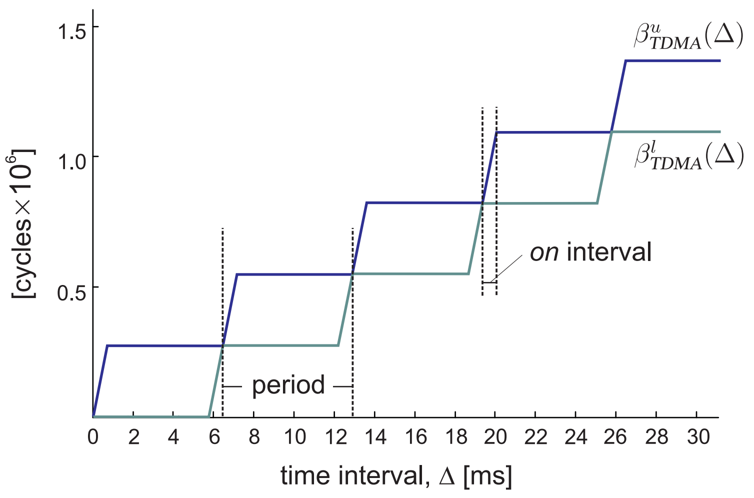
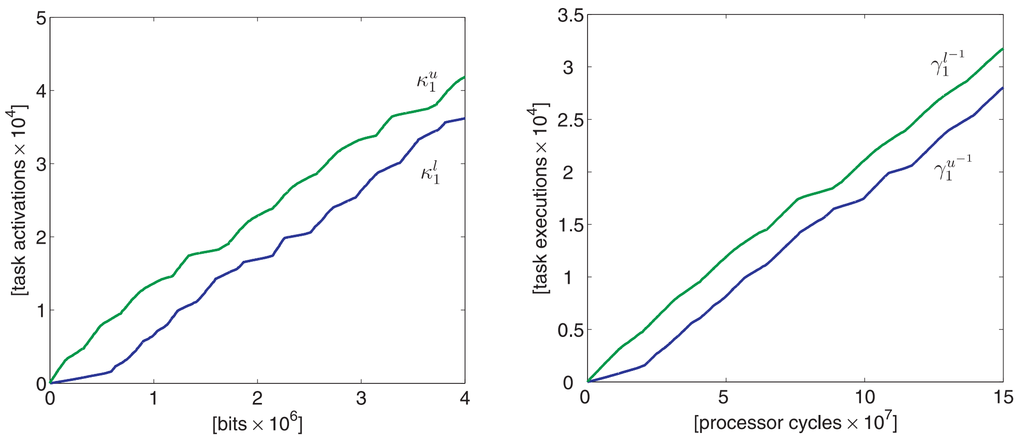


**Fig. 33:** Example resource-based service curves. Here, $\beta^l_{TDMA}$ and $\beta^u_{TDMA}$ are the lower and the upper resource-based service curves characterizing service offered to one of the streams being scheduled on a PE using the TDMA scheduler. The scheduler's period, $p$, is set to the equivalent of $2.5 \times 10^6$ processor cycles, and the clock rate of the PE is 390 MHz. The *on* interval indicates the time over which the PE processes the stream within any period. The value of $\beta^l_{TDMA}$ is initially 0, corresponding to the maximum time the PE is unavailable to the stream.

rate), resource-based service curve $\beta^l(\Delta) = \beta^u(\Delta) = f\Delta$ constitutes the total service offered by the processor. Then the service curve for stream $x$ is $\beta^l_x(\Delta) = \beta^u_x(\Delta) = w_x f\Delta$; the service curve for stream $y$ is $\beta^l_y(\Delta) = \beta^u_y(\Delta) = w_y f\Delta$. Therefore, the lower and upper service curves for both streams coincide and are straight lines with slopes $w_x f$ and $w_y f$, respectively.

When $p$ has a finite value, the resource-based service curves take the form of a staircase function, and the lower and upper curves no longer coincide. Fig. 33 gives an example of such a service curve. Here, $p$ is set to a time interval over which the PE offers a total of $2.5 \times 10^6$ cycles. Note that period $p$ does not determine the amount of service (i.e. number of processor cycles) provided to the scheduled streams in a long term. It does influence only short term variations of the service. The long term service provided to a stream is fully determined by PE's clock rate $f$ and the TDMA weight $w$ assigned to this stream, i.e. by value $wf$. While designing a TDMA scheduler, it is important to choose this value such that, *in a long term*, the stream would receive not less service than it requires, otherwise, a buffer overflow is bound to happen at some point in time.

To fully specify the problem, besides characterizing the TDMA schedulers, we also need to characterize the video tasks executing on both PEs, the timing properties of the video stream at the $\mathrm{PE1}$'s input and the video interface which reads the fully decoded video stream from the playout buffer $B_v$ at the $\mathrm{PE2}$'s output.



(a) Consumption curves (b) Pseudo-inverse execution demand curves

**Fig. 34:** Characterization of the video decoding task executed on PE1.

For a constant bit rate input video stream, which we use here as an example, event-based arrival curves $\bar{\alpha}^l_{\textit{video\_in}}(\Delta) = \bar{\alpha}^u_{\textit{video\_in}}(\Delta) = r_c\Delta$ (the lower and upper arrival curves coincide), where $r_c$ denotes the bit rate of the compressed video stream at the PE1's input. In this case study, we consider video sequences with $r_c = 4 \cdot 10^6$ bits/sec.

The video output interface periodically reads decoded macroblocks from playout buffer $B_v$ at a constant rate $r_{mb}$. Hence, the event-based service curves characterizing this reading process $\bar{\beta}^l_{\textit{video\_out}}(\Delta) = \bar{\beta}^u_{\textit{video\_out}}(\Delta) = r_{mb}\Delta$. Rate $r_{mb}$ is determined by the frame rate and the resolution of the decoded video clip. In our setup, $r_{mb} = 39600$ macroblocks/sec.

The video decoding tasks executing on PE1 and PE2 are characterized by VCCs $(\kappa_1, \pi_1, \gamma_1)$ and $(\kappa_2, \pi_2, \gamma_2)$, respectively. The VCCs $(\pi_1^l, \pi_1^u)$, $(\pi_2^l, \pi_2^u)$ and $(\kappa_1^l, \kappa_1^u)$ are straight lines with slopes which correspond to the constant-rate production (consumption) of one stream object per task activation. In contrast to this, $\kappa_1^l$ and $\kappa_1^u$ have complex shapes since, per one activation of the task performing VLD and IQ functions, PE1 consumes a variable number of bits from its input buffer $B_1$. Similarly, $(\gamma_1^l, \gamma_1^u)$ and $(\gamma_2^l, \gamma_2^u)$ have complex shapes, because both the MPEG-2 decoding tasks running on PE1 and PE2 have variable execution demands. As an example, Fig. 34 shows $(\kappa_1^l, \kappa_1^u)$ and pseudo-inverse of $(\gamma_1^l, \gamma_1^u)$, corresponding to an MPEG-2 video sequence which we used in our experiments.

The VCCs $(\kappa_1^l, \kappa_1^u)$, $(\gamma_1^l, \gamma_1^u)$ and $(\gamma_2^l, \gamma_2^u)$ were obtained by analyzing traces generated from the initial simulation step described in Section 5.3. In this case study, this step comprised of simulating the execution of PE1 and PE2 for a representative MPEG-2 video clip using the SimpleScalar instruction set simulator [8]. To derive the VCCs, the traces collected from the SimpleScalar simulation were analyzed using the technique described in Section 3.5.2. Here, we once

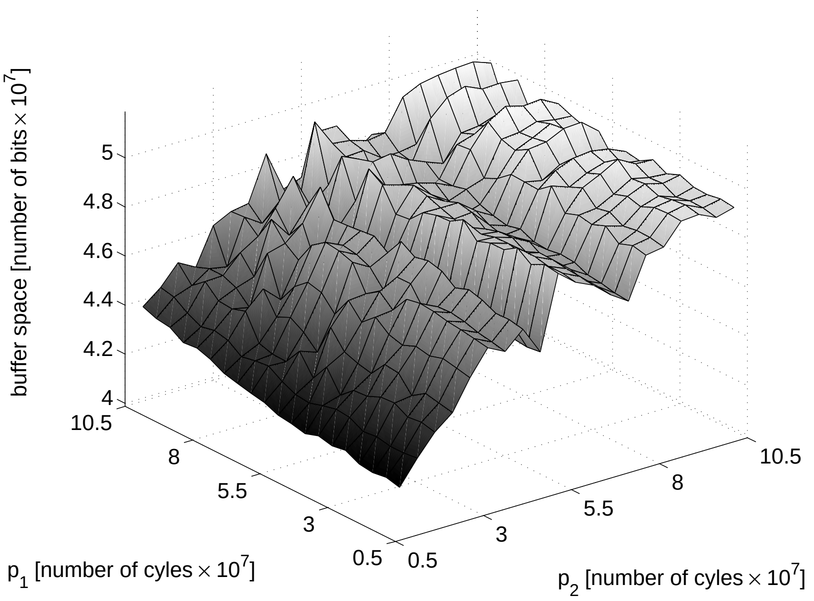


**Fig. 35:** The surface showing the dependency of memory requirements on the values of the periods $p_1$ and $p_2$ of the TDMA schedulers.

again point out that although obtaining the VCCs requires the simulation of the MPEG-2 decoder tasks, we need to do it only once, using a representative video clip (or a set of clips). Once this specification is obtained, multiple instances of the platform architecture (with different configurations) can be analyzed using only analytical means. Furthermore, we can avoid a time-consuming simulation of the whole multiprocessor system—rather, we simulate an *abstract model* of the platform for which we need to employ only an instruction set simulator.

Having characterized the TDMA schedulers and the workload properties of the video decoding chain, we can compute the maximum buffer space required for processing the video stream in the architecture shown in Fig. 29 as described in [115].

We are interested in studying how the amount of buffer space required for processing any MPEG-2 video stream depends on the granularity of the TDMA schedulers implemented on $\mathrm{PE1}$ and $\mathrm{PE2}$. Hence, a design point is determined by a pair of TDMA periods $(p_1, p_2)$. For each pair $(p_1, p_2)$ we iteratively compute the maximum backlogs in the FIFO buffers and scale the obtained values by the maximum size (in bits) of the stream objects associated with the buffers. The results of this computation for the representative MPEG-2 video sequence are shown in Fig. 35.

By inspecting the 3D surface shown in Fig. 35 we can see the expected trend: decreasing the values of the TDMA periods $p_1$ and $p_2$, in general, leads to a reduction in the memory needed to implement the buffers. However, we also can see that this reduction is not uniform across the entire range of the period values.

Even for a simple scheduling discipline like TDMA, there are large irregularities in the design space. This makes it virtually impossible to come up with an appropriate tradeoff, based only on a designer's experience on how the memory requirements typically change with small changes in the parameters of the schedulers. Since on-chip buffers have large area requirements, such an information about the design space is, however, essential for determining optimal platform management policies. Using our framework it is therefore possible to discover the irregularities in the design space, and from it arrive at an appropriate tradeoff—in this case between scheduling overheads and buffer requirements. This capability of the framework can be attributed to the underlying concept of VCCs which can be used to precisely represent the different types of variabilities associated with multimedia processing on multiprocessor SoC platforms.

Finally, we note that evaluating a single design point $(p_1, p_2)$ by simulating an abstract transaction-level model of the platform architecture in SystemC [153] (using the system simulator described in Appendix A) for a $2$ sec long video clip required almost an hour of simulation time. This simulation time was around 100 times longer than the time needed for evaluating a design point with our analytical framework implemented using a combination of Mathematica[5] and Matlab[6] models. (We believe that an efficient C/C++ implementation would be at least 5-10 times faster than our current prototype implementation of the analytical framework.) Considering the time involved in simulating even a single design point for a relatively short video clip, it is almost infeasible to obtain a *design surface* such as the one shown in Fig. 35 in a reasonable time using purely simulation-based techniques. In contrast, using the proposed analytical framework system designers can learn about the structure of the entire design space in tens of minutes.

## 5.5 Checking Feasibility of Stream Schedulers

In this section, we introduce the concept of *service bounds* [102, 103, 111]. The service bounds allow to quickly verify whether in principle there exists a scheduling policy that can satisfy QoS requirements of a stream (such as the delay and buffer constraints), and to check feasibility of a given scheduler against these requirements. Furthermore, they can direct the design space exploration process by providing the information by how much a given scheduler does not (or does) satisfy a given set of QoS requirements.

In this section, we limit the presentation to the service bounds obtained from the buffer constraints; however, the principles presented in this section can be applied to derive the service bounds also for the delay constraints.

[5]`http://www.wolfram.com`
[6]`http://www.mathworks.com`

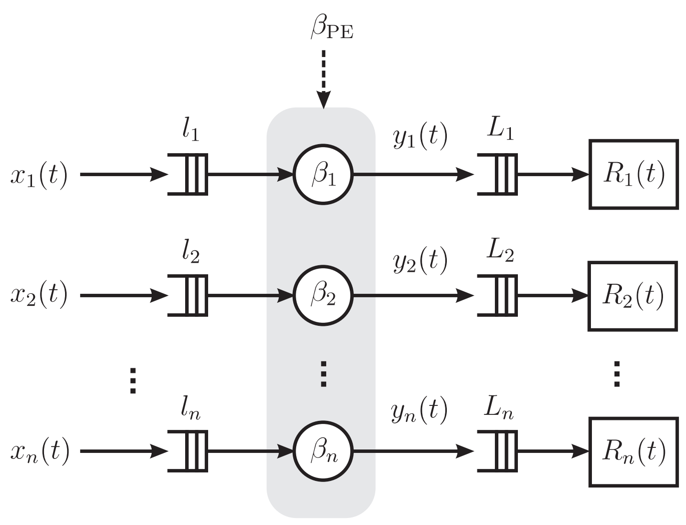


**Fig. 36:** An abstract view of a PE which processes $n$ streams.

### 5.5.1 Problem formulation

For the sake of generality, we consider any media stream to include a potentially infinite sequence of stream objects. A stream object could be a macroblock, a video frame, an audio sample, or a network packet, depending on the part of the architecture where the stream exists. For example, in Fig. 29, stream objects are network packets when the relevant stream is that entering the network interface. In contrast, the stream objects are partially processed macroblocks when the relevant stream is that written into buffer $B_2$.

Fig. 36 shows an abstract view of a PE that processes $n$ streams. Functions $x_i(t)$, $i = 1, 2, \ldots, n$, specify the streams entering this PE. These functions denote the total number of stream objects that arrive at the input buffers in Fig. 36 over time interval $[0, t]$. $l_i$ denotes the number of stream objects that the input buffer, for stream $i$, can store (i.e. $l_i$ is the size of the input buffer). In Fig. 36, the PE writes the processed streams into playout buffers which are then read by real-time output devices. $L_i$ denotes the playout buffer size for stream $i$. Function $y_i(t)$ specifies the processed output stream entering the playout buffer. Like $x_i(t)$, this function denotes the number of stream objects exiting the PE over time interval $[0, t]$. The real-time output device associated with stream $i$ consumes stream objects from the playout buffer at a rate specified by function $R_i(t)$, which denotes the number of stream objects consumed during time interval $[0, t]$. We note that, for all $i = 1, 2, \ldots, n$, $x_i(t)$, $y_i(t)$, $R_i(t)$ are increasing functions[7] of $t$.

$\beta_i$ is a tuple $(\beta_i^l, \beta_i^u)$, where $\beta_i^l$ and $\beta_i^u$ are lower and upper *resource-based* service curves characterizing the service offered by the PE to stream $i$. The shape of $\beta_i$ is completely determined by the scheduling policy implemented on the PE

[7]Refer to Section 3.2.1 for the meaning of ”increasing function”.

to schedule the different streams and possibly other tasks processed on this PE.

We require the PE to process the streams under the following *buffer constraints*.

1. The PE's input buffers must never overflow.
2. The playout buffers, at the PE's output, must neither overflow nor underflow.

Our evaluation framework can solve the following two problems:

- For the PE in Fig. 36, given functions $x_i(t)$ and $R_i(t)$, and buffer sizes $l_i$ and $L_i$ for stream $i$, the first problem is to compute functions $\bar{\sigma}_i^l$ and $\bar{\sigma}_i^u$, which we refer to as *service bounds*. The service bounds have to guarantee that if the actual service provided to the stream satisfies them, then none of the buffers overflow and the playout buffer never underflows.
- Once we have obtained the service bounds for each stream being processed by the PE, the second problem is to check whether a given scheduler (specified by the set of resource-based service curves $\beta_1, \beta_2, \ldots, \beta_n$) is *feasible*. The given scheduler is feasible if it satisfies the above buffer constraints for all the streams. If the scheduler is feasible, a designer can then further evaluate it by considering other factors, such as scheduling overhead and implementation complexities.

### 5.5.2 Service bounds

In this subsection, we describe how to compute upper and lower *service bounds* that have to be satisfied for each input stream if the buffer constraints associated with the stream are to be satisfied.

Assume stream $i$ receives service $\bar{\beta}_i$ from the PE in Fig. 36. $\bar{\beta}_i$ is a tuple $(\bar{\beta}_i^l, \bar{\beta}_i^u)$, where $\bar{\beta}_i^l$ and $\bar{\beta}_i^u$ are the lower and upper *event-based* service curves characterizing the service provided by the PE to stream $i$. Two factors determine $\bar{\beta}_i$:

- the execution time of the stream objects belonging to stream $i$, and
- the scheduling policy implemented on the PE.[8]

Note the distinction between event-based service curves $\bar{\beta}_i$, which are unknown in our setup, and resource-based service curves $\beta_i$, which are fully determined by a given scheduling policy. In this subsection, we will deal only with event-based service curves.

[8]Although there may be other factors influencing the service provided to a stream on the PE, such as the number of consumed and produced stream objects per task execution, for simplicity we do not consider them here. In particular, we assume in this section that a task processing a stream consumes and produces only one stream object per one execution. Our framework, however, is not restricted to this special case.

Now, consider the streams $i = 1, 2, \ldots, n$ in Fig. 36. For simplification, we drop identifier $i$. We express the playout buffer underflow constraint for the stream as

$$y(t) \geq R(t), \quad \forall t \geq 0 \tag{5.6}$$

Similarly, we express the constraint on the playout buffer overflow as

$$y(t) \leq R(t) + L, \quad \forall t \geq 0 \tag{5.7}$$

Finally, we express the constraint on the overflow of the input buffer associated with the stream as

$$y(t) \geq x(t) - l, \quad \forall t \geq 0 \tag{5.8}$$

Combining (5.6) and (5.8), we obtain constraint

$$y(t) \geq R(t) \vee (x(t) - l), \quad \forall t \geq 0 \tag{5.9}$$

If lower event-based service curve $\bar{\beta}^l$ represents the minimum service that the PE guarantees to the stream, then ([85])

$$y(t) \geq (\bar{\beta}^l \underline{\otimes} x)(t), \quad \forall t \geq 0 \tag{5.10}$$

Hence, the minimum value of $y(t)$ at any time $t$ is $(\bar{\beta}^l \underline{\otimes} x)(t)$. Then, substituting for $y(t)$ in the constraint (5.9), we obtain

$$(\bar{\beta}^l \underline{\otimes} x)(t) \geq R(t) \vee (x(t) - l), \quad \forall t \geq 0$$

Since for any increasing functions $f$,$g$ and $h$, $g \underline{\otimes} h \geq f$ if and only if $h \geq f \overline{\oslash} g$ [85], we can further reformulate the above constraint as

$$\bar{\beta}^l(\Delta) \geq ((R \vee (x - l)) \overline{\oslash} x)(\Delta), \quad \forall \Delta \geq 0$$

or, equivalently, if we expend the min-plus deconvolution operator, we have

$$\bar{\beta}^l(\Delta) \geq \sup_{\Delta \geq 0}\{(R(t + \Delta) \vee (x(t + \Delta) - l)) - x(t)\}, \quad \forall \Delta \geq 0$$

Finally, after rearranging the above inequality we obtain

$$\bar{\beta}^l(\Delta) \geq (R \overline{\oslash} x \vee (x \overline{\oslash} x - l))(\Delta), \quad \forall \Delta \geq 0 \tag{5.11}$$

If the upper event-based service curve $\bar{\beta}^u$ represents the maximum service that the stream can receive from the PE, then ([85])

$$y(t) \leq (\bar{\beta}^u \underline{\otimes} x)(t), \quad \forall t \geq 0$$

holds. Therefore, using $(\bar{\beta}^u \underline{\otimes} x)(t)$ as the maximum value of $y(t)$, we can reformulate the constraint on the playout buffer overflow, (5.7), as

$$(\bar{\beta}^u \underline{\otimes} x)(t) \leq R(t) + L, \quad \forall t \geq 0$$

or equivalently,

$$\bar{\beta}^u(\Delta) < (R \,\overline{\oslash}\, x)(\Delta) + L, \quad \forall \Delta \geq 0 \tag{5.12}$$

Inequalities (5.11) and (5.12) give lower and upper bounds on the values of $\bar{\beta}^l$ and $\bar{\beta}^u$ that satisfy the buffer constraints associated with the stream. These bounds represent the service bounds, $\bar{\sigma}^l$ and $\bar{\sigma}^u$, which we were aiming to find in this subsection. Hence, for stream $i$, we have

$$\bar{\sigma}_i^l(\Delta) = (R_i \,\overline{\oslash}\, x_i \vee (x_i \,\overline{\oslash}\, x_i - l_i))(\Delta) \tag{5.13}$$
$$\bar{\sigma}_i^u(\Delta) = (R_i \,\overline{\oslash}\, x_i)(\Delta) + L_i \tag{5.14}$$

Thus, any feasible scheduler implemented on the PE must satisfy

$$\bar{\beta}_i^l(\Delta) \geq \bar{\sigma}_i^l(\Delta), \quad \forall \Delta \geq 0 \tag{5.15}$$
$$\bar{\beta}_i^u(\Delta) < \bar{\sigma}_i^u(\Delta), \quad \forall \Delta \geq 0 \tag{5.16}$$

for all $i = 1, 2, \ldots, n$.

**Computing service bounds for class of streams**

Service bounds $\bar{\sigma}_i^l$ and $\bar{\sigma}_i^u$, obtained above, are only for a concrete instance of stream $i$, which is specified by cumulative arrival function $x_i(t)$. Hence, they can guarantee satisfaction of the buffer constraints only for this specific instance. However, we would like to derive the service bounds for a whole *class of streams* that can appear at input $i$ of the PE. We assume that this class is specified by event-based arrival curves $\bar{\alpha}_{x_i}^l$ and $\bar{\alpha}_{x_i}^u$, i.e. for any $x_i(t)$

$$\bar{\alpha}_{x_i}^l(\Delta) \leq x_i(t + \Delta) - x_i(t) \leq \bar{\alpha}_{x_i}^u(\Delta), \quad \forall \Delta, t \geq 0$$

always holds. Furthermore, for the specification of the real-time output device reading processed stream $i$ from the playout buffer, instead of using cumulative function $R_i(t)$, we use event-based service curves $\bar{\beta}_{R_i}^l$ and $\bar{\beta}_{R_i}^u$. $\bar{\beta}_{R_i}^l(\Delta)$ and $\bar{\beta}_{R_i}^u(\Delta)$ return minimum and, respectively, maximum number of stream objects that the output device can read from the playout buffer within any time interval of length $\Delta$.

Consider the lower service bound determined by (5.13). First, we note that self-deconvolution $(x_i \,\overline{\oslash}\, x_i)(\Delta) \leq \bar{\alpha}^u(\Delta)$. Second, since $R_i(t) \leq \bar{\beta}_{R_i}^u(t)$ and $x_i(t) \geq \bar{\alpha}_{x_i}^l(t)$, we have

$$(R_i \,\overline{\oslash}\, x_i)(\Delta) \leq (\bar{\beta}_{R_i}^u \,\overline{\oslash}\, \bar{\alpha}_{x_i}^l)(\Delta), \quad \forall \Delta \geq 0$$

Hence, we can reformulate (5.13) as follows.

$$\bar{\sigma}_i^l(\Delta) = (\bar{\beta}_{R_i}^u \,\overline{\oslash}\, \bar{\alpha}_{x_i}^l \vee (\bar{\alpha}_{x_i}^u - l_i))(\Delta) \tag{5.17}$$

Now consider the upper service bound determined by (5.14). Because of $R_i(t) \geq \bar{\beta}_{R_i}^l(t)$ and $x_i(t) \leq \bar{\alpha}_{x_i}^u(t)$, the following inequality holds

$$(R_i \,\overline{\oslash}\, x_i)(\Delta) \geq (\bar{\beta}_{R_i}^l \,\overline{\oslash}\, \bar{\alpha}_{x_i}^u)(\Delta), \quad \forall \Delta \geq 0$$

Thus, we can replace (5.14) with

$$\bar{\sigma}_i^u(\Delta) = (\bar{\beta}_{R_i}^l \,\overline{\oslash}\, \bar{\alpha}_{x_i}^u)(\Delta) + L_i \tag{5.18}$$

### 5.5.3 Feasibility check

Now we come to the second problem. Given a scheduler to be implemented on a PE, does the resulting service offered to each media stream processed on this PE match the service that the stream requires?

At this point, we would like to note that before even considering a particular scheduler the service bounds can already tell us whether at all there exists a scheduler which can satisfy the buffer constraints for a given set of streams. If the following conditions evaluate to true for all streams in the set (i.e. for all $i$), then *in principle* there exists a feasible scheduler:

$$\bar{\sigma}_i^l(\Delta) \leq \bar{\sigma}_i^u(\Delta), \quad \forall \Delta \geq 0,\ i = 1, 2, \ldots, n \tag{5.19}$$

(5.19) will be false in case the buffer constraints are conflicting. Hence, only after verifying (5.19) it makes sense to proceed with feasibility checks of particular schedulers.

The service required by a stream is what we obtained in the previous subsection as the service bounds. However, we computed this requirement in terms of the number of stream objects to be processed within any given time interval. The service that a scheduler provides to a stream, on the other hand, is naturally expressed in resource-based units, e.g. in terms of the number of processor cycles. Hence, we need a way to express this service in terms of the number of stream objects. Because of the variability in the execution requirements of different stream objects belonging to a stream, this is difficult. We address this problem by characterizing the variability using the execution demand curves defined in Def. 7.

Let $\gamma_i^l$ and $\gamma_i^u$ denote the lower and upper execution demand curves characterizing the task which processes stream $i$ on the PE in Fig. 36. As stated in the problem definition, the service that a scheduler offers to stream $i$ on the PE is specified by resource-based service curves $\beta_i = (\beta_i^l, \beta_i^u)$. I.e., $\beta_i^l(\Delta)$ and $\beta_i^u(\Delta)$ denote the minimum and maximum number of processor cycles available to stream $i$ within any time interval of length $\Delta$. Then, for the scheduler to be feasible for stream $i$,

$$(\gamma_i^{u^{-1}} \odot \beta_i^l)(\Delta) \quad \geq \quad \bar{\sigma}_i^l(\Delta), \quad \forall \Delta \geq 0 \tag{5.20}$$

$$(\gamma_i^{l^{-1}} \odot \beta_i^u)(\Delta) \quad < \quad \bar{\sigma}_i^u(\Delta), \quad \forall \Delta \geq 0 \tag{5.21}$$

These inequalities should hold for all the streams processed by the PE that contains the scheduler.

### 5.5.4 Case study: Evaluating TDMA schedulers

We used our framework to evaluate different schedulers on $\mathrm{PE2}$ of the set-top box in Fig. 29. Again, for simplicity, we restrict ourselves to TDMA schedulers.

In Fig. 29, $\mathrm{PE2}$ executes tasks for both the video and the audio streams and, therefore, represents a shared resource in the platform architecture. Identifying an appropriate scheduler for $\mathrm{PE2}$ is thus an issue that the system designer must address. To avoid degradation of the sound and picture quality, such a scheduler must ensure that no audio or video samples are lost due to an overflow of any of the buffers and that playout buffers never underflow. Such a scheduler might be difficult to identify because of the high variability in the execution time of the different tasks running on $\mathrm{PE2}$ and the burstiness of the two streams that it processes.

Because $\mathrm{PE2}$ processes only two streams, any TDMA-based scheduler is completely specified by weights $w_1$ and $w_2$ and period $p$. Determining $w_1$ and $w_2$ is relatively straightforward. The long-term average rate at which $\mathrm{PE2}$ processes either of the two streams must exactly equal the corresponding output devices long-term average consumption rate for that stream. Either a buffer overflow or underflow is bound to occur at some point if these long-term rates do not match. Designers should therefore choose weights $w_1$ and $w_2$ to match these rates. However, there can be short-term mismatches in the processing and consumption rates because of the burstiness of the streams and the variability in their execution requirements from $\mathrm{PE2}$. The tolerable amount of mismatch depends on the sizes of the internal and playout buffers associated with each stream, and period $p$.

Finding an appropriate value of $p$ is not straightforward. There is generally a set of such values satisfying all buffer constraints. However, given any value of $p$, our framework can determine whether the resulting scheduler is feasible. After determining a set of feasible values of $p$, the system designer can use other evaluation criteria, such as incurred scheduling overhead or power consumption, to narrow that set.

#### System and workload specification

The system configuration for the platform architecture in Fig. 29 is as follows. Each PE is a reduced-instruction set computing (RISC) core. $\mathrm{PE1}$ has application-specific extensions for MPEG-2 processing and runs at a clock rate of 200 MHz. $\mathrm{PE2}$ has application-specific extensions for video-processing functions and runs at a clock rate of 390 MHz.

Fig. 37 shows an abstract view of processing element $\mathrm{PE2}$ in the system model of the set-top box in Fig. 29. The two input buffers with sizes $l_v$ and $l_a$, in Fig. 37, correspond to buffers $B_2$ and $B_3$ in Fig. 29. The two playout buffers of sizes $L_v$ and $L_a$ correspond to buffers $B_v$ and $B_a$. Tab. 7 gives the corresponding buffer sizes.

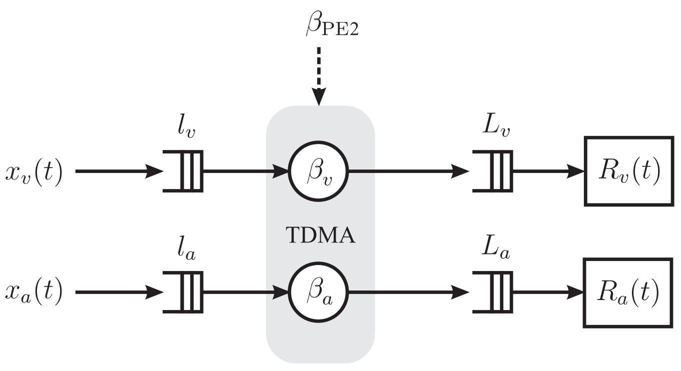


**Fig. 37:** An abstract view of PE2 in the set-top box application scenario in Fig. 29.

| Buffer | Notation | Size |
|---|---|---|
| $B_2$ | $l_v$ | 4000 macroblocks |
| $B_v$ | $L_v$ | 3200 macroblocks |
| $B_3$ | $l_a$ | 4 frames |
| $B_a$ | $L_a$ | 4 frames |

**Tab. 7:** Buffer sizes for the platform architecture in Fig. 29.

| Stream | Parameter | Specification |
|---|---|---|
| MPEG-2 video[†] | constant bit rate | 8 Mbps |
| | frame rate | 25 fps |
| | picture resolution | 704×576 |
| | clip duration | 15 sec |
| MP3 audio | constant bit rate | 256 kbps |
| | sampling frequency | 44.1 kHz |
| | clip duration | 15 sec |

[†] `susi_080.m2v`, available at
`ftp.tek.com/tv/test/streams/Element/MPEG-Video/625/`

**Tab. 8:** Specification of the two media streams processed by the platform architecture in Fig. 29.

Tab. 8 gives the parameters related to the two streams given in Fig. 29. The streams correspond to an MPEG-2 video and an MP3 audio clip. These represent typical clips that the set-top box in Fig. 29 must process.

In Fig. 37, cumulative arrival functions $x_v(t)$ and $x_a(t)$ specify the video and the audio streams at the PE2's input, whereas $R_v(t)$ and $R_a(t)$ specify the video and the audio output devices reading the playout buffers. We obtained the $x_v(t)$ function in Fig. 37 by measuring the execution times of the VLD and IQ tasks for each macroblock in the video sequence and by accounting for

- the constant arrival rate of the compressed bitstream at the PE1's input in Fig. 29;

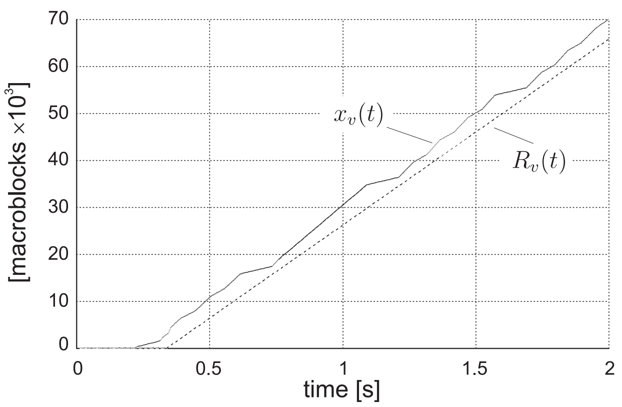


**Fig. 38:** Specification of the video stream: functions $x_v(t)$ and $R_v(t)$ from Fig. 37.

and

- the number of bits per macroblock, which is variable because of the VLD task.

Further, in our setup, $\mathrm{PE1}$ executed only the task performing the VLD and IQ functions. I.e., the full processing capacity of $\mathrm{PE1}$ was devoted exclusively to this task, and no scheduler, therefore, was employed to schedule $\mathrm{PE1}$.

Fig. 38 shows the resulting function, $x_v(t)$. Similarly, function $R_v(t)$, shown in Fig. 38, specifies the consumption of the video stream by video output device. The value of this function is $0$ for the first $0.34$ seconds, corresponding to a playout delay. After this delay, the function increases with a constant slope, representing a periodic consumption pattern of $39.600$ macroblocks per second. (One macroblock corresponds to a $16 \times 16$ pixel block in a frame; thus, one frame with resolution $704 \times 576$ pixels contains $1.584$ macroblocks. Therefore, $25$ frames per second result in $39.600$ macroblocks per second.)

The execution demand curves shown in Fig. 39 capture the total execution requirements of the task performing the IDCT and MC functions on $\mathrm{PE2}$. Rather than using a constant value, we use the execution demand curves to capture the variation in the total execution requirements of this task. To obtain these curves, we first collected a trace of execution times for the task and then analyzed this trace with the method described in Section 3.5.2.

Functions $x_v(t)$ and $R_v(t)$, combined with the execution demand curves, completely specify the video stream. We obtain the specification of the audio stream similarly. Once such a specification of the streams is available, designers can use our framework to evaluate any scheduler using the process in Fig. 30 and without resorting to further simulations.

### Evaluating TDMA schedulers with different periods

Given functions $x_v$, $x_a$, $R_v$, and $R_a$ for the representative video and audio clips, Fig. 40 shows the service bounds, $\bar{\sigma}^l$ and $\bar{\sigma}^u$, for the video and audio streams.

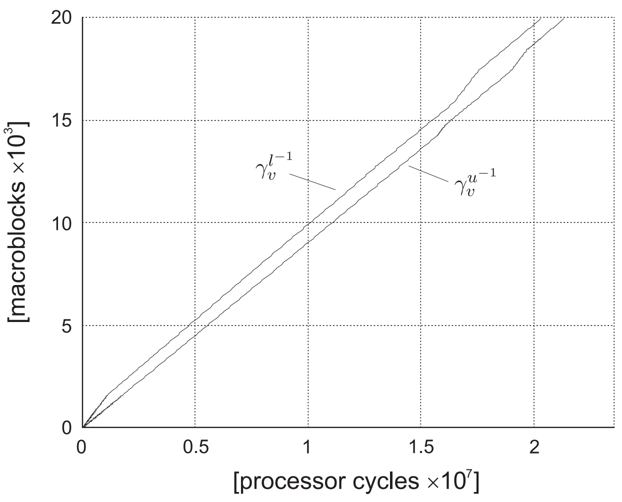


**Fig. 39:** Pseudo-inverse execution demand curves $(\gamma_v^{u^{-1}}, \gamma_v^{l^{-1}})$ capturing the execution requirements for the video task running on $\mathrm{PE2}$ in Fig. 29.

These service bounds represent the number of stream objects that $\mathrm{PE2}$ *must* process within any time interval of a given length. The figure also shows the offered *event-based* service curves $\gamma_v^{u^{-1}} \odot \beta_v^l$ and $\gamma_v^{l^{-1}} \odot \beta_v^u$, $\gamma_a^{u^{-1}} \odot \beta_a^l$ and $\gamma_a^{l^{-1}} \odot \beta_a^u$. These service curves represent the number of stream objects that $\mathrm{PE2}$ *will* process using a TDMA scheduler with a period of $2.5 \times 10^6$ processor cycles. TDMA weights $w_v$ and $w_a$, associated with the two streams, are $0.109$ and $0.891$.[9] (Fig. 33 shows the corresponding resource-based service curves $\beta_v^l$ and $\beta_v^u$ for the video stream. In Fig. 33, $\beta_{TDMA}^l = \beta_v^l$ and $\beta_{TDMA}^u = \beta_v^u$.)

Fig. 40 illustrates conditions (5.20) and (5.21). However, a better graphical representation of these conditions is to plot them as differences $\bar{\sigma}^u - \gamma^{l^{-1}} \odot \beta^u$ and $\gamma^{u^{-1}} \odot \beta^l - \bar{\sigma}^l$. If any of these differences takes a negative value, then the corresponding scheduler parameters are potentially infeasible. We refer to such plots as *difference plots*. Fig. 41(a) shows the difference plots corresponding to the configuration in Fig. 40. By inspecting the difference plots in Fig. 41(a), we can see that for the TDMA scheduler with period $2.5 \times 10^6$ processor cycles the feasibility conditions are satisfied. By comparing the difference plots in Fig. 41(a) with the difference plots in Fig. 41(b), which were obtained for a scheduler with period $7 \times 10^6$ processor cycles, we can conclude that the larger the scheduler's period is, the higher chances for buffer underflows and overflows are. In fact, in Fig. 41(b) we can see that the scheduler with period $7 \times 10^6$ processor cycles is potentially infeasible—either a buffer overflow or an underflow may occur while processing the video stream.

[9]Although in our setup $w_v + w_a = 1$, in general, the sum of the two weights does not need to be strictly equal to one. It is only necessary that $w_v + w_a \leq 1$. If $w_v + w_a < 1$, then some portion of the PE's bandwidth will be allocated neither to the video nor to the audio stream. This portion can then be used to execute some other tasks, or it can (has to) be simply wasted.

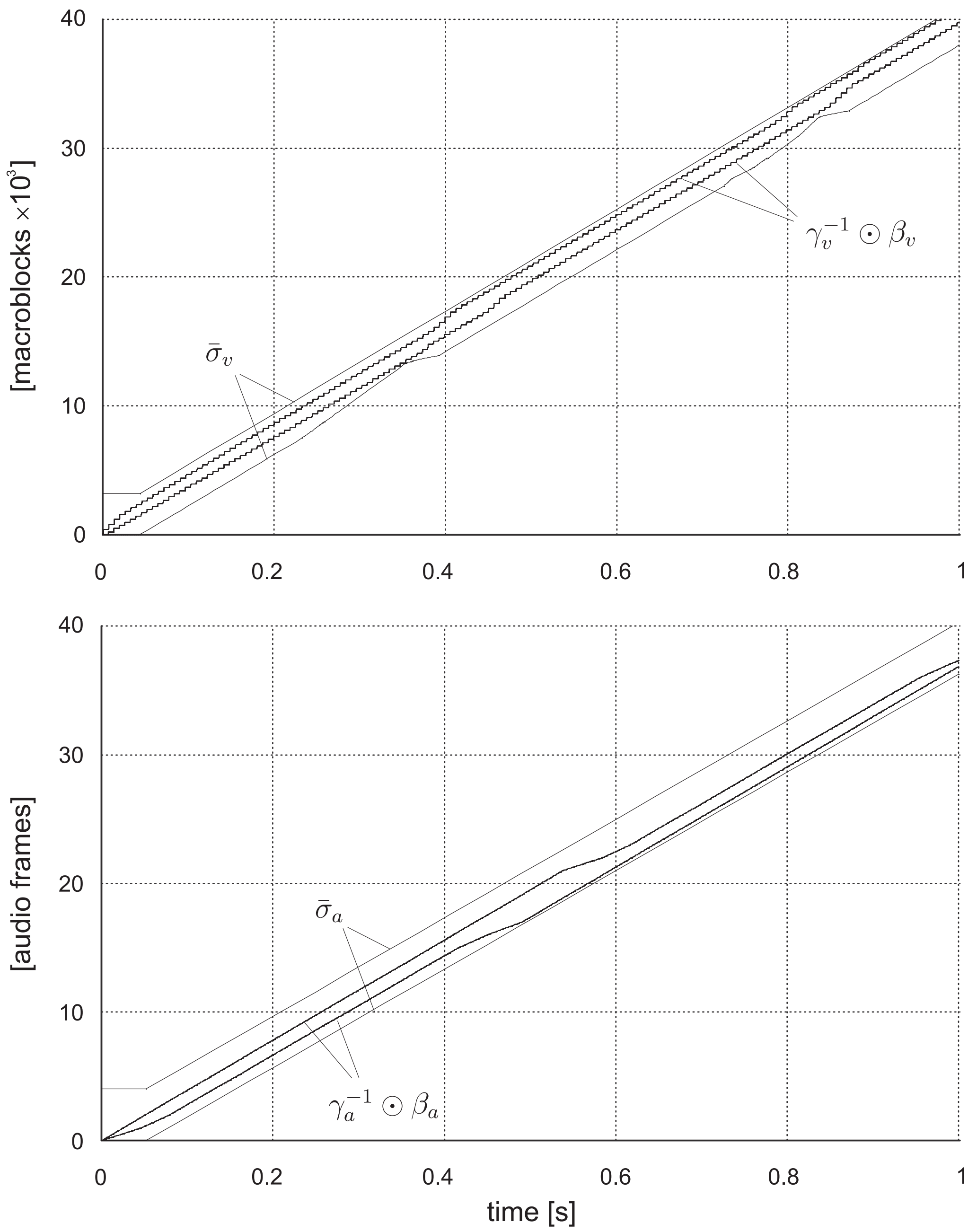
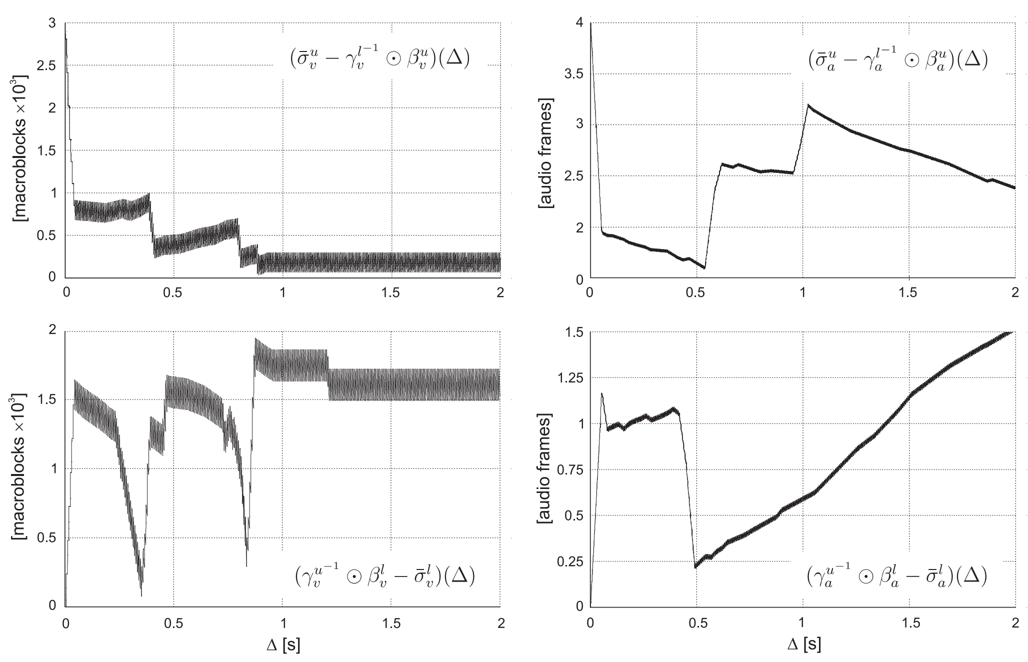
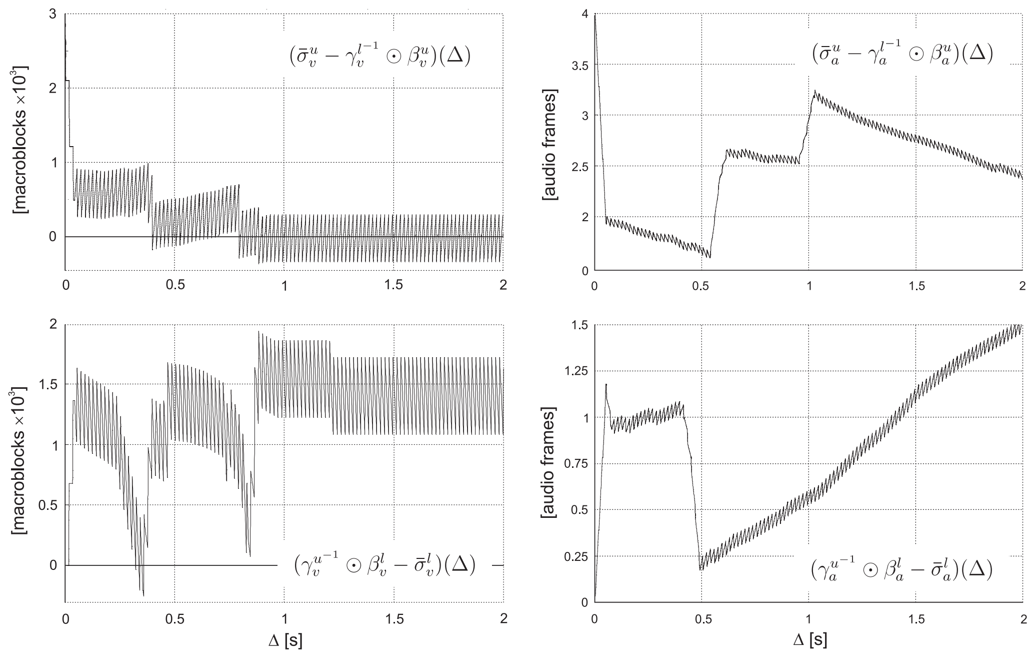


**Fig. 40:** Service requirements specified by service bounds $\bar{\sigma}$ and the actually provided service specified by curve $\gamma^{-1} \odot \beta$ for the video (top) and audio (bottom) streams. The upper and lower curves corresponding to $\gamma^{-1} \odot \beta$ lie completely between the upper and lower service bounds, $\bar{\sigma}$. This implies that the service provided to a stream matches its requirements; hence, the scheduler satisfies all the buffer constraints. For an infeasible scheduler, the resulting upper and lower curves of $\gamma^{-1} \odot \beta$ would not completely lie between the upper and lower curves of $\bar{\sigma}$.



(a) Difference plots for the TDMA scheduler with period $2.5 \cdot 10^6$ cycles.



(b) Difference plots for the TDMA scheduler with period $7 \cdot 10^6$ cycles.

**Fig. 41:** Difference plots for the video stream (left column) and for the audio stream (right column) obtained for TDMA schedulers with different periods.

| Scheduler parameters | | | Schedulability test | Buffer backlog measured in simulation | | | |
|---|---|---|---|---|---|---|---|
| Period, (cycles) | Video weight | Audio weight | | $B_2$ | $B_v$ | $B_3$ | $B_a$ |
| $1 \times 10^6$ | 0.109 | 0.891 | Passed | 3610 | 2437 | 2 | 2 |
| | 0.115 | 0.885 | Failed | 2844 | 3299* | 2 | 2 |
| | 0.106 | 0.894 | Failed | 4812* | 1979 | 2 | 3 |
| $2.5 \times 10^6$ | 0.109 | 0.891 | Passed | 3736 | 2559 | 2 | 2 |
| | 0.115 | 0.885 | Failed | 2966 | 3402* | 2 | 2 |
| | 0.106 | 0.894 | Failed | 4899* | 2110** | 2 | 3 |
| $7 \times 10^6$ | 0.109 | 0.891 | Failed | 4040* | 2540 | 2 | 2 |
| | 0.115 | 0.885 | Failed | 3292 | 3300* | 2 | 2 |
| | 0.106 | 0.894 | Failed | 5144* | 2023** | 2 | 3 |

* Buffer overflow
** Buffer underflow

**Tab. 9:** Results obtained with our framework compared to simulation results, for different configurations of a TDMA scheduler implemented on PE2 of the architecture shown in Fig. 29. Buffer backlog is measured in number of macroblocks for video, and in number of frames for audio.

**Validating the analytical framework**
To validate our framework, we evaluated several different TDMA-based schedulers, having different values of $w_v$, $w_a$, and $p$. Tab. 9 summarizes the results. For each scheduler configuration, the table shows

- whether our framework evaluated the scheduler as feasible or infeasible, and
- the corresponding simulation results that measure the maximum and minimum buffer fill levels (from which we can identify buffer overflows and underflows).

To obtain these simulation results, we used a transaction-level model of the architecture described in Appendix A. The models of processors PE1 and PE2 are from a customized version of the SimpleScalar instruction set simulator [8]; we used the simulator's `sim-profile` configuration. The PEs use the portable instruction set architecture (PISA) with application-specific extensions for MPEG-2 decoding and video processing. In the table, the buffer backlogs are in number of macroblocks for the video stream, and number of frames for the audio stream. From Tab. 9, it is apparent that designing an appropriate scheduler can greatly influence buffer space requirements. If the design space is relatively large, especially for scheduling multiple PEs, resorting to purely simulation-based techniques is no longer feasible. Our framework can provide systematic guidance in such cases.

## 5.6 Summary

In this chapter, we proposed a framework for design space exploration and optimization of platform management policies for multimedia MpSoC architectures. This framework relies on the VCC-based workload model and uses the extended MPA framework developed in Chapter 3. It features a combination of simulation and analytical performance evaluation phases. The simulation phase is performed only once, before the design space exploration loop is started. The outcome of this phase is a set of VCCs capturing relevant workload characteristics. These VCCs are then used within the time-critical exploration loop for fast analytic performance evaluation and optimization of platform management policies. The performance evaluation and optimization of the platform management policies rely on two techniques: The first technique computes upper bound on the buffer memory requirements associated with a given platform management policy, while the second technique allows for fast schedulability checks for a given PE within the architecture. Both techniques account for the QoS requirements associated with processing media streams on buffer-constrained multiprocessor architectures. The utility of the framework was demonstrated in this chapter through the case studies of a set-top box application scenario involving the TDMA scheduling policy.

# 6

# Energy-Efficient Stream Processing

Chapters 4 and 5 demonstrated applications of VCCs in system-level design of media processors. Unlike these applications, where VCCs were used in an *off-line* setting, this chapter demonstrates an *online* application of VCCs. "Online application" means that VCCs are used in a multimedia embedded system while it is operating, i.e. at run time. Based on the run-time information about the workload, certain system parameters can be dynamically adapted such that to optimally suit current user needs and to improve the quality of service offered by the system to its users. Reducing energy consumption of embedded systems through online adaptations to the varying workload is one important incarnation of this basic idea. In this chapter, we explore this direction — we show how VCCs can be used to perform such adaptations to achieve energy savings in media processors. Although this chapter has a clear focus on the energy-aware adaptations, the main concept behind the presented approach can be applied in a broader context of adaptive stream scheduling for real-time embedded systems.

The focus of this chapter is on a scheme for dynamic voltage scaling (DVS) for processing media streams on architectures with restricted buffer sizes. The main advantage of this scheme is its ability to provide *hard* QoS guarantees while still achieving considerable energy savings. VCCs are central to the whole method. They allow to handle multimedia workloads characterized by both, the data-dependent variability in the execution time of multimedia tasks and the burstiness in the on-chip traffic arising out of multimedia processing. The main novelty of the scheme lies in a online DVS strategy which uses for the adaptations *dynamic VCCs*, i.e. VCCs that are computed at run time based on the "conventional" *static VCCs* and the workload history. The DVS scheme is fully scalable and has a bounded application-independent run-time overhead.

This chapter has the following structure:

- Section 6.1 introduces our DVS technique, outlining main differences to similar approaches.
- Section 6.2 gives an overview of existing DVS techniques.
- Using a motivating example, Section 6.3 describes the problem addressed by our method.
- Section 6.4 describes the method.
- Section 6.5 presents results of an experimental evaluation of the proposed DVS technique, including a comparison with a similar method.
- Finally, some concluding remarks come at the end of this chapter, in Section 6.6.

## 6.1 Introduction

Multimedia applications constitute a significant portion of the workload running on battery-powered devices such as PDAs, mobile phones and portable audio-video players [36, 77]. A major challenge faced by the designers of such devices is the need for minimizing energy consumption and at the same time handling computationally expensive multimedia workloads and providing QoS guarantees. The bursty and highly irregular nature of such a workload, coupled with stringent memory and cost constraints associated with portable devices makes this problem even more difficult.

One important research direction aimed at solving this problem relies on dynamically changing the processor's clock frequency and voltage in response to a time-varying workload [27]. This technique, called Dynamic Voltage Scaling (DVS), rests on the fact that reducing the supply voltage of CMOS circuits results in approximately quadratic reduction in dynamic energy dissipation. Although this energy reduction does not come for free (a lower supply voltage leads to increased gate delays), DVS is known to be more effective than another energy saving technique, called Dynamic Power Management (DPM) [16]. In contrast to DVS, DPM simply puts a processor into a low power state when the processor is idle. With the availability of variable-voltage processors [3, 58, 60, 164], research on DVS scheduling techniques has gained a lot of momentum.

Govil et al. [46] proposed to look at different DVS scheduling techniques from two perspectives: the way how those techniques *predict* the workload and how they *smooth* it. In order to make a decision at which speed to run the processor during next interval of time, a DVS scheduler first has to make some assumptions about the workload on this interval. Making such assumptions can be regarded as predicting the future workload. On the other hand, workload smoothing refers to a *policy* followed by the DVS scheduler while deciding the processor speed. The concrete decision is based on the assumptions about the future workload, while the policy is typically determined by the performance goals and constraints associated with the application(s) executed on the processor.

One promising approach for *smoothing out* the workload is to employ buffers. This technique usually achieves considerable energy savings, but at the expense of increased processing delay. Nevertheless, it is especially useful in a large class of multimedia applications, which can tolerate such delays.

Recently, a number of DVS techniques has appeared in the literature which use buffers for smoothing out the workload [49, 59, 104, 106, 144, 174]. These techniques can be broadly classified into three groups based on the way how they perform the *workload prediction*. [49, 144] predict the future workload based on stochastic models. [106, 174] employ feedback control loops to track workload changes and extrapolate the future workload. [59, 104] rely on off-line worst-case characterization of tasks and statically use this characterization at run-time for making conservative scheduling decisions (i.e. essentially they do not use *prediction* as such, but assume at any time that the worst case workload will happen).

Although it has been shown that the above lines of work lead to considerable energy savings, all of them still suffer from a number drawbacks. The schemes based on stochastic prediction models and feedback control loops are powerful in handling workloads which are characterized by both, the data-dependent variability in the execution time of multimedia tasks and the burstiness in the on-chip traffic arising out of multimedia processing. However, usually it is difficult to provide hard QoS guarantees with such schemes. On the other hand, schemes that rely on worst-case characterization of the workload can provide hard QoS guarantees. However, they account only for the task execution time variability, assuming that real-time tasks arrive strictly periodically.

In this chapter, we present a DVS scheduling technique which addresses the above mentioned shortcomings of the previous approaches. It takes into account both, the burstiness in a stream and the data-dependent variability in the execution time of a task. On the other hand, our scheme offers a guaranteed QoS along with energy savings that are comparable with those obtained by previous approaches. Furthermore, one of the main assumptions made by many existing DVS schemes has been the availability of large buffers. However, in reality, many portable devices have severe cost and memory constraints. We address this issue by targeting our DVS scheme specifically towards *buffer-constrained architectures*.

Our scheme relies on an off-line analysis to determine *bounds* on the variability of the workload associated with a *class* of media streams. These bounds are represented by VCCs introduced in Chapter 3. At run-time, by using a bounded amount of history of the actually incurred workload, the VCCs obtained from the off-line analysis are revised. Such revised VCCs, which we refer to as *dynamic*

*VCCs*, are then used to adjust the processor's voltage and clock frequency. These dynamic VCCs can be much tighter than their respective *static VCCs*, i.e. those VCCs obtained from the off-line analysis. At the same time, the dynamic VCCs are "safe" in terms of guaranteeing QoS constraints.

The main results of this chapter are:

- A strategy for online processor rate adaptations for energy-efficient processing of media streams on buffer-constrained architectures.
- A formalization of this strategy as an algorithm which employs dynamic VCCs to provide hard QoS guarantees to the processed media streams.
- An efficient algorithm for run-time computation of dynamic VCCs from static VCCs using the workload history.
- An experimental evaluation of the resulting DVS scheme, including its comparison to another up-to-date DVS scheme [174] and an estimation of the run-time overhead.

Although we present our VCC-based DVS scheme in the context of a simple setup, where DVS is implemented on a single processor running a multimedia task, it can also be applied to more involved architectures such as on-chip networks [17, 31, 79, 144] and multiple clock domain processors [64, 123, 138, 174].

## 6.2 Related Work

Although dynamic voltage scaling emerged almost a decade ago (see, e.g. [21, 46, 122, 170]), it still remains a very active research area (e.g [5, 59, 81, 144]). Early explorations of this technique were primarily directed towards non-real-time computing systems [46, 124, 170]. Later on, DVS schemes have been developed for real-time systems, including off-line and on-line scheduling algorithms for a single processor [55, 61, 87, 177], multiple processors [69, 107, 176, 180] and heterogeneous distributed platform architectures [4, 47, 136]. Some DVS schemes were designed for hard real-time applications [69, 90, 139, 142], while others targeted soft real-time applications [56, 178]. Yet another class of DVS techniques were specifically devised for systems processing media streams [29, 30, 59, 106, 143]. With constantly shrinking technology feature sizes, there is a growing concern about rapid increase in leakage power dissipation of CMOS circuits [2, 20], which cannot be reduced by scaling the supply voltage. Consequently, new research directions, reconsidering DVS techniques and combining them, for example, with adaptive body-biasing [73, 110], recently has started to appear in the literature [5, 66, 175].

As mentioned before, it is useful to view different DVS schemes from two perspectives: how they predict the workload and which policy they use to smooth it [46]. These characteristics are tightly related to the energy-performance trade-offs made in the system. For instance, tasks in non-real-time systems do not have stringent timing constraints, therefore the prime concern of DVS schemes developed for this kind of systems is to maintain some *average* level of performance (e.g. average task response times) while maximizing the energy savings [42, 46, 124, 145, 170]. Since these DVS schemes do not need to provide hard performance guarantees, they allow for some inaccuracy in the workload predictions. Hence, these schemes rely on different sorts of statistics for predicting the workload. [46, 124, 170] propose and experimentally evaluate various DVS schemes which calculate how busy the processor was during a number of past intervals and then apply various techniques, such as a weighted average, to predict the future workload from these calculations. [145] extends the work in [46, 124, 170] by introducing a concept of *workload history filtering* and proposing several prediction strategies based on this concept. [42] further improves the above workload prediction mechanisms by taking into consideration activities of individual tasks. The majority of the DVS schemes just described employ relatively simple smoothing policies, such as setting the processor speed high enough to complete the predicted work before next adaptation point in time [46].

In hard real-time systems, guaranteeing that tasks complete within their deadlines is of prime concern [22]. Consequently, DVS schemes for hard real-time systems employ notably different mechanisms than those used in non-real-time systems. In particular, since hard performance guarantees have to be provided, such DVS schemes cannot rely on statistics to predict the future workload. In fact, they perform a kind of degenerate form of prediction—at any time these DVS schemes assume that the *worst-case* workload will happen. Hence, they fully rely on worst-case characterization of tasks. As in traditional real-time scheduling [100], most of the DVS techniques designed for hard real-time systems characterize tasks by their worst-case execution time (WCET) and *explicitly* defined arrival times and deadlines. Since such a characterization is inherently *static*, the only mechanism which these DVS schemes can use at run time to save the energy is the workload smoothing.

The main challenge in hard real-time scheduling of variable-voltage processors is to smooth the workload as much as possible but still to guarantee that all tasks complete within their deadlines. The workload smoothing is equivalent to increasing the processor utilization by minimizing the *worst-case slack time* (WST) and the *workload-variation slack time* (VST) [90]. The worst-case slack time is the time during which the processor is idle even if all tasks run at their WCET. It is inherent to many real-time schedules. The workload-variation slack time is the idle time which occurs as a result of tasks not always running at their WCET. Different DVS techniques for hard real-time systems differ in the way how they exploit these two kinds of slack time and what assumptions they make about the task set.

A number of DVS techniques produce static off-line schedules [54, 61, 81, 177]. They can handle real-time tasks with arbitrary arrival times and deadlines under the assumption that the relative timing between arrivals and deadlines within the task set is fixed. For highly variable workloads, energy savings achieved by these off-line techniques may be very modest. This is mainly due to the fact that they can exploit only WST [90].

*Online* DVS scheduling algorithms have a potential to exploit both kinds of slack time, WST and VST [9, 55, 87, 90, 139, 141, 142]. DVS techniques published in [55, 142] perform processor rate adaptations only at task boundaries. They always assume that a task will run at its WCET. As a result, these techniques can utilize only a limited amount of VST. Both techniques assume periodic task sets. The scheduler in [55] also accepts sporadic requests on a best-effort basis. [87] reports an on-line DVS scheduler based on the earliest deadline first (EDF) policy [100]. Unlike [55, 142], the scheduler in [87] can handle tasks with arbitrary arrivals. Whenever a task is rescheduled after its preemption, the DVS scheduler in [87] recomputes the processor speed setting for this task using task's remaining worst-case execution time. Hence, if there are a lot of preemptions in the system, the processor speed may be recomputed several times for a given task instance, leading to better utilization of VST. However, a large number of preemptions also means that the scheduling overhead incurred by this scheme may be considerable.

Another line of work represents so called *intra-task* DVS techniques [9, 90, 139, 141]. Within each task, they insert points at which the processor speed will be adjusted at run time. During task execution, the processor speed is adjusted depending on the currently active execution path within the task. This allows intra-task DVS techniques to relatively fully exploit both WST and VST. Although these techniques may suffer from high scheduling overheads (especially if tasks are of a small granularity and arrive at high rates), they represent an interesting trend towards a more detailed characterization of real-time tasks.

In the DVS techniques outlined above, better smoothing of real-time workloads (i.e. minimization of WST and VST) comes at a cost of more frequent processor speed adaptations. For highly variable and computationally intensive multimedia workloads, such methods might not be a first choice.

An alternative approach to workload smoothing is to use *buffering* [28, 49, 59, 104–106, 144, 174]. With buffers the frequency of processor speed adaptations can be significantly reduced. Certainly, buffering does not come for free. It requires additional memory and increases processing delays. Nevertheless, buffers are natural in processing media streams [134]. They play a central role in stream-oriented models of computation [70, 89]. Furthermore, many multimedia applications can tolerate buffering delays [59]. For such applications it may be more important to maintain a constant rate of stream objects at input or output of a PE than to satisfy some explicit deadlines. However, as we discussed in Chapter 5, buffer overflows and underflows may be of concern for this processing style.

One of the first DVS schemes employing buffers has appeared in [49]. A FIFO buffer is placed at the input of a PE whose clock rate and voltage are to be regulated. The DVS scheme computes a moving average of the execution demand to estimate the minimum sufficient processing rate for the samples currently in the buffer. This technique assumes that exact execution demands of individual samples are known *a priori*, i.e. before they have been processed. Only under this critical assumption and for periodic arrivals, the DVS scheme in [49] can guarantee absence of buffer over- and underflows. Based on the same assumptions, [28] improves the rate estimation algorithm of [49] by extending it to the case when the selection of operating frequency/voltage levels is limited to a set of discrete values.

[104] proposes to insert buffers between tasks processing a video stream in a pipelined fashion on a single PE. The buffers serve to maintain a constant output rate while allowing energy savings on a processor that has only few fixed frequency levels. In addition, inserting the buffers improves response times for sporadic tasks executed on the same processor. [104] constructs frequency-assignment graphs capturing relevant information such as buffer states, and then develops efficient graph-walking algorithms to to find optimal frequency/voltage settings at run time. It assumes that all frames to be processed are available at the start of the operation (e.g. stored in memory or on a hard disk) and that the tasks have constant execution demands.

[59] shows that hard real-time DVS techniques not employing buffers cannot fully utilize VST and therefore there is a potential to achieve higher energy savings. To fully exploit both kinds of slack time, WST and VST, [59] proposes to use buffers to delay the processing such that whenever the processor has completed execution of one task instance, it *always* finds next task instance waiting in a buffer. Hence, if a preceding task instance has finished earlier than its WCET resulting in a slack time, then the next task instance is able to fully utilize this slack time while running at a lower clock rate. To guarantee that the buffers never overflow and underflow, [59] has to assume that tasks arrivals are purely periodic.

In order to give the performance guarantees, DVS techniques in [28, 49, 59, 104] have to assume either periodic arrivals or, even worse, a complete *a priori* knowledge of the workload. These assumptions rarely hold in practice. Other buffer-based DVS schemes published in [105, 106, 144, 174] are free from these assumptions, however, as we will see, they fail to provide hard performance guarantees.

[144] presents a buffer-based combined DVS-DPM scheme which fully relies on stochastic workload characterization: stream arrivals and execution demands are characterized by probability distributions. These distributions are obtained by fitting statistics collected through extensive stochastic simulations to standard

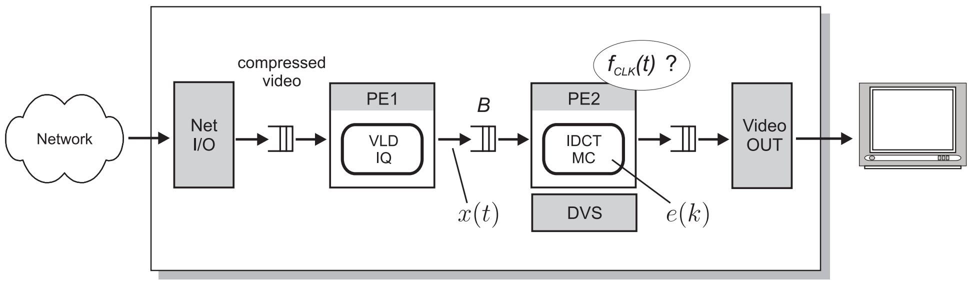


**Fig. 42:** MPEG-2 decoder implemented on two PEs. Supply voltage and clock rate of $\mathrm{PE2}$ can be controlled (DVS). $x(t)$ and $e(k)$ are cumulative arrival and execution demand functions.

probability distributions. At run time the workload history is collected and used to detect, on the basis of these distributions, changes in the workload intensity. The processor rate is then adjusted accordingly. Clearly, due to its use of stochastic models the method in [144] can provide only probabilistic guarantees. It also depends on the accuracy with which real workload is approximated using standard probability distributions.

DVS schemes presented in [105, 106, 144, 174] use different flavors of a PID (Proportional-Integral-Derivative) controller [37] to regulate the processor rate. The processor with the buffer at its input (or output) represents the controlled plant. Typically, the buffer occupancy level serves as a feedback signal to the controller. The main difficulty in these techniques is non-linearity of the controlled plant. Because of this non-linearity, it becomes extremely difficult to formally proof the properties of a control algorithm and analytically find optimal settings for its parameters (such as controller gains). Furthermore, providing hard performance guarantees (e.g. with respect to the buffer and delay constraints) under highly bursty workloads requires an overshoot-free controller with a short reaction time. However, such a controller is difficult to design because the requirements to have a short response time and to completely avoid overshoots are conflicting.

## 6.3 Motivating example

As a motivating example consider the system shown in Fig. 42. (A similar system has been already considered in Chapter 3 in Fig. 17 and in Chapter 4.) This system performs decoding of MPEG-2 video streams. It includes two processing elements $\mathrm{PE1}$ and $\mathrm{PE2}$. They can be embedded processor cores specialized for specific tasks such as video processing or any other kind of processing elements. A compressed video stream first enters $\mathrm{PE1}$, which executes a part of the MPEG-2 decoding algorithm. The task running on $\mathrm{PE1}$ performs VLD and IQ functions. After processing on $\mathrm{PE1}$, the video stream enters buffer $B$ at $\mathrm{PE2}$'s input. At this place in the system the stream exists as a sequence of partially decoded *macroblocks*. $\mathrm{PE2}$ consumes from $B$ one macroblock at a time and applies to it IDCT and MC functions. Finally, the fully decoded video stream emerges at $\mathrm{PE2}$'s output.

Our objective is to minimize the energy consumed by $\mathrm{PE2}$ without deteriorating the quality of the processed video stream. The stream's quality is preserved if buffer $B$ at the $\mathrm{PE2}$'s input never overflows and if the processing delay, experienced by the stream on $\mathrm{PE2}$, does not exceed some specified value. Our ultimate goal is to design a *predictable system*. This means that we want to ensure that the system satisfies the above mentioned QoS requirements under *all* possible load scenarios and not only in the average case.

We assume that $\mathrm{PE2}$ supports DVS, i.e. its clock rate and supply voltage can be changed at run time.[1] Such changes can be controlled by the software on $\mathrm{PE2}$ or by some other hardware or software entity, external to $\mathrm{PE2}$. We refer to time instants at which the processor speed is altered as *adaptation points*.

We assume that the adaptation points are fixed in time. For obtaining energy savings while providing the QoS guarantees, such an assumption is less favorable than the assumption that we can adapt the processor's clock rate at any time. Our method though can handle both cases. In any case, the spacing of the adaptation points in time in our method is completely decoupled from the execution state and granularity of tasks on $\mathrm{PE2}$. This means that the adaptations are not restricted to occur at task boundaries, and their frequency, in general, is independent of the rate at which the video stream arrives at the $\mathrm{PE2}$'s input.

A DVS scheduler can reduce the energy dissipated on $\mathrm{PE2}$ by exploiting the variability of the workload imposed on this PE. This variability comes from two sources. First, the execution time of the task running on $\mathrm{PE2}$ is variable. Second, the data-dependent variability of the execution time of the task running on $\mathrm{PE1}$ causes the stream of macroblocks at the $\mathrm{PE2}$'s input to be *bursty*.

One traditional way of reducing the energy dissipated on $\mathrm{PE2}$ would be to fully average out the workload imposed on it using buffer $B$. If buffer $B$ is sufficiently large, it can completely absorb the workload fluctuations. This allows $\mathrm{PE2}$ to run at a low *constant* clock rate which is just sufficient to sustain the long-term average arrival rate of the stream. In this mode, one can ensure that $B$ never gets empty (as a playout buffer). In this case, no cycles are wasted even during low-load periods, i.e. the available slack is *fully* exploited. This strategy would yield the most energy savings on $\mathrm{PE2}$. However, such a strategy is often unaffordable since it requires large buffers for processing bursty multimedia workloads like MPEG streams. As an example, our experiments showed that the complete averaging of the workload imposed by DVD-quality videos on $\mathrm{PE2}$ in Fig. 42 required in the worst case the buffer space of at least 8100 macroblocks (or about 3.7 MByte). Such a large buffer would be too expensive to imple-

[1]Throughout this chapter whenever we say that the processor's rate is changed we assume that to reduce energy its supply voltage is changed accordingly.

ment in some embedded SoC architectures. Furthermore, from the application perspective, the delay incurred by the video stream on PE2 as a result of such averaging might not be tolerable. (It is about 5 full video frames in our setup shown Fig. 42.)

In contrast, we assume that our architecture is *buffer-constrained*, i.e. the buffer space at the PE2's input is inadequate for the complete workload averaging. Hence, unless we allow buffer overflows we cannot constantly run the processor at the average rate: a burst in the stream's arrival pattern or in its execution demand can easily cause an overflow. To avoid the overflows, we could service the stream at some constant *safe rate* which is high enough to successfully handle the bursts under the given buffer constraint. Clearly, such a safe rate would be higher than the average rate, and therefore during periods with the average or low load some amount of the processor cycles would be wasted for waiting on the empty buffer. In some cases, we could save some energy by putting the processor into a low-power idle state whenever the buffer is empty, and then let it run again whenever there is something to process in the buffer (i.e. use DPM). However, in many cases the switching overhead between processor power states is too high (in the range of milliseconds [16]) compared to the stream arrival rate (in the range of microseconds in our setup shown in Fig. 42) thereby making this strategy infeasible. On the other hand, a DVS strategy which could exploit the slack during low-load periods would yield higher energy savings.

By now it should be clear that despite of the buffer constraint there is a potential to save the energy by running PE2 at a lower rate during low-load periods. However, this potential should be realized very carefully since a burst may suddenly arrive and cause a buffer overflow. Hence, our goals are conflicting: on one hand we want to stay at the average level of performance for saving the energy, but on the other hand we have to provide QoS guarantees and, therefore, need to be ready at any time to handle the worst case. Designing a scheduling strategy which can meet the both goals represents a challenging problem and involves delicate tradeoffs.

The main challenge in designing a safe DVS strategy for a system with constrained buffers, as the one described above, is in the fact that it is a priori unknown how the workload will behave in an interval between two adaptation points. Even if we knew exactly how many stream objects in total will arrive within the interval, this information would be insufficient to guarantee that the buffer will not overflow. For providing such a guarantee we need to know *how* the stream objects will arrive within the interval. For instance, they may arrive in a dense burst, right in the beginning of the interval. If in the previous interval the processor has not cleared enough buffer space to accommodate this burst, an overflow is bound to happen. Many existing DVS techniques which are capable of providing the QoS guarantees and which employ buffers for the energy reduction avoid this problem by assuming that the stream arrives into (or departs from) the buffer at a constant rate [59, 104]. This assumption, however, greatly simpli-

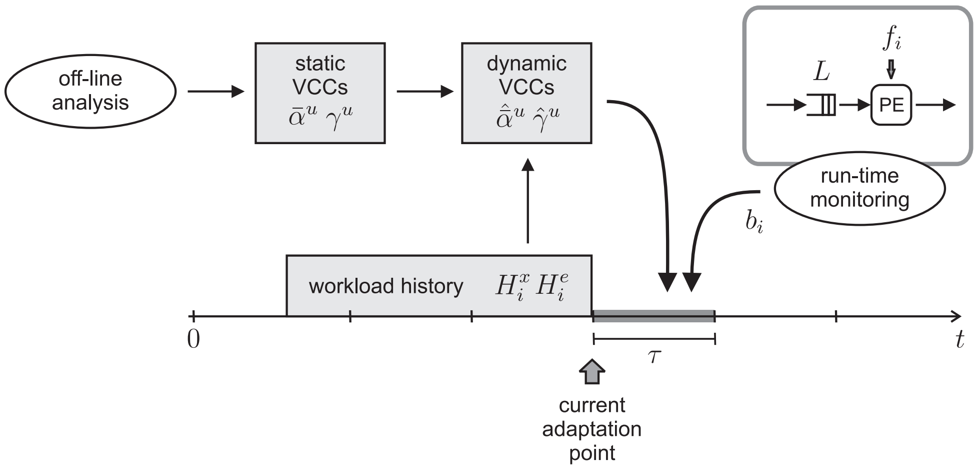

off-line analysis
static VCCs $\bar{\alpha}^u \; \gamma^u$
dynamic VCCs $\hat{\bar{\alpha}}^u \; \hat{\gamma}^u$
$f_i$
$L$
PE
run-time monitoring
$b_i$
workload history $H_i^x \; H_i^e$
0
$t$
$\tau$
current adaptation point

**Fig. 43:** Overview of the method

fies the problem and often does not hold in practice. Furthermore, by making this assumption, the existing techniques lose the opportunity to gain additional energy savings by exploiting the variability in the arrival process of the stream. They exploit only the slack resulted from the variability of the task execution time.

## 6.4 Adaptive Run-Time Scheduling with VCCs

Fig. 43 shows an overview of our method. Our algorithm dynamically adapts the clock rate of the processor to the workload variation using two mechanisms: (i) run-time monitoring of the buffer fill level (i.e. buffer $B$ in Fig. 42) and (ii) online improvement of static VCCs based on the workload history. It exploits both types of workload variability—the slack in the execution time and the irregular arrival patterns of the stream.

Central to our method is the concept of VCCs introduced in Chapter 3. This concept is the key to providing QoS guarantees and achieving good average performance. Using VCCs at run-time and taking into account the current backlog in the buffer and the workload history, our algorithm makes *safe, but not too pessimistic* decisions at the adaptation points. Such a reduced pessimism is possible because VCCs represent a more detailed characterization of the workload than traditional task and event models, as we have shown in Chapter 3.

In this chapter, we distinguish between two types of worst-case bounds: *static VCCs* and *dynamic VCCs* (see Fig. 43). The static VCCs are obtained at the design time through an off-line analysis, and then used by our DVS scheduler at run-time. In this sense, they are similar to conventional worst-case characterization of tasks, such as the worst-case execution time and the minimum interarrival time or period. The dynamic VCCs is a novel concept that we introduce in this chapter. The novelty of this concept lies in the fact that these VCCs are obtained *at run-time*. They represent an online improvement of the static VCCs. The dynamic VCCs are obtained using the workload history. Depending on the workload situation, these VCCs can be much tighter than the corresponding static VCCs. They allow our scheduler to be less pessimistic about the future workload and through this to achieve considerable energy savings. Furthermore, the dynamic VCCs provide the same level of guarantee as that provided by the static VCCs from which these dynamic VCCs were derived.

Following subsections give details about our method.

### 6.4.1 Workload and service characterization

We shall consider a stream to be composed of a potentially infinite sequence of *stream objects*. Depending on the application at hand a stream object can be an audio sample, a video (macro)block or a whole frame, etc. A stream can be modeled by two cumulative functions $x(t)$ and $e(k)$ (see Fig. 42). $x(t)$ denotes the total number of stream objects that arrived at the buffer $B$ during the time interval $[0, t]$, whereas $e(k)$ denotes the total number of execution cycles requested from the processor by $k$ consecutive stream objects starting from the first stream object in the sequence. Our objective is to characterize a whole *class* of streams that the processor has to handle. We achieve this by using upper event-based arrival curve $\bar{\alpha}^u$ and upper execution demand curve $\gamma^u$, defined in Section 3.3. For any stream $i$ belonging to the class, $\bar{\alpha}^u$ and $\gamma^u$ have to satisfy

$$x_i(t+\Delta) - x_i(t) \leq \bar{\alpha}^u(\Delta) \quad \forall t, \Delta \in \mathbb{R}_{\geq 0} \tag{6.1}$$
$$e_i(k+\epsilon) - e_i(k) \leq \gamma^u(\epsilon) \quad \forall k, \epsilon \in \mathbb{Z}_{>0} \tag{6.2}$$

Thus, tuple $(\bar{\alpha}^u, \gamma^u)$ represents a particular class of streams. In practice, a class may encompass all streams belonging to one application scenario. For instance, all MPEG-2 video sequences with identical parameters like resolution, frame rate, bit rate etc. can belong to one class.

Besides the workload model represented by the arrival and execution demand curves, we need a similar abstraction for the service offered to process this workload. Note that the rate of a DVS processor can change over time. Hence, we have to model these changes properly. For this we use the concept of a *service curves*. The theoretical framework presented in this chapter involves both, event-based and resource-based lower service curves, denoted $\bar{\beta}^l$ and $\beta^l$, respectively. In the simplest case, when the processor constantly runs at a clock rate $f$, the service curve $\beta^l(\Delta) = f \cdot \Delta$. For a processor which may be switched off for maximum $\delta$ time units and in the rest of the time runs at a constant speed $f$, e.g. as in the case of DPM, $\beta^l(\Delta) = \sup\{f \cdot (\Delta - \delta), 0\}$.

### 6.4.2 Safe service rate

To simplify exposition of the method, we introduce the concept of *safe service rate* and make the following assumption: all stream objects in a stream impose the same execution demand on the processor. This assumption implies that if the processor clock rate is constant on some time interval, then the service rate measured in number of stream objects per time unit which the processor offers to the stream is also constant on that interval. We will relax this assumption later, in Section 6.4.4, by accounting for the execution demand variability.

**Def. 16:** **(Safe service rate)** *Service rate is safe if* continuously *servicing a stream at this rate guarantees that the buffer at the input of the processor never overflows and the delay constraint associated with the processed stream is satisfied.*

Our goal is to determine the *minimum* safe rate. For this, suppose that the processor services a stream at a *constant* rate $R$, where $R$ is measured in number of stream objects that can be serviced per time unit. Hence, we can model the offered service as $\bar{\beta}^l(\Delta) = R \cdot \Delta$. Then, from (2.10) we can find the minimum service rate $R_L$ which ensures that the buffer of size $L$ never overflows:

$$R_L = \sup_{\forall \Delta \geq 0} \left\{ \frac{\bar{\alpha}^u(\Delta) - L}{\Delta} \right\} \tag{6.3}$$

Similarly, from (2.9) we can determine the minimum service rate $R_D$ satisfying the delay constraint $D$:

$$R_D = \sup_{\forall \Delta \geq 0} \left\{ \frac{\bar{\alpha}^u(\Delta)}{D + \Delta} \right\} \tag{6.4}$$

Thus, the minimum safe service rate $R_{\mathit{safe}} = \max\{R_L, R_D\}$, i.e.

$$R_{\mathit{safe}} = \sup_{\forall \Delta \in \mathbb{R}_{\geq 0}} \left\{ \frac{\bar{\alpha}^u(\Delta)}{D + \Delta}, \frac{\bar{\alpha}^u(\Delta) - L}{\Delta} \right\} \tag{6.5}$$

To guarantee satisfaction of the buffer and delay constraints, $L$ and $D$, it is sufficient (but not necessary) that the processor offers a service rate which is not lower than $R_{\mathit{safe}}$ determined by (6.5). In other words, $R_{\mathit{safe}}$ guarantees the constraint satisfaction in the worst case, e.g. during a burst in the stream arrival pattern. However, the appearance of the worst case is bounded, and therefore it is not necessary for the processor to offer to the stream a safe service rate through all the time. Whenever the worst case does not happen, the processor could offer a service rate which is lower than the minimum safe rate. In doing so, care must be taken to timely react to changes in the workload by increasing the service rate if necessary. This principle forms a basis for the dynamic processor rate adaptations which we will consider in the next subsection.

### 6.4.3 Adapting processor speed at run time

To save energy, during *low-load periods* our scheduler tries to run the processor at a rate which matches stream's arrival rate. The scheduler uses the buffer fill level as an indicator of the stream's arrival rate. At each adaptation point, the scheduler tries to set the processor rate such that the buffer fill level is *close* to zero. Since any such rate tends to match the arrival rate and is lower than the minimum safe rate $R_{\mathit{safe}}$, this strategy results in energy savings during the low-load periods.

If a workload *burst* starts arriving, the processor frequency is increased accordingly. This is done in a safe way, based on the information about the current buffer fill level and the expected future worst-case workload. The scheduler tries to *fully* exploit the available buffer space during the bursts by being as "lazy" as possible. At each adaptation point, it sets the processor rate such that it is *just* sufficient to avoid a buffer overflow in the worst case. That is, if the worst case did really happen, the buffer would reach its full state but would never overflow within the adaptation interval. Since the worst case happens rarely, this "lazy" strategy results in energy savings during the high-load periods.

Let $b_i$ denote the backlog in the buffer at $i$th adaptation point and suppose that the $i$th adaptation interval is of length $\tau$. Then, the above rate-adaptation strategy can be realized if at the $i$th adaptation point the processor rate is set to $R_{L,i}$ which is computed as follows:

$$R_{L,i} = \sup_{0<\Delta\leq\tau} \left\{ \frac{\bar{\alpha}^u(\Delta) - L + b_i}{\Delta}, 0 \right\} \tag{6.6}$$

This formula is in principle the same as (6.3), but it accounts for the initial backlog $b_i$ and is valid only for the $i$th adaptation interval of length $\tau$.

(6.6) ensures that the buffer will not overflow *within* the $i$th adaptation interval. However, it might happen that due to a burst the backlog at the end of the $i$th adaptation interval is close to its maximum allowed value. In this case, avoiding buffer overflows may require the processor to run at a high rate during the next adaptation interval. This rate might be higher than the maximum rate $R_{max}$ supported by the processor. Thus, a deadlock situation may occur. To avoid such a situation, at the $i$th adaptation point the scheduler has also to consider what might happen in the worst case *after* the $i$th adaptation interval. For this, at the $i$th adaptation point it sets the processor rate such that this rate is at least as high as $R_{min,i}$ which is computed as follows.

$$\delta_i = \frac{L - b_i}{R_{max}} + \inf_{\forall\Delta\geq\tau} \left\{ \Delta - \frac{\bar{\alpha}^u(\Delta)}{R_{max}} \right\} \tag{6.7}$$

$$R_{min,i} = R_{max} \frac{\tau - \delta_i}{\tau} \vee 0 \tag{6.8}$$

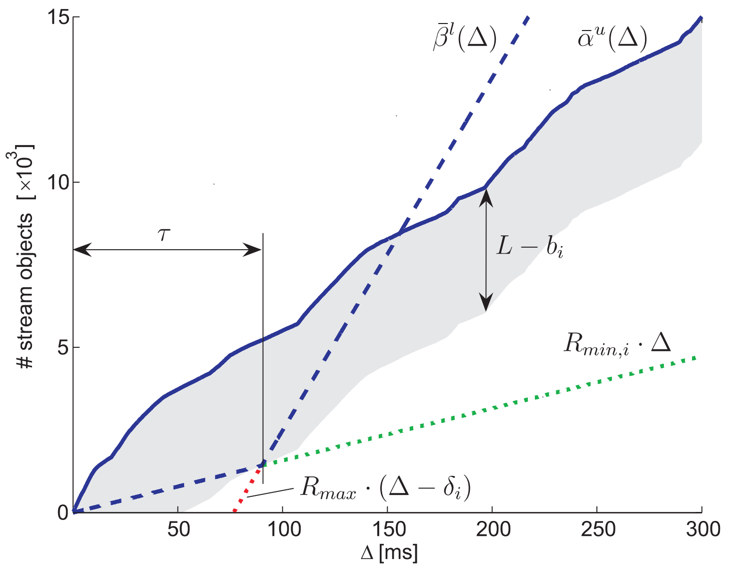


**Fig. 44:** Calculation of the minimum service rate $R_{min,i}$ for the $i$th adaptation interval.

Fig. 44 illustrates the above formulas. $R_{min,i}$ ensures that whenever the worst-case load *really* happens in the $i$th interval and the buffer is (almost) full at its end, we still can prevent the buffer from overflowing. For this, after the $i$th adaptation interval, we just have to run the processor at $R_{max}$. This holds because in the worst case the stream is guaranteed to receive a service which is not less than the service curve

$$\bar{\beta}^l(\Delta) = \sup_{\forall \Delta > 0} \{R_{min,i} \cdot \Delta, R_{max} \cdot (\Delta - \delta_i)\}$$

and

$$\bar{\alpha}^u(\Delta) - \bar{\beta}^l(\Delta) \leq L - b_i \quad \forall \Delta > 0$$

The opposite situation might also occur. If there is a lot of free space in the buffer, (6.6) and (6.8) may return zero. In this case, one could switch off the processor until the next adaptation point. However, this would not be an optimal energy saving strategy. Even if the stream's arrival rate is low during the time when the processor is switched off, a number of stream objects will accumulate in the buffer. This will necessitate the processor to run at a higher rate during the subsequent adaptation intervals thereby making this strategy not optimal for energy savings. The optimal strategy is to run the processor at a rate exactly matching the stream's arrival rate. Since the exact arrival rate is never known for future adaptation intervals, such a strategy is difficult to realize. Hence, different approximations have to be made. In our case, (6.6) and (6.8) returning zero may indicate a low-load period. The lowest rate at which the stream might arrive during that period, and therefore at which we should run the processor, is

$$R^l_{min} = \frac{\bar{\mu}_\tau}{\tau} \tag{6.9}$$

where $\bar{\mu}_\tau$ denotes the minimum number of stream objects that might arrive within *any* interval of length $\tau$. [2]

Finally, our algorithm can be summarized as follows:

$$R_i = \max\{R_{L,i}, R_{min,i}, R^l_{min}, R_D\} \tag{6.10}$$

where $R_i$ is the rate set at the $i$th adaptation point.

### 6.4.4 Accounting for variable execution demand

The above discussion was based on the assumption that all stream objects impose exactly the same execution demand on the processor. Although in reality this rarely happens, this assumption helped to illustrate the principles of the proposed rate adaptation algorithm by simplifying the formulation of the service rate constraints in the preceding subsections. However, if we were to apply this assumption in practice, we would also have to assume the *worst-case* execution demand for each stream object. As our experiments showed in Section 3.6, such an assumption may result in overly pessimistic bounds. Therefore, to avoid this problem, our method employs the upper execution demand curve $\gamma^u$, which represents a more detailed characterization of the worst-case execution demand imposed by a stream on the processor.

As discussed in Section 3.4, the workload transformation $(\gamma^u \odot \bar{\alpha}^u)(\Delta)$ gives an upper bound on the number of processor cycles that can be requested within any time interval of length $\Delta$ by any stream belonging to the class characterized by tuple $(\bar{\alpha}^u, \gamma^u)$. Using this workload transformation and Thm. 1, and by following the same principles as were used for deriving (6.4), (6.6), and (6.8), we can obtain constraints on the processor *clock* rate:

$$f_D = \sup_{\forall \Delta \geq 0} \left\{ \frac{\gamma^u \odot \bar{\alpha}^u(\Delta)}{D + \Delta} \right\} \tag{6.11}$$

$$f_{L,i} = \sup_{0 < \Delta \leq \tau} \left\{ \frac{\gamma^u \odot ((\bar{\alpha}^u(\Delta) - L + b_i) \vee 0))}{\Delta} \right\} \tag{6.12}$$

$$f_{min,i} = \sup_{\forall \Delta \geq \tau} \left\{ \frac{f_{max}(\tau - \Delta) + \gamma^u \odot ((\bar{\alpha}^u(\Delta) - L + b_i) \vee 0)}{\tau} \right\} \tag{6.13}$$

where $f_D$, $f_{L,i}$ and $f_{min,i}$ are clock frequencies, corresponding to event-based service rates $R_D$, $R_{L,i}$ and $R_{min,i}$ derived in the previous subsection under the assumption that all stream objects impose the same execution demand on the processor.

[2] In fact, $\bar{\mu}_\tau = \bar{\alpha}^l(\tau)$, where $\bar{\alpha}^l$ is the lower event-based arrival curve of the stream. To reduce clutter, we do not introduce in this chapter any lower VCCs, although they also can be useful in realizing an energy-conscious stream scheduling strategy.

Similarly, clock rate constraint $f^l_{min}$, corresponding to $R^l_{min}$, can be formulated as

$$f^l_{min} = \frac{\mu_\tau}{\tau} \tag{6.14}$$

where $\mu_\tau$ denotes the minimum number of processor cycles that might be requested by the stream within *any* interval of length $\tau$. [3]

Finally, our clock rate adaptation algorithm can be summarized as follows.

$$f_i = \max\{f_{L,i}, f_{min,i}, f^l_{min}, f_D\} \tag{6.15}$$

where $f_i$ is the value of the processor clock rate set by our DVS scheduler at the $i$th adaptation point.

### 6.4.5 Using dynamic VCCs

The algorithm described in the previous subsections uses *static VCCs* $\bar{\alpha}^u$ and $\gamma^u$. It was derived under the assumption that at any point in time nothing is known about the past workload. Now suppose that we keep a finite-length workload history. By exploiting this history we can improve the energy savings without jeopardizing the safety property of the algorithm. We use the history to revise static VCCs $\bar{\alpha}^u$ and $\gamma^u$ into their dynamic equivalents $\hat{\bar{\alpha}}^u$ and $\hat{\gamma}^u$. This subsection explains how this is exactly done.

The basic idea behind deriving the dynamic VCCs from the static VCCs using the knowledge about the workload's behavior in the past is fairly simple and can be illustrated with the following example.

**Ex. 4:** *Suppose that the upper bound on the workload imposed by a stream on a processor within any time interval of length $\Delta$ equals to $A$ processor cycles. Since this bound holds at all intervals of length $\Delta$, it is a static worst-case bound. Now, suppose that we observe the system at some point in time $t$. If we know that the execution demand requested by the stream during time interval $[t - \Delta_p, t]$ ($\Delta_p < \Delta$) was $B$ processor cycles, then we can guarantee that over the next time interval $(t, t + \Delta - \Delta_p]$, the stream will not request more than $A - B$ cycles. The value $A - B$ represents a dynamic worst-case bound over the interval $(t, t + \Delta - \Delta_p]$. At time $t$, the scheduler can safely use this dynamic worst-case bound for computing the frequency at which the processor needs to be run during the interval $(t, t + \Delta - \Delta_p]$.*

In Ex. 4, we employed a static bound which provided us with the information about the worst-case workload only on intervals of length $\Delta$. We used this information as *a constraint* to compute the dynamic worst-case bound for a future interval of a smaller length. Note that we can derive *many* such constraints from

[3] $\mu_\tau = (\gamma^l \odot \bar{\alpha}^l)(\tau)$, where $\bar{\alpha}^l$ is the lower event-based arrival curve of the stream and $\gamma^l$ is its lower execution demand curve.

static VCCs $\bar{\alpha}^u$ and $\gamma^u$, since they capture the worst-case workload on intervals of different lengths. At any given time instant, by taking into consideration the past workload, we can then select from those constraints the *tightest* ones to form the dynamic VCCs, $\hat{\bar{\alpha}}^u$ and $\hat{\gamma}^u$.

Suppose that the system is at the beginning of the $i$th adaptation interval of length $\tau$. The upper bound $\mathcal{X}_i$ on the number of stream objects that can arrive within $\tau$ is

$$\mathcal{X}_i(\tau) = \inf_{0 \le j \le N} \{\bar{\alpha}^u(j\theta + \tau) - H_i^x(j)\} \tag{6.16}$$

where $H_i^x$ is the *arrival history* at the $i$th adaptation point, $\theta$ is the *resolution of the arrival history*, and $N$ is the number of constraints that the scheduler considers for computing $\mathcal{X}_i$. $\theta$ can be interpreted as a sampling period with which arrivals are monitored.

The arrival history $H_i^x$ represents a set of $N$ sliding windows, with the $j$th window spanning the interval $[t_i - j\theta, t_i)$ and returning the number of stream objects that arrived within it. Formally,

$$H_i^x(j) = x(t_i) - x(t_i - j\theta), \quad j = 0, 1, .., N \tag{6.17}$$

Note that from (6.1), (6.16) and (6.17) it follows that $\mathcal{X}_i(\tau) \le \bar{\alpha}^u(\tau)$ for all $i \in \mathbb{Z}_{\ge 0}$ and for all $\tau \in \mathbb{R}_{\ge 0}$. Hence, for the processor rate calculation, instead of using $\bar{\alpha}^u(\tau)$ we can use $\mathcal{X}_i(\tau)$. Furthermore, since $\bar{\alpha}^u$ is an increasing function[4], $\mathcal{X}_i(\tau)$ can be used to improve $\bar{\alpha}^u$ not only for $\tau$, but also for other interval lengths that are smaller than $\tau$:

$$\hat{\bar{\alpha}}^u(i, \Delta) = \inf_{\forall \Delta \in [0, \tau]} \{\mathcal{X}_i(\tau), \bar{\alpha}^u(\Delta)\} \tag{6.18}$$

(6.18) represents the *dynamic arrival curve* used by our scheduler at the $i$th adaptation point.

By following the same principle, we can improve the execution demand bound $\gamma^u$. Let $H_i^e(j)$ denote the *execution demand history* of length $M$ and resolution $\psi$. $\psi$ is defined in terms of number of stream objects. The upper bound $\mathcal{E}_i$ on the number of processor cycles that can be requested by any sequence of $k$ consecutive stream objects after the $i$th adaptation point can be computed as follows

$$\begin{aligned} \mathcal{E}_i(k) &= \inf_{0 \le j \le M} \{\gamma^u(j\psi + k) - H_i^e(j)\} && (6.19) \\ H_i^e(j) &= e(l) - e(l - j\psi), \quad j = 0, 1, .., M && (6.20) \end{aligned}$$

where $l$ is the total number of stream objects that have been completely processed up to the $i$th adaptation point. As a result we get the *dynamic execution demand bound*:

$$\hat{\gamma}^u(i, \eta) = \inf_{\eta \in [0, \hat{\bar{\alpha}}^u(i, \tau)]} \{\mathcal{E}_i(\eta), \gamma^u(\eta)\} \tag{6.21}$$

[4]Refer to Section 3.2.1 for the precise meaning of the term ”increasing function”.

$\hat{\bar{\alpha}}^u$ and $\hat{\gamma}^u$ can used in all formulas of Sections 6.4.4 and 6.4.3 in place of $\bar{\alpha}^u$ and $\gamma^u$, respectively.

### 6.4.6 Notes on implementation

The DVS algorithm described in the previous subsections can be implemented either in SW or in HW or using a combination of the two. For any implementation, a number of considerations has to be made. These include: (i) taking into account voltage/frequency transition overhead; (ii) working in discrete time; (iii) working with discrete frequency levels; (iv) determining granularity and length of the workload history; (v) downloading the static VCCs at run time if the application scenario changes, etc. Some of these issues are briefly addressed in this subsection.

**Accounting for the voltage/frequency transition overhead**
Switching between different voltage and frequency levels may take some time $\varepsilon$. During this time the processor cannot service the stream. Hence, at the $i$th adaptation point the actual processing starts after $\varepsilon$ time units. This *time-out* can be easily modeled by the service curves. In particular, for the computation of $f_D$ and $f_{L,i}$, instead of resource-based service curve $\beta^l(\Delta) = f \cdot \Delta$, we have to consider resource-based service curve

$$\beta^l(\Delta) = \sup_{\forall\Delta\geq 0} \{f \cdot (\Delta - \varepsilon), 0\}$$

This will correspondingly change the formulas (6.11) and (6.12) for $f_D$ and $f_{L,i}$:

$$f_D = \sup_{\forall\Delta\geq 0} \left\{ \frac{\gamma^u \odot \bar{\alpha}^u(\Delta)}{D - \varepsilon + \Delta} \right\}$$

$$f_{L,i} = \sup_{\Delta:\varepsilon<\Delta\leq\tau} \left\{ \frac{\gamma^u \odot ((\bar{\alpha}^u(\Delta) - L + b_i) \vee 0))}{\Delta - \varepsilon} \right\}$$

For the computation of $f_{min,i}$, we have to consider that resource-based service curve $\beta^l(\Delta)$ consists of four linear segments:

$$\beta^l(\Delta) = \begin{cases} 0 & \forall\Delta : 0 \leq \Delta \leq \varepsilon \\ f \cdot (\Delta - \varepsilon) & \forall\Delta : \varepsilon < \Delta \leq \tau \\ f \cdot (\tau - \varepsilon) & \forall\Delta : \tau < \Delta \leq \tau + \varepsilon \\ f_{max} \cdot (\Delta - \tau - \varepsilon) & \forall\Delta : \tau + \varepsilon < \Delta \end{cases}$$

Then (6.13) will change to

$$f_{min,i} = \sup_{\forall\Delta\geq\tau+\varepsilon} \left\{ \frac{f_{max}(\tau + \varepsilon - \Delta) + \gamma^u \odot ((\bar{\alpha}^u(\Delta) - L + b_i) \vee 0)}{\tau - \varepsilon} \right\}$$

| Videos | | | | Parameters |
|---|---|---|---|---|
| # | file name | # | file name | MP@ML |
| 1 | bbc3_080.m2v | 4 | susi_080.m2v | 8 Mbps CBR |
| 2 | cact_080.m2v | 5 | tens_080.m2v | 25 fps |
| 3 | mobl_080.m2v | | | 704×576 pixel |
| Source: `ftp.tek.com/tv/test/streams/Element/MPEG-Video/` | | | | |

**Tab. 10:** MPEG-2 video sequences used in the experiments.

We can use similar approach to devise an energy-saving strategy combining DVS and DPM. $\varepsilon$ would then correspond to the maximum time interval during which the processor can be switched off. Also, note that service curves allow for modeling other effects which might be more complex than the time-out due to voltage/frequency transitions or idle intervals in DPM. For example, sharing of a PE by multiple tasks can be included in the analysis. However, these issues go beyond the scope of this thesis.

**Working in discrete time**
All formulas presented up to this point in this section assume computation in *continuous time*, i.e. for $\Delta \in \mathbb{R}_{\geq 0}$. This, however, is impractical. Our method can also work in the discrete time. It is sufficient to compute the clock frequency constraints (6.11), (6.12), and (6.13), only for a few values of $\Delta$. For this, instead of using $\bar{\alpha}^u$ which is defined on $\mathbb{T} = \mathbb{R}_{\geq 0}$, we use its discrete equivalent, defined on $\mathbb{T} = \mathbb{Z}_{\geq 0}$. (3.3) formalizes the discretization procedure. Suppose that $\bar{\alpha}^u_{\mathbb{Z}_{\geq 0}}(k) = \bar{\alpha}^u_{\mathbb{R}_{\geq 0}}(\Delta_k)$, $\Delta_0 = 0$ for $k = 0$, and $\Delta_k < \Delta_{k+1}$ for all $k \geq 0$, then, to be conservative while computing the clock frequency constraints, we have to use $\bar{\alpha}^u_{\mathbb{Z}_{\geq 0}}(k+1)$ in place of $\bar{\alpha}^u_{\mathbb{R}_{\geq 0}}(\Delta_k)$ for all $k \geq 0$. Also, note that the discretization points $\Delta_k$ need not be equally spaced in time, making possible better approximations and optimization of the number of computations.

**Working with discrete frequency levels**
In the above discussion, we assumed that the clock frequency can scale continuously. However, some existing commercial DVS processors have only discrete frequency/voltage operating points [3, 60, 164]. Adapting our method to this sort of architectures is straightforward: at each adaptation point the actual processor clock rate must be set to the smallest possible value that is larger than the computed clock rate $f_i$.

## 6.5 Experimental Results

### 6.5.1 Experimental setup

For the evaluation of our DVS technique, we conducted several experiments using a simulator of the MPEG-2 decoding system shown in Fig. 42 and described in Section 6.3. The simulator consisted of a SystemC [153] transaction-level model in which $\mathrm{PE1}$ and $\mathrm{PE2}$ were modeled using the `sim-profile` configuration of the SimpleScalar ISS [8]. Appendix A provides details about the simulation environment.

Tab. 10 gives the set of MPEG-2 video sequences for which we conducted the experiments. This set includes videos imposing different workload patterns on the architecture. For each video, we collected traces corresponding to functions $x_i(t)$ and $e_i(k)$. Using trace analysis technique described in Section 3.5.2, we obtained *two* curves, $\bar{\alpha}^u$ and $\gamma^u$, representing the *whole* set of videos. For illustration, Fig. 44 depicts the resulting $\bar{\alpha}^u$.

In all experiments, our DVS scheduler performed frequency and voltage adaptations with a constant period $\tau$. We neglected the time and energy overhead associated with the adaptations and assumed that the frequency and voltage of $\mathrm{PE2}$ can change continuously (i.e. at very fine steps).

For the energy consumption estimation, we adopted the model from [104]. According to [104], the energy

$$E \propto \int_0^{n\tau} v_{dd}^2 \cdot f\, dt = \sum_{i=1}^{n} (v_{dd,i})^2 f_i \cdot \tau \propto \sum_{i=1}^{n} (v_{dd,i})^2 f_i \qquad (6.22)$$

where $v_{dd,i}$ and $f_i$ are voltage and frequency values set by the scheduler at the $i$th adaptation point for the $i$th adaptation interval, and $n$ is the total number of adaptation intervals (such that $n\tau$ is the duration of the entire video clip). (6.22) assumes that during idle periods, i.e. when the input buffer is empty and no stream object is being processed, the PE stays active (e.g. checking for the buffer status). However, during these idle periods we could further reduce energy by putting the PE into a low power state. We assume that such a low power state corresponds to the state in which the circuit switching activity is zero, i.e. the PE is switched off completely; hence the energy consumption

$$E \propto \sum_{i=1}^{n} (v_{dd,i})^2 f_i \cdot \tau_i^* \qquad (6.23)$$

where $\tau_i^*$ is the total time during which the PE is *not* switched off within the $i$th adaptation interval. Further, for the estimation of the normalized energy we assumed that $f_i \propto v_{dd,i}$.

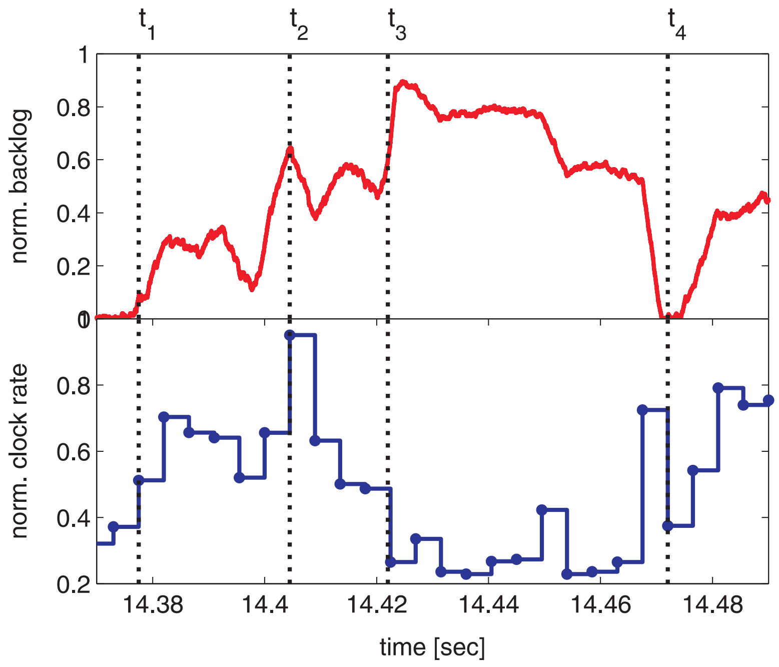


**Fig. 45:** Experimental results. A fragment of a frequency schedule produced by our DVS algorithm (top) and the corresponding to it buffer fill level (bottom).

### 6.5.2 Qualitative examination

Fig. 45 shows a fragment of a frequency schedule produced by our DVS algorithm and the corresponding to it buffer fill level which we observed in one of our experiments. The figure illustrates how the two mechanisms of our method— the run-time monitoring of the buffer fill level and the dynamic VCCs — work together to reduce the energy consumption. Dots on the frequency schedule plot show the adaptation points. By inspecting this plot we can make the following observations.

- Before time $t_1$, the load is low and the backlog is close to zero, hence, the processor runs at a low rate.

- At $t_1$ a burst starts arriving. In response to this burst, the scheduler increases the clock rate, but not by too much: it lets the buffer fill up to some level, and then tries to ”balance” at this level.

- Shortly before $t_2$, the load abruptly increases even further and the buffer fills up very quickly (within one adaptation interval). Therefore, at $t_2$ to avoid a buffer overflow the scheduler increases the processor rate significantly, but only for a short time. In the interval from $t_2$ to $t_3$, the load is still high and the processor runs at a rate which approximately matches this load.

- At $t_3$ the buffer becomes almost full. Despite this, at $t_3$ our DVS algorithm decides to run the processor at a very low rate. *How can this be possible?* The answer is the effect of the dynamic VCCs. At $t_3$ they tell the scheduler that the burst is over and it is safe to run the processor at a low rate because the next burst will not arrive very soon. Consequently, in the interval from $t_3$ to $t_4$, $\mathrm{PE2}$ runs at relatively low rates even though the buffer is nearly full during that interval.

- Shortly before $t_4$, the scheduler again has to increase $\mathrm{PE2}$'s speed. This is because at $t_4$ the dynamic VCCs tell the scheduler that a new burst might start arriving soon and therefore a sufficient space must be cleared in the buffer to accommodate this burst. Indeed, shortly after $t_4$ a new burst starts arriving and the adaptation cycle repeats.

### 6.5.3 Quantitative comparison

We compared the energy savings achieved by our technique with those achieved by the DVS scheme published in Wu et al. [174]. Wu et al. uses a *PID controller* which tracks changes in the buffer fill level and correspondingly regulates processor's speed and voltage. This scheme is similar to ours in a sense that (i) it can handle both the stream burstiness and the data-dependent variability in the task execution demand; (ii) it is suitable for buffer-constrained architectures; and (iii) it also uses fixed adaptation intervals. Furthermore, to the best of our knowledge, at the time of writing, the scheme of Wu et al. represents one of the advanced DVS techniques recently published. Thus, we found it suitable for the comparison. From a user's perspective, the only difference between this scheme and ours is its *unpredictability* in terms of satisfying the specified QoS constraints, i.e. it cannot provide hard QoS guarantees while our scheme can do this. However, an implementation of this scheme might be associated with smaller SW/HW and energy overheads than of our DVS scheme.

We have implemented the PID controller as described in [174]. The adaptation interval lengths of the PID controller and that of our scheme were set to the same value, $\tau = 4.5$ms (which roughly corresponds to nine adaptations per video frame). In our scheme, the arrival history contained $N = 150$ samples with the resolution $\theta = \tau$, whereas the execution demand history contained $M = 200$ samples with the resolution $\psi = 1$.

Fig. 46 shows the results of this comparative study. In this figure, we refer to our scheme as *dVCC* and to the PID controller scheme as *PID*. We simulated both these schemes in two configurations. In one configuration, if the buffer was empty, $\mathrm{PE2}$ continued to run at the rate set at the latest adaptation point (i.e. some cycles were wasted in the idle state). In the other configuration, $\mathrm{PE2}$ was switched off completely for periods when there was nothing to process (i.e. no cycles were wasted). In Fig. 46, the data corresponding to the latter configuration is indicated with dashed lines and with a suffix *_iDPM*. The switching on and off of $\mathrm{PE2}$ was assumed to occur in zero time under the control of an ideal ("oracle") DPM. Although unrealistic, such a configuration is useful for

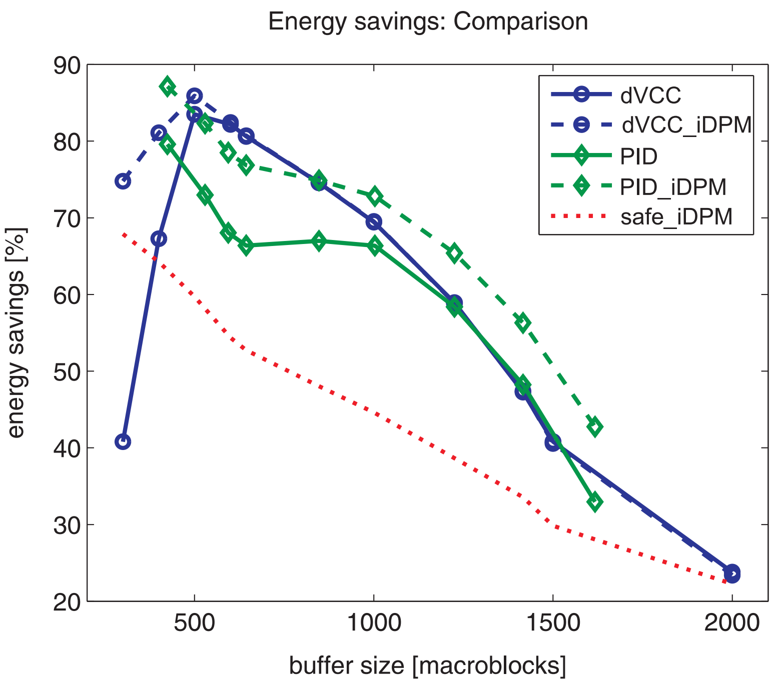


**Fig. 46:** Experimental results. Energy savings achieved by our DVS scheme (denoted as *dVCC* and *dVCC_iDPM*) and the PID-based DVS scheme [174] (denoted as *PID* and *PID_iDPM*).

the analysis because it indicates how well a DVS technique can exploit the slack during low-load periods.

As a baseline for measuring the energy savings we used the energy dissipated by $\mathrm{PE2}$ constantly running at the safe rate $f_{safe}$ computed for a given buffer size $L$ using (3.47). We also measured energy savings obtained by the ”oracle” DPM with instantaneous on–off switching of $\mathrm{PE2}$, running at a fixed $f_{safe}$. In Fig. 46, the corresponding graph is shown with a dotted line and is labelled as *safe_iDPM*.

By inspecting the plots in Fig. 46 we can make the following observations.

1. For relatively small buffer sizes ($L \simeq 400 \ldots 1300$), both DVS schemes, *dVCC* and *PID*, result in more than 50% energy savings. With increasing buffer size the savings decrease. This is mainly because the energy consumption of the baseline architecture rapidly decreases when $L$ grows, since a lower $f_{safe}$ is needed to avoid buffer overflows. If $L$ is sufficiently large to completely average out the workload, *any* DVS scheme is likely to be not worthwhile.

2. In comparison to *safe_iDPM*, for $L \simeq 400 \ldots 1300$ the energy savings of both *dVCC* and *PID* are 10. . . 30%. This indicates that both techniques can effectively exploit the slack during low-load periods. We can also see that for $L > 600$ *dVCC* fully exploits the slack (compare to *dVCC_iDPM*), whereas *PID* potentially could perform better.

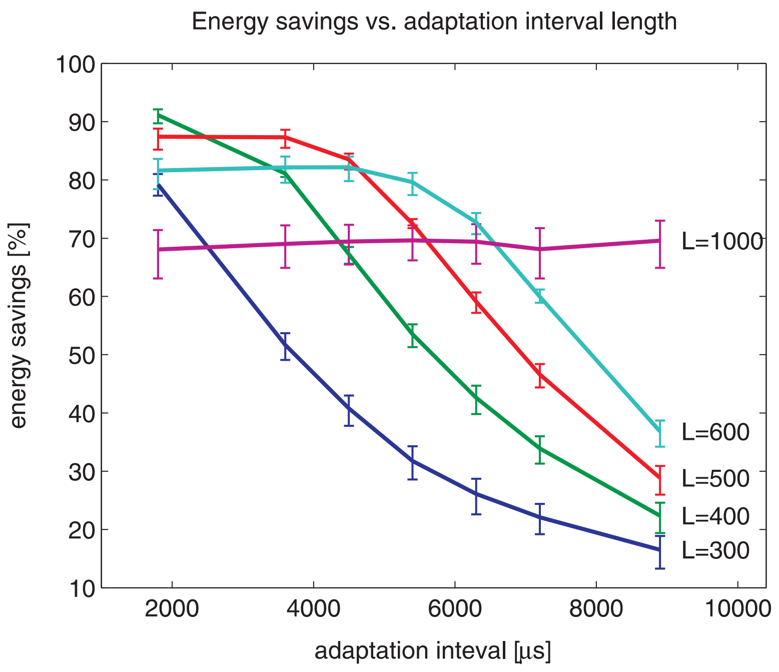


**Fig. 47:** Experimental results. Energy savings vs. adaptation interval length, for different buffer sizes $L$.

3. For $L > 450$ energy savings from *dVCC* are at least as high as from *PID*, while for $L < 450$ *PID* saves 15% more energy than *dVCC*. This is the price for the hard QoS guarantees provided by *dVCC*. If the user specifies $L$ as the buffer constraint, *dVCC* guarantees that the maximum buffer fill level will never exceed $L$. In contrast, *PID* cannot guarantee this. In *PID* the user can only specify a *target*, not the *maximum*, buffer fill level. The PID controller will then try to keep the backlog at this level. Bursty workloads and stringent buffer constraints require the PID controller to quickly respond to workload changes. This leads to (large) oscillations around the target level during adaptations, which means that if the target level has been set improperly a buffer overflow may occur. As an example, the left most point in the *PID* graph was obtained by setting the target buffer fill level to 20 macroblocks, while the maximum backlog registered in the buffer for this setting was about 450 macroblocks. This problem with *PID* might be less acute for smoother workloads and larger buffers.

### 6.5.4 Energy savings vs. implementation overhead

The estimated computational requirement of our DVS algorithm, for parameters $(\tau, N, \psi, M)$ set as described above, (i.e. $\tau = 4.5$ms, $N = 150$, $\psi = 1$, $M = 200$), is about 0.5 MIPS. This overhead scales linearly with the values of these parameters. It is also relatively low in comparison to the average workload imposed on $\mathrm{PE2}$ by a DVD-quality video stream (about 45 MIPS in our setup).

Fig. 47 shows measured tradeoff plots between the adaptation interval length $\tau$ and the energy savings obtained by our scheme for different buffer sizes $L$. In Fig. 47, we can see that smaller adaptation intervals lead to higher energy savings. Achieving the energy reduction for smaller buffer sizes comparable to that achieved for larger buffer sizes necessitates more frequent adaptations and therefore results in higher run-time overhead. This is again the price for the guaranteed QoS.

## 6.6 Summary

In this chapter, we described a new DVS scheduling scheme specifically targeted towards processing media streams on architectures with restricted buffer sizes. In contrast to previously proposed DVS schemes, our scheme provides hard QoS guarantees and accounts for both, the variability of the task execution demand and the burstiness of processed streams. Our experiments showed that the scheme achieves energy savings comparable to those obtained by previous approaches. The advantages of our scheme can be attributed to the novel combination of the off-line worst-case workload characterization based on VCCs with the run-time improvement of the worst-case bounds. The implementation and run-time overhead of our scheme, although modest, might be slightly higher than that of previous schemes. However, this is the price that has to be paid for predictability of the system, i.e. for its ability to provide hard QoS guarantees.

# 7

# Conclusions

In this thesis, we have addressed the problem of workload modeling for system-level design of heterogeneous multiprocessor embedded computers whose main functionality involves real-time processing of media streams. The complex, variable nature of the multimedia workloads imposed on these computers greatly complicates their system-level design. The central result of this thesis is a *workload model* that helps to reduce this design complexity by providing a set of simple but powerful abstractions to accurately capture the variability characteristics of the multimedia workloads. We have formally defined this workload model, demonstrated its advantages over existing workload characterization methods and showed some of its applications in the system-level design of heterogeneous multiprocessor system-on-chip (MpSoC) architectures. In particular, we would like to point out the following main outcomes of this work:

- We have introduced the concept of *Variability Characterization Curves* (VCCs)—a generic model for the worst- and best-case variability characterization of entire classes of increasing functions and sequences — and based on this concept defined our model for the multimedia workload characterization (*multimedia VCCs*).

- We have used our workload model to enhance modeling capabilities of the *Modular Performance Analysis* [159, 160] framework based on *Real-Time Calculus* [24, 121, 157, 158]. This has resulted in the definition of new modeling constructs, called *workload transformations*, which have enabled an accurate and efficient performance analysis of heterogeneous multiprocessor embedded systems under variable multimedia workloads. Our experimental results showed that the workload transformations allow to obtain significantly tighter performance bounds than those achievable without using our workload model.

- We have demonstrated application of our workload model in design of resource management policies for multimedia MpSoCs. Our workload model permits to formulate and address a class of scheduling problems which has not been addressed before: *stream scheduling on buffer-constrained architectures with hard QoS guarantees*. We have proposed a design framework for efficient exploration and optimization of this class of schedulers and showed its utility using case studies involving TDMA scheduling disciplines.

- Based on our workload model we have developed a *run-time processor rate adaptation strategy* which can be used in conjunction with the dynamic voltage scaling for energy-efficient media stream processing on buffer-constrained architectures. In comparison to other methods addressing similar problems, our scheme can handle multimedia workloads characterized by both, the data-dependent variability in the task execution time *and* the burstiness in the on-chip traffic arising out of multimedia processing; and at the same time it can provide hard QoS guarantees. An experimental evaluation showed that our scheme can achieve considerable energy savings, comparable to those obtainable with another state-of-the-art DVS scheme (which is, however, unable to provide hard QoS guarantees).

- Finally, we have demonstrated how our workload model can be used to design *representative workload scenarios* for simulation-based system-level performance evaluation of multimedia MpSoCs. We have proposed a method for (automatic) classification of media streams based on the variability characteristics of the workload they may impose on the architecture. We are not aware of any other method addressing this important issue in the system-level design of MpSoCs.

# A

# Simulation Framework

This chapter describes the simulation framework which was used in experimental case studies presented throughout this thesis.

Fig. 48 shows an overview of the simulation framework and its relations with the methods developed in this thesis. Two major components of the simulation framework are

- an *instruction set simulator* (ISS) and
- a *system simulator*.

ISS is used to collect information about behavior of application tasks mapped for execution on to programmable processing elements of a target MpSoC architecture. This information is collected and stored in the form of *traces*. These traces are then used in two different ways. First, we can compute from them VCCs, which are needed by the VCC-based methods presented in this thesis. Second, the traces serve as an input to the system simulator to simulate execution of the application on the target MpSoC architecture.

The system simulator allows to measure various performance indexes of a given application-to-architecture mapping (system configuration). These measurements can then be compared with corresponding numbers obtained from mathematical performance models.

The following two sections give details on ISS and the system simulator.

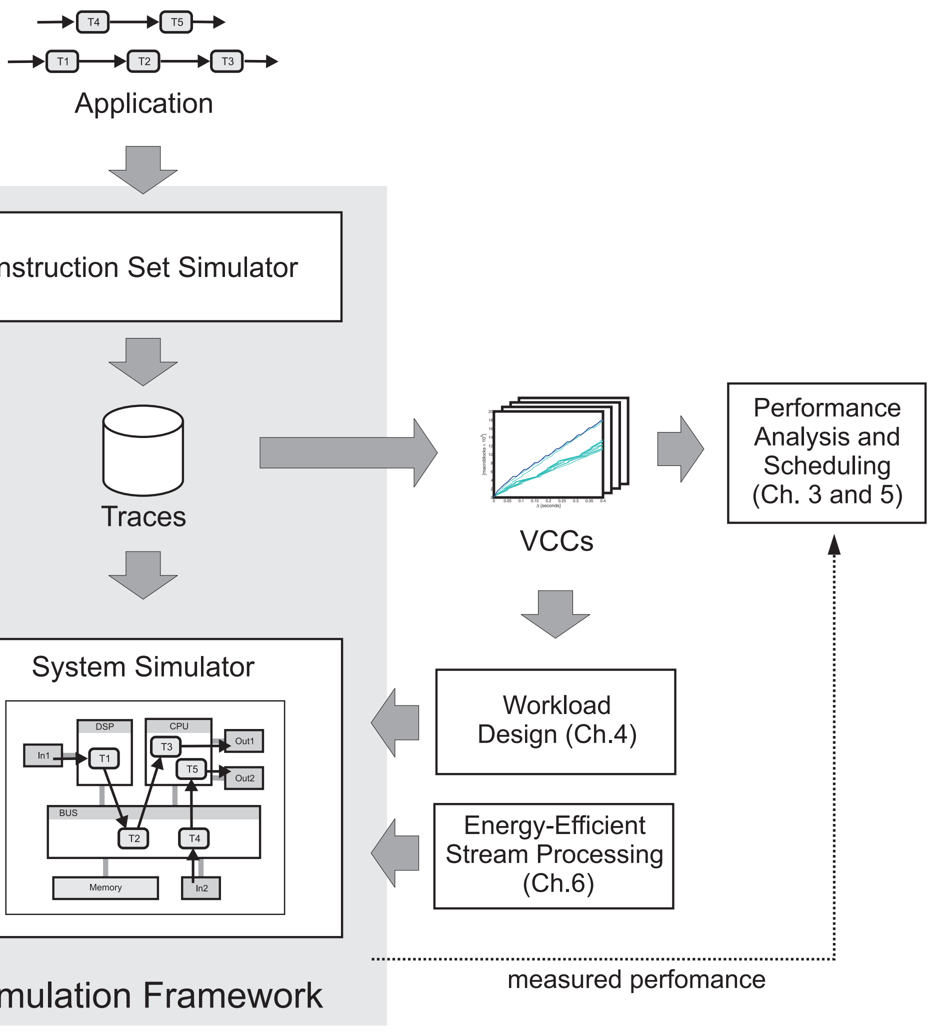


**Fig. 48:** Simulation framework and its relations with methods developed in this thesis.

## A.1 Instruction set simulator

We used the SimpleScalar ISS [8] to model programmable processing elements within a target MpSoC architecture. On its own, the SimpleScalar ISS does not support simulation of heterogeneous multiprocessor architectures [8]. However, it can be used to model with sufficient accuracy execution of individual software tasks on a given processor type. SimpleScalar supports several instruction set architectures and permits modeling of various microarchitectural features (e.g. pipelines, caches, branch predictors, etc.) Furthermore, it is extensible in a sense that users can easily customize simulation models of the processors by introducing new instruction types and by adding new or configuring already existing microarchitectural features.

For experimental case studies presented in this thesis, we employed the *sim-profile* configuration of the SimpleScalar. This configuration assumes that execution of every instruction in a program code takes exactly one processor cycle, i.e. it does not account for possible stalls due to, for example, pipeline hazards or cache misses. Although the *sim-profile* configuration might be too inaccurate for the microarchitecture design, for the system level at which we considered the MpSoC architectures, it represents an appropriate choice.

For modeling processing elements we used an instruction set similar to that used in MIPS3000 processors, but without floating point support. This instruction set had application-specific extensions for video decoding, e.g. instructions for bitstream access, IDCT computation and special block-based memory addressing modes. To collect useful information about the data-dependent behavior of application tasks, we had to augment the SimpleScalar with logging and trace collection facilities.

## A.2 System simulator

Our simulation framework allows to easily construct a simulation model of a given application-to-architecture mapping. We refer to such a simulation model as the system simulator. The system simulator represents a *transaction-level model* [34] of the system in which implementation details of individual hardware components and application tasks are abstracted away. Instead, the hardware architecture is represented by a set of (coarse-grain) computation and communication resources with certain processing capabilities, while the application tasks are modeled as clients requesting these resources (i.e. generating demands for these resources). At this level of abstraction, there is no principle difference between computation and communication resources (or tasks).

An application is represented by a set of concurrent tasks communicating via unidirectional FIFO channels. The execution model is the same as the one described in Section 3.3.1 besides the fact that the FIFO channels have finite sizes and block write operations whenever they are full. The finite channel sizes allow to model the buffer space constraints resulting from mapping of the application onto the hardware architecture. A write operation into a FIFO channel is regarded as a transaction. A task may accomplish several such transactions during its execution.

When a task is activated it generates a request for the resource onto which it is mapped. In the current implementation, the amount of resources requested is either fixed to a constant value or obtained from the traces collected using the ISS described in the previous section. A resource is distributed to tasks in accordance with the scheduling (or arbitration) policy implemented on that resource. The framework supports both preemptive and non-preemptive scheduling policies. In principle, task switching and operating system overheads can be modeled within the framework but were not modeled in the experiments presented in this thesis.

The implementation of the system simulator just described fully rests on the

SystemC C++ library [153] and correspondingly uses of its discrete event simulation engine. We have added an abstraction level on top of the SystemC library which allows to quickly construct a new application-to-architecture mapping or to easily change an existing system configuration.

# Bibliography

[1] L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, 2004.

[2] A. Agarwal, C. H. Kim, S. Mukhopadhyay, and K. Roy. Leakage in nano-scale technologies: mechanisms, impact and design considerations. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, pages 6–11, New York, NY, USA, 2004. ACM Press.

[3] Advanced Micro Devices, Inc. AMD PowerNow! technology. `http://www.amd.com`, 2005.

[4] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. M. Al-Hashimi. Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In *Design, Automation and Test in Europe (DATE)*, page 10518, Washington, DC, USA, 2004. IEEE Computer Society.

[5] A. Andrei, M. T. Schmitz, P. Eles, Z. Peng, and B. M. A. Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *Design, Automation and Test in Europe (DATE)*, pages 514–519, Washington, DC, USA, 2005. IEEE Computer Society.

[6] A. Artieri, V. DAlto, R. Chesson, M. Hopkins, and M. C. Rossi. Nomadik open multimedia platform for next-generation mobile devices. STMicroelectronics Technical Article TA305, `http://www.st.com`, 2003.

[7] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, page 123, Washington, DC, USA, 1998. IEEE Computer Society.

[8] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[9] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Design, Automation and Test in Europe (DATE)*, page 168, Washington, DC, USA, 2002. IEEE Computer Society.

[10] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. John Wiley, Sons, New York, 1992.

[11] S. K. Baruah. A general model for recurring real-time tasks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, pages 114–122, 1998.

[12] S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1):93–128, 2003.

[13] S. K. Baruah, D. Chen, S. Gorinsky, and A. K. Mok. Generalized multi-frame tasks. *Real-Time Systems*, 17(1):5–22, 1999.

[14] A. C. Bavier, A. B. Montz, and L. L. Peterson. Predicting MPEG execution times. In *SIGMETRICS*, pages 131–140, 1998.

[15] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC cosimulation and emulation of multiprocessor SoC designs. *IEEE Computer*, 36(4):53–59, 2003.

[16] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, June 2000.

[17] L. Benini and G. D. Micheli. Powering networks on chips: energy-efficient and reliable interconnect design for SoCs. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS)*, pages 33–38, New York, NY, USA, 2001. ACM Press.

[18] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, 1999.

[19] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[20] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[21] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In *HICSS (1)*, pages 288–297, 1995.

[22] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag Telos, 2004.

[23] S. Chakraborty. *System-Level Timing Analysis and Scheduling for Embedded Packet Processors*. PhD thesis, ETH Zurich, Apr. 2003.

[24] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, Mar. 2003. IEEE Press.

[25] S. Chakraborty, S. Künzli, L. Thiele, A. Herkersdorf, and P. Sagmeister. Performance evaluation of network processor architectures: Combining simulation with analytical estimation. *Computer Networks*, 41(5):641—-665, 2003.

[26] S. Chakraborty and L. Thiele. A new task model for streaming applications and its schedulability analysis. In *Design, Automation and Test in Europe (DATE)*, pages 486–491, 2005.

[27] A. P. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[28] L. H. Chandrasena, P. Chandrasena, and M. J. Liebelt. An energy efficient rate selection algorithm for voltage quantized dynamic voltage scaling. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS)*, pages 124–129, New York, NY, USA, 2001. ACM Press.

[29] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 732–737, 2002.

[30] K. Choi, R. Soma, and M. Pedram. Off-chip latency-driven dynamic voltage and frequency scaling for an MPEG decoding. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, pages 544–549, 2004.

[31] W. J. Dally and B. Towles. Route packets, not wires: on-chip inteconnectoin networks. In *Proceedings of the 38th Conference on Design Automation (DAC)*, pages 684–689, New York, NY, USA, 2001. ACM Press.

[32] B. P. Dave and N. K. Jha. CASPER: Concurrent hardware-software co-synthesis of hard real-time aperiodic and periodic specifications of embedded system architectures. In *Design, Automation and Test in Europe (DATE)*, pages 118–124, 1998.

[33] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *Proceedings of the 34th Conference on Design Automation (DAC)*, pages 703–708, 1997.

[34] A. K. Deb, A. Jantsch, and J. Öberg. System design for dsp applications in transaction level modeling paradigm. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, pages 466–471. ACM Press, 2004.

[35] R. P. Dick and N. K. Jha. MOCSYN: Multiobjective core-based single-chip system synthesis. In *Design, Automation and Test in Europe (DATE)*, pages 263–270, 1999.

[36] K. Diefendorff and P. K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997.

[37] R. C. Dorf and R. H. Bishop. *Modern Control Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[38] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design & Test of Computers*, 18(5):21–31, 2001.

[39] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 83–94. IEEE Computer Society, 2002.

[40] H. El-Rewini, H. H. Ali, and T. G. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37, 1995.

[41] C. J. Fidge. Real-time schedulability tests for preemptive multitasking. In *Selected papers from the 4th Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, pages 61–93, Norwell, MA, USA, 1998. Kluwer Academic Publishers.

[42] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. *Wireless Networks*, 8(5):507–520, 2002.

[43] S. Goddard and K. Jeffay. Managing latency and buffer requirements in processing graph chains. *The Computer Journal*, 44(6):486–503, 2001.

[44] K. Goossens, J. Dielissen, J. van Meerbergen, P. Poplavko, A. Rădulescu, E. Rijpkema, E. Waterlander, and P. Wielage. Guaranteeing the quality of services in networks on chip. In A. Jantsch and H. Tenhunen, editors, *Networks on Chip*, pages 61–82. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[45] A. D. Gordon. *Classification*. Chapman & Hall/CRC, 1999.

[46] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Annual Intern. Conf. on Mobile Computing and Networking*, pages 13–25. ACM Press, 1995.

[47] F. Gruian and K. Kuchcinski. LEneS: task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the 2001 Conference on Asia South Pacific Design Automation (ASP-DAC)*, pages 449–455, 2001.

[48] J. C. P. Gutiérrez and M. G. Harbour. Offset-based response time analysis of distributed systems scheduled under EDF. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–12, 2003.

[49] V. Gutnik and A. P. Chandrakasan. Embedded power supply for low-power DSP. *IEEE Transactions on VLSI Systems*, 5(4):425–435, 1997.

[50] S. Ha and E. A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, 1997.

[51] A. Hamann and R. Ernst. TDMA time slot and turn optimization with evolutionary search techniques. In *Design, Automation and Test in Europe (DATE)*, pages 312–317, Washington, DC, USA, 2005. IEEE Computer Society.

[52] F. Harmsze, A. H. Timmer, and J. L. van Meerbergen. Memory arbitration and cache management in stream-based systems. In *Design, Automation and Test in Europe (DATE)*, pages 257–262, 2000.

[53] J. Helmig. Developing core software technologies for TI's OMAP platform. Texas Instruments, `http://www.ti.com`, 2002.

[54] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *Proceedings of the 35th Annual Conference on Design Automation (DAC)*, pages 176–181, New York, NY, USA, 1998. ACM Press.

[55] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 653–656, New York, NY, USA, 1998. ACM Press.

[56] S. Hua, G. Qu, and S. S. Bhattacharyya. Energy reduction techniques for multimedia applications with tolerance to deadline misses. In *Proceedings of the 40th Conference on Design Automation (DAC)*, pages 131–136, New York, NY, USA, 2003. ACM Press.

[57] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 254–265. ACM Press, 2001.

[58] IBM PowerPC. `http://www.chips.ibm.com/products/powerpc/`, 2005.

[59] C. Im, S. Ha, and H. Kim. Dynamic voltage scheduling with buffers in low-power multimedia applications. *Transactions on Embedded Computing Systems*, 3(4):686–705, 2004.

[60] Intel Corporation, Enhanced Intel SpeedStep technology. `http://www.intel.com`, 2005.

[61] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 197–202, New York, NY, USA, 1998. ACM Press.

[62] S. Ishiwata, T. Yamakage, Y. Tsuboi, T. Shimazawa, T. Kitazawa, S. Michinaka, K. Yahagi, H. Takeda, A. Oue, T. Kodama, N. Matsumoto, T. Kamei, M. Saito, T. Miyamori, G. Ootomo, and M. Matsui. A single-chip MPEG-2 codec based on customizable media embedded processor. *IEEE Journal of Solid-State Circuits*, 38(3):530–540, 2003.

[63] H. Iwasaki, J. Naganuma, K. Nitta, K. Nakamura, T. Yoshitome, M. Ogura, Y. Nakajima, Y. Tashiro, T. Onishi, M. Ikeda, and M. Endo. Single-chip MPEG-2 422P@HL CODEC LSI with multi-chip configuration for large scale processing beyond HDTV level. In *Design, Automation and Test in Europe (DATE)*, pages 20002–20007, 2003.

[64] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 158–168, Washington, DC, USA, 2002. IEEE Computer Society.

[65] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 304–314, 1999.

[66] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, pages 275–280, New York, NY, USA, 2004. ACM Press.

[67] M. Jersak and R. Ernst. Enabling scheduling analysis of heterogeneous systems with multi-rate data dependencies and rate intervals. In *Proceedings of the 40th conference on Design automation (DAC)*, pages 454–459, New York, NY, USA, 2003. ACM Press.

[68] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[69] N. K. Jha. Low power system scheduling and synthesis. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 259–263, Piscataway, NJ, USA, 2001. IEEE Press.

[70] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[71] A. Kalavade and P. Moghé. A tool for performance estimation of networked embedded end-systems. In *Proceedings of the 35th Conference on Design Automation Conference (DAC)*, pages 257–262. ACM/IEEE, June 1998.

[72] K. Keutzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12), 2000.

[73] C. Kim and K. Roy. Dynamic Vth scaling scheme for active leakage power reduction. In *Design, Automation and Test in Europe (DATE)*, page 163, Washington, DC, USA, 2002. IEEE Computer Society.

[74] S. Kim, C. Im, and S. Ha. Schedule-aware performance estimation of communication architecture for efficient design space exploration. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 195–200, New York, NY, USA, 2003. ACM Press.

[75] S. Kim, C. Im, and S. Ha. Efficient exploration of on-chip bus architectures and memory allocation. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 248–253, New York, NY, USA, 2004. ACM Press.

[76] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Boston Kluwer Academic Publishers, 1997.

[77] C. E. Kozyrakis and D. A. Patterson. A new direction for computer architecture research. *Computer*, 31(11):24–32, 1998.

[78] M. Krunz and S. K. Tripathi. On the characterization of VBR MPEG streams. *SIGMETRICS Perform. Eval. Rev.*, 25(1):192–202, 1997.

[79] S. Kumar. On packet switched networks for on-chip communication. In A. Jantsch and H. Tenhunen, editors, *Networks on Chip*, pages 85–106. Kluwer Academic Publishers, Hingham, MA, USA, 2003.

[80] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[81] W.-C. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *Transactions on Embedded Computing Systems*, 4(1):211–230, 2005.

[82] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. In *Workload characterization of emerging computer applications*, pages 145–163. Kluwer Academic Publishers, 2001.

[83] K. Lahiri, A. Raghunathan, and S. Dey. System level performance analysis for designing on-chip communication architectures. *IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems*, 20(6):768–783, 2001.

[84] A. A. Lazar, G. Pacifici, and D. E. Pendarakis. Modeling video sources for real-time scheduling. *Multimedia Systems*, 1(6):253–266, 1994.

[85] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the Internet*. Springer-Verlag New York, Inc., 2001.

[86] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[87] C.-H. Lee and K. G. Shin. On-line dynamic voltage scaling for hard real-time systems using the EDF algorithm. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, pages 319–327, Washington, DC, USA, 2004. IEEE Computer Society.

[88] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[89] E. A. Lee and T. M. Parks. Dataflow process networks. In *Readings in hardware/software co-design*, pages 59–85. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[90] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Conference on Design Automation (DAC)*, pages 806–809. ACM Press, 2000.

[91] S. H. Lee, K.-Y. Whang, Y.-S. Moon, and I.-Y. Song. Dynamic buffer allocation in video-on-demand systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 343–354. ACM Press, 2001.

[92] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium (RTSS)*, pages 201–213. IEEE, 1990.

[93] J. P. Lehoczky. Real-time queueing theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS)*, page 186, Washington, DC, USA, 1996. IEEE Computer Society.

[94] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[95] S. Leibson and J. Kim. Configurable processors: A new era in chip design. *IEEE Computer*, 38(7):51–59, 2005.

[96] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[97] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[98] Y. Li and W. Wolf. A task-level hierarchical memory model for system synthesis of multiprocessors. In *Proceedings of the 34th Conference on Design Automation (DAC)*, pages 153–156, 1997.

[99] P. Lieverse, T. Stefanov, P. van der Wolf, and E. F. Deprettere. System level design with Spade: an M-JPEG case study. In *2001 International Conference on Computer-Aided Design (ICCAD)*, pages 31–38, 2001.

[100] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[101] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[102] Y. Liu, A. Maxiaguine, S. Chakraborty, and W. T. Ooi. Processor frequency selection for SoC platforms for multimedia applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, pages 336–345, Lisbon, Portugal, Dec. 2004. IEEE Computer Society.

[103] Y. Liu, A. Maxiaguine, S. Chakraborty, and W. T. Ooi. Processor frequency selection in energy-aware SoC platform design for multimedia application. Technical Report TRC8/04, National University of Singapore, Nov. 2004.

[104] Y.-H. Lu, L. Benini, and G. D. Micheli. Dynamic frequency scaling with buffer insertion for mixed workloads. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1284–1305, November 2002.

[105] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 156–163, New York, NY, USA, 2002. ACM Press.

[106] Z. Lu, J. Lach, M. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *Proceedings of the 21st International Conference on Computer Design (ICCD)*, page 489. IEEE Computer Society, 2003.

[107] J. Luo and N. K. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 357–364, Piscataway, NJ, USA, 2000. IEEE Press.

[108] J. Madsen and P. Bjørn-Jørgensen. Embedded system synthesis under memory constraints. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES)*, pages 188–192, 1999.

[109] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *Transactions on Embedded Computing Systems*, 3(4):706–735, 2004.

[110] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 721–725, New York, NY, USA, 2002. ACM Press.

[111] A. Maxiaguine, S. Chakraborty, S. Künzli, and L. Thiele. Evaluating schedulers for multimedia processing on buffer-constrained SoC platforms. *IEEE Design & Test*, 21(5):368–377, Sept. 2004.

[112] A. Maxiaguine, S. Künzli, S. Chakraborty, and L. Thiele. Rate analysis for streaming applications with on-chip buffer constraints. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 131–136, Yokohama, Japan, Jan. 2004.

[113] A. Maxiaguine, S. Künzli, and L. Thiele. Workload characterization model for tasks with variable execution demand. In *Design, Automation and Test in Europe (DATE)*, pages 1040–1045, Paris, France, Feb. 2004. IEEE Computer Society.

[114] A. Maxiaguine, Y. Liu, S. Chakraborty, and W. T. Ooi. Identifying ”representative” workloads in designing MpSoC platforms for media processing. In *Proceedings of the 2nd Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)*, pages 41–46. IEEE, 2004.

[115] A. Maxiaguine, Y. Zhu, S. Chakraborty, and W.-F. Wong. Tuning SoC platforms for multimedia processing: identifying limits and tradeoffs. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis (CODES+ISSS)*, pages 128–133. ACM Press, 2004.

[116] T. Meyerowitz, C. Pinello, and A. Sangiovanni-Vincentelli. A tool for describing and evaluating hierarchical real-time bus scheduling policies. In *Proceedings of the 40th Conference on Design Automation (DAC)*, pages 312–317, New York, NY, USA, 2003. ACM Press.

[117] A. K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

[118] International Standard Organization, ”Information Technology – Generic Coding of Moving Pictures and Associated Audio Information – Part 2: Video,” ISO/IEC 13818-2.

[119] MPEG Software Simulation Group. `http://www.mpeg.org`, 2005.

[120] M. Naedele. *On the Modeling and Evaluation of Real-Time Systems*. PhD thesis, ETH Zurich, Mar. 2000.

[121] M. Naedele, L. Thiele, and M. Eisenring. Characterising variable task releases and processor capacities. In *Proceedings of the 14th IFAC World Congress 1999*, Beijing, July 1999.

[122] L. S. Nielsen and C. Niessen. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, 1994.

[123] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. J. IV, D. Franklin, V. Akella, and F. T. Chong. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, page 150, Washington, DC, USA, 2004. IEEE Computer Society.

[124] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 76–81, New York, NY, USA, 1998. ACM Press.

[125] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere. Exploring embedded-systems architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001.

[126] A. D. Pimentel, S. Polstra, F. Terpstra, A. W. van Halderen, J. E. Coffland, and L. O. Hertzberger. Towards efficient design space exploration of heterogeneous embedded media systems. In *Embedded Processor Design Challenges*, pages 57–73, 2002.

[127] P. Pop, P. Eles, and Z. Peng. Performance estimation for embedded systems with data and control dependencies. In *Proceedings of the 8th International Workshop on Hardware/Software Co-Design (CODES)*, pages 62–66, 2000.

[128] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. In *Design, Automation and Test in Europe (DATE)*, pages 10184–10189, 2003.

[129] P. Pop, P. Eles, and Z. Peng. Schedulability-driven communication synthesis for time triggered embedded systems. *Real-Time Systems*, 26(3):297–325, 2004.

[130] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES)*, pages 187–192, 2002.

[131] S. Prakash and A. C. Parker. Synthesis of application-specific multi-processor systems including memory components. *Journal of VLSI Signal Processing Systems*, 8(2):97–116, 1994.

[132] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, 1995.

[133] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4), 2003.

[134] M. J. Rutten, J. T. J. van Eijndhoven, E. G. T. Jaspers, P. van der Wolf, E.-J. D. Pol, O. P. Gangwal, and A. Timmer. A heterogeneous multi-processor architecture for flexible media processing. *IEEE Design & Test of Computers*, 19(4):39–50, 2002.

[135] M. J. Rutten, J. T. J. van Eijndhoven, and E.-J. D. Pol. Robust media processing in a flexible and cost-effective network of multi-tasking co-processors. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, page 223, Washington, DC, USA, 2002. IEEE Computer Society.

[136] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publishers, Boston, 2004.

[137] K. Sekar, K. Lahiri, and S. Dey. Dynamic platform management for configurable platform-based system-on-chips. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 641–649, 2003.

[138] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*, pages 356–367, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[139] J. Seo, T. Kim, and K.-S. Chung. Profile-based optimal intra-task voltage scheduling for hard real-time applications. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, pages 87–92, New York, NY, USA, 2004. ACM Press.

[140] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[141] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th Conference on Design Automation (DAC)*, pages 438–443, New York, NY, USA, 2001. ACM Press.

[142] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC)*, pages 134–139. ACM Press, 1999.

[143] T. Simunic, L. Benini, A. Acquaviva, P. W. Glynn, and G. D. Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th Conference on Design Automation (DAC)*, pages 524–529, 2001.

[144] T. Simunic, S. P. Boyd, and P. Glynn. Managing power consumption in networks on chips. *IEEE Transactions on VLSI Systems*, 12(1):96–107, 2004.

[145] A. Sinha and A. P. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In *Proceedings of the 14th International Conference on VLSI Design (VLSID)*, page 221, Washington, DC, USA, 2001. IEEE Computer Society.

[146] N. T. Slingerland and A. J. Smith. Design and characterization of the Berkeley multimedia workload. *Multimedia Systems*, 8(4):315–327, 2002.

[147] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[148] K. Sreenivasan and A. J. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, 1974.

[149] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., New York, NY, USA, 2000.

[150] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[151] M. T. J. Strik, A. H. Timmer, J. L. van Meerbergen, and G.-J. van Rootselaar. Heterogeneous multiprocessor for the management of real-time video and graphics streams. *IEEE Journal of Solid-State Circuits*, 35(11):1722–1731, 2000.

[152] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[153] Open SystemC Initiative. `http://www.systemc.org`, 2002.

[154] R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES)*, pages 147–152, 2001.

[155] D. Talla, C.-Y. Hung, R. Talluri, F. Brill, D. Smith, D. Brier, B. Xiong, and D. Huynh. Anatomy of a portable digital mediaprocessor. *IEEE Micro*, 24(2):32–39, 2004.

[156] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proceedings of the 39th Design Automation Conference (DAC)*, pages 880–885, New Orleans LA, USA, June 2002. ACM Press.

[157] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert. Embedded software in network processors - models and algorithms. In *Proceedings of the 1st International Workshop on Embedded Software (EMSOFT)*, pages 416–434, London, UK, 2001. Springer-Verlag.

[158] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, Geneva, Switzerland, Mar. 2000.

[159] L. Thiele and E. Wandeler. Performance analysis of embedded systems. In *The Embedded Systems Handbook*. CRC Press, 2004.

[160] L. Thiele, E. Wandeler, and S. Chakraborty. A stream-oriented component model for performance analysis of multiprocessor DSPs. *IEEE Signal Processing Magazine, special Issue on Hardware/Software Co-design for DSP*, 22(3):38—-46, May 2005.

[161] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, and J. W.-S. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 164 – 173. IEEE Computer Society, 1995.

[162] K. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[163] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 40(2-3):117–134, 1994.

[164] Transmeta Corporation, LongRun technology. `http://www.transmeta.com`, 2005.

[165] G. Varatkar and R. Marculescu. On-chip traffic modeling and synthesis for MPEG-2 video applications. *IEEE Transactions on VLSI Systems*, 12(1), January 2004.

[166] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 450–459, Toronto, Canada, May 2004.

[167] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. *Journal of Real-time Systems*, page to appear, Mar. 2005.

[168] E. Wandeler and L. Thiele. Abstracting functionality for modular performance analysis of hard real-time systems. In *Proceedings of the Asia and South Pacific Desing Automation Conference (ASP-DAC)*, pages 697—-702, Shanghai, P.R. China, Jan. 2005.

[169] E. Wandeler and L. Thiele. Characterizing workload correlations in multi processor hard real-time systems. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, USA, Mar. 2005.

[170] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *OSDI*, pages 13–23, 1994.

[171] F. Wolf. *Behavioral Intervals in Embedded Software: Timing and Power Analysis of Embedded Real-Time Software Processes*. Kluwer Academic Publishers, 2002.

[172] W. Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41th Conference on Design Automation (DAC)*, pages 681–685, 2004.

[173] W. Wolf. Multimedia applications of multiprocessor systems-on-chips. In *Design, Automation and Test in Europe (DATE)*, pages 86–89, 2005.

[174] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of ASPLOS-XI*, pages 248–259, New York, NY, USA, 2004. ACM Press.

[175] L. Yan, J. Luo, and N. K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 30, Washington, DC, USA, 2003. IEEE Computer Society.

[176] C.-Y. Yang, J.-J. Chen, and T.-W. Kuo. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *Design, Automation and Test in Europe (DATE)*, pages 468–473, Washington, DC, USA, 2005. IEEE Computer Society.

[177] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, page 374, Washington, DC, USA, 1995. IEEE Computer Society.

[178] W. Yuan and K. Nahrstedt. Integration of dynamic voltage scaling and soft real-time scheduling for open mobile systems. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 105–114, New York, NY, USA, 2002. ACM Press.

[179] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1396, 1995.

[180] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, page 84, Washington, DC, USA, 2001. IEEE Computer Society.

[181] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings of the 6th International Workshop on Hardware/software Codesign (CODES/CASHE)*, pages 9–13, Washington, DC, USA, 1998. IEEE Computer Society.

[182] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 54–61, 1998.

[183] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI – A system model for heterogeneously specified embedded systems. *IEEE Transactions on VLSI Systems*, 10(4):397 – 389, August 2002.

[184] V. D. Zivkovic, E. A. de Kock, P. van der Wolf, and E. F. Deprettere. Fast and accurate multiprocessor architecture exploration with symbolic programs. In *Design, Automation and Test in Europe (DATE)*, pages 10656–10661, 2003.