# Predictable Component-Based Software Design of Real-Time MPEG-4 Video Applications

Egor Bondarev[a], Milan Pastrnak[a,b], Peter H.N. de With[a,b] and Michel R. V. Chaudron[a]

[a]Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
[b]LogicaCMG Nederland, P.O. Box 7089, 5605 JB, Eindhoven, The Netherlands

## ABSTRACT

Component-based software development is very attractive, because it allows a clear decomposition of logical processing blocks into software blocks and it offers wide reuse. The strong real-time requirements of media processing systems should be validated as soon as possible to avoid costly system redesign. This can be achieved by prediction of timing and performance properties. In this paper, we propose a scenario simulation design approach featuring early performance prediction of a component-based software system. We validated this approach through a case study, for which we developed an advanced MPEG-4 coding application. The benefits of the approach are threefold: (a) high accuracy of the predicted performance data; (b) it delivers an efficient real-time software-hardware implementation, because the generic computational costs become known in advance, and (c) improved ease of use because of a high abstraction level of modelling. Experiments showed that the prediction accuracy of the system performance is about 90% or higher, while the prediction accuracy of the time-detailed processor usage (performance) does not get lower than 70%. However, the real-time performance requirements are sometimes not met, e.g. when other applications require intensive memory usage, thereby imposing delays on the retrieval from memory of the decoder data.

**Keywords:** Real-time system, performance prediction, MPEG-4 coding, component-based software, predictable design

## 1. INTRODUCTION

Current multimedia systems industry has to face extremely high time-to-market pressure. A major problem is that system development times often exceed the technology innovation cycle due to elaborate and complex software architectures and implementations. Once the product becomes available for market introduction, it may be completely outdated because of the emergence of next generation products. Therefore, the focus of industrial development is shifting towards improved system design techniques, because they allow evaluation of the system functionality and performance already at very early development phases, which reduces technical risks and time-to-market.

Advanced design techniques are imperative especially for time-critical software-intensive media systems. The real-time requirements imposed on these systems, such as frame skipping and latency limitations, can only be validated after system implementation. To avoid system redesign for ensuring performance properties, we concentrate on the accurate *prediction* of the extra-functional properties at the early design phase. Since MPEG video processing is increasingly software-based and involves intensive processing, an efficient real-time software-based implementation of this technology is of primary importance.

Besides performance prediction, the reuse of already existing software blocks potentially saves costs. Contemporary media processing systems consist of a set of logically identical blocks (Pixel Sampler, DCT module, Quantizer, VLC module, etc). Not seldom, during the development of a new system, the blocks are implemented from scratch. Instead, reusing these blocks could save a significant development time. A component-based software paradigm[1] addresses these issues, enabling decomposition of functionality into components and wide reuse of these components for various systems. However, note that component-based technology complicates the prediction of resource usage and timing properties. In component-based systems, the actual behaviour and resource

Further author information:
Egor Bondarev: E-mail: e.bondarev@tue.nl, Telephone: +31 40 247 2480

usage are determined by an ensemble of *internally* and also *externally* developed components. The external components are black-boxes of which structure and behaviour are usually not known in detail. Consequently, we focus on the accurate prediction of extra-functional properties of component-based systems at the early design phase.

Performance prediction has been extensively addressed in system architecture research. The survey in[2] gives an overview on the current prediction-enabling techniques for design. The modular performance analysis described in[3] allows exploring software and hardware alternatives for performance optimization, though it does not give high prediction accuracy. The "Spade" technique[4] provides an example of a simulation-based performance analysis methodology for signal processing systems. It addresses the co-design of software and hardware, but still does not enable prediction of real-time properties. In[5] and,[6] we find efficient performance prediction techniques in the component-based software domain, which are based on component modelling and static scheduling analysis. However, for complex applications, the models become too large for understanding and efficient use in analysis. Last but not least, the PRIMA-UML methodology[7] applies queuing networks and extends UML with a model for system performance validation. For our problem statement we want to satisfy four aspects simultaneously: (1) real-time properties prediction, (2) high accuracy, (3) ease of use of the design methodology and (4) applicability to component-based architectures. None of the above-mentioned proposals satisfies these four aspects at the same time.

In this paper, we propose an easy-to-use scenario simulation approach for accurately predicting the timing and resource usage properties of the designed component-based system. We validated the technique by a case study developing an advanced MPEG-4 video application. The proposed approach is based on three concepts: (a) *models* for the system component's behaviour and resource usage, (b) *execution scenarios* of the complete system, in which the resources are potentially overloaded, (c) *simulation* of these scenarios, resulting in timing behaviour of the design system. The modelling enables a high-level description, thus making this prediction technique easy to use. Additionally, the application of the execution scenarios reduces the system state space for exploration, and helps to focus only on the relevant time-critical execution configurations, in which the risks of system overload are high. As a design paradigm, we have adopted component-based software architecture, because of its structural flexibility for large complex applications and the enabling possibilities for software reuse. The MPEG-4 decoder case study presented later in this paper, revealed that it is indeed possible to predict real-time properties with sufficiently high accuracy. The case study also uncovered system management issues, such as influence of the operating system and memory management aspects.

The remainder of this paper is as follows. In Section 2, we address a so-called Robocop software component-based architecture that was deployed for the MPEG-4 coder case study. Section 3 presents the proposed scenario simulation approach enabling performance predictions at the design phase. In Section 4, we describe the MPEG-4 case study in more detail, which was used for validation of the approach. The validation results are provided in Section 5. Finally, Section 6 presents conclusions from the case study, and suggests future directions for research.

## 2. COMPONENT-BASED SOFTWARE ARCHITECTURE

Within the international Space4U project*, we have developed a Robocop Component-Based Architecture (CBA),[8] which was adopted for conducting our research on predictable software design. This architecture is developed for middle-ware in consumer devices, with the emphasis on robustness and reliability. The Robocop CBA is similar to CORBA[11] and Koala,[12] but enables more efficient realization of real-time and performance constraints via modelling techniques. For example, a component designer can supply a *component behaviour model* along with the component executable code. A designer composes an application from the number of components and can predict the application behaviour, using the set of such component behaviour models.

The Robocop CBA is highly efficient, particularly for multimedia processing systems and Systems-on-Chip. First, it allows decomposing the processing software into functional blocks and then mapping these blocks on the

*Space4U is part of the ITEA research program funded by the European Union.

architecture in the optimal way. Second, the supporting Robocop Run-Time Environment has built-in Quality-of-Service implementation. Finally, the freedom of defining the necessary types of models allows addressing not only the processor usage, but also other attributes important for multimedia systems (e.g. memory, bus load and robustness).

Let us now define the Robocop component model in more detail. A Robocop component is a set $M$ of possibly related models, as depicted in Fig. 1. Each individual model $m$ provides a particular type of information about the component. Models can be represented in readable form (e.g. documentation) or in binary code. One of the models is the *executable model* that contains the executable component. Other examples are: *resource model*, *functional model*, and *behaviour model*.

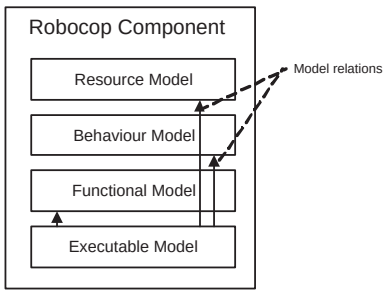


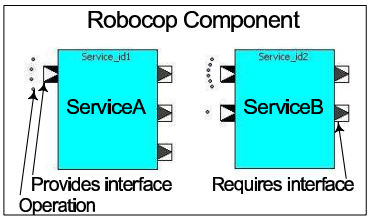

**Figure 1.** Example of Robocop component model.

A component offers functionality through a set of 'services' $P$ (see Fig. 2 and Fig. 3(a)). Services are static entities, which are the Robocop equivalents of *public classes* in object-oriented (OO) programming. More formally, we can specify an arbitrary executable model $m$ by:

$m$ = $P$,
where $m$ is an *Executable Model* and
$P$ is a set of individual $p$'s (services).

**Figure 2.** Specification of an executable model.

Services are instantiated at run-time, using a service manager. The resulting entity is called 'service instance', which is a Robocop equivalent of an *object* in OO programming. A Robocop service may define several interfaces (ports). We distinguish a set of 'provides' ports $PR$ and a set of 'requires' ports $REQ$. The former defines interfaces that are offered by the service, while the latter defines interfaces that the service needs from other services in order to operate properly. An interface is defined as a set of implemented operations *impl_opr*. The binding between service instances in the application is made via a pair of provides-requires interfaces. A service $p$ being part of the above-mentioned executable model is specified in Fig. 3(b).




$p$ = *(PR, REQ)*,
where $PR$ is a set of *pr* (provided ports) and
$REQ$ is a set of *req* (required ports),
*pr* = (*name, interface*),
*req* = (*name, interface*),
*interface* = $O$,
where $O$ is a set of *impl_opr* (implemented operations).

(b)

**Figure 3.** (a) Example of executable component, (b) Specification of Robocop service.

Note that a *Robocop service* is equivalent to a *component* in COM or CORBA, i.e. a service is a subject of composition, and it has input and output ports. A *Robocop component* is a deployable container that packages these services. Therefore, in the Robocop context, the term *composition* stands for a composition of services.

The Robocop implies no implementation-level constraints. The architecture has no limitations on programming languages and platforms. A service can implement any number of threads. Besides this, both synchronous and asynchronous communication are possible.

## 3. PREDICTION-ENABLING APPROACH

In the component-based software, a real-time application developer should satisfy given real-time, performance and functional requirements, when he builds his application on the basis of available components. The scenario simulation approach enables early predictions of the performance properties of a designed application, which help to reason about its quality attributes at early stages of development. The approach is based on three concepts:

- *Models* of the system component's behaviour and resource usage,
- *Execution scenarios* of the complete system in which the resources are potentially overloaded,
- *Simulation* of these scenarios, resulting in timing behaviour of the designed system.

In order to support the smooth interaction and deployment of the above concepts, we have developed a tool, called Real-Time Integration Environment (RTIE). The workflow of our approach (see Fig. 4) is described in[9] in detail. In this section, we briefly describe the workflow, which is based on four phases.

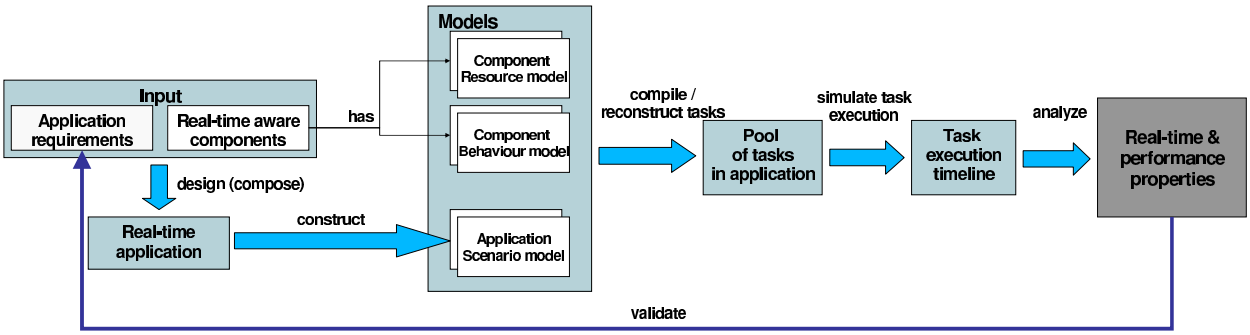



**Figure 4.** Workflow phases of the scenario simulation approach.

### 3.1. Component Specification Phase

A *component developer* specifies the *behaviour* and *resource* models of a real-time aware component at the stage of the component development. These models should be supplied (sold) along with the executables of the component. The resource model contains processing, bandwidth and memory requirements of each operation implemented by the component. The behaviour model specifies for each implemented operation a sequence of external calls to operations of other interfaces. The external call is a usual method invocation made inside the implemented operation. Besides this, the behaviour model may specify thread triggers, if they are implemented by the services of the component. For a detailed specification refer to[9] . In Section 4 of this paper, we give an example of both models.

### 3.2. System Composition and Scenario Identification

An *application (system) developer* graphically composes a real-time application from the set of available components using the RTIE tool. The composition flow passes through the selection, instantiation and binding the services that will satisfy the functional requirements and may satisfy the extra-functional requirements. For a composition, the developer defines a set of resource-critical scenarios[†] and for each of them specifies an *application scenario model*. In the scenario, the designer may specify stimuli (events or thread triggers) that influence the system behaviour. For a stimulus, the designer may define the burst rate, minimal interarrival time, period, deadline, offset, jitter, task priority, and so on. Finally, for each critical scenario, a developer initializes (gives a value to) all input parameters of the constituent components and stores the value into the corresponding scenario model. Consequently, the result of this phase is a set of critical execution scenarios, which sometimes may differ in the parameter values, or in a burst rate of a certain event.

### 3.3. Generation of Scenario Tasks

The *application scenario*, *component resource* and *component behaviour models* are jointly compiled by the RTIE tool. The objective of the compilation is to reconstruct (generate) the tasks running in the application. Prior to compilation, the task-related data is spread over different models. For instance, the *task periodicity* may be specified in an application scenario model, whereas the information about the *operation call sequence* comprising the task is spread over relevant component behaviour models. The compiler reconstructs all necessary properties of the tasks, like deadline, period, priority and operation call sequence. Some details on the task generation are given in Section 4.

### 3.4. Scenario Simulation and Analysis

An application developer schedules (by the RTIE tool) the generated task pool, simulating the execution of the defined scenario. The simulation scheduling policy of the application execution should be compliant with the scheduling policy of the operating system. The resulting data from the scheduler is a *task execution timeline*. This timeline allows extracting the real-time, memory- and bus-related performance properties of an application (system). A comparison between the predicted data and application requirements allows us to quantitatively assess the design of the application. If any of the requirements are not satisfied, a developer may optimize the composition, find other design alternatives (components), negotiate the requirements and repeat the workflow.

In the next section, the above-mentioned aspects are used to decompose and validate an MPEG-4 coding application. The scenario defines the execution mode of the MPEG-4 decoder, we generate decoder tasks (e.g. decoding, rendering) and simulate their execution. We show how the decoder, renderer and buffer resource and behaviour models are specified.

## 4. MPEG-4 CODING APPLICATION

For validation of our performance prediction design technique, we conducted a case study for which we developed a state-of-the-art MPEG-4 coding application. We used the full specification of the standard, featuring arbitrary-shaped video objects. Applying the scenario simulation approach, we predicted the performance and real-time property of the designed decoder. After the integration phase, we compared the predicted results with the real execution data.

Let us now provide some details of advanced video object-oriented processing. Fig. 5 depicts the computation graph for arbitrary-shaped video object decoding as used in MPEG-4. Similar to MPEG-2, objects are divided into macroblocks (MB). The diagram shows special processing stages for decoding the shape and motion of the video objects, in addition to the usual texture decoding. Each decoding job iteration starts with macroblock type decoding (MBtype Dec). The ShapeMC stage computes the motion compensation for the Shape part and provides referenced MB for the Context Arithmetic Decoding (CAD) stage. The CAD provides an MPEG-4 compliant shape representation of the macroblock. The shape for the macroblock is represented by 16x16 binary sub-image, which is sent to the outputs of the Shape job. The Coded Block Pattern (CBP) extracts information about parts of the texture that need to be updated.

[†]Critical scenarios are the application execution configurations that may introduce processor, memory or bus overload.

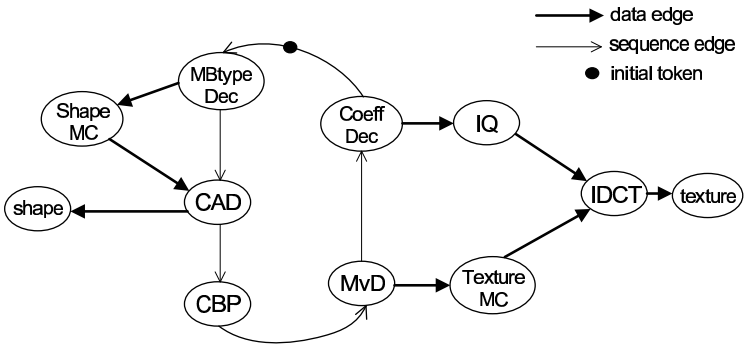



**Figure 5.** Computation graph of arbitrary-shaped MPEG-4 video object decoder.

Texture decoding involves five steps: Motion vectors Decoding (MvD), IDCT Coefficients Decoding (Coeff Dec), Texture Motion Compensation (TextureMC), Inverse Quantization (IQ) and Inverse DCT (IDCT). Stages MBtype Dec, CAD, CBP, MvD, and Coeff Dec are executed sequentially, because each stage depends on another stage, to specify the next position in the input bit stream. Therefore, we introduce a loop surrounding these stages, indicating the order between them. In[10] we discuss the decoder structure in detail.

Having defined the MPEG-4 logical processing blocks and their communication, we follow the proposed scenario simulation design methodology (see Section 3) to build the coding application with predictable performance.

### 4.1. Component specification

In this subsection we distinguish the individual components and specify them independently, including their interfaces. To satisfy the application functional requirements, we developed four Robocop CBA components: Reader, Buffer, Decoder and Renderer (see Fig. 6). For simplicity of explanation, each component contains one service with the same name. The Reader service has IRead *provides* interface (implementing readFrame() and startReadingThread() operations) and IBufferAccess *requires* interface. The Buffer service has only IBufferAccess provides interface, implementing buffering operations popElement() and pushElement(). The Decoder service provides IDecode interface with operations decodeFrame() and startDecodingThread(). In addition, the Decoder service requires two buffers for operation via IBufferAccess interface. Finally, the Renderer service provides IWrite interface with writeFrame(), startWritingThread() operations and requires IBufferAccess interface.

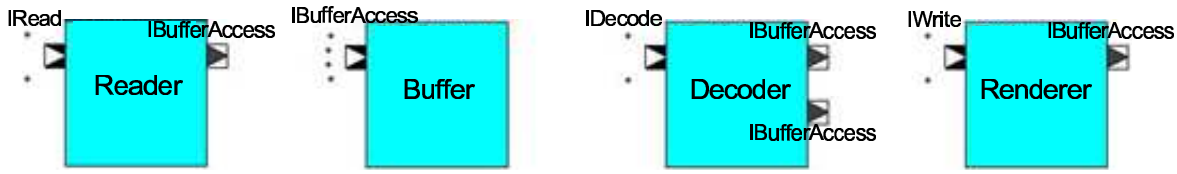



**Figure 6.** Components developed for MPEG-4 application.

Each component was accompanied with their corresponding *resource* and *behaviour* models (see simplified version in Fig. 7). The resource model specifies resource requirements per individual component operation, while the behaviour model also describes the underlying calls to other operations per component operation as well as thread triggers (if existing) implemented by this operation. The resource usage data per operation has been extracted by testing and profiling of each individual component. The operation behaviour data has been generated from the component source code. Reading the Decoder model, we can see that the operation IDecode.decodeFrame() calls IBufferAccess.popElement() operation first (takes encoded frame from the buffer), than decodes the frame in the core of the decodeFrame() and finally calls IBufferAccess.pushElement() to store the

decoded frame to a buffer. All calls are synchronous. The maximum CPU claim of the operation decodeFrame() itself equals to 5.69 ms. Note that the CPU claims of called pushElement() and popElement() operations are specified in the Buffer model - 9.25 ms and 11.51 ms, respectively[‡]. Another example is a startDecodingFrame() operation. It implements a periodic timer which periodically (40 ms) triggers the operation core execution. That means once the operation is called, the firing timer is created and the operation core will be executed periodically.

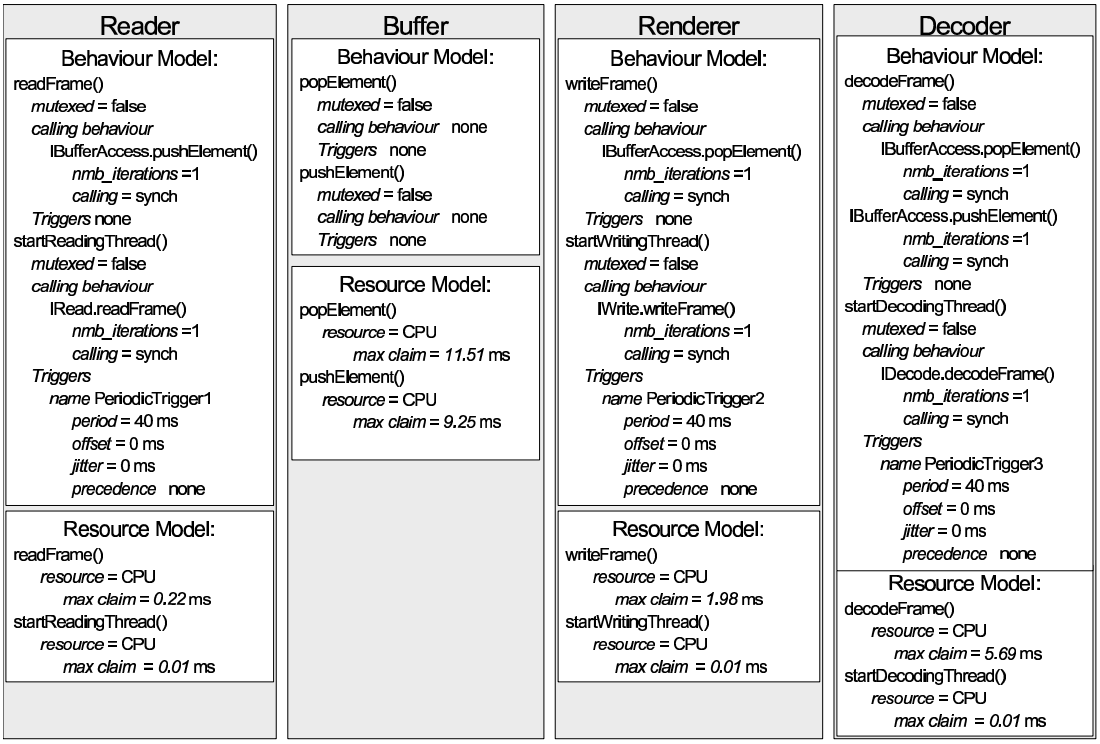



**Figure 7.** Behaviour and resource models of the developed components.

## 4.2. Composition and Scenario Identification

For the scenario identification, the services were (graphically, with the RTIE tool) composed into a new CBA structured MPEG-4 application. The composition process consists of two activities: (a) instantiation of services and (b) binding the instances via their interfaces (see Fig. 8). As indicated by the figure, the Reader instance is bound to Buffer1 to store the encoded video frames. The Decoder instance is bound to Buffer1 to get these frames, and to Buffer2 to store the decoded pixels. The Renderer instance is bound to Buffer2 to get the decoded pixels and render them on the display. After composition, a *critical scenario* has been selected. We predefined a normal execution mode with the resolution 340×280 and frame rate 10 frames/sec as a critical scenario. Note that it is possible to select multiple scenarios. Furthermore, a *scenario model* was specified for the chosen mode (see Fig. 9). It consists of an application *composition structure* and a number of control inputs (*stimuli*). Those inputs (event, periodic timer, interrupt, etc.) lead to the frame-periodic execution of one of the component operations. In our case, we designed three application-level periodic timers that call Reader.readFrame(), Decoder.decodeFrame() and Renderer.writeFrame() operations with periodicity 100 ms, thereby establishing the well-known pipes-and-filters execution architecture. Note that we specified a *deadline* for each task instance triggered by a stimulus (also 100 ms). This is the *real-time property* of the system we are going to validate in the later simulation and analysis phases.

[‡]The relatively high processing claim of the buffering operations is explained by the prototyping implementation - the storage and retrieval were built in a pixel-by-pixel fashion.

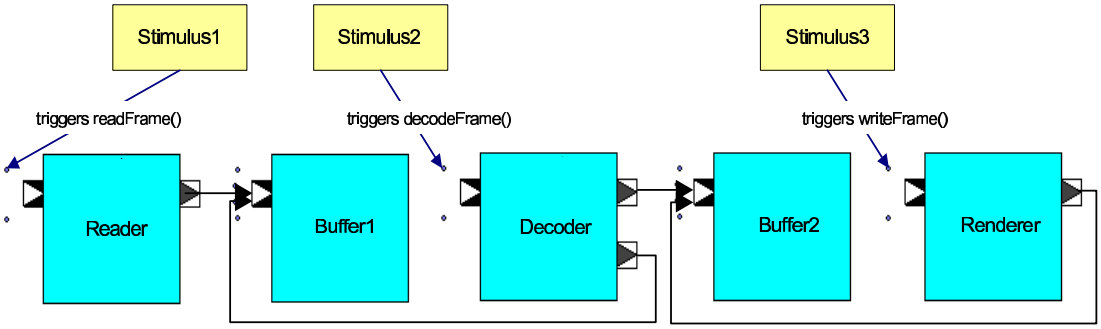



**Figure 8.** Structure of a critical execution scenario.

Scenario Model "decoding mode":

*Composition_Structure*
- *Service_Instances*
  - Reader
  - Reader
  - Buffer1
  - Buffer2
  - Decoder
  - Renderer
- *Binding*
  - Reader; IBufferAccess; Buffer1; IBufferAccess
  - Decoder; IBufferAccess; Buffer1; IBufferAccess
  - Decoder; IBufferAccess; Buffer2; IBufferAccess
  - Renderer; IBufferAccess; Buffer2; IBufferAccess

*Events* none

*Triggers*
- Stimulus1
  - *triggered_opr* = IRead.readFrame()
  - *periodicity* = periodic
  - *period* = 100 ms
  - *deadline* = 100 ms
  - *precedence* none
- Stimulus2
  - *triggered_opr* = IDecode.decodeFrame()
  - *periodicity* = periodic
  - *period* = 100 ms
  - *deadline* = 100 ms
  - *precedence* none
- Stimulus3
  - *triggered_opr* = IWrite.writeFrame()
  - *periodicity* = periodic
  - *period* = 100 ms
  - *deadline* = 100 ms
  - *precedence* none

**Figure 9.** Specification of the scenario model. The scenario plays MPEG-4 video with a rate of 10 frames/sec.

### 4.3. Model Compilation and Task Generation

We jointly compiled all relevant component resource and behaviour models together with the scenario model in order to generate a set of tasks running in the scenario. A *task* is defined by the period (or minimal interarrival time if aperiodic), deadline, offset, precedence constraints with other tasks and a sequence of operation calls through components made by each task instance. The former parameters are inherited from the stimulus parameters in the scenario model. The task call sequences are reconstructed from the individual call sequences of constituent operations specified in the behaviour models. A processor load of a constituent operation is known from the corresponding resource model, so that the RTIE tool can calculate the total execution time of a task instance. The RTIE visualizer draws the generated tasks (see Fig. 10) to help in understanding and analysis of the task behaviour in the scenario.

For instance, the decoding task (see Fig. 10(c)), triggered by Stimulus2, contains three operation calls of the Decoder, Buffer1 and Buffer2 service instances. The CPU utilization times of each operation called are known from the corresponding resource model (see Fig. 7). Thereby, the calculated total execution time of this task is 26.25 ms. The period and deadline of the task (100 ms) are derived from the parameters of Stimulus2 specified in the scenario model (see Fig. 9).

### 4.4. Scenario Simulation

The execution of the generated tasks was simulated with a rate-monotonic virtual scheduler provided by the RTIE tool. The resulting execution *timeline* is depicted in Fig. 11. The timeline indicates all running tasks

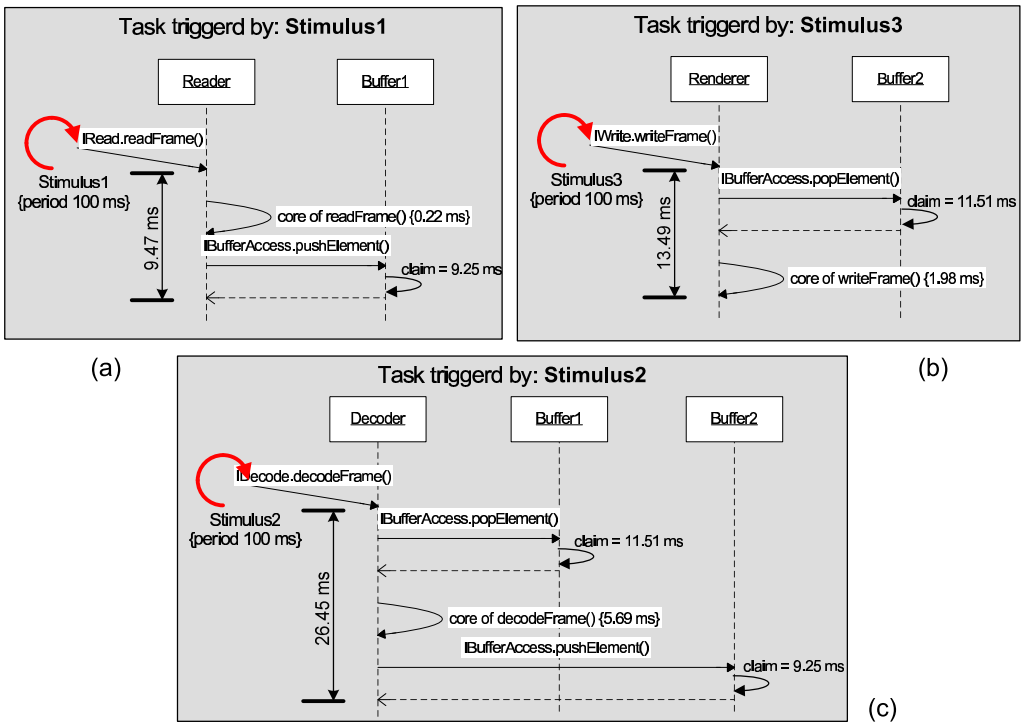



**Figure 10.** Tasks generated for the scenario. (a) reading task triggered by Stimulus1, (b) decoding task triggered by Stimulus2, (c) rendering task triggered by Stimulus3

of the decoder, their predicted *start* and *completion times*, as well as deadlines. The gray boxes on the figure represent the occupation of the CPU caused by their corresponding tasks. The black vertical line in a gray box means a transition from one operation executed to another within the task. The simulation time was set to 1500 seconds. The figure depicts only the first 400 ms of the simulation execution. From the simulation results we concluded that the deadlines for all three tasks are met along the whole simulation time (also for non-plotted intervals). The general performance property (processor utilization) was derived from the timeline data. The total CPU utilization was predicted to be 49.4%. In the next section, we discuss and compare the predicted results vs. real execution data in more detail.

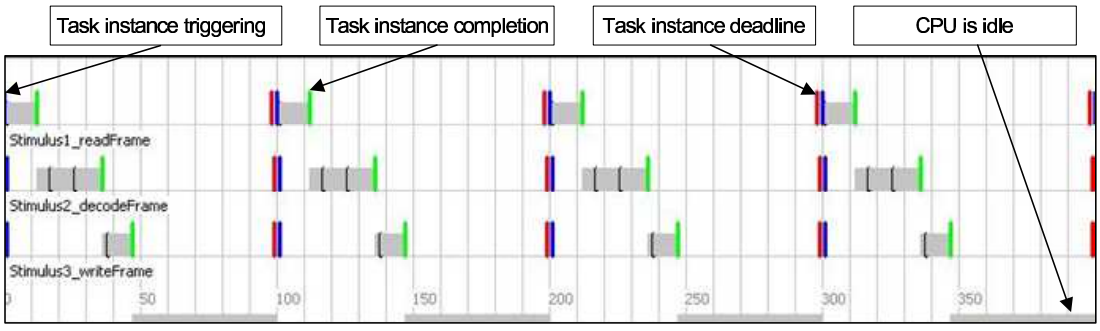



**Figure 11.** Task execution timeline for the selected scenario

## 5. EXPERIMENTS AND RESULTS

After prediction at the design phase, we integrated the real-time aware components into a real MPEG-4 decoder application and executed this on a Linux/32 platform. We aimed at obtaining two types of data: (a) timeline data (which task is executed at what moment) and (b) processor utilization data.

First, we compared the acquired (by the Linux Trace Toolkit) timeline data with the predicted execution timeline. The analysis showed a slight difference in the predicted-*vs.*-real execution moments of the tasks. This difference is explained by numerous OS kernel activities (pagefaults, timestamps, i/o calls) interrupting the decoder execution. Thus, the approach does not allow accurate prediction of the starting/completion times of individual tasks. However, the predicted patterns of the task execution coincided well with the real execution patterns. Besides this, the task real-time requirements (no missing deadlines) has been met as predicted. This was partly due to the large CPU slack (50%) available. In case of higher decoder CPU utilization, the real-time requirements would be jeopardized by the kernel activity. It is clear that for those high-load condition scenarios, the OS kernel activities should be integrated in our design approach.

Second, we analyzed the accuracy of the predicted performance properties (see Table 1). The prediction accuracy error on the average processor utilization appeared to be about 10%, and proved to be reasonably stable. The accuracy error on the *time-detailed* processor utilization (granularity = 1 sec) varied within 30%. We have found that a reason for the strongly varying processor-usage is the variable number of macroblocks for decoding of arbitrary- sized video objects (as we mentioned above, the CPU *claims* of each operation are specified for a worst-case scenario). This observation resulted into the concept of an input-parameter-dependent resource modelling. This new concept is being explored at the moment.

**Table 1.** The predicted vs. real execution data comparison of the MPEG-4 coding application.

| **Type of measurements** | **Predicted exec. data** | **Real data** | **Prediction tolerance** |
|---|---|---|---|
| Average CPU utilization | 49.4% | 45.1% | 8.7% |
| Reading Task av. utilization | 9.5% | 8.7% | 8.4% |
| Decoding Task av. utilization | 26.4% | 23.6% | 10.6% |
| Rendering Task av. utilization | 13.5% | 12.8% | 5.2% |
| Time-detailed CPU util. (lower bound) | 49.4% | 34.3% | 30.0% |
| Time-detailed CPU util. (upper bound) | 49.4% | 49.1% | 0.6% |

Finally, we discovered an interesting phenomenon in the decoder timing behaviour under memory overload conditions. In the Linux OS, a concept of virtual memory is used for dealing with memory overload. This virtual memory is implemented by *page swapping* - releasing memory slots by storing memory data on a disk. Returning this data to memory upon request takes relatively long time. In the MPEG-4 decoder this causes serious latency (1-2 frames) in the processing, because the data cannot be read instantaneously. The resulting phenomenon is that latency requirements are not met, while the CPU usage is very low. The conclusion is that processor, memory and bus usage need to be analyzed jointly to identify the aspects that hamper accurate predictable system operation.

## 6. CONCLUSION

We have exploited a scenario simulation approach that enables prediction of real-time properties of an advanced software-intensive MPEG-4 coding system. The average real-time behaviour of the MPEG-4 decoder can be predicted with high accuracy (within 90%). The prediction accuracy error on the time-detailed processor utilization varied within 30%. As an extra benefit, the timing results give detailed performance information at the design phase. For obtaining the time-detailed accuracy, we have found that integration of a timing model for system activities into our modelling technique is indispensable, because our experiments revealed a large variation of starting and completion times.

The knowledge about the generic computational costs resulting from our approach, provides important guidelines for efficient software/hardware co-design of multimedia coding systems. The case study revealed that the processor, memory and bus usage need to be analyzed jointly, instead of processor analysis only. A second important result is that input parameter dependencies should be taken into account. Third, the system-level activities cannot be neglected during the modelling phase.

For future research, we study the execution of our CBA-based implementation on a programmable multiprocessor (multimedia) system with prediction of its real-time features and bus/memory usage. In order to do this, we will decompose the Decoder component into a number of smaller individual components. Each component can be mapped on a certain processing node in order to achieve an efficient SW/HW partitioning in terms of performance. This fine-grained component decomposition will allow as to address advanced analysis topics, e.g. (a) performance estimation of a multitude of complicated multimedia processing tasks, (b) trade-offs in processor vs. communication bus loads, and (c) efficient architectural mapping of software components.

## REFERENCES


1. C. Szyperski, *Component software : beyond object-oriented programming*, ISBN 0-201-74572-0, New York, Addison-Wesley, 2002.
2. S. Balsamo, A. Di Marco, P. Invenardi, "Model-Based Performance Prediction in Software Development: A Survey", *IEEE Trans. Software Eng.*, vol. 30, pp. 295-310 (2004).
3. E. Wandeler, L. Thiele, M. Verhoef, "System Architecture Evaluation Using Modular Performance Analysis - A Case Study", *Proc. 1th ISOLA Symposium*, 2004.
4. P. Lieverse, P. van der Volf, K. Vissers, "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems", *Journ. VLSI Signal Proc. Signal, Image and Video Proc.*, vol. 29, pp. 197-207, 2001.
5. K.C. Wallnau, "Volume III: A Technology for Predictable Assembly from Certifiable Components", *CMU/ESI-2003-TR-009 report*, April 2003.
6. S.A. Hissam, *et al.*, "Packaging Predictable Assembly with Prediction-Enabled Component Technology", *CMU/ESI-2001-TR-024 report*, November 2001.
7. V. Cortellessa, R. Mirandola, "PRIMA-UML: a performance validation incremental methodology on early UML diagrams", *Elsevier Science B.V.*, February 2002.
8. Public homepage of the Robocop project. [http://www. extra.research.philips.com/euprojects/robocop].
9. E. Bondarev, J. Muskens, P.H.N. de With and M.R.V. Chaudron, "Predicting Real-Time Properties of Component Assemblies: a Scenario-Simulation Approach", *Proc. 30th Euromicro Conf., CBSE Track*, ISBN 0-7695-2199-1, pp. 40-47, September 2004.
10. M. Pastrnak and P. Poplavko and P.H.N. de With and D. Farin, "Data-flow timing Models of Dynamic Multimedia Applications for Multiprocessor Systems", *4th IEEE International Workshop on System-on-Chip for Real-Time Applications (SoCRT)*, ISBN 0-7695-2182-7, pp. 206-209, July 2004.
11. T. Mowbray and R. Zahavi, *Essential Corba*, John Wiley and Sons, New York, 1995.
12. R. van Ommering *et al.*, "The Koala component model for consumer electronics software", *IEEE Trans. Computer*, 33 (3), 78-85, Mar. 2002.
13. D. Box, *Essential COM*, Object Technology Series. Addison-Wesley, 1997.