MASTER

Modular composition of imaging application on commercial-off-the-shelf programmable hardware platforms

Cracana, S.C.

Award date:
2014

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Modular composition of imaging application on commercial-off-the-shelf programmable hardware platforms

Master Thesis

S.C. Crăcană

Supervisor:
Prof. Dr. J.J. Lukkien

Tutor:
Dr. M.M.H.P. van den Heuvel

Version 1.0

Eindhoven, 29 Aug 2014
This work was supported in part by the European Union’s ARTEMIS Joint Undertaking for CRYSTAL – Critical System Engineering Acceleration – under grant agreement No. 332830.
Acknowledgments

I would like to express appreciation to my supervisor, Johan Lukkien, for his valuable and constructive suggestions during the planning and development of this work and for allowing me to start the project.

I would like to thank my tutor, Martijn van den Heuvel for his patient guidance, encouragement and useful critiques throughout all this project. I also want to extend my appreciation to Hrishikesh Salunkhe and Alok Lele from the Software Architecture and Networking Department of Technical University of Einhoven for introducing me to the area of data flow analysis and for their willingness to give their time so generously.

Furthermore, I would like to thank for the support given by the company Barco (especially to Dominique Segers and Benjamin Federau) and to the GStreamer development community for their reactivity and useful replies.

Finally, I am grateful to my friends and my family for supporting me over the years.
Abstract

The primary purpose of this study is to investigate methods for performance analysis and prediction of modular imaging software for medical devices and to improve a prototype development framework for such devices. For such medical devices, we need to derive conservative performance predictions because they must run under strict regulations and performance guarantees. First, we provide an architectural description of the development framework and the goals of the development tools and run-time environment. Additionally, we propose an extension of the framework to enable performance analysis and performance prediction of imaging applications. Secondly, we explain the match between Synchronous Data Flow (SDF) models (that we use for analysis and prediction) and the multimedia processing framework (GStreamer). Lastly, we validate our claims by executing various experiments with an example application and compare the measured results with the outputs of the prediction models.

The prototype development framework embeds several frameworks to enable a high level programming interface through which processing algorithms can be tied together and linked to an application in order to deliver a desired video stream. Individual algorithms can be developed and analyzed independently for deriving run-time characteristics (such as worst case execution time or data consumption/production rate). An algorithm can be analyzed as a model or a fully functional executable. Consequently, the analysis of applications using SDF models can be based on a combination of models and fully functional executables.

In order to analyze GStreamer imaging pipelines using SDF models we enforce several configurations to the framework in order to obtain fixed production/consumption rates, extract the execution time of individual elements and also to follow the memory allocation. Furthermore, we also configure the number of threads being used and the communication channel sizes between threads in order to match SDF assumptions.

Lastly, we perform experiments to validate our work by checking whether or not the guaranteed performance prediction is met by means of observed worst-case measurements. We are also checking the available trade-offs between memory allocation and minimal achievable throughput discovered with SDF modeling and how they impact the run-time. Additionally, we compare two analysis methods found in literature for computing the buffer sizes on the communication channels for the scenarios where an algorithm can or cannot be stopped from executing when the data output channel is full. We compute buffer sizes such that it is guaranteed that no data is lost (given some constraints) we again verify whether or not our predictions are consistent with the memory allocation with the consumption of the algorithms at run-time.
## 1 Introduction
1.1 Context and background ........................................ 1
1.2 Motivation .................................................... 2
1.3 Goals .......................................................... 3
1.4 Problem description .............................................. 4
1.5 Contributions .................................................. 5
1.6 Outline .......................................................... 5

## 2 Related Work
2.1 Computation Models .............................................. 6
2.2 Modeling of multimedia applications ......................... 7
2.3 Pipes and Filters Architecture ............................... 7

## 3 Architecture Description of software-defined imaging pipelines
3.1 Stakeholders and Concerns ..................................... 10
3.2 Barco software display development process ................. 12
  3.2.1 Components engineering ............................ 12
  3.2.2 Component integration .............................. 12
  3.2.3 System integration testing ....................... 12
3.3 Logical view .................................................... 13
  3.3.1 Qt .................................................. 13
  3.3.2 GStreamer ......................................... 15
3.4 Framework extension and work flow summary ............. 18

## 4 Preliminaries
4.1 SDF-modeling .................................................. 20
  4.1.1 Throughput ........................................... 23
4.2 Storage Requirements ......................................... 23
  4.2.1 Back-pressured systems ............................ 23
  4.2.2 Non back pressure systems ....................... 25
4.3 GStreamer-SDF mapping ..................................... 27
  4.3.1 Queues Properties ................................. 27
  4.3.2 Decoder max-threads .............................. 28
5 GStreamer implementation, SDF modeling and the analysis
5.1 Barco use-case application and reference pipeline
5.1.1 Limiting the data rate of the data source in the application
5.1.2 Processing elements of the reference pipeline
5.1.3 Process View of the reference pipeline
5.2 SDF Analysis of the reference pipeline
5.2.1 SDF-Model
5.2.2 Design Space Exploration for non back-pressure
5.2.3 Buffer-sizing for non-back pressure

6 Evaluation
6.1 Environment setup
6.1.1 Hardware and software
6.1.2 Inputs
6.1.3 Run-time metrics
6.2 Experimental setup
6.2.1 Basic setup
6.2.2 Extracting execution times of elements
6.2.3 Pipeline partitioning
6.2.4 Throughput and buffer-size design space for reference pipeline
6.2.5 Comparison between back-pressure and non-back pressure
6.2.6 Validation experiments
6.3 Results
6.3.1 Execution times of elements
6.3.2 Partitioning
6.3.3 Throughput and buffer-size design space for reference pipeline
6.3.4 Comparison between back-pressure and non back-pressure
6.3.5 Validation experiment

7 Conclusions
7.1 Future Work
7.1.1 Scalability of GStreamer
7.1.2 Towards an open environment
7.1.3 Increasing the accuracy of the model

Bibliography

Appendix
A System Description
A.1 Hardware information
A.2 Running Kernel Version
A.3 Linux distribution
A.4 Libraries and dependencies
B GStreamer
B.0.1 GStreamer Probing
B.0.2 gst-tracelib
B.0.3 GStreamer DEBUG info
B.0.4 Probing for multiple buffer-rates
B.0.5 Commands and configurations for generating video tests
C x264 Encoding Options
## Glossary

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALSA</td>
<td>Advanced Linux Sound Architecture</td>
<td>17</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td>15</td>
</tr>
<tr>
<td>COTS</td>
<td>Custom Of The Shelf</td>
<td>2, 3</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>24</td>
</tr>
<tr>
<td>EOS</td>
<td>End Of Stream</td>
<td>17, 21</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
<td>1, 2</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
<td>3</td>
</tr>
<tr>
<td>HDMI</td>
<td>High-Definition Multimedia Interface</td>
<td>2</td>
</tr>
<tr>
<td>LVDS</td>
<td>Low Voltage Differential Signaling</td>
<td>2</td>
</tr>
<tr>
<td>QOS</td>
<td>Quality Of Service</td>
<td>17</td>
</tr>
<tr>
<td>R&amp;D</td>
<td>Research and Development</td>
<td>1</td>
</tr>
<tr>
<td>RGB</td>
<td>Red Green Blue</td>
<td>17, 22</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
<td>2</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
<td>5, 6, 24, 26</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
<td>2, 3</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Software oriented imaging pipeline. Processing steps (in blue) implemented in software \hspace{1cm} 2  
1.2 Architecture transformation. Custom FPGA hardware is replaced by COTS hardware and introducing a software framework for processing \hspace{1cm} 3  
1.3 Modification of the proposed architecture by adding a Performance analysis module \hspace{1cm} 4  
2.1 Various modes for triggering activities in a pipeline \hspace{1cm} 9  
3.1 Barco software display system process \hspace{1cm} 13  
3.2 Logical View of the prototyped framework \hspace{1cm} 14  
3.3 Overview of Qt framework Architecture \hspace{1cm} 15  
3.4 Gstreamer Architecture Overview. Core components: plugins, tools and application use-cases \hspace{1cm} 16  
3.5 GStreamer communication flow \hspace{1cm} 17  
3.6 Architecture extension: performance analysis of the software pipelines \hspace{1cm} 19  
5.1 Use-case application overview \hspace{1cm} 30  
5.2 Overview Barco Use-Case Application \hspace{1cm} 31  
5.3 Reference pipeline \hspace{1cm} 32  
5.4 Execution flow of the reference pipeline without concurrency \hspace{1cm} 37  
5.5 Queues positioning in the reference pipeline \hspace{1cm} 38  
5.6 Execution flow of reference pipeline with three queues \hspace{1cm} 39  
5.7 SDF model of the reference pipeline with limited auto-concurrency and a distribution $\delta=(AB,BC,CD)$ \hspace{1cm} 40  
5.8 Self-timed execution of the reference pipeline SDF model with $\delta=(1,1,1)$ \hspace{1cm} 41  
5.9 Dependency graph of the reference pipeline SDF model with $\delta=(1,1,1)$ \hspace{1cm} 42  
5.10 Pareto space for reference pipeline model \hspace{1cm} 42  
5.11 Simulation time line for the first periodic iteration. Marked overlapping firings actor C \hspace{1cm} 44  
6.1 Timestamps for calculating the execution time for processing the N-th frame of a video \hspace{1cm} 46  
6.2 Partitionings of reference pipeline with two queues \hspace{1cm} 49  
6.3 Corresponding SDFGs for the three partitionings \hspace{1cm} 53  
6.4 Memory usage visualization with TimeDoctor for back pressure and non back pressure reference pipeline over one execution \hspace{1cm} 55  
6.5 Worst case memory usage of non back pressure pipeline (TimeDoctor Visualization) \hspace{1cm} 55  
6.6 Memory usage at run-time of a pipeline without a dominating source (TimeDoctor visualization) \hspace{1cm} 56  
A.1 Hardware topology of the platform used for experiments \hspace{1cm} 63  
B.1 Data probes measurements \hspace{1cm} 66  
B.2 Video pipeline elements \hspace{1cm} 67
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Filesource pseudo-code</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Decoder element pseudo-code</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>Function calls for adding the elements to the reference pipeline and linking them</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Reference pipeline scheduling activation</td>
<td>35</td>
</tr>
<tr>
<td>B.1</td>
<td>QML Hello World Application</td>
<td>68</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This report describes the work done for improving a prototype framework for imaging application development by enabling its performance analysis using a set of formal methods. The framework is proposed by the company Barco in the context of their shift from a hardware-centric towards a software-centric product development. This way, a typical application is back-ended by an imaging processing pipeline defined and implemented in software instead of custom hardware. The structure of a pipeline and its configuration influences the resource usage of the system running it and the overall performance of the application. For such a framework, it would be useful to establish a theoretical model that could describe this relationship and be used at design time of an application. This represents the focus of our work.

We start describing the context of this project in Section 1.1 and the motivation in Section 1.2. Next, we continue with the goals of this project followed by a more detailed problem description in Section 1.4 and the contributions of this work in Section 1.5. Finally, we end this chapter with a general outline of the report (Section 1.6).

1.1 Context and background

The European project CRYSTAL (CRitical sYSTem engineering AcceLeration)\(^1\) takes up the challenge to establish a set of interoperability specifications and a reference technology platform as an European standard for safety-critical systems. Establishing a new standard for a large scale in markets that are already consolidated cannot be achieved by individual organizations, therefore 71 partners from across 10 European countries are part of the project. Among them, Eindhoven University of Technology is collaborating with the Barco company for the medical sector. Barco undertook a set of activities in the context of the development of video applications for consumer health-care devices. Their current display platforms are based on proprietary hardware using Field Programmable Gate Arrays (FPGAs). This is cost-effective when there are only a limited number of products to be supported, but requires a huge research and development (R&D) effort for production and maintenance when the number of product variants increases. Another drawback is that hardware development typically also requires longer development cycles.

Barco therefore works on the creation of a software-oriented framework for medical display platforms. It must provide the means for scalable development of applications for various types of products, speeding up time-to-market and reducing development time. This includes creating the graphical interface, the underlying logic, performance analysis and optimization. In order to display a medical image correctly, an input signal needs several transformations that are linked together in a so-called image pipeline that is required to provide an output compliant with typical medical devices regulations. A high level overview of a medical display image pipeline is given in Figure 1.1. Currently, the image pipeline is implemented using hardware components (implemented on FPGAs). With this thesis proposal, it is intended to achieve a software-only

\(^1\)http://www.crystal-artemis.eu/
implementation of the processing steps marked in blue in the diagram while the other steps remain dedicated to hardware components (the processing steps from the figure are called Software Imaging Components).

Figure 1.1: Software oriented imaging pipeline. Processing steps (in blue) implemented in software

Source: [1]

1.2 Motivation

Application development for embedded systems always needs to take into account resource utilization and performance requirements while delivering the desired functionality. These issues need to be reconsidered every time requirements like hardware resources, processing steps or expected performance change. This indicates a need for tools that can handle requirement variations without affecting development cycles. To give an example of this situation we consider the set of requirements for a multimedia medical device that Barco uses for a set of imaging devices [1]:

1. Input video stream formats to be supported
   - SD - 576i: MPEG2 Transport Stream up to 6 Mbit/s
   - HD - 720p: H.264 Transport Stream up to 15 Mbit/s
   - HD - 1080i: H.264 Transport Stream up to 20 Mbit/s

2. Input video transport formats to be supported
   - Multi-cast User Datagram Protocol (UDP), Real-time Transport Protocol (RTP)

3. Output screen resolutions
   - 1366x768 and 1920x1080 via Low-Voltage Differential Signaling (LVDS) or High-Definition Multimedia Interface (HDMI) with fluent video playback

4. Use Case: Switching between multi-cast channels
   - Latency requirement of maximum 200 for switching between channels (time for displaying the first frame).

5. COTS hardware
6. Video player

Modular QT/QML/GStreamer based video player with hardware acceleration if COTS platform chosen allows it.

![Architecture transformation diagram]

Figure 1.2: Architecture transformation. Custom FPGA hardware is replaced by COTS hardware and introducing a software framework for processing.

With this use case, assuming we decide upon a HD-1080i input stream, using UDP for network transport and a quad core platform (with a given set of hardware resources). Typically, we would want to implement a product that meets the corresponding throughput an latency requirement (as listed above). Besides this, we would need to analyze whether a certain throughput for displaying the video can be achieved (e.g. 25 fps) and how to configure the software for achieving this. However, we could also assume a different product is needed, for a lower price but with similar functionality. This new product would require cheaper hardware and eventually lower performance: for example a product that would use a dual core platform. Then, again the same questions hold regarding the performance, resource usage or the configuration of the software. Essentially, the same problems need to be solved again for only a small change in the requirements. We consider this a strong motivation for our work towards the improvement of the prototype framework by integrating a theoretical model that can describe the relationship between different processing elements enhancing the development process.

On top of that, medical devices must run under strict performance requirements (in terms of throughput or latency) because they need to be compliant with certification regulations imposed by governmental policies. This adds the extra requirement for the tools and the software to give conservative performance predictions (e.g. give guarantees for minimal throughput).

1.3 Goals

The main goals of this projects are:

1. Provide a method for predicting the performance of the imaging software. In particular, this means to obtain a model that adequately describes the relation between the steering parameters (of the software implementation), the hardware resources and the performance metrics for a given pipeline instance.
2. Build a model of a pipeline, validate it and make a comparison between the predicted metrics and the observed ones in a prototype environment.

3. Define the methodology (the steps to be executed) in order to create a model and describe what parameters need to be controlled to match performance prediction.

4. Define the parameters that need to be controlled for matching the performance model.

1.4 Problem description

We would like to modify the proposed framework by adding a Performance Analysis component according to Figure 1.3. The dotted element is the module that we want to introduce in order to analyze a pipeline instance against hardware resources and throughput requirements. Finally, it should yield answers to a set of questions like the expected performance and possible configurations.

![Figure 1.3: Modification of the proposed architecture by adding a Performance analysis module](image)

This raises the following questions:

**Performance analysis questions**

PAQ1 How to model the performance of a pipeline instance? We need to choose a mathematical abstraction to model the throughput and the application model of a pipeline.

PAQ2 What are the input parameters of a such model?

PAQ3 What are the outputs of a model? Once we have a model, what are the expected outputs and how they are correlated to the software framework’s performance.

PAQ4 What is the mapping between the model and the run-time? Establish the relation between the inputs of the model and the parameters of the software.

PAQ5 What are the restrictions of the modeling? What are the limitations of the current methodology.

**Configuration questions**

CQ1 What are the configuration options of the framework for a pipeline?

CQ2 How do they influence the pipeline at run-time? This is specific to each component of the framework and it is relevant for matching the computation model and the run-time model.

CQ3 What needs to be changed in the framework in order to enable more accurate modeling?
1.5 Contributions

The contributions of this thesis are as follows:

- Give an architectural description of the prototype framework (Chapter 3);
- Proposal of an architectural enhancement by introducing a performance analysis component and methodology for configuring the software accordingly (Chapter 3);
- Mapping between SDF models and software multimedia pipelines developed with GStreamer (Chapter 4);
- Integration of resource analysis of GStreamer pipelines at run-time (Chapter 6);
- Prediction and conservatives bounds on performance composable video processing algorithms within a multimedia software framework (GStreamer) (Chapter 5).

1.6 Outline

In this chapter we describe the context of this project and the motivation behind it. We briefly explain the problem that we want to solve and the steps taken to reach the goals of the project.

Chapter 2 presents related work, by presenting a set of computation models, what is our choice and what could be used for future work. Afterwards, we present other examples of existing work in the area of performance modeling of multimedia applications and the connection with our work. We end the chapter with presenting the pipes and filter architectural pattern which is typically adopted for software implementing stream processing tasks.

Chapter 3 covers the architecture description of the framework. We present general information about some of its subsystems (Gstreamer, QML, Matlab) and how they can be used together (interfaced). Finally, we present our proposal for the architecture of the system, and the methodology for using it.

In Chapter 4 we cover basic definitions for SDF modeling, notations and relevant theoretical findings that we will use in our analysis for the SDF models. We end the chapter describing the mapping between Gstreamer pipelines and SDF models.

Chapter 5 considers a pipeline based on the requirements from Barco for which we create a SDF model and then present the analysis algorithms for finding the throughput-buffer size trade-offs and the buffer sizing.

Chapter 6 evaluates our implementation of an example application being modeled in SDF and implemented in GStreamer. We first describe the environment of the setup (details about the hardware platform and the used software), then describe the experiments to validate and evaluate (methodology steps in Chapter 3) the modeling and the analysis algorithms.

The last chapter contains our conclusions and the future work and directions for the project.
Chapter 2

Related Work

This chapter covers related work to this thesis. First, we present a brief comparison of data-flow models and motivate our choice for SDF as a modeling method in our project. Secondly, we present other examples in the area of modeling software multimedia applications but using different methods and how they could fit in our work. In the last section we present the pipes and filter architectural pattern. We consider it related work because its modularity makes software systems (that follow this pattern) easier to be analyzed and hence predicted. We also highlight the major aspects to be considered when analyzing the mapping between SDF and GStreamer.

2.1 Computation Models

In [2] the author provides a comparison of data-flow models. The software framework (GStreamer) we use in this project is meant for building applications that process data streams thus fitting the data-flow paradigm. The criterion used in [2] are the expressiveness (what kind of systems can be described with a model in terms of possible interactions and dependencies), succinctness (how compact the descriptions are), analyzability (availability of analysis algorithms) and efficiency (execution time for algorithms). Synchronous Data Flow (SDF) is a model first proposed in 1986 [3] that assumes that all computation and data communication are scheduled statically. Consequently, algorithms are expressed as SDF Graphs (SDFG) fulfilling certain conditions and can always be converted into an implementation that is guaranteed to take finite time and use finite memory (embedded software requiring minimal usage). This type of modeling is well-suited for digital signal processing and communications systems which often process an endless supply of data. The nodes in a graph represent tasks executing and the edges represent communication between nodes, always consuming and producing fixed amounts of data. Homogeneous SDF (HSDF) models assume that each execution element can produce and consume one data unit at once (represent a subset of SDF). HSDF Graphs (HSDFG) are more succinct than SDFGs (the equivalent HSDFG of an SDFG may require substantially more nodes) but they require more execution time for analysis of throughput and latency. Both these models correspond to subclasses of Petri Nets which is a well-established general purpose model with existing theoretical[4] results that may be applied to them.

Cyclo-Static Data Flow models represent an extension of the previous models by relaxing the assumption of fixed production and consumption rates. Furthermore, Boolean Data Flow Graphs model data dependent behavior and might be interesting for future work of this thesis. Scenario Aware Data Flow models have the same goal and they can express more complex control structures. Such advanced models are beyond the scope of this thesis, but may be interesting for future work.

Kahn Process Networks [5] can model processes that can communicate at any moment in time and also depending on the content of the data. To this extent, KPN is very expressive but it does not provide methods for analyzing properties that would be interesting for our project such as throughput or latency.
All in all, we decide to use SDFGs to model our software because of the trade-off it provides between expressiveness, analyzability and low computation time for analysis algorithms. In this work, we will investigate the performance (in terms of throughput) of an example imaging application implemented using GStreamer.

### 2.2 Modeling of multimedia applications

Weffers et al. [6] provide a mathematical model for the behavior of media processing chains of tasks that communicate through memory buffers. The proposed model is based on *traces* which represent the execution order of the task. The authors provide methods for obtaining the capacities for the queues between elements, number of context switches and response times for individual elements and entire chain. Still, variable computation times of tasks and timing constrains of chains are not taken into account and multiprocessor platforms are not investigated. Another limitation is that this work assumes only systems with a back-pressure mechanism and would not be suitable for live streaming applications (explained further in Chapter 5).

Reduction of memory requirements for multimedia applications is studied in [7] on an application model that assumes time-driven heads and tails in pipelines with bounded end-to-end latencies. A general mechanism for reducing memory requirements is presented on the assumption of shared memory pools. Within our context, this work could be reused for those parts of the imaging pipeline that execute on a single-processor case. However, this is left as an exercise for future work.

Video components typically impose real-time constraints with highly transient variations in the processing of their stream rendering [8]. Allocating a static amount of processing resources to video applications is therefore unsuitable for consumer electronics, because it leads either to frame misses or to an over-provisioning of resources. To enable cost-effective video processing in traditional consumer electronic terminals [9], many quality-of-service (QoS) strategies [10] have been developed in order to estimate the required processing resources by the processing pipeline dynamically. These strategies typically estimate resources at the level of individual frames and a frame is then entirely processed, for which processing resources may or may not be sufficient. In the latter case, a work-preserving approach is often taken in which the processing of the current frame is completed and a next frame is skipped [10]. However, for medical imaging applications, as considered in the current paper, the potential loss of video content and quality compromises are unacceptable due to strict certification regulations.

### 2.3 Pipes and Filters Architecture

Complexity of embedded systems makes their analysis and performance prediction a challenging task. One step towards better predictability of software is to choose a modular architecture which allows a *divide et impera* approach. The *Pipes and Filters* pattern [11] offers a structure suitable for systems that process data streams. The processing is divided in independent sequential steps that are encapsulated in filter components. Data is passed between adjacent filters which consume and deliver data incrementally. Typically, the input to the system is a data *source* and the output of the system is a data *sink*. These two are connected by intermediate filters through pipes depending on the requirements of the output of the system. The sequence of filters combined by pipes is called a *processing pipeline*.

The *filter* component represents the building block of the system. It performs transformations on its input data either by performing some computations and adding information, or it refines data by compressing or extracting. Another option is that it transforms the data by changing the delivering format. Nevertheless, concrete implementations can contain a mix of these basic operations. This component has will be encountered in both the model and the implementation and we need to identify the mapping between their executions.

The activity of a filter can be caused by several events:
- The subsequent pipeline element pulls output data from the filter.
- The previous pipeline element pushes new input data to the filter.
- The filter is active in a loop, pulling its input from and pushing its output down the pipeline.

Based on this, filters can be categorized as passive (first two cases) and active (last case). Basically, the difference is that active filters start processing on their own as a separate program or thread, while passive ones need to be called from other elements. This behavior must be captured by the model and the implementation.

Pipes are used for setting connections between filters. When two active filters are joined, the role of a pipe is to synchronize them with a first-in-first-out buffer. For both active and passive filters, they represent data dependency and also show the order of execution in a pipeline. The pipes represent the memory that needs to be allocated and will represent one of the outputs of the performance analysis component.

A pipeline can have a mix of passive and active elements which influences its control flow. We present the possible scenarios for the flow control in a pipeline with one data source and one data sink which are connected through two filters:

- **S1: Push pipeline** - the source is driving the pipeline (Figure 2.1a)
- **S2: Pull pipeline** - the sink is driving the pipeline (Figure 2.1b)
- **S3: Mixed pipeline** - a filter is driving the pipeline (Figure 2.1c)
- **S4: All filters active pipeline** - each element is active running on a separate thread (Figure 2.1c)

In each scenario, the first filter executes $f_1$ for processing data and the second one executes $f_2$. In Figure 2.1a the pipeline is in push mode, meaning that the activity starts with the data source. Filter activity is triggered by writing data to the passive filters and hence, the flow is the following: the source writes data to the first filter, the filter executes its function and pushes the new data further on the second filter which will execute its own processing and pass the data to the source. In the end, the control returns to the source.

A second scenario (Figure 2.1b) is when the sink coordinates the control flow by requesting data. At first, the request for read is passed through, from filter 2 to filter 1 and then to the source. Thereafter, the same flow as in the first scenario follows, the only difference being that at the end of the processing, the sink triggers again the activity. Both these first two cases represent a sequential execution of some processing steps ($f_1$ and $f_2$).

In the third scenario (Figure 2.1c we have mixed push-pull pipeline. The second filter is in pull/push and will represent the active element starting the activities in the first side of the pipeline and push data towards the sink when this data is available. In this situation, we also have only execution thread that is controlled by the second filter.

Lastly, the most interesting scenario (Figure 2.1) contains only active elements. All of them pull and push data so therefore will run in separate processes. This requires a mechanism for synchronization. Consequently, buffering pipes are introduced between them, enabling parallel execution. Furthermore, the sizes of the buffers influence the performance of the pipeline (throughput) and the memory allocation.

Various implementations for this pattern exist: the Unix shell programs, compilers, application for signal processing, functional programming and distributed systems. GStreamer also implements this architecture and one of its testing tools (gst-launch has a syntax influenced by the Unix shell - Appendix B.0.5). It is important for us to identify the parameters in GStreamer that determine the control flow (which scenarios are supported) and how threading and memory allocation is handled.
Figure 2.1: Various modes for triggering activities in a pipeline
Source: [12]
Chapter 3

Architecture Description of software-defined imaging pipelines

In this chapter we discuss the proposed framework and its elements. We first refer to a subset of its stakeholders and their concerns regarding the framework in Section 3.1. In Section 3.2 we present the development process envisioned by Barco process view at a high level and identify the step where the framework fits. We continue with Section 3.3 which includes the logical view of the system were we describe the relevant components to our work in the framework (QT, GStreamer), their application model, available tools and the overall contribution to the framework and also the interfacing. Lastly, in Section 3.4 we present a new architectural extension and the methodology that will be investigated in this project.

3.1 Stakeholders and Concerns

To better understand the framework Barco is working on, it is important to start with its main users or persons interested in using it (stakeholders) and their concerns or what they would expect from the framework. Table 3.1 contains the relevant stakeholders for our work. Each of these actors have particular concerns. Application developers are interested in:

- Fast and simple development: use powerful programming languages;
- Re-usability: reuse existing working elements so that they can cover new requirements using existing work;
- Flexibility: cross platform development so that an application is developed once and can be run on different platforms;
- Performance: estimate performance of software at design time using models or (partially) developed functional components

Algorithm developers need to:

- Use a high level environment for testing and creating of algorithms;
- Stay away from implementation details. Work on concepts;
- Separate concerns: test and develop the algorithms separately from the application that will use them.
<table>
<thead>
<tr>
<th>Actor</th>
<th>Description</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications developer</td>
<td>Uses the framework for developing applications</td>
<td>• GUI creation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Pipeline implementation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Integration of custom algorithms</td>
</tr>
<tr>
<td>Algorithms developer</td>
<td>Uses the framework for creating processing steps</td>
<td>• Development of custom algorithms for image processing</td>
</tr>
</tbody>
</table>

Table 3.1: Stakeholders and their concerns

A software-oriented design process aims at addressing these concerns. Implementing a pipes and filters architecture ensures modularity and independent development. Interfacing the right tools can ease the collaboration between these two types of stakeholders enabling them to stick to the software tools they are already used to without affecting their productivity. Additionally, a modular architecture allows a better performance prediction and analysis of the system.
3.2 Barco software display development process

Segers et al. [1] describes the desired software-centric development process of Barco at a very high level (see Figure 3.1). It is represented as series of activities that are executed to deliver a product. Here is a short description of each step [1]:

1. Requirements: During this activity the Stakeholder Requirements are captured, maintained and managed.

2. Release Planning: Based on the Stakeholder Requirements the product development is planned with respect to releases and their features.

3. Sprint Planning: A Work Breakdown Structure is created for the product release of which the development is started.

4. System Engineering during this activity the System Requirements are created and an executable System Architecture is defined and verified by simulation. The System Architecture describes the complete system with all the engineering disciplines such as mechanical, electrical and software.

5. Components Engineering: The architectural components that are defined during System Engineering are handed over to the engineering disciplines where they will be defined, designed, created and tested.

6. Component Integration: The components that make up the system are integrated, thus assembling a part of or the complete system.

7. System Integration Test: The system is tested with respect to the system requirements that are created during System Engineering.

8. System Acceptance Test: the system is validated with respect to the stakeholder requirements that are captured during the Requirements activity.

9. Impact Analysis: Change proposals are evaluated, the impact is determined and the proper actions are defined and planned.

The prototype framework will be used in the Component Engineering and Integration steps (Component and System integration) and therefore we zoom-in into these steps in the following subsections.

3.2.1 Components engineering

This step describes how components are created. The product is divided into constituent components, each with their own requirements. The development process is centered on developing these individual components. This step should take into account methods for extracting performance information from the component that will be needed in the system integration step.

3.2.2 Component integration

In this step, performance is predicted for an application composed of independently developed components. Based on the predictions, different configurations can be considered given the run-time resources.

3.2.3 System integration testing

In this step, the application is tested in tested the real run-time environment. The predictions from the Component Integration step are checked against the run-time measurements.
3.3 Logical view

The logical view "serves to identify common mechanisms and design elements across the various parts of the system" [13], showing the structure of the system through its components and their interactions. We describe here all the components used for developing an application and how they interact within Barco’s development framework. This software framework is currently at an experimental phase and has not been documented before.

An overview of the framework is shown in Figure 3.2. Typically for creating an application, a developer uses QT for defining the user interface and GStreamer for creating the underlying imaging processing software. When more complex algorithms are needed, Matlab is used for developing and testing them and they are integrated in the imaging software (GStreamer).

In the subsequent subsections we describe the User Interface component, the Imaging Pipeline Component and the interfacing between them

3.3.1 Qt

Qt [14] is a cross-platform application and Graphical User Interface (GUI) development framework and it is at the top layer the Barco development framework. The main programming languages that can be used are C++, Java and JavaScript-based language called Qt Meta Language (QML). Besides them, there are also a set of bindings for other languages available. An overview of the architecture of Qt can be seen in Fig 3.3.

The libraries are divided into modules. All of them rely on a main one called QtCore which provides basic functionality enabling communication between objects (through signals and slots),
queryable and designable object properties and others (for a complete list of Qt Core features see [15]). The other modules provide means for GUI development, database interaction, network communication etc.. As shown in Fig 3.3, the platform offers a set of tools. One of the most important tools is the QtDesigner, used for designing and building GUIs from a Qt library (containing various widgets - elements of interaction for the GUI) in a what-you-see-is-what-you-get manner (WYSIWYG). Further on, Qt Linguist is a tool for multi-language support, QtAssistant can be used for documentation generation and qmake is a cross platform build tool. (for a complete list, see [16]).

Qt Meta Language and QuickStreamer

QML is a high level, scripted language designed for defining the user interface of an application [17] in terms of appearance and behavior. This is done as a tree of objects with properties. The elements of the tree represent the building blocks of an application. Listing in Appendix B.1 shows an example of QML syntax. For each element, we can set an identifier, properties and handlers to be executed for specific events that will define the behavior of the interface.

The module that enables the use of QML is called "Qt QML". It defines and implements the language and engine infrastructure, and provides an Application Programming Interface (API) to enable application developers to extend the QML language with custom types and integrate QML code with JavaScript and C++. The Qt QML module provides both a QML API and a C++ API. On top of Qt QML we have, Qt Quick that is the standard library of types and functionality for QML. It includes visual types, interactive types, animations, models and views, particle effects and shader effects. A QML application developer can get access to all of that functionality with a single import statement.

Because of the large use in applications of both Gstreamer and Qt (and QML), some methods for integrating the two of these emerged. One method for integrating Qt (QML) and Gstreamer is implemented through a Qt module called QuickStreamer and it’s based on creating custom QML types that represent Gstreamer elements. The details of the implementation can be seen at [18]. With this module, the developer can use the QT framework as a one stop shop for defining a
pipeline for applications in QML. For this we give a pipeline example and how can it be created using C language and a tool from GStreamer and using QML.

### 3.3.2 GStreamer

In this section, we describe the layer responsible for image processing of the Barco framework. We will go more into depth covering the main concepts, since it is strongly related to the performance an application has and the work presented in this thesis is build on top of it.

GStreamer is an open-source framework for creating media applications that handle audio, video or any kind of data flow in a modular way. The basic idea of GStreamer is to link together various elements in a graph-based way to obtain a stream that meets some desired requirements. The arrangements and the linking of the elements is called a pipeline and it basically defines the flow of data. The GStreamer architectural overview can be seen in Fig.3.4. The core of the framework provides the infrastructure for the system (pipeline architecture, media handling, base classes, types etc.) and exposes a set of interfaces that the other elements of the architecture must comply with in order to integrate in the system. Based on the core functionality there are different types of plug-ins for communication, formats, codecs and filters, that are already available or new elements which can be developed and hooked up into the architecture easily due to its modular design. Besides this, a set of tools are available for testing pipelines (gst-launch) and for investigating the existing elements and their properties (gst-inspect).

#### Elements

Elements are the building block of the Gstreamer pipeline architecture. Gstreamer provides elements as loadable blocks of code. A registry is used to store the details of the existing elements in an binary registry file and in this way a program using Gstreamer does not have to load all of them at start but only the elements it uses.

Elements can be broadly classified in the following categories:
Figure 3.4: Gstreamer Architecture Overview. Core components: plugins, tools and application use-cases

Source: [19]

- Source elements - Media creator components. E.g. Advanced Linux Sound Architecture (ALSA) source, File Source, XImage source;
- Sink elements - Media assimilation components: ALSA Sink, File Sink, Ximage Sink;
- Transform elements - Media transformation components: Volume Control, RGB Control etc.;
- Processing elements - Media manipulation components: buffer elements, decoder elements.

After elements are created, they are added to a pipeline element which represent a container for a list of linked elements and takes care of different aspects such as initialization/de-initialization, scheduling (pad activation in Section 5.1.3) or communication (offers a bus for elements to communicate upstream and downstream).

Matlab algorithms conversion into GStreamer elements

Matlab [20] is a broadly used computing environment used in many research and industry contexts from engineering to economy that allows fast development of algorithms, modelling etc. Therefore, it is often the case that applications need to use algorithms created in Matlab. In our situation, it would be needed to use custom algorithms for video processing in an application developed using the Gstreamer. This has been done successfully by the industry [21]. The basic idea is to transform the Matlab code into a Gstreamer element that would then be integrated into the pipeline architecture. This is achieved using a C wrapper along with the MATLAB compiler and a set of specifications when writing the code in Matlab.

Inter elements communication

Inter element communication is managed using pre-specified media stubs in the element. They are called pads. Pads are used for link negotiation and data flow between GStreamer elements. They represent the ports of an element where links can be set with other elements and through which data can flow to/from those elements. The type of a pad is defined by its direction and by
its availability. Direction can be either "sink" (in) or "source" (out). In graphical representations, sink pads are drawn on the left side of an element and the source pads are on the right side. Data flows from left to right. Furthermore, both types of pads (sink or source) have specific data handling capabilities and they can restrict the type of data that can flow through them. As a consequence, links can be made only between pads with compatible allowed types. The type data that can be sent within the framework can be seen in Fig. 3.5 and they are described as:

Figure 3.5: GStreamer communication flow
Source: [19]

- Control communication - information for controlling media processing, called events (End Of Stream - EOS, Quality of Service - QOS, Latency);
- Content communication - transfer of media content that needs to be processed, called buffers;
- Application communication - messages from the application to a pipeline for asking information about processing (e.g. duration of streaming), called queries and information; sent to the application by the elements (warnings, errors, tags) called messages.

Memory allocation

Memory allocation is important in multimedia applications, especially for embedded devices where the resource usage needs to be minimized. Additionally, when using special purpose hardware resources, the memory use and access methods is also specific to each one. In this context, GStreamer aims at providing a flexible memory management. We will introduce the low-level memory handling element, GstMemory and its allocation mechanism. Then, we continue to a higher level to the memory elements used to exchange data between GStreamer elements based on GstMemory, called GstBuffers. The GstBuffers represent the data passed between filters in the pipes and filter architecture and the tokens in the SDF model. For managing efficiently memory buffers of the same size, GStreamer uses GstBufferPools. We describe the basics for this mechanism and end with the method used for negotiating memory management options between elements.

First of all, the low level object for managing access to a memory area is GstMemory. It points to a region of memory that can have a maximum size set at creation time and that cannot be changed later on. The content is accessed through an offset and a size of the actual data which both can be changed. These objects are allocated through an allocator object, GstAllocators. Different allocators exist for system memory, shared memory or Direct Memory Access backed memory. For special types of memory, different allocators need to be implemented.

A GstBuffer is a lightweight object that is passed between elements and contains memory and meta data. A buffer represents the media content that is pushed or pulled by elements and can be made out of one or more GstMemory objects. Accessing the data of the buffer can happen by
retrieving individual GstMemory objects or one large continuous memory area with containing all objects.

The GstBufferPool object provides a convenient base class for managing lists of reusable buffers. Essential for this object is that all the buffers have the same properties such as size, padding, metadata and alignment. A buffer-pool object can be configured to manage a minimum and maximum amount of buffers of a specific size. A buffer-pool can also be configured to use a specific GstAllocator for the memory of the buffers.

**Scheduling**

Scheduling overlaps with the activity triggering presented in Section 2.3. In GStreamer this is based on pads actively pushing (push-mode scheduling) or pulling in data (pull-mode scheduling) from other pads. The framework supports elements with pads in any of the scheduling modes where not all pads need to be operating in the same mode. If all pads of an element are activated in push-mode, the element is push mode as a whole. For a source, this means that it has to start a task that pushes buffers on its source pad towards downstream elements (Scenario I from Section 2.3). Alternatively, the sink pads can operate in pull mode. For a sink, it matches the second scenario from Section 2.3. The sink will start a thread and ask for data from upstream elements. This requires all the up-stream elements to be able to operate in pull mode. For the third scenario, an element in the middle of a pipeline has to have its sink in pull mode, its source in push mode, the elements upstream enabled in pull mode and the elements downstream in push mode (matching scenario 3). The forth control flow scenario from the pipes and filters pattern can be implemented in GStreamer introducing queues between elements which creates thread boundaries in pipelines.

**Threads**

GStreamer is a multi-threaded framework. This means that, internally, it creates and destroys threads as it needs them, for example, to decouple streaming from the application thread. Moreover, elements are also free to create threads for their own processing (for example, a video decoder could create 4 threads to take full advantage of a CPU with 4 cores). Furthermore, when building the pipeline an application can specify explicitly that a branch (a part of the pipeline) runs on a different thread (for example, to have the audio and video decoders executing simultaneously). This can be done by placing a queue which represents a thread boundary in the framework. Another situation when threads can be used is when a throughput requirement is not met having the pipeline on a single thread and all the elements executing sequentially. In this case, queues can be inserted between elements to split the work among the elements to meet the requirement (if possible).

### 3.4 Framework extension and work flow summary

We propose an additional module to be included in the prototype framework (Performance Analysis block in Figure 3.6). Its purpose is to predict the performance of a given pipeline instance while giving a relation between memory allocation and achievable throughput through a design space exploration (explained in Section 5.2.2 and Section 5.2.3). We explain in this Subsection the work flow for the framework and the proposed extension in terms of a set of steps to be followed in order to develop an application in a predictable way. We describe the the main steps, from gathering requirements to the performance analysis of an application and give references in the the report where specific steps are explained in more detail and examples are given.

1. **Gather requirements of the software**
   In this step requirements are derived from discussions with the clients and also taking into account different regulations for medical devices. An example is given in the Motivation section of Chapter 1. The requirements will ultimately dictate what processing needs to be performed by an application and what are the performance requirements.
2. **Design pipeline**  
Based on the requirements of the application (obtained in Step 1), a GStreamer pipeline architecture is created. Mainly, this means to connect a set of GStreamer plugins in a pipeline that would produce a desired output. An example of a pipeline for a medical device is given in Chapter 5.1 and it will be used throughout this report.

3. **Obtaining execution times of plugins**  
After the pipeline is constructed, we need the execution times of the selected plugins. This information is required for the modeling of the pipeline (Step 5). The best case and the worst case execution times of the pipeline can be obtained by running the pipeline with a set of test video sequences and measuring the execution of each element for processing a GStreamer buffer. It is important to ensure a fixed production and consumption rate of buffers per plugin (assumptions required for SDF modeling). A way to enforce this is described in Section 5.1.1.

4. **Partitioning the pipeline**  
This step is about deciding the level of parallelism of the designed pipeline. This is done by setting sets what elements will run on separate threads. We how this is done for GStreamer and give a guideline on how to do it Section 6.2.3.

5. **Create SDF model**  
Based on the chosen partitioning and the execution times of plugins an SDF model can be created for the pipeline. The mapping between SDF models and GStreamer is detailed in Section 4.3 and examples of SDF models based on different partitionings of a GStreamer pipeline are given in Section 6.3.2.

6. **Memory / throughput trade-offs analysis**  
Once an SDF model is created, an algorithm for obtaining the possible trade-offs between memory allocation and throughput can be executed. We consider (i) the case when a synchronization method is available for suspending plugins execution when their output channel has reached the maximum capacity (detailed in Section 5.2.2) and (ii) the case when this mechanism cannot be enforced and predictability is ensured by limiting the input rate of the pipeline (explained in Section 5.2.3).

---

![Figure 3.6: Architecture extension: performance analysis of the software pipelines](Image)

---

19
Chapter 4

Preliminaries

In this chapter we describe the preliminaries for the analysis of GStreamer pipelines using SDF models. We introduce in Section 4.1 the definitions, assumptions and theoretical findings from data flow literature for predicting the performance of such pipelines (and real-time guarantees for minimum throughput) and existing trade-offs between memory allocation and throughput for given pipeline instances. In Section 4.2 we discuss storage requirements of SDF models and the conditions under which guarantees for no data loss and minimal throughput can be given. In Section 4.3 we explain the mapping between a SDF model and a GStreamer pipeline. We also explain the parameters of some elements in a pipeline that influence the performance and the memory allocation. This chapter gives the background for the next one where we take a specific GStreamer pipeline instance, create a model for it and run the analysis algorithms.

4.1 SDF-modeling

We use SDF models to analyze the behavior of GStreamer pipelines. We introduce here the related definitions and annotations that will be used in the remainder of this report. We are consistent with the notations used in [2] that which provides the algorithm for throughput-memory allocation trade-offs exploration for back-pressured systems. Additionally, we present some required notions for introducing the method from [22] for buffer-sizing for non-back pressure systems.

A SDF model is a graph with nodes called actors and representing functions that execute by consuming tokens (data units) from their inputs and writing the results at their outputs (ports). The number of tokens consumed for the execution of actors is called input rate and the produced one is output rate (annotated on the input and output edges of actors). The edges represent the channels of a SDF model and reflect the communication channels between actors. If a channel contains tokens, it is specified graphically by a dot on the edge and a number representing the amount of tokens. The storage space of the channels is in theory unbounded. Since this is not reasonable for real implementations (which have finite resources), bounds can be enforced on the channels. Furthermore, the execution of an actor is called firing. Firings are atomic and are assumed to have fixed execution times.

We now present the formal notions that we need to describe the algorithms for computing buffer requirements and analyze the throughput (examples for the definitions and annotation will be given in the next chapter). As a starting point we consider the set of connection points (ports) of actors to be called Ports and for each \( p \in \text{Ports} \) an associated finite rate \( \text{Rate}(p) \in \mathbb{N} \setminus \{0\} \).

**Definition 1 - Actor.**

An actor \( a \) is a triple \((In, Out, \tau)\) where \( In \subseteq \text{Input ports} \) (denoted \( In(a) \)), \( Out \subseteq \text{Output ports} \) (denoted \( Out(a) \)) with \( In \cap Out = \emptyset \). \( \tau \in \mathbb{N} \setminus \{0\} \) represents the execution time and is denoted as \( \tau(a) \). Additionally, when various execution times of actors are considered, we write \( \tau(a, i) \) and consider it to be the execution time for the \( i + 1 \)th firing.
Definition 2 - SDFG.

A SDFG is a tuple \((A,C)\) where \(A\) is a finite set of actors and \(C \subseteq \text{Ports} \times \text{Ports}\) a finite set of channels meeting the following properties:

1. A channel’s source must be an output source of an actor.
2. All channels are connected to ports of actors.

For an actor \(a = (\text{In}, \text{Out}, \tau) \in A\) we denote the set of all channels that are connected to the ports in \(\text{In}\) as \(\text{InC}(a)\) and all the channels connected to ports in \(\text{Out}\) as \(\text{OutC}(a)\).

In real applications, the execution time of code is variable and therefore the assumption for actors having fixed execution time does not hold. Traditionally, this value is replaced with the Worst Case Execution Time (WCET) and this allows to obtain real-time guarantees (e.g. for back-pressure models the buffer sizes obtained with WCET are valid for any Variable Execution Time [23]). Furthermore, when back-pressure is not available, the variation in the execution time influences the buffer sizing and just a WCET analysis is not enough (we will treat these two situations separately in the further sections).

When an actor \(a\) starts its firing, it removes \(\text{Rate}(p)\) tokens from all channels \((p,q) \in \text{InC}(a)\). When its execution ends, the actor will produce \(\text{Rate}(p)\) tokens on every channel \((p,q) \in \text{OutC}(a)\). Rates can be chosen in a way such that deadlocks occur or tokens accumulate on channels (hence they can execute with unbounded storage memory). The required condition for an SDFG to execute within bounded memory is called consistency [24]. We are interested in modeling such systems since they describe applications that can virtually run for an infinite period of time yet using a finite amount of physical memory.

Definition 3 - Repetition Vector and Consistency.

A repetition vector of an SDFG \((A,C)\) is a function \(\gamma : A \to \mathbb{N} \setminus \{0\}\) such that

\[
\forall (p,q) \in C, (p \in \text{Out}(a)) \land (q \in \text{In}(b)) \land (a,b \in A) : \text{Rate}(p) \cdot \gamma(a) = \text{Rate}(q) \cdot \gamma(b)
\]

(4.1)

A repetition vector \(\gamma\) is called non-trivial iff \(\forall a \in A, \gamma(a) > 0\). A SDFG is called consistent if it has a non-trivial repetition vector. The smallest repetition vector of a consistent SDFG is called the repetition vector.

Definition 4 - Timed SDFG.

A timed SDFG is a triple \((A,C,\Upsilon)\) consisting of a SDFG \((A,C)\) and a function \(\Upsilon : A \to \mathbb{N}\) that assigns an execution time for every \(fa \in A\).

Definition 5 - Time Bounded SDFG.

A time bounded SDFG is a denoted by \((A,C,\hat{\Upsilon},\Upsilon,\check{\Upsilon})\) consisting of a SDFG \((A,C)\) and the functions

- \(\hat{\Upsilon} : A \to \mathbb{N}\) where \(\hat{\Upsilon}(a)\) gives the BCET of \(a\)
- \(\check{\Upsilon} : A \to \mathbb{N}\) where \(\check{\Upsilon}(a)\) gives the WCET of \(a\)
- \(\Upsilon : A \times \mathbb{N} \to \mathbb{N}\) where \(\Upsilon(a,k)\) gives the execution time for the \(k + 1^{th}\) firing of \(a\)

and \(\forall a \in A, \forall k \in \mathbb{N} : \hat{\Upsilon}(a) \leq \Upsilon(a,k) \leq \check{\Upsilon}(a)\).

In order to capture the timing behavior of an SDFG we need to keep track of the distribution of tokens over the channels, the start and end time of actor executions and the progress of time. For the quantity of data related to channels we introduce the following definition:
Definition 6 - Channel quantity.
A channel quantity on the set $C$ of channels is a mapping $\delta : C \rightarrow \mathbb{N}$. If $\delta_1$ is a channel quantity on $C_1$ and $\delta_2$ is a channel quantity on $C_2$ with $C_1 \subseteq C_2$, we write $\delta_1 \preceq \delta_2$ if and only if for every $d \in C_1, \delta_1(d) \leq \delta_2(d)$. Channel quantities $\delta_1 + \delta_2$ and $\delta_1 - \delta_2$ are defined by point wise addition of $\delta_1$ and $\delta_2$ and subtraction of $\delta_2$ from $\delta_1$ ($\delta_1 - \delta_2$ defined if $\delta_2 \preceq \delta_1$).

The number of tokens read (consumed) when a firing of an actor $a$ starts can be described by a channel quantity $Rd(a) = \{(p, \text{Rate}(p)|p \in In(a)\}$. Similarly, the amount of tokens written (produced) at the end of a firing by $Wr(a) = \{(p, \text{Rate}(p)|p \in Out(a)\}.$

Definition 7 - State.
The state of a SDFG $(A, C)$ is a pair $(\delta, \upsilon)$. Channel quantity $\delta$ associates the number of tokens in each channel of $C$ in that state. $\upsilon : A \rightarrow \mathbb{N}^R$ is used for keeping track of time by associating with each actor $a \in A$ a multi-set representing the remaining times of different firings of $a$.

The use of a multi-set for keeping track of the progress of an actor allows simultaneous firings of actors. This can be limited by adding self loops with a number of initial tokens equivalent with the concurrency degree. The dynamics of an actor is described by transitions. There are three types of transitions: start of firing, end of firing, in progress.

Definition 8 - Transition.
A transition of a SDFG $(A, C)$ from a state $(\delta_1, \upsilon_1)$ to state $(\delta_2, \upsilon_2)$ is denoted by $(\delta_1, \upsilon_1) \xrightarrow{\beta} (\delta_2, \upsilon_2)$ where label $\beta \in (A \times \{\text{start, end}\}) \cup \{\text{clk}\}$ denotes the type of transition.

- Label $\beta = (a, \text{start})$ corresponds to the firing start of actor $a \in A$. The transition may occur if $Rd(a) \preceq \delta_1$ and will result in $\delta_2 = \delta_1 - Rd(a), \upsilon_2 = \upsilon_1[\{a \mapsto \upsilon_1(a) \cup \{Y(a)\}]$. Meaning that $\upsilon_1$ with the value for $a$ replaced by $\upsilon_1(a) \cup \{Y(a)\}$ (is the multi set union).

- Label $\beta = (a, \text{end})$ corresponds to the firing end of actor $a \in A$. The transition may occur if $0 \in \upsilon_1(a)$ and will result in $\gamma_2 = \gamma_1 + Wr(a), \upsilon_2 = \upsilon_1[\{a \mapsto \upsilon_1(a) \setminus \{0\}]$. ($\setminus$ is the multi set difference).

- Label $\beta = (\text{clk})$ denotes a clock transition. It is enabled if no end transition is enabled and results in $\delta_2 = \delta_1, \upsilon_2 = \{a, \upsilon_1(a) \cup \{1\} | a \in A\}$ with $\{\} \cup \{\} = \{\} \cup \{1\} \cup \{\} = \{a\} \cup \{\upsilon_1(a) \cup \{1\} \cup \{\}\}$ returns a multi-set of natural numbers containing the elements of $\upsilon_1(a)$ reduced by one. (its purpose being to capture the progress of actors’ execution over time)

Definition 9 - Execution.
An execution of a SDFG is an infinite alternating sequences of states and transitions $s_0 \xrightarrow{\beta_0} s_1 \xrightarrow{\beta_1} ...$ for a designated initial state $s_0$.

The execution that gives the maximal throughput requires that each actor fires as soon as it is enabled and that all active firings make progress (meaning actors executing in parallel). This kind of executions is called self-timed execution.

Definition 10 - Self-timed Execution.
In a self-timed execution, clock transitions only occur if no start transition is enabled and only clock transitions that reduce the remaining execution time of all active firings can occur.

Definition 11 - Iteration.
Given a timed SDFG $(A, C, T)$ with a repetition vector $\gamma$, an iteration is a set of actor firings such that $\forall a \in A$ the set contains $\gamma(a)$ firings of $a.$
4.1.1 Throughput

Throughput of an SDF graph represents how often a selected actor produces output tokens. There are many algorithms for computing the throughput on an SDFG. They usually transform a SDFG into a Homogeneous SDFG (HSDFG) where all consumption and productions rates are 1 and then compute its maximal cycle mean (MCM [25]). The HSDFG is seen as a directed weighted graph with the weights on the nodes representing the execution times of actors and the weights on the edges represent the initial tokens on the channels. The cycle mean of a cycle in such a graph is calculated by dividing the sum of the weights of all nodes over the sum of the weights of all edges over that cycle. The maximum cycle mean over all cycles is the MCM. 1/MCM gives the maximal throughput the graph can achieve. Alternatively, this can also be done by executing a SDFG in a self-timed manner and dividing the number of firings a chosen actor over the number of clocks in one cycle of the periodic phase. It is proven in [2] that the state space of any SDFG with a so-called storage distribution (see next subsection) has a transient phase and a periodic phase (that has infinitively many transitions) with a finite number of states.

4.2 Storage Requirements

One important aspect related to allocation of storage for a SDF model is whether the model assumes a back-pressure mechanism in the system it describes. If back-pressure is available, the system has a mechanism to block (i.e. it cannot continue execution) an actor when the output channel where it is supposed to produce new data is full; the actor will be granted access to continue as soon as there is sufficient free space on that channel. With this assumption, the model is guaranteed not to consume more memory than what it was statically assigned and also no loss of data. Moreover, using the same memory sizing for a system without back pressure cannot provide same guarantees. When back-pressure is not available, actors will not suspend their execution if the channels where they have to output results are full, resulting in corruption or loss of data. In particular, we need to consider this case when analyzing GStreamer pipelines because they can have so called live sources that cannot stop their execution without suffering from data loss (example in Section 6.6).

Furthermore, when considering the throughput calculation for back-pressure models, it is enough to consider only the worst-case execution times for actors. Considering the worst-case delay that can occur gives guarantees on the minimum throughput. However, for non back-pressure models, the variations in the execution times if actors (bounded by a worst-case and a best-case execution time) must be considered in order to give a bound on the data that can be maximally produced on a channel by a producer before it can be consumed. Such a bound guarantees that no produced data is lost, even in the absence of a back-pressure mechanism.

The remainder of this Section is divided as follows:

- In Section 4.2.1 we recapitulate the buffer-sizing techniques for back-pressured systems by Stuijk et al [26].
- In Section 4.2.2 we recapitulate the buffer-sizing techniques for non-back-pressured systems by Salunkhe et al. [22].

4.2.1 Back-pressured systems

SDFGs have unbounded storage space which cannot be assumed for real implementation. However, channels storage can be bounded in several ways.

1. Use a memory shared between all channels: the required space will be determined by the maximum number of tokens stored at the same time during the execution. This can be used for single-processor systems in which actors can share the memory space and has been used as an assumption in [27]. An example implementation can be found in [28].
2. Use a separate memory for each channel (empty space from one channel cannot be used for another one).

The second method can be used in multiprocessor systems where memory cannot always be shared between processors. The channel capacity must be determined per channel over an entire schedule and the total amount of memory is obtained by their sum. However, this calculation is conservative in the sense that the needed memory calculated in this way is never lower than when memory is shared.

**Definition 12 - Storage Distribution.**

A storage distribution of an SDFG \((A, C)\) is a channel quantity \(\delta\) that associates with every \(c \in C\) the maximum number of tokens that can be stored on \(c\).

**Definition 13 - Size Distribution.**

The size of a storage distribution \(\delta\) is given by \(|\delta| = \sum_{c \in C} \delta(c)|.\)

Let’s assume a channel \((p, q)\) from actor \(a\) to actor \(b\) has \(d\) tokens initially. After \(n\) firings of \(a\) and \(m\) firings of \(b\), the channel will have a number of tokens given by the following equation:

\[
n \cdot \text{Rate}(p) - m \cdot \text{Rate}(q) + d
\]

We can rewrite this as:

\[
\left\lfloor \frac{d + n \cdot \text{Rate}(p) - m \cdot \text{Rate}(q)}{k} \right\rfloor \cdot k + d \mod k,
\]

where

\[
k = \gcd(\text{Rate}(p), \text{Rate}(q)).
\]

This shows that the number of tokens in a channel (the storage space) depends on the gcd of the rate at which actors \(a\) and \(b\) produce and consume tokens. This is called the step size of a channel.

The bound on the storage of each channel can be modeled in an SDFG \((A, C)\) by adding for each edge \((p, q) \in C\) from an actor \(a_1 \in A\) to an actor \(a_2 \in A\) an edge \((p_3, q_3)\) from \(a_2\) to \(a_1\) with the \(\text{Rate}(p) = \text{Rate}(p_3)\) and \(\text{Rate}(q) = \text{Rate}(q_3)\). The initial tokens on the edge \((p_3, q_3)\) determines the bound for the storage space of the edge \((p, q)\). The subscript "\(\delta\)" denotes elements used to model storage space. However, there is no memory allocated for these self-loops.

When considering the tokens consumed during for a firing, the ones from the edges modeling storage space should be included. This consumption can be seen as allocation of storage for writing the results of a computation. At the end of the firing, the actor will produce the output tokens. Again this includes tokens produced for the channels modeling storage spaces. This is equivalent to the release of space at the end of the firing.

The maximal throughput of an SDFG is limited by channel capacities since the firing of an actor may depend on tokens’ availability on its input channels that model the storage space. Therefore, by increasing the storage space of that channel, we may increase the throughput. This kind of dependency is called casual dependency.

**Definition 14 - Casual dependency.**

A firing of an actor \(a\) casually depends on the firing of actor \(b\) through the channel \(c\) if the firing of \(a\) consumes a token from \(c\) produced by the firing of \(b\) on \(c\) without a clock transition between the start of firing of \(a\) and the end of the firing of \(b\).

Casual dependencies in the periodic phase of an execution affect throughput since they might tamper an actor to fire repeatedly. This means that the throughput can be increased by resolving the dependencies between actor firings. Additionally, it is enough to solve them in one period, since these dependencies are equal in all periods. They can be captured with a dependency graph.
Definition 15 - Dependency Graph.
Given a timed SDFG \((A_\delta, C_\delta, \Upsilon)\) with \(\delta\) storage distribution and a sequence of states and transitions \(p\) corresponding to one period of its self-timed execution, the \textit{casual dependency graph} \((D, E)\) contains a node \(a_k\) for the \(k^{th}\) firing in \(p\) of actor \(a \in A_\delta\). The set of dependencies contains an edge iff there exists a casual dependency between the corresponding firings.

A cycle in a dependency graph show casual dependencies between actors.

Definition 16 - Casual Dependency Cycle.
A casual dependency cycle is a cycle in the dependency graph.

We are interested in casual dependencies of channels that model storage bounds because they mean that increasing the storage space of such a channel will increase the throughput.

Definition 17 - Storage dependency.
Given a SDFG \((A_\delta, C_\delta)\) and its dependency graph \(\Delta\), a channel \(c \in C_\delta\) has a storage dependency in \(\Delta\) iff there is a casual dependency in some dependency cycle of \(\Delta\) through channel \(c\).

Definition 18 - Minimal Storage Distribution.
A storage distribution \(\delta\) with throughput \(\tau\) is minimal iff for any other storage distribution \(\delta'\) with throughput \(\tau'\), \(|\tau'| < |\tau|\) implies \(\tau' < \tau\) and \(|\tau'| = |\tau|\) implies \(\tau' \leq \tau\).

All these concepts are needed for describing the algorithm that calculates all the minimal storage distributions.

4.2.2 Non back pressure systems

GStreamer can have both live and non-live sources. A live source means that it cannot be stopped from executing (similar to a non-back-pressure system). Then the buffer size calculated for a backpressure system will not suffice because of execution times variations which lead to higher buffer sizes requirements. If they are not allocated, a system without back pressure cannot guarantee no loss of data. Since products for medical domain have strict regulations for their certifications, we would be interested in finding buffer sizes that can guarantee this. The topic is treated in [22] for HSDFGs and we introduce here the assumptions and the conditions for this calculation. The definitions from the previous section for SDFGs are true also for HSDFGs because any SDFG can be transformed into a HSDFG ([25]).

We introduce some new definitions.

Definition 19 - Initial tokens function.
For a SDFG \((A, C)\) the initial number of tokens function \(d : C \to \mathbb{N}\) gives the initial number of tokens on the channel \(c = (i,j)\) from actor \(i\) to actor \(j\).

Definition 20 - Start time function.
For a time bounded HSDFG \((A, C, \breve{\Upsilon}, \Upsilon, \hat{\Upsilon})\) and its self-timed execution, the start time function \(s : (A \times \mathbb{N} \to \mathbb{N}) \times A \times \mathbb{N} \to \mathbb{N}\) with \(s(\Upsilon, a, k)\) gives the start time for the \((k + 1)^{th}\) firing of actor \(a\) in the execution by the following equation:

\[
\begin{align*}
s(t, a, k) = \max_{(x, a) \in C} \left\{ \begin{array}{ll}
\max_{s(t, x, k - d(x, a)) + \Upsilon(x, k), k \geq d(x, a)} & 0, \\
0, & k < d(x, i)
\end{array} \right. \quad (4.5)
\end{align*}
\]

Definition 21 - Finish time function.
For a time bounded HDGF \((A, C, \breve{\Upsilon}, \Upsilon)\), a given timing function \(t\), its self-timed execution and a start time function \(s\) then the finish time for the \((k + 1)^{th}\) firing of an actor \(a\) is given by the function \(f(a, k) = s(a, k) + t(a, k)\).  

25
Conditions

When no bounds are modeled in a HSDFG, then is not strongly connected and hence a periodic behavior cannot be guaranteed. Other conditions need to be imposed in order to have a guaranteed periodic execution of a graph and finite bounds.

Let G be a graph that is not strongly connected. We consider a set $SCC_G = \{SC_1, SC_2, ..., SC_n\}$ containing all strongly connected subgraphs of G. Then we can have a component graph $G_{SCC}$ defined by $(A_{SCC}, C_{SCC})$ for which $A_{SCC}$ has an actor $a_i$ for each $SC_i \in SCC_G$.

For each node in $G_{SCC}$ we can compute its Maximal Cycle Mean (MCM) $\mu$. Considering two adjacent actors ($a_i$ and $a_j$ connected by a channel $(i, j)$), the one with a higher MCM will set the low limit for its execution. This is called the induced MCM for the actor in the component graph. The induced MCM by $a_j$ is calculated with:

$$\mu_{idc}(a_j) = \max(\mu(a_j), \max_{(a_i, a_j) \in A_{SCC}} \mu_{idc}(a_i))$$ (4.6)

It is proven that a non-strongly connected data flow graph it is also characterized by a transient phase and a periodic phase but this does not guarantee buffer boundedness. The required condition is that given the source node in the $G_{SCC}$, its induced MCM is higher than any other induced MCMs by the other components:

$$\mu_{idc}(v_{src}) \geq \mu(v_i) \forall v_i \in V_{SCC}$$

For computing the buffer size bounds it is necessary also that the MCM for the WCET is equal to the MCM for the BCET of the subgraph corresponding to the source. (the formal conditions are presented in [22]). If this is not true, then intervals for calculating the buffer bounds (see next subsection) are ever increasing over iterations.

To summarize, for the models of the GStreamer pipelines the following conditions need to hold in order to guarantee finite buffer bounds and to calculate them:

- the source has to be dominant (have the highest execution time in the model) over all actors
- the WCET and the BCET of the source has to be equal.

Buffer size bound

Let us consider a self-timed execution of a timed HSDFG $(A, C, \bar{\Upsilon}, \Upsilon, \hat{\Upsilon})$. For a channel between actors $i$ and $j$ with $(i, j)$ initial tokens, the total number of tokens in the $k$th iteration is given by the number of firings of $i$ in the interval

$$[s(i, k), f(j, k + d(i, j))]$$ (4.7)

In order to obtain the maximum value (the buffer bound) for this channel we need to have this interval as large as possible and the most number of firings of $i$ in this interval. This would require to consider in the interval between the best case execution time for the $k$-th firing of an actor $i$ ($\bar{s}(i, k)$) and the worst case execution time of the $k$-th firing of the actor $i$ ($\hat{f}(j, k + d(i, j))$) and count the number of firings of actor $i$ in this interval ($\bar{s}(i, k'), \forall k'$). The bound is given by calculating the maximum over all firings over a channel.

$$\bar{b}(i, j) = \max_{k \in \mathbb{N}} \sum_{k'=0}^{\infty} \begin{cases} 1, & \bar{s}(i, k) \leq \bar{s}(i, k') \leq \hat{f}(j, k + d(i, j)) \\ 0, & \text{otherwise} \end{cases}$$ (4.8)

where $i$ has a finite number of firings under best case self timed execution ([22]).
4.3 GStreamer-SDF mapping

We explain here the configurations of GStreamer elements that need to be set in order to comply with the assumptions of SDF and enable a mapping between the two. Also, these parameters influence the behavior of the pipeline so they are considered the steering parameters of a pipeline that influence its performance and the memory allocation. Eventually, these configurations represent together with the predicted performance, the outcomes of the Performance Analysis in the architecture proposal.

4.3.1 Queues Properties

As explained in Section 3.3.2, placing a queue between two elements creates a thread boundary. This is one of the ways GStreamer is forced to create new threads and run elements in separate threads. We describe now the properties of the queue element that need to be set in order to obtain the buffer boundedness modeled with back edges and initial tokens in SDF.

Size

The queue has a set of properties for limiting the number of tokens it can store (equivalent to the bounds of channels)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max-size-buffers</td>
<td>Maximum number of buffers the queue can store</td>
</tr>
<tr>
<td>Max-size-time</td>
<td>Maximum amount of data the queue can store in nanoseconds (in case of video playing)</td>
</tr>
<tr>
<td>Max-size-bytes</td>
<td>Maximum number of bytes the queue can store</td>
</tr>
</tbody>
</table>

Table 4.1: Queue size related properties

Max-size-buffers specifies the size of the queue in terms of GstBuffers. This represents the property that matches the bound in terms of maximum number of tokens of a channel. The others (Max-size-time, Max-size-byte) properties should be disabled (set to 0).

Leaky

The leaky properties of queues defines whether the pipeline implements a back-pressure mechanism or not (the possible values are shown in Table 4.2).

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GST_QUEUE_NO_LEAK</td>
<td>Back-pressure enabled:</td>
</tr>
<tr>
<td>GST_QUEUE_LEAK_UPSTREAM</td>
<td>Back-pressure disabled - new buffers dropped</td>
</tr>
<tr>
<td>GST_QUEUE_LEAK_DOWNSTREAM</td>
<td>Back-pressure disabled - oldest dropped</td>
</tr>
</tbody>
</table>

Table 4.2: Queue properties for specifying the back-pressure in a pipeline

For the first value (GST_QUEUE_NO_LEAK), the streaming thread is blocked when the queue is full and waits until buffers are consumed from the element downstream. This is equivalent to having back-pressure enabled. However, this is possible only for sources that can be stopped (e.g. file sources). On the other hand, when a live source is used (a camera or network source), it
cannot be stopped or blocked if there is not enough data to store the date so therefore data will be lost. This is the case with the second and third values from Table 4.2 which specify how data should be dropped (older buffers should be replaced by the new arrived ones or new ones should be dropped).

4.3.2 Decoder max-threads

The decoder element has a parameter for specifying whether it can use multiple threads for decoding. This is the configuration that sets the auto-concurrency of the element as we assumed this for our modeling.

In this section we describe the mapping between GStreamer pipelines and SDFGs. Both the framework and SDF models are complying with the pipes and filter architecture making the mapping straightforward (summarized in Table 4.3). As previously mentioned (Section 4.1), an actor represents independent functionality that runs on an execution unit (a thread on a processor core). Because one or more elements can run on a same thread in GStreamer, the list of elements in a pipeline that run on the same thread is mapped on one actor. With this mapping, the number of queues in the pipeline (queues introduce thread boundaries, Section 3.3.2 ) dictates how many actors the model will have. If a pipeline has $N$ queues, then the number of actors in the SDF model is $2N + 1$. If the threshold properties are disabled, then the execution of the pipeline matches the self-timed of the corresponding SDFG. Depending on the placement of these queues some actors can encapsulate the sequential execution of several consecutive elements. For a one-to-one mapping between elements and actors this would mean that the execution of the actors is not self timed.

The next element in our mapping is regarding the data units (tokens passed between actors in SDFG and GstBuffers in GStreamer) used in communication between filters (elements or sets of consecutive elements in GStreamer and actors for SDF). The mapping one-to-one, i.e. one buffer in GStreamer. Furthermore, the channels between actors correspond to the queues in the pipelines. It worths mentioning here that SDFG allow cycles while in GStreamer the cycles are done through communication between elements through events and not by directly pushing GstBuffers. Furthermore, the production/consumption rates of actors correspond to he number of buffers being pushed on the source and sink pad of an element. Since the rates are considered to be fixed in SDF we needed to make some adjustments in GStreamer to meet this requirement. (explained in Section 5.1.2).

Coming back to our goals of finding a model to describe the relationship between the steering parameters of the system and its metrics. We conclude that the main steering parameters of a pipeline are the GStreamer equivalent of actors, i.e., the elements running on separate threads, ((Table 4.3)) together with the the queue sizes. The metric we are interested in is the throughput of the pipeline.
<table>
<thead>
<tr>
<th>SDF element</th>
<th>GStreamer equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>One or more linked elements running on the same core</td>
<td>Functionality or code to be executed</td>
</tr>
<tr>
<td>Token</td>
<td>GstBuffer</td>
<td>Data unit</td>
</tr>
<tr>
<td>Channels</td>
<td>Queues</td>
<td>Data dependencies/execution order</td>
</tr>
<tr>
<td>Rate</td>
<td>Number of buffers pushed/pulled on one execution of an element</td>
<td>Data units consumed/produced</td>
</tr>
<tr>
<td>Back-pressure</td>
<td>Queues maximum</td>
<td>Execution conditions of the actors/elements</td>
</tr>
<tr>
<td>Execution time of actor for firing once</td>
<td>Execution time of elements(s) for producing a buffer</td>
<td>Execution conditions of the actors/elements</td>
</tr>
<tr>
<td>Throughput of SDFG</td>
<td>Throughput of pipeline</td>
<td>Total number of firings of the sink of the pipeline over total execution time of pipeline</td>
</tr>
</tbody>
</table>

Table 4.3: GStreamer - SDF mapping
Chapter 5

GStreamer implementation, SDF modeling and the analysis

In this chapter we start from the requirements from Barco and build an application running a GStreamer pipeline. We describe this application in Section 5.1. This section includes the constituent elements, some details regarding implementation and describes the execution flow. Then, based on the mapping established in the previous chapter, we create a SDF model to map the pipeline (Section 5.2). In order to analyze the obtained model we use two algorithms existing in literature. We calculate the minimum achievable throughput of this model under various levels of concurrency and we present and apply algorithm for obtaining the throughput-buffer size trade-offs considering back-pressure available [26]. Finally, we present and apply an algorithm for calculating the buffer sizes in case of non-back pressure to guarantee no loss of data [22].

5.1 Barco use-case application and reference pipeline

In the first chapter (see Section 1.2) we briefly presented a set of requirements for an entertainment application for patients and explained the motivation of our work. In this section we present details about the implementation of this use-case. A high-level overview is shown in Figure 5.1.

![Figure 5.1: Use-case application overview](image)

The general idea of this application is to enable patients to watch different types of videos over a network. A server streams H264 compressed video content over UDP. It can broadcast content
either from a file or from a video camera. The client will connect to the server and receive the
video, decode it, perform some transformation before displaying it.

A GStreamer pipeline for this scenario is presented in Figure 5.2 and shows the steps that need
to be executed in more detail and the corresponding elements that should perform some processing
steps. One the one hand, the server is responsible for multiplexing between two sources (a file
and a camera), compressing the content from the selected source, divide it in packets for network
communication and then send them over an UDP connection to a client. On the other side, the
client application opens an UDP connection to receive the packets from the server, parses them
for restoring the content and then forward it to a decoder. After decoding, the application will
perform a color-space conversion and a resolution scaling before displaying it.

We continue in this section describing available solutions for having a data rate constrained
network connection and how we model this in the reference pipeline (Section 5.1.1). Then we
describe the processing elements of the reference pipeline explaining their main functionality and
role in the reference pipeline (Section 5.1.2). Thereafter, in Section 5.1.3 we present the process
view of the pipeline by describing its execution flow under various levels of parallelism.

![Figure 5.2: Overview Barco Use-Case Application](image)

### 5.1.1 Limiting the data rate of the data source in the application

Limiting the data rate of the source in a SDF model is one of the condition to enable calculation
of buffer-sizes for a non-back pressure system (proven by Salunkhe et. al. [22] and presented in
Section 4.2). For real applications, this is equivalent to having a traffic shaper element implemented
at the network connection level.

Traffic shaping can also be established by means of implementing synchronization between
client and server, so that the server will stop sending packets when the client cannot handle
more. This has been done [29] for MPEG streaming over UDP/TCP. In fact, this implements
a back-pressure mechanism over TCP/UDP. However, it also requires a controlled environment
(and changes) at both client and server side.

In COTS hardware, for example in CISCO routers [30], it is possible to enable "rate-limiting
features". According to CISCO’s documentation, they typically allow configuration for two types
of mechanisms:

- A shaper which typically delays excess traffic using a buffer (or queuing mechanism) to hold
  packets and shape the flow when the data rate of the source is higher than expected.
• A policer which typically drops traffic.

Note that a shaper is called "greedy" by Wandeler et al. [31] if it does not buffer the data longer than necessary (for example using CISCO’s real-time traffic classes). Wandeler et al. have also investigated how much memory is required for such kind of shapers in order to guarantee real-time traffic without data loss.

Consequently, a wide field of research has investigated predictable network streaming of real-time multimedia traffic which is beyond the scope of this thesis. For simplicity, we model a traffic shaper by means of a constant delay between video frames.

We are interested in modeling the performance of the client application since it represents the software that will run on a medical device for which we are interested in predicting its execution and memory requirements. In order to ease the modeling and the evaluation, we simplify the application in the following way: we remove the server side and replace it by a source on the client side that simulates the behavior of the server. To this extent, we use a file source that reads compressed video content from a file and forwards it to an element that inserts time delays between frames to simulate of the server. Consequently, the UDP source and the element responsible for restoring the video content from network packets is replaced by a file source element and a traffic shaper element. The simplified version of the client application is shown in Figure 5.3.

Besides simplification of the application model, having a delay at the source that allows us to control the amount of data being pushed enables us to fix the consumption and production rate of the pipeline (assumption required by SDF modeling). We set for each reading from the file to read exactly one frame (which is pushed as a buffer through the pipeline). In this way, we can have one to one production and consumption rates for each element in the pipeline. Essentially, this modification simulates a network connection with a constant and constrained rate (no bursts in the network traffic).

5.1.2 Processing elements of the reference pipeline

We present here the concrete GStreamer pipeline that we want to model as a GStreamer pipeline. It will be called the reference pipeline in the remainder of this paper. In the following subsections we present briefly briefly the implementation of each element.

![Figure 5.3: Reference pipeline](image)

The (file)source

The first element of the pipeline is a source element that reads data blocks from a specified path using read [32] function. The default block size is 4096 Bytes. This element will be driving the data-flow of the pipeline (is activated in push mode) and therefore it will create a thread for reading the content of the file in chunks of block-size (the property that specifies how much to read in one iteration) bytes and pushing them as GstBuffers on the source pad of the next element in the pipeline. This element will produce (every time it executes) a buffer that contains a compressed frame. This way, we enforce that all the elements in the pipeline will output a buffer for each buffer (frame) they receive, thus having fixed consumption and production rates (as required by SDF). The pseudo-code for the file source can be seen in Listing 5.1. The implementation includes also handling the case when the read block is smaller than the allocated buffer.
Listing 5.1: Filesource pseudo-code

At line 1 of Listing 5.1 the variable `bytes_to_read` is initialized with the total size in bytes of the file that contains an input video for the pipeline. This is obtained by calling the `sizeof` function with the location of the file to be read (`file_location`) as an argument. This value is used for establishing the stop condition for the loop described in the lines from 3 to 14. In each iteration of this loop, memory for a GSTBuffer that will hold a frame of the video sequence is allocated (Line 8). The function `allocate_buffer` reserves `block-size` bytes (equal to the size of a video frame) and returns the starting address of the allocated memory. At line 8, a frame is copied from the file to the memory allocated for the buffer (calling `read_from_file`). Then this buffer is pushed to the next element in the pipeline (the traffic shaper). If these two elements are executing on the same thread, then the source, will have to wait the delay introduced by the traffic shaper and then continue execution from this line. When the push is made, a callback will be executed and will change value of `block-size` to the size of the next frame in the video (if exists, if not is will be set to 0) so that every time the proper memory is read and allocated. After the buffer is pushed, the remaining bytes that have to be read is decreased by the number of bytes that were read for the current frame (`bytes_read`. This value is returned by `read_from_file` function at line 10). Eventually, the offset from where the reading function starts reading every time is increased by the size of the frame that was read, ensuring that next time it will start reading the next frame (if any). By the time all the frames were read and pushed as buffers through the pipeline, the `bytes_to_read` will reach 0 and the execution will end.

The traffic shaper
As mentioned at the beginning of this section, the traffic shaper of the pipeline introduces a fixed delay after each buffer. We implement this by using the GStreamer `identity` element which has a parameter called "sleep-time" for setting the length of the delay it adds for each received buffer before pushing it further. This, the identity element together with the file source simulate the behavior of a data-rate constrained network source.

The (caps)filter
The next element in the pipeline passes data without any modification and just limits the format of the content that can be passed between two elements (capsfilter). The main property of this element, `caps`, represents the type of data streamed between pads. Having this property fixed in the filter, forces other elements that can modify their capabilities to produce or receive certain data. Basically, the caps filter makes a check only upon changes of the format. In our case, this is when the pipeline is initialized. After initialization, its computation time is negligible (demonstrated in the evaluation Section 6.3). We will consider this element making a check on each buffer it
receives when we explain the execution flow later in this section but this check takes places only at beginning of the execution of the pipeline.

The (h264)decoder

Next to the filter we have a H264 [33] decoder. This element is based on the x264 [34] implementation of the standard. Since the source drives the streaming in this pipeline, this element is scheduled in push mode and buffers will be pushed on its source pad by the upstream element. Listing 5.2 describes the main steps that take place in the decoder.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>buffer received on source pad</td>
</tr>
<tr>
<td>2</td>
<td>add buffer to codec cache</td>
</tr>
<tr>
<td>3</td>
<td>do</td>
</tr>
<tr>
<td>4</td>
<td>parse data in decoder cache</td>
</tr>
<tr>
<td>5</td>
<td>if (parsing = success)</td>
</tr>
<tr>
<td>6</td>
<td>decode frame</td>
</tr>
<tr>
<td>7</td>
<td>if (decoding = success)</td>
</tr>
<tr>
<td>8</td>
<td>push buffer with frame on pad sink</td>
</tr>
<tr>
<td>9</td>
<td>}</td>
</tr>
<tr>
<td>10</td>
<td>} while (parsing = fail)</td>
</tr>
</tbody>
</table>

Listing 5.2: Decoder element pseudo-code

At line 1 a buffer is received from the push of an upstream element (the caps filter). This buffer is added to a cache memory used by the decoder (line 3). Then in the loop between lines 5 and 25 the decoder will try to parse the data (line 7). If this succeeds, it means there is enough data in the decoder’s cache to decode one frame. The data is removed from the cache and decoded (line 13). If the decoding is performed successfully then the decoded frame is pushed as a new buffer towards the next element. Then the execution continues, by trying again to parse the remained data in the cache. With our current configuration of the file source (that reads every time exactly one frame from the file sources) this loop will execute every time once. Therefore, for each received buffer, the decoder will eventually push one buffer (hence ensuring a fixed rate).

The (color-space)converter

The decoder outputs frames in YUV representation while the sink expects RGB. This is solved by adding this element into the pipeline and ensures that before the pipeline is executing, the elements negotiate their caps and the appropriate capability is set for each pad.

The Scaler

For the scaler we consider a simple model that has fixed execution time for processing each buffer. Similar to the traffic shaper, this element is implemented with an identity element that will simply delay the propagation of buffers with a fixed delay.
The (XImage)sink

The last element of the pipeline is the sink. In our case, this is a standard X Window System-based video sink.

5.1.3 Process View of the reference pipeline

Considering now the aforementioned pipeline we present (i) how the activation of each element is done and (ii) the execution flow of the elements in a pipeline. For the execution flow we take two cases. The simple case is with the the reference pipeline without any queues, having all the code executed sequentially. Secondly, we show the effect of adding queues (thread boundaries) to a pipeline.

Activation of processing elements

Setting the scheduling of the elements of a GStreamer pipeline (Section 3.3.2) is called pad activation. Before starting the streaming, the pipeline calls its element to activate and initialize for the data flow. Before this, they need to be added to the pipeline and linked. Adding the elements to a pipeline is done announcing what elements it has to manage and establishing links between elements’ pads. This way, when an element pushes a buffer on its sink pad, the source pad of the next element gets called and in turn runs the core processing function of that element (called chain function) that will eventually push data to the next element it is linked to. The GStreamer API calls these two steps are shown in Listing 5.3. The first function call adds to the pipeline a list of elements which are afterwards linked together with the second function call. The order of the parameters of both functions have to match the order of pipeline architecture (the order of elements in Figure 5.3).

`gst_bin_add_many (pipeline, Src, Traffic-shaper, Filt, Dec, Cap, Scaler, Sink);`

Listing 5.3: Function calls for adding the elements to the reference pipeline and linking them

The pipeline will activate its elements in a sink to source order. Each element will activate its pads from source pads to sink pads to make sure that when the sink pads are ready and already pushing data, the source pads are also ready to push data downstream.

By default all these elements can be activated in push-mode. The activation calls depicted in Listing 5.4. Subsequently, the first element to activate its pad is the XImageSink element which must activate its sink pad. This continues upstream in the pipeline, with the scaler, the color space converter element, the decoder, the traffic shaper and the caps filter each of them activating first heir sinks and their pads and ending with the file source element that activate its source pad. Note that in the implementation, each element in the pipeline must be activated, regardless of the activation mode (push or pull). The modes are which is in the end validated for consistency the pipeline.

```
ximagesink.sinkpad.activate_pad(); // will activate by default in push mode
scaler.sinkpad.activate_pad();       // push-mode
scaler.sourcepad.activate_pad();     // push-mode
csp.sinkpad.activate_pad();         // push-mode
csp.sourcepad.activate_pad();       // push-mode
dec.sinkpad.activate_pad();         // push-mode
dec.sourcepad.activate_pad();       // push-mode
traffic.sinkpad.activate_pad();      // push-mode
```

1 gst_bin_add_many (pipeline, Src, Traffic−shaper, Filt, Dec, Cap, Scaler, Sink);
2 gst_element_link_many (Src, Traffic−shaper, Filt, Dec, Cap, Scaler, Sink);

35
Until now, the reference pipeline has no queues. This means that all elements would execute sequentially in one thread. The main steps of the execution are shown in the diagram from Figure 5.4. We abstract here many aspects such as initialization, memory allocation or communication between elements. The sole purpose is to show the execution flow of the pipeline. This execution flow corresponds to the first scenario presented in Section 2.3.

The start of the execution is denoted by a black circle, with a red edge and a "Start" label. The black circle with a double red edge and the "End" label represents the end of execution of the pipeline, i.e. the moment when all frames of the video have went through the processing steps, reached the end of the pipeline and got displayed by the sink. Each activity is represented by a yellow rounded and usually corresponds to processing executed by an element from the pipeline. A yellow rhombus shows a conditional change of the execution flow. The label represents the condition, and the two outgoing edges (labeled with "yes" and "no") correspond to the path taken according to the evaluation of the condition. Finally, the edges with arrows represent the flow and order of execution of the activities.

At start we assume all elements are created, initialized, linked together, added to the pipeline and ready to process the stream (playing the video). The first step is done in the source element that reads one frame from a file that contains a compressed video. We abstract here the traffic shaper element (which inserts time delays) and consider it part of the reading. If the read outputs that EOF was reached, then it means there is no data to be read and hence no frames left to be processed. Therefore, the execution of the pipeline should end. Otherwise, if End Of File (EOF) was not reached, then the new created buffer will be checked for its capabilities (by the capsfilter element) to ensure that next elements can handle it. We assume here that the capabilities check is always successful to simplify the diagram. The next step is the decoding (executed by the decoder element). Afterwards, the decoded frame is scaled and then passed to the XimageSink element that will display it on the video output. Afterwards, the process starts again from the beginning and will eventually finish when the EOF is reached. As we can see, and all execution of elements is done sequentially. In the following subsection we explain how the flow changes when it is influenced by queues in the pipeline (i.e. enabling concurrent execution).

### Execution Flow of reference pipeline with queues

In this section we consider adding three queues (Q1, Q2 and Q3) in the reference pipeline as follows (Figure 5.5):

- **Q1** between the traffic shaper and the filter;
- **Q2** between the decoder and the color space converter;
- **Q3** between the color space converter and the scaler element;

For the queues, we consider the leaky property of each queue is set to NO_LEAK and therefore when a queue is full, the pushing thread will block (Queue Properties in Section 4.3), i.e., effectively implementing a back-pressure mechanism in software. Also, the execution flow assumes the queue...
sizes to be constrained, i.e. at some point queues can become full. Since some elements will execute now on different threads, the execution flow changes since some of the processing elements can be executed in parallel (assuming there is data available in the input queue and there is space in the output queue of processing elements). The new activity diagram for the execution flow of the pipeline is shown in Figure 5.6. This execution model maps the 4th scenario in Section 2.3 and to the principles of self-timed execution model of SDF from Section 4.1.1. Consequently, elements linked together in different thread bounds execute independently (represent actors) and they always execute as soon as there is data to process in their input queues (i.e. self timed execution). This enables us to use the SDF theoretical findings for analyzing GStreamer pipelines.

The file source, the filter and the traffic shaper and the caps filter execute on one thread, the decoder and the color space converter are running on one thread each and finally, the scaler and the sink are running on the 4-th thread. Accordingly, the diagram has 4 activities running in parallel.

The first parallel activity is for the filter, the traffic shaper and the caps filter. Again, we do not include the traffic shaper execution as we consider it as part of the reading. First, the file

Figure 5.4: Execution flow of the reference pipeline without concurrency
source element will read a buffer and if it reaches the EOF then it sends an End of Stream (EOS) buffer to the Q1 for signaling the streaming is over and after this buffer no frame will be further sent. Similarly, for any push to a queue the element will wait for free space (not presented in the diagram for brevity). Moreover, if EOF was not reached, then the buffer is checked for capabilities (always succeeds). Afterwards, the element is waiting for Q1 to have free space for a buffer and then it will send the buffer (this waiting actually takes place in the Q1’s sink pad code). After the push is made, the flow goes back to the beginning, i.e., it tries to read the next buffer and repeat the steps until EOF is reached.

The second activity that runs in parallel corresponds to the thread assigned to the decoder. In the beginning, the execution blocks until there is some data to be read from the queue (check if Q1 is empty). If the queue is not empty it means there is valid data for the Q1’s sink pad to read. When data is read, the buffer is removed from the queue (thus creating free space for one buffer so that elements from the previous thread can push buffers). If the buffer contains an EOS buffer, it means that no further data will be send to the queue and then the execution must be finished. The queues act in a First In First Out (FIFO) manner meaning that elements are pulled out in the order they were pushed in. This means that at this point, if the EOS was reached, all the elements before it have been already processed and the execution can end. Before ending, the EOS signal must be sent to the next elements so they could end their execution accordingly. In case the read buffer was a regular data buffer, then the buffer is decoded and as before, it waits for free space in a queue (in Q2) to push it forward.

A similar flow exists in the third (color-space conversion) and the forth (scaling and displaying) parallel activities. The only difference is that in the last one, since it is the end of the pipeline, after EOS, there is no need to send any signal further (there are no elements in the pipeline after the sink).

Finally, when all these 4 activities reach the same point (denoted by a red line perpendicular with the execution flow edges each activity), all frames have been processed (displayed) and execution is finished.

Note that we described the execution flow for the pipeline in its period phase. However, it takes some time before the processing pipeline is filled, i.e., during the transient phase not all 4 activations are able to execute completely (they will be blocked waiting for elements to be pushed on their queues).
5.2 SDF Analysis of the reference pipeline

In this section we create an SDF model corresponding to the reference pipeline and explain the mapping (Section 5.2.1). Furthermore we present two algorithms from the literature for analysis considering back pressure enabled (Section 5.2.2) and when back pressure is not present in a system (Section 5.2.3).

5.2.1 SDF-Model

As previously mentioned in Section 4.3, one actor is mapped on one thread in GStreamer. This means that some elements will be executed sequentially inside one thread. In particular, we map the processing elements that have small execution times on one actor. Looking at Figure 5.7,
Actor A corresponds to the execution of the first three elements of the reference pipeline (the file source, the filter and the identity element realizing the traffic shaper) and is considered the source of the model. Actor B corresponds to the decoder. Actor C to the color-space conversion element. Finally, the scaling and the sink are included in D. We map more elements on one actor since some of them have execution times that can be neglected (e.g. filter). We introduce 3 queues in the pipeline corresponding to channels AB (data passing between the traffic shaper and decoder), BC (data passing between the decoder and the color-space converter), CD (color-space converter and scaler). We explained how the file source pushes buffers that contain exactly one video frame (Section 5.1.2). This results in having each element producing one buffer out (token) on its sink pad for each buffer it gets pushed on it source pad (hence all rates are 1). The self edges on each actor with one initial token represent a limited auto concurrency of actors (i.e. there is only one instance of an actor executing at one time). Edges $AB_\delta$, $BC_\delta$, $CD_\delta$ and their initial number of tokens model the distribution size of the channels AB, BC and CD. The execution times per firings of each actor are included in Figure 5.7 and these are chosen based on real measurements of the corresponding GStreamer elements (explained in Chapter 6) and for the traffic shaper we consider a delay of 30 milliseconds and for the upscaler converter 20 milliseconds. In the remainder of the document we will assume that one token is equivalent to one frame (GStBuffer) and one clock is equivalent to 1 millisecond.

Figure 5.7: SDF model of the reference pipeline with limited auto-concurrency and a distribution $\delta = (AB_\delta, BC_\delta, CD_\delta)$

With this information we can determine the throughput-buffer size design space using the algorithm by Stuijk et al. [26] with back-pressure, which we will demonstrate in Section 5.2.3. Additionally, by considering WCET and BCET of the defined actors we can compute buffer sizing for the non back-pressure case. This is relevant when the source of a GStreamer pipeline is a live source (such as a live-camera). In fact, it is only relevant for the first part of the pipeline, because the part after the live source can be controlled by a traffic shaper and by the back-pressure mechanism of GStreamer.

5.2.2 Design Space Exploration for non back-pressure

We present now the algorithm by Stuijk et al. [26] that allows the finding of all minimal storage distributions of a SDFG and their throughput (proven in [2]) and we show how it applies to our reference pipeline. The main steps are presented in the pseudo code of Algorithm 1. Typically, it could start the exploration of the design space from a distribution $\delta$ which in theory could have no size (result in a deadlock and hence no throughput) on each channel ($\delta = (0,0,...,0)$). Since GStreamer cannot have empty queues, our starting point is a distribution having 1 token on each channel. The starting call of Algorithm 1 is with the first parameter a SDFG corresponding to the reference pipeline (the one in Figure 5.7 but without the back edges $AB_\delta$, $BC_\delta$ and $CD_\delta$) and the...
Algorithm 1: Algorithm for finding minimal storage distributions

1: **Inputs:** A SDFG G with storage distribution \( \delta \)
2: **Output:** Set D of pairs (storage distribution, throughput)
3: **Preconditions:** G is consistent
4: **procedure** \text{FindMinStorageDist}(G, \delta)
5: Create SDFG \( G_\delta \) that models \( \delta \)
6: Compute throughput \( \tau \) of \( G_\delta \)
7: \( D \leftarrow D \cup (\delta, \tau) \)
8: Compute dependency graph \( \Delta \) of \( G_\delta \)
9: Let \( S \) be the set of storage dependencies in \( \Delta \)
10: **for all** \( c \in S \) **do**
11: \( \delta_{\text{explore}} = \delta \)
12: \( \delta_{\text{explore}}(c) = \delta(c) + \text{setp.size}(c) \)
13: \( D \leftarrow D \cup \text{FindMinStorageDist}(G, \delta_{\text{explore}}) \)

storage distribution \( \delta = (1, 1, 1, 1) \). First step (line 1) is to compute \( G_\delta \) as explained in Section 4.2. The resulted graph is shown in Figure 5.7 (with \( AB_\delta = 1 \), \( BC_\delta = 1 \) and \( CD_\delta = 1 \)). Then, for this graph we calculate the maximal achievable throughput, \( \tau \), as described in Subsection 4.1.1. Figure 5.8 shows the transition state space for the SDF model of the reference pipeline with both transient and periodic phase. The initial state contains the initial storage distribution \((0,0,0,1,1,1)\). We do not include here the storage for the self edges since we do not consider them as part of the real memory usage. In the first state, only actor A can fire. Clock transitions have an extra annotation that mentions the number of clock periods that passed to reduce the number of transitions in the figure. The throughput is given by the number of firings of D in the periodic phase (1) divided by the number of clock cycles (55). Thus, the throughput is 0.018 (tokens/clock). Given that we considered 1 clock to be equivalent of 1 ms this means that the throughput of the pipeline is of 18.18 frames per second.

![Image](image-url)
the dependency graph as explained in Section 4.2.1 (Figure 5.9) (line 8). We observe that channel AB has a storage dependency (there is a cycle in the dependency graph between the first firings of A and B actors). Therefore, set S computed at line 9 will have only one element. The loop from lines 10 to 13, goes through all the channels that yield a storage dependency (all of them) and creates a distribution size ($\delta_{\text{explore}}$) that fixes each dependency by adding to the corresponding channel a number of tokens equal to its value from the repetition vector (equation 4.4 in Section 4.2.1). In our case, the new distribution size ($\delta_{\text{explore}}$) that will be used for the recursive call at line 13, will have the size of channel AB increased by 1. With this addition, the dependency for this channel will be solved and there will be no cycle in the dependency graph in the next iteration on this channel.

The algorithm continues recursively until all existing pairs are explored. At the end of the execution we will have 4 pairs: $((1,1,1), 0.018), ((2,1,1),0.028), ((2,2,1),0.031)$ and $((2,2,2),0.032)$ which define the Pareto space space for the reference pipeline (Figure 5.10).

Figure 5.9: Dependency graph of the reference pipeline SDF model with $\delta=(1,1,1)$

Figure 5.10: Pareto space for reference pipeline model
5.2.3 Buffer-sizing for non-back pressure

We present now the algorithm by Salunkhe et al. [22] for computing the buffer sizes for a system without back pressure (a GStreamer pipeline with a live source). As inputs, the algorithm takes an HSDFG that would model a pipeline, and the WCET and BET for each actor. Beside this, as a precondition, the source actor has to be the slowest in the graph and its WCET and BCET have to be equal (as explained in Section 4.2.2). The algorithm should return a distribution vector $\delta_{NB}$ so that the graph would guarantee that under the given conditions, no data would be dropped (the channels will never require bigger sizes than they are assigned). The algorithm computes for each channel of the model, the required size based on the formula from Equation 4.8.

\begin{algorithm}
\caption{Algorithm for computing the buffer sizes for Non-Back pressure Systems}
1: Inputs: A bounded HSDFG $G$ and an initial number of tokens given by a function $d$
2: Output: A storage distribution vector $\delta_{NB}$ for a non-back pressure model
3: Preconditions: $G$’s source is the slowest actor in the graph
4: procedure FINDSTORAGE$_{NB}(G, \delta)$
5: Let $C$ be the set of channels in $G$
6: $\delta_{NB} = \emptyset$
7: for all $c = (i,j) \in C$ do
8: $\delta_{NB}[c] = \bar{b}(i,j)$
\end{algorithm}

The algorithm expects a graph $G$ and a number of initial tokens on each channel. For GStreamer all channels are empty in the beginning ($\forall a \in A: d(a) = 0$). For calculating $\bar{b}(i,j)$ (equation 4.8) we use simulations of the self-timed executions of the graphs with the BCET and one for the WCET. For its execution we consider the graph for the reference pipeline (Figure 5.7) with the WCET and BCET in Table 5.1 as an input.

<table>
<thead>
<tr>
<th>Actor</th>
<th>WCET</th>
<th>BCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>B</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>21</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.1: WCET and BCET for the reference pipeline (in clocks)

We present in Figure 5.11 the values for the start and finish time functions (as defined in Section 4.2.2) obtained by simulation for this graph for the BCET execution (upper part in the picture) and WCET execution (lower in the picture). On the horizontal axis we denote the time passing (from left to right) and we present the value in number of clocks corresponding to the starting and finishing time of actors. For computing the maximum buffer size needed, it is sufficient to carry out the simulation until the end of the second periodic iteration, since all the tokens produced in the first periodic iteration will have been consumed in the second periodic iteration [22]. We discuss here only the calculation for the first iteration.

For example, for calculating the needed size for the AB channel we need to see how many starts of execution in the BCET simulation we have for the actor A in the interval $[\hat{s}(A, 1), \hat{f}(B, 1))$. According to the simulation this is delimited between the red dotted lines (Figure 5.11). We can see that in this interval, A starts a second firing (in the BCET) which means that the channel would require at least a size of two. Similarity, for the channel CD we count the number of starts of actor C in the BCET simulation in the the interval given by $[\hat{s}(C, 1), \hat{f}(D, 1))$. Delimited between the two blue lines in the figure and resulting again in a channel of size two. After calculating for all channels, we obtain that all channels should have size two in order to guarantee no data loss. This is consistent with the claim made by Salunkhe et al. that the buffer sizes obtained by
Figure 5.11: Simulation time line for the first periodic iteration. Marked overlapping firings actor C employing back-pressure can be smaller than or equal to the buffer sizes obtained for the non-back pressure systems.
Chapter 6

Evaluation

This chapter describes the evaluation part of this project. Section 6.1 describes the environment setup. Thereafter, in Section 6.2 we present the basic software setup used for experiments and the subsequent variations and what is their purpose for experimenting. Lastly, we present the results in Section 6.3 for the described experiments and discuss them briefly.

6.1 Environment setup

In this section we describe the hardware platform on which the experiments were performed and details about the software environment configurations. Furthermore, we continue with the description of the input sequences used for testing and explain the run-time metrics of interest and how we measure them.

6.1.1 Hardware and software

All experiments are performed on a x86-64 quad core processor system with two (hardware) threads per core (see Appendix B1 for complete hardware descriptions). The reference pipelines being evaluated in this section, require a number of software threads less than the number of hardware threads. Furthermore, the application running the pipeline is set to have the highest priority in the system and the threads get different processor affinities (in order to get executed on separate cores and simulate the behavior of actors in SDFGs). With this configuration we want to ensure that threads get executed as soon as possible in a similar manner actors get in a self-timed execution.

The details about the software (GStreamer and kernel versions, OS distribution) used for this project are are mentioned in Section 2, 3 and 4 of Appendix A.

6.1.2 Inputs

For our experiments we use a set of synthetic video sequences. For the synthetic tests we used the GStreamer’s videotestsrc element to generate video sequences containing different patterns (white, checkers, noise and zone-plate). Appendix B.0.5 contains information about how we generated them. All sequences contain 1000 frames and are compressed with the encoder from [34] with an arbitrary fixed configuration (described in Appendix C).

6.1.3 Run-time metrics

We describe here how we extract data at runtime from a GStreamer pipeline for the actors of its corresponding SDF models and how we measure the throughput and the latency.
Execution times of elements

For creating the data flow models of GStreamer pipelines we need the WCET and BCET of actors. As mentioned in Section 4.3, the execution of an actor is equivalent to the time it takes an element (or set of elements) to process a buffer. For exemplification, we consider an actor mapped on a processing element (as in Section 5.1.3) and in Figure 6.1 we illustrate the relevant moments in time for processing the $N^{th}$ frame (buffer) of a video sequence. $T_1$ is a time-stamp taken just before pushing the buffer on the element’s sink and $T_2$ is a time-stamp taken just after decoding is done and the new frame is pushed towards the next element. In this situation, the execution time for producing the $N^{th}$ buffer (equivalent to the firing time of the corresponding actor for the $N^{th}$ firing) is given by $T_2 - T_1$. If we do this measurement for all the frames of various video sequences (provided that the element runs for completely in this period) we can obtain a reasonable value for the minimum ($\bar{\tau}$), maximum ($\hat{\tau}$) and the mean execution times.

In order to introduce the timestamps at these points in the execution, we attached a callback on each elements’ pads. This callback is triggered every time push/pop of buffers occurs and in the execution of the callback we log the information accordingly. (Details about the callback mechanisms in Appendix B.0.1). Furthermore, for getting information about how many buffer are pushed/pulled in the pipelines, total execution time and similar information we used the gst-tracelib library. (Details in Appendix B.0.2).

If we have a set of elements that are mapped on the same actor, then for obtaining the WCET, BCET and the mean execution time we add the corresponding values of the individual elements mapped on the same actor. For example, if three elements (file source, caps filter and traffic shaper) are mapped on one actor, then the WET of that actor is the sum of WCET of each element that is mapped by the actor. Similarly, this is done also for BCET and the mean execution time. In the case of multiple elements mapped on one actor, we consider the sum of individual execution times to be appropriate since they execute sequentially (in one thread) and the time for transferring one buffer from one element to another is small compared to the processing time and can be neglected (communication is done by passing a pointer to main memory).

Timestamps are taken from the current value of the pipeline’s clock which is a hardware wall-time clock that starts from the moment the pipeline pushes the first buffer on its source. The wall clock is basically a monotonic system-wide clock. The access to the clock is made through Linux API specific calls ( clock_gettime()).

Pipeline throughput

The throughput of a pipeline is given by the number of frames at the output on the display over the total time to produce them. If we consider $T_1$ the time-stamp taken when the first buffer was read from the pipeline and $T_2$ the time-stamp when the last buffer was output to the sink then the total time to produce the frame is given by $T_2 - T_1$. The throughput is obtained by dividing the total number of frames (for our input sequences this is 1000) by $T_2 - T_1$. 
Memory Usage
We measure the memory usage of a pipeline by taking the sum of the maximum buffers each queue can store. For simplicity, we assume that each thread has its own memory that cannot be accessed by other elements. In other words, we assume that if a buffer gets pulled from a queue by a consumer element, then the position in the queue cannot be used by another element until the consumer produces a corresponding output. An accurate memory usage measurement would be to examine at each moment in time how many elements each queue has and calculate the maximum. However, we have chosen this conservative metric, because it matches the standard assumptions that we have adopted for our SDF performance models. In the future, it may be interesting to improve the SDF performance models as well as the corresponding memory-usage measures in order to obtain less conservative results.

In order to monitor the memory usage, we log all the operations on queues (push and pop) and use the Time Doctor tool [36] for visualization. It allows us to observe the memory usage of the pipelines at run-time. We log a push on the queue when an element is pushed on the queue’s source pad and an element being taken out when the buffer is pushed on the queue’s sink pad.

Converting SDF metrics to GStreamer metrics
When calculating the execution times of the actors being derived from the GStreamer elements, we consider a clock transition to be equivalent to 1 milliseconds. For calculating the WCET of actors, for each contained element we take the smallest integer greater than the maximum execution time of that actor (ceiling). Alternatively, for the BCET we take the biggest integer smaller than the minimum execution time (floor). This means,

\[
WCET(\text{actor}) = \lceil \text{maximum execution time for processing a frame} \rceil
\]

and

\[
BCET(\text{actor}) = \lfloor \text{minimum execution time for processing a frame} \rfloor
\]

Both are rounded to 1 millisecond boundaries.

6.2 Experimental setup
In this section we define our set of experiments. Furthermore, we explain what parameters we intend to modify in order to observe their influence upon the system. Our expectations are captured by hypotheses and checked with the theoretical results and the run-time results.

6.2.1 Basic setup
For all our experiments we analyze the pipeline reference described in Section 5.1. For the traffic shaper and the scaling element we chose a fixed execution time based on the goal of the experiment (and it is mentioned in the subsections). Furthermore, each experiment is repeated 5 times in order to increase the reliability of the obtained data. Furthermore, during some runs we obtain some isolated unexpected values for the measurements (i.e. the processing time for producing some frames is 5 times larger than the average execution time). This might be caused by some bug in our implementation or in GStreamer and was not investigated. The outlying values were not taken into consideration when computing these results.

6.2.2 Extracting execution times of elements
As previously mentioned, an actor can be mapped on one or more elements and the execution times of actors are calculated by adding the execution times of the corresponding elements. This is also the step in the work-flow from section 3.4. We will measure the execution times as mentioned in Section 6.1.3 for various inputs.
6.2.3 Pipeline partitioning

Once a pipeline is designed, we have to choose how many actors the corresponding model will have. Our assumption is that the number of actors is smaller or equal to the number of hardware threads (independent execution units). Now, we consider that for the reference pipeline we have a quad core architecture. Consequently, we will add two queues (Q1 and Q2) to the pipeline and analyze three different partitions in order to exhibit the influence of this choice and explain intuitively how to make for other pipelines. We call partitioning of the pipeline choosing between which elements to place queues (thread boundaries). In other words, once we decided how many actors will be used for modeling a pipeline, we have to decide what actors are mapped on what elements.

We assume actors A, B, C and the following partitioning variants:

- **P1**: Actor A - file source, traffic shaper, Actor B - filter, Decoder, Actor C: color-space converter, scaler and sink (top pipeline in Figure 6.2);
- **P2**: Actor A - file source, traffic shaper, filter, Actor B - Decoder, Actor C: color-space converter, scaler and sink (middle pipeline in Figure 6.2);
- **P3**: Actor A - file source, traffic shaper, filter, Actor B - Decoder, Actor C: color-space converter, scaler and sink (bottom pipeline in Figure 6.2);

We consider these partitionings solely for exploring various options of mapping elements on actors and how the minimal throughput achievable varies. Note that we assume only the back pressure case for this system (and therefore the source does not need to be dominating in the graph). The traffic shaper will have a 15 millisecond delay and the scaler will have 5 milliseconds. The other elements are considered to have the execution times from Table 6.2. For the experiments, we will consider that we set the queues with the highest distribution sizes obtained with Algorithm 1. The distribution sizes are $\delta_{P1} = (2, 2)$, $\delta_{P2} = (2, 1)$, $\delta_{P3} = (2, 2)$ for partitionings P1, P3 and P3, respectively.

**Hypothesis 1.**

Based on the extracted execution times of actors, we expect that P1 has the highest minimal achievable throughput among the partitionings P1, P2, P3. (considering the queue sizes that enables them to reach their maximum throughput).

Additionally we consider the 4th partitioning corresponding to the queues placement from section 5.1.2 (with Q1, Q2, Q3) that we will use in further subsections.

- **P4**: Actor A - file source, traffic shaper, Actor B - filter, Decoder, Actor C: color-space converter, Actor D: scaler and sink (in Figure 5.3);

For P4 we will consider the traffic shaper to have a 30 milliseconds execution time and the scaler to have 20 milliseconds. We will consider as the best partitioning, the one with the highest throughput. Inquisitively, this is obtained by balancing the load (execution of actors) among threads.

6.2.4 Throughput and buffer-size design space for reference pipeline

Once we decide upon a partition for a pipeline, we would like to see how the queue sizes (max number of buffers that can be stored) can influence the overall throughput. For this we consider the pipeline partitioning denoted as P4 in Section 6.2.3 and the two possible distribution sizes that theoretically would give two different (minimum) throughputs (presented in Setion 5.2.2). As mentioned in Section 4.3, GStreamer does not fully comply with the SDF memory allocation assumption. We recall that an SDF actor produces output tokens at the end of firing. This
Figure 6.2: Partitionings of reference pipeline with two queues

includes also the production of tokens on the channels that model storage space. In other words, while an actor is executing, the tokens being consumed by it is considered not to be available until the actor finishes execution. On contrast with SDF, a GStreamer element will first release the token from the queue (by pulling a pointer out) and then it will start its execution. Therefore, an element connected to the other end of the queue will see the corresponding slot available in the queue before the consuming element will even start its execution. This results in the fact that the status of the queues (how many elements they store at some point in time) might not show precisely the memory allocated since a queue might appear to have free space, but in fact the data being still used by some element. Another effect is that elements with consumption rates of 1 will not have their throughput influenced by increasing their output channel sizes by 1. For example, let’s consider a channel between two actors, from the downstream actor to the upstream actor (each mapped on a GStreamer element with data production/consumption of 1) and a channel that models the size of this channel with one initial token. Furthermore, let’s assume the channel is full (there is one element in it) and the downstream actor starts its execution (consuming it). The upstream actor cannot continue (has tokens on its input channel modeling the size) until the downstream actor finishes execution (and releases the token on the channel modeling the size). In GStreamer, since an actor will release first a queue element and then finish its execution, the upstream element will not have to wait for the finishing of the downstream element. As a consequence, since the difference between the two storage distributions from Section 5.2.2 is of one token, we would expect that in both cases (queues with sizes (1,1,1) and queues with sizes (2,1,1)) would yield the same throughput. We embody this as a hypothesis that we will check with the measurements.

Hypothesis 2.
The difference between the average run time measured throughput of the reference pipeline with
the Partitioning 4 for the (1,1,1) and (2,2,2) distributions should be negligible.

6.2.5 Comparison between back-pressure and non-back pressure

In this experiment we compare the buffer sizes calculation for back pressure and non back pressure for the SDF model of Partitioning 4. For this we look at the distribution size obtained by Algorithm 1 with for the highest throughput (calculated in Section 3.4). We enforce this configuration on the GStreamer pipeline and then measure the highest memory usage observable with TimeDoctor. We should expect that by setting the back pressure, the worst case memory usage is less or equal with the calculated buffer sizing.

**Hypothesis 3.**

Partitioning 4 of the reference pipeline with back pressure enabled and queue sizes matching the distribution (2,1,1) should yield at runtime a throughput greater or equal to the predicted throughput (calculated considering WCET of elements).

**Hypothesis 4.**

Partitioning 4 of the reference pipeline with back pressure enabled and queue sizes matching the distribution (2,1,1) should yield at runtime a worse case memory usage (maximum number of buffers queue stored over the execution) less or equal to (2,1,1).

Similarly, we calculate the buffer sizing for the partitioning P4 considering non-back pressure. Note that we consider that the graph meets the boundedness conditions, i.e. having a dominating source and same WCET and BCET for the source actor. The last condition is met by having a periodic execution of the source with a fixed period. To measure memory usage at runtime, we set sizes equal to 1000 (as if there is unlimited memory) and then measure the worst case usage of memory in the queues over time. We would expect that the run-time memory usage to be less or equal to the modeling result:

**Hypothesis 5.**

Partitioning 4 of the reference pipeline with no back pressure enabled and unlimited queue sizes should yield at runtime a throughput greater or equal to the predicted throughput computed with back pressure and the largest distribution (2,2,2) (calculated considering WCET of elements).

**Hypothesis 5 is based on the fact that the self execution of the SDF without back pressure is equivalent with the model having infinite resources (all storage dependencies solved).**

**Hypothesis 6.**

Partitioning 4 of the reference pipeline with no back pressure enabled and unlimited queue sizes (set to total numbers of frames of the input sequence) should yield at runtime the worst case memory usage of queues less or equal to the corresponding buffer sizings obtained by the non-back pressure calculation (2,2,2).

Hypothesis 6 basically claims that with the buffer sizing calculated with Algorithm 2 for the case of a non back pressure system, no data can be lost or corrupted.

6.2.6 Validation experiments

We know that for a HSDFG without a strict dominant source, its self-timed execution without back pressure needs infinite memory resources in order to guarantee that no tokens are corrupt or lost.

We want to see how a GStreamer pipeline behaves to this extent. For this, we again consider the partitioning from Section 5.1.3, a large size for each queue (equal to the total number of frames generated frames) and the file source pushing data as soon as possible (the traffic shaper adds no delays). With this setting, we analyze the evolution of buffers in each queue.
Hypothesis 7.
Partitioning 4 of the reference pipeline running with unlimited queue sizes and a non-dominating source will exhibit ever increasing of data in the queues.

6.3 Results

In the following subsection we present the results for the experiments described in Section 6.2.

6.3.1 Execution times of elements

We present here results for measuring the execution times for the filter, the color-space converter and the decoder element in Table 6.2. The XImage sink element is not included because its execution time for processing one buffer can be neglected. The main functionality of this element is to copy a buffer (frame) from a memory area to another memory area (XImage structure) which in turn gets displayed by the XImage server ([35]). Execution time of this element is much less under 1 ms. We have therefore considered its minimum as being 0 and maximum 1 in our SDF analysis. Similarly, the execution time of the file source can be neglected since it only measures the time for reading a number of bytes from a file but we will consider it to be 1 in both worst case and best case in order to meet the requirements for finite buffer sizes in the case of non back pressure.

The values are calculated as explained in Section 6.1.3, presented for each input sequence and they are expressed in milliseconds. For the traffic shaper and the up-scaler have fixed execution times (delays) as mentioned in the previous section (depending on the experiment).

<table>
<thead>
<tr>
<th>Input</th>
<th>Execution time (ms)</th>
<th>Input</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Avg</td>
</tr>
<tr>
<td>White</td>
<td>0.10</td>
<td>0.35</td>
<td>0.15</td>
</tr>
<tr>
<td>Checkers</td>
<td>0.09</td>
<td>0.24</td>
<td>0.12</td>
</tr>
<tr>
<td>Zone plate</td>
<td>0.09</td>
<td>0.26</td>
<td>0.14</td>
</tr>
<tr>
<td>White noise</td>
<td>0.10</td>
<td>0.35</td>
<td>0.14</td>
</tr>
</tbody>
</table>

(a) Caps Filter element (b) Decoder element

<table>
<thead>
<tr>
<th>Input</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>White</td>
<td>5.79</td>
</tr>
<tr>
<td>Checkers</td>
<td>5.79</td>
</tr>
<tr>
<td>Zone plate</td>
<td>5.78</td>
</tr>
<tr>
<td>White noise</td>
<td>5.80</td>
</tr>
</tbody>
</table>

(c) Color space converter

Table 6.1: Execution times of elements used in reference pipeline (in ms)

We summarize now the number of cycles for all elements in the pipeline that we use for calculating the execution times of actors for various partitionings. For each element we chose the maximum for all inputs and the maximum for all inputs. For the average value, we calculate the means of all averages from all inputs (for which we consider the floor value).
### Table 6.2: Summary of execution times of all elements (in clocks)

<table>
<thead>
<tr>
<th>Element</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>File source</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Caps filter</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Decoder</td>
<td>2</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>Color-space converter</td>
<td>6</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>XImage sink</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 6.3.2 Partitioning

We look now at run-time results of the three partitionings presented in Section 6.2.3. The values for the delays added by the traffic shaper and the upscaler element converter are the ones previously mentioned in Section 6.2.3.

The corresponding SDF graphs for the three partitionings are presented in Figure 6.3 (respecting the conversion explained in Section 6.1). We calculated the maximum achievable throughput of these three SDFGs using the algorithm in section 5.2.2 considering the WCET of elements. These maximum values represent the minimum guaranteed throughput of the pipelines. The throughput of these graphs is limited by the slowest actors. Consequently, Partition 1 (Figure 6.3a) yields the highest throughput (0.038 tokens/clock) since its slowest actor is faster than the slowest actors from the other partitionings. Consequently, Partition 3 (6.3c) has a higher throughput (0.027 tokens/clock) than the one obtained by Partition 2 (0.024 tokens/clock) but still not higher than the throughput of the Partition 1 (Hypothesis 1).

We measured the throughput for these instances and we summarize the results in Table 6.3. The run-time measurements are taken as the average over 5 runs and over all input sequences. This measurements confirms Hypothesis 1, P1 has the best throughput followed by P3 and P2 (confirming Hypothesis 1). Additionally, the measured throughputs are better than their corresponding minimal throughput obtained by data-flow analysis (with WCET) which is expected since our analysis is conservative.

Furthermore, this experiment confirms the intuition that increasing the throughput for a pipeline can be achieved by enabling parallelism and by choosing a balanced mapping of elements among threads. In our case, the minimal calculated throughput of the pipeline obtained with Partitioning 2 can be improved by 56% by choosing Partitioning 1 for the pipeline. For the run-time measurements, the improvement in throughput from Partitioning 2 to 1 is of 76%.

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Minimum throughput</th>
<th>Average Run-time throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.38</td>
<td>0.53</td>
</tr>
<tr>
<td>P2</td>
<td>0.24</td>
<td>0.30</td>
</tr>
<tr>
<td>P3</td>
<td>0.27</td>
<td>0.40</td>
</tr>
</tbody>
</table>

Table 6.3: Predicted minimum throughput and run-time measured throughput of three different pipeline partitionings

We translate the throughput into Frames Per Second (FPS) (with the conversion explained Section 6.1). This means that based on the analysis, Partition 1 is guaranteed to run with at least 38 FPS, Partition 2 with 24 FPS and Partition 3 with 27 FPS. If we consider to have a hard requirement of 25 FPS (Section 1), then only Partition 1 can meet this requirement. However, the end-users could not perceive any noticeable difference between the streaming of the three pipelines since all of them reach frame rates above the threshold for which humans can identify artifacts due to low frame rate.
6.3.3 Throughput and buffer-size design space for reference pipeline

Table 6.4 contains the measurements (average run-time throughput) for Partitioning 4 with for the two extreme distributions in Section 5.2.2 ((1,1,1) and (2,2,2)). The measured values are conservative with the minimum predicted ones (Hypothesis 3) but the difference between them is negligible (0.0002 tokens/cycle which is equivalent to 0.2 FPS) (Hypothesis 2).

<table>
<thead>
<tr>
<th>Distribution size</th>
<th>Predicted min. throughput</th>
<th>Avg. run-time throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1,1)</td>
<td>0.018</td>
<td>0.0317</td>
</tr>
<tr>
<td>(2,2,2)</td>
<td>0.032</td>
<td>0.0319</td>
</tr>
</tbody>
</table>

Table 6.4: Predicted minimum throughput and run-time throughput of three different pipeline partitionings

Consequently, this means that because of the way queues function, we are not able to opt for a lower throughput and less memory in the given mapping and implementation.

6.3.4 Comparison between back-pressure and non back-pressure

We present now the results regarding memory usage for the back pressure and the non back pressure versions of the Partitioning 4. Table 6.5 contains the distribution derived for back pressure.
and non back pressure, the worst case memory usage at run-time and the predicted one.

There are several observations regarding the result of this experiment. First of all, both calculations are conservative in respect to average throughput at run-time and the predicted minimal one (Hypothesis 3 and 6).

For the non back pressure, we measured a worst case memory usage of (1,2,2) over all input sequences. This is less than the value calculated with the non-back pressure analysis (Hypothesis 6). Reaching the maximum values would require variations between execution times that are less likely to produce at run time. The values obtained by the analysis represent the bounds that give guarantees that it is not required to allocate more memory than the calculated distribution indicates in order to ensure no loss of data (given the WCET and the BCET of elements and fulfilling the buffer boundedness conditions).

<table>
<thead>
<tr>
<th>Back Pressure</th>
<th>Distribution</th>
<th>Worst case run-time memory usage</th>
<th>Predicted throughput</th>
<th>Average run-time throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>(2,1,1)</td>
<td>(1,1,1)</td>
<td>0.028</td>
<td>0.0314</td>
</tr>
<tr>
<td>Disabled</td>
<td>(2,2,2)</td>
<td>(1,2,2)</td>
<td>0.031</td>
<td>0.0318</td>
</tr>
</tbody>
</table>

Table 6.5: Predicted minimum throughput and run-time throughput of three different pipeline partitionings

Figure 6.4 presents the queue behavior at run-time for the back pressure and the non back pressure pipeline captured with TimeDoctor. In Figure 6.4a are shown the activities of push and pop over the AB, BC, CD queues (Q1, Q2 and Q3 in Figure 5.5) for all execution of the pipeline for the noise input sequence. The maximum value for each queue oscillates between 0 and 1 for all three queues (number of stored frames) and never goes beyond that (being forced by the back pressure mechanism). Alternatively, in Figure 6.4b where we illustrate the Time Doctor for the non-back pressure case memory usage varies up to 2 frames per queue at most.

Figure 6.5 shows the analysis capabilities of the TimeDoctor tool. The screen shot here is zooming the moment of the execution when the queues in the non back pressure pipeline were storing more than one element. This tool could be used for observing how many queues were filled at a particular moment in time and what was the total memory usage of the pipeline.

6.3.5 Validation experiment

We present in Figure 6.6 the memory usage over a run with the zone plate input. As mentioned in 6.2.6 the source adds no delay after each buffer. The figure shows a screen shot of the TimeDoctor tool output. In the upper part of the picture we have the status of queue AB (see Partitioning 4 in Section 6.2.3 for queue placements and actor mapping) over time, i.e. the number of buffers a queue stores (basically shows the evolution of the memory according to push/pop events). We observe that the queue is being quickly filled in the beginning. This happens because the traffic shaper does not introduce any delay and therefore the file source pushes data with a much more higher rate than the decoder can consume (actor B). Therefore, the queue is accumulating data and reaches a peak of almost all frames in the video sequence (959 buffers) (Hypothesis 7). If the pipeline would run infinitely, it would require infinite memory for this queue. This illustrates the fact that not having a dominating source in a SDF model without back-pressure cannot guarantee finite channel sizes.

Furthermore, in the middle part of the figure, Queue BC has always at most 1 buffers in it. This happens because element B dominates element C (decoder element is slower than color space element) and therefore, between two executions of B, C has enough time to process the produced buffer. For the CD queue, a similar situation occurs as for the AB queue. Data on the channel is produced at a higher rate than the consumption rate and therefore we have again an increasing accumulation of data over time. Since the difference between the rates here are lower than for
the AB channel, the maximum peak is of 119 buffers. However, this would still require infinite memory usage if the pipeline would run for an infinite time (ever accumulating of data on the channel).

Again, visualizing the data through the tool makes it easier to understand how memory usage varies over time and can be useful in development.
Figure 6.6: Memory usage at run-time of a pipeline without a dominating source (TimeDoctor visualization)
Chapter 7

Conclusions

In the past decade, commercial-off-the-shelf (COTS) hardware platforms performance has increased being driven by the innovation in competitive consumer electronics markets. Implementing medical devices using such platforms together with modular software frameworks becomes this way more attractive in terms of efforts for developing, maintaining and supporting more product variants. Still, these devices, require to comply with strict regulations regarding their performance and must give real-time guarantees. In this work, we analyzed how a multimedia framework (GStreamer) could fit in the design process of this kind of products and to what extent synchronous data flow (SDF) models can be used for (i) inferring performance guarantees of the run-time and (ii) establishing a relationship between the software design choices and the real-time performance.

In this thesis, we first provided a high level architecture description of the framework envisioned by our industrial partner, Barco, for creating imaging application. This is composed of several tools and frameworks which should enable modular development of imaging applications for various hardware platforms in a highly integrated way. The presented frameworks and tools are well established and proven to be working in the industry (Qt, GStreamer), while the integration of these tools are either newly developed and may lack some functionality (e.g. qt-quickstreamer) or were not tested for this work but are implemented by the industry (e.g. the wrapper between Matlab algorithms and GStreamer plugins). This work is based on analyzing this framework and aims at exploiting the capabilities of today’s computer hardware to execute multiple functions concurrently.

We analyzed GStreamer pipelines at design time using SFD models obtaining minimal achievable throughput and queue sizing (for back-pressure and non-back pressure systems). We managed to identify the configurations needed to be set in order to have a valid mapping between the run-time and the model. The results obtained by existing analysis algorithms in the literature are consistent with the run-time measurements. GStreamer allows to enable/disable back-pressure in different parts of its pipelines so a mixed analysis is suitable. The non-back-pressure analysis can be performed only on the part of the pipeline that is directly connected to a live source, while the rest can be configured to have back pressure (thus reducing memory allocation). However, the performance analysis of a pipeline without back-pressure allows for a conservative analysis even when developers are unaware of the existence of a back-pressure mechanism (or forget to enable it) in GStreamer.

Many COTS networking hardware have support for predictable network streaming. We assumed the presence of such support and show how to model and simulate this behavior in GStreamer. Configuring and deploying a real network infrastructure with such support is beyond the scope of this work and requires further study.
7.1 Future Work

This project is a first step towards performance analysis of compositional algorithms implemented within a software framework (GStreamer) using data flow models. In this section we would like to mention some open issues of our work what could represent future research direction.

7.1.1 Scalability of GStreamer

We would like to analyze more complex pipeline structures. This could include a larger number of elements and branches. For example, GStreamer has been used to create pipelines with thousands of threads for gravitational waves research [37]. However, it is to be seen if GStreamer is also scalable when it comes to predictable performance. It would be interesting to look at more complex pipelines, with more threads that have to share the same processor. A follow-up challenge is how to make the execution of big pipelines predictable (i.e. scheduling of threads). We assumed in our experiments that the pipeline maps to as many actors as independent hardware execution units (hardware threads). In a real situation, it is likely that there less execution units than actors and therefore, the scheduling of actors on same core would need to be handled. This would also require an extension to the framework in order to enable setting priorities among elements.

7.1.2 Towards an open environment

A general assumption in our work is that the architecture of a pipeline does not change at run-time. However, GStreamer supports so called anytime pads which allow adding elements to a pipeline at run-time (e.g., based on the content of the stream). This could represent a situation that other data-flows models could be used (such as Scenario Aware Data Flows). In order to allow changes in the pipeline, we have several problems to be solved:

- each element must have an interface describing its consumption/production rate of data and it must specify at least a WCET
- an admission test, e.g. predicting the new performance based on SDF models, must be executed prior to accepting a change request to the pipeline, otherwise no performance guarantees can be given.

7.1.3 Increasing the accuracy of the model

Our modeling abstracts many details of the run time. The more details we include, the more accurate the outputs of the models would be. There are several options for increasing accuracy of the model:

1. better modeling of the execution environment (either with or without real software components).
2. better modeling of the software components.
3. combination of 1. and 2.

For improving the the modeling, memory access could be also included in the modeling for elements. This is a reasonable choice because embedded devices often have types of memory with different access/response time (for example: caches versus main memory). Furthermore, in our model, we assumed limited auto-concurrency of all elements. This is limiting because a decoder can run several threads for decoding and thus speeding-up the process.

Additionally, our reference pipeline had some synthetic elements (the traffic shaper and the scaler) and it would be valuable to have an experimental setup composed only of real elements. It is desirable to have “real” implementations of components, but also more precise behavioral
models can be placed in a pipeline. Often data flow graphs are highly application specific, whereas in this work we focused more to the architectural trade offs. In fact, we could have a hierarchy of models, real implementations or a mix, which could show the strength of the modularity of the framework. What kind of mixed setups are useful it would be interesting for further studying.

Finally, for any modifications brought to the modeling, the mapping between GStreamer and the models has to be further investigated and checked for consistency.
Bibliography


[38] “x264 encoder settings,” http://mewiki.project357.com/wiki/X264_Settings, 2012. 69
Appendix A

System Description

A.1 Hardware information

Figure A.1 describes the hardware resources of the platform where the experiments were ran. There are 4 Cores, each having two processing units (two hardware threads), one shared L3 cache memory for all cores and then for each core an L2 cache and then two L1s cache memories for the data and the instructions.

![Hardware Topology of the Platform Used for Experiments](image)

Figure A.1: Hardware topology of the platform used for experiments

<table>
<thead>
<tr>
<th>Architecture:</th>
<th>x86\64</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU op-mode(s):</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order:</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s):</td>
<td>8</td>
</tr>
<tr>
<td>On-line CPU(s) list:</td>
<td>0–7</td>
</tr>
<tr>
<td>Thread(s) per core:</td>
<td>2</td>
</tr>
<tr>
<td>Core(s) per socket:</td>
<td>4</td>
</tr>
<tr>
<td>Socket(s):</td>
<td>1</td>
</tr>
<tr>
<td>NUMA node(s):</td>
<td>1</td>
</tr>
<tr>
<td>Vendor ID:</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family:</td>
<td>6</td>
</tr>
<tr>
<td>Model:</td>
<td>58</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
</tr>
<tr>
<td>Stepping:</td>
<td>9</td>
</tr>
<tr>
<td>CPU MHz:</td>
<td>1200.000</td>
</tr>
<tr>
<td>BogoMIPS:</td>
<td>4788.94</td>
</tr>
<tr>
<td>Virtualization:</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache:</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache:</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache:</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache:</td>
<td>6144K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s):</td>
<td>0–7</td>
</tr>
</tbody>
</table>

### A.2 Running Kernel Version

```
Linux scr-850w 3.11.0-17-generic #31~precise1~Ubuntu SMP Tue Feb 4 21:25:43 UTC 2014
x86_64 x86_64 x86_64 GNU/Linux
```

### A.3 Linux distribution

<table>
<thead>
<tr>
<th>Distributor ID:</th>
<th>Ubuntu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Ubuntu 12.04.4 LTS</td>
</tr>
<tr>
<td>Release:</td>
<td>12.04</td>
</tr>
<tr>
<td>Codename:</td>
<td>precise</td>
</tr>
</tbody>
</table>

### A.4 Libraries and dependencies

```
QMake version 3.0
Boost version: 1.55.0
QT 5.2.1
C compiler GNU 4.6.3
CXX GNU 4.6.3
PkgConfig: /usr/bin/pkg-config (found version "0.26")
gstreamer-0.10, version 0.10.36
gstreamer-base-0.10, version 0.10.36
gstreamer-plugins-base-0.10, version 0.10.36
gstreamer-app-0.10, version 0.10.36
gstreamer-interfaces-0.10, version 0.10.36
gstreamer-video-0.10, version 0.10.36
gstreamer-pbutils-0.10, version 0.10.36
glib-2.0, version 2.32.4
gobject-2.0, version 2.32.4
doxygen version "1.7.6.1"
pkg-config version "0.26"
```
Appendix B

GStreamer

B.0.1 GStreamer Probing

Probing is a mechanism available in GStreamer to inspect, modify or drop the data that flows through pads. Basically, a probe will act as a callback that can be attached to a pad. At runtime, the probe will notify on various activities based on the parameters used when the probe was attached (i.e. buffers, events, queries).

A data probe is a callback activated when buffers are being pushed in or out from an element. This mechanism is a flexible method (it doesn’t require modifications existing elements) for measuring the time it takes an element to process a buffer. The callbacks are triggered by the GStreamer core before the buffers are pushed into in and out from the element. Therefore, measuring requires data probes to be added on both sink and source pads of an element. The difference between the source and sink timestamps represent the execution time of an element for processing one buffer. This method works only for production and consumption rates of 1 (enforced as mentioned in 5.1.2). We explain in more detail in Appendix B.0.4 the situation.

B.0.2 gst-tracelib

Gst-tracelib is a library that hooks into some GStreamer key functions and logs the behavior. When the application exits it can display some general statistics. Further analysis can be done based on the data written to the log file.

It provides statistics and information about the following

- data flow, messages, queries and events.
- caps
- pipeline topology changes
- resource usage

B.0.3 GStreamer DEBUG info

GStreamer has debug option for obtaining information about a running pipeline. It can be very verbose providing information about the status of the pipeline from caps negotiation to data pushing and memory allocation. All the information is timestamped using the pipeline’s clock which is a real-time clock. The code of each element needs was analyzed and the relevant debug messages are taken into account for the measurements. We use this method to get the timing information for double checking the probing method output.
B.0.4 Probing for multiple buffer-rates

We explain the probing measurements and why it doesn’t work for when the consumption/production rates are not 1. Let’s assume that in the pipeline from Figure B.1 the decoder needs 2 buffers pushed in before it can push a buffer out. We describe the execution flow of the pipeline in Figure B.1. First, the element will read first buffer from the file (Buffer 1), and then push it to the filter. Consequently, the filter will push it to the decoder’s sink. As described in Listing 5.2, the decoder will cache the new arrived buffer, parse the cache and decode a frame. If succeeded, it will push a buffer containing the decoded frame towards the next element. Otherwise, the execution will return to the first element (the source) and start again the same path until it reaches again the decoder. We assumed, that when the second buffer gets pushed on the decoder, the decoder will produce a buffer (Buffer 1 on the decoder’s sink). For measurements, we would attach a probe on the decoder’s sink and source. The execution time for producing a first buffer would be then TS3 - TS1 which would be wrong since it would includes also the time for producing the second buffer and filtering it. What we want to measure is T (TS3 - TS2). For fixing this issue we can either make sure that each buffer has exactly enough data for decoding one frame or analyzing a given pattern and change the measurement calculations accordingly. On the other hand, other elements that modify the buffer they get in on the fly, or they make a modification on it and just push it forward (e.g. color space conversion, up-scaling, etc.) can have their execution time measured using GStreamer probes

![Figure B.1: Data probes measurements](image)

B.0.5 Commands and configurations for generating video tests

For generating the synthetic video sequences we created a pipeline using `gst-launch` The following listing contains the general command:

```
gst-launch -1.0 videsrc pattern=${PATTERN} num-buffers=${BUFF} TO_PLAY ! video/x-raw, format=I420, width=1920, height=1080 ! filesink location=${YUV_OUT_NAME}
```

Where

- **PATTERN** represents the video pattern. We used "zone-plate", "checkers-8", "white" and "zone-plate kx2=20 ky2=20 kt=30"
- **BUFF_TO_PLAY** specifies the number of frames (1000 in our tests)
- **YUV_OUT_NAME** specifies the location where the file should be created

These files have a YUV raw format. They need to be compressed afterwards.

**Pipeline example**

We consider a pipeline that has to apply the following transformation to an input stream

- Gamma correction
• Hue and contrast adjustment
• Text overlay displaying some information
• Text overlay displaying streaming time

This can be done with in GStreamer using the existing plugins. The desired pipeline is depicted in Fig B.2 Here are some ways of realising this pipeline.

1. Using GStreamer command line tool *gst-launch*

   ```
gst-launch -1.0 videotestsrc ! \
    gamma gamma=0.8 ! \
    textoverlay text="Test Image" ! \
    timeoverlay ! \
    videobalance hue=0.7 ! \
    ximagesink \
   /
   ```

2. Using C code

   ```
#include <gst/gst.h>
int main(int argc, char *argv[]) {
  GstBus *bus;
  GstreamMessage *msg;
  GstStateChangeReturn ret;

  /* Initialize GStreamer */
  gst_init (&argc, &argv);

  /* Create the elements */
  source = gst_element_factory_make("videotestsrc", "source");
  gama = gst_element_factory_make("gamma", "gama");
  [...]

  /* Link the elements */
  pipeline = gst_pipeline_create();
  /* Add the elements */
  gst_bin_add_many (pipeline, source, gama, textoverlay, timeoverlay, sink, NULL);

  /* Get the bus */
  bus = gst_pipeline_get_bus (pipeline, GST_BUS_ALL);
  msg = gst_bus_poll (bus, GST_POLL_TYPE_READ_WRITE, GST_CLOCK_TIME_NONE);

  gst_object_unref (pipeline);
  gst_object_unref (bus);
}
```

3. In QML using GStreamer bindings

```
import QtQuick 2.0
import QuickStreamer 1.0

Rectangle {
    width: 800
}
```
We can see that this approach is convenient because the developer can create the user interface and the logic (underlying pipeline) in the same manner using QML. This should increase the productivity and ease the testing or changing an application.

Listing B.1: QML Hello World Application
Appendix C

x264 Encoding Options

We present here the settings we used in creating the input sequences and their values.

The three major picture types used in the different video algorithms are I, P and B. They are
different in the following characteristics:

- I frames are the least compressible but don’t require other video frames to decode.
- P frames can use data from previous frames to decompress and are more compressible than
  Iframes.
- B frames can use both previous and forward frames for data reference to get the highest
  amount of data compression.

bframes = 0

This sets the maximum number of concurrent B-frames that x264 can use. Our stream will have
no B frames. We want to force the decoder to require maximum execution time.

keyframes = 1

This sets the maximum interval between I-frames. Since we select it to be 1 and the number of
bframes=0 it means that our video sequence would have only I frames.

bitrate = 51200

x264 will attempt to encode the video to target the given bitrate as the overall average. The
parameter given is the bitrate in kilobits/sec. This should be chosen according to the network
connection and the quality needs.

The encoder has a long list of parameters and options of trading between quality of a video
and the size which is described at [38].