Preliminary design and validation of a modular framework for predictable composition of medical imaging applications

M.M.H.P. van den Heuvel†, S.C. Crăcană†, H.L. Salunkhe†, J.J. Lukkien†, A. Lele† and D. Segers‡
†Eindhoven University of Technology, Eindhoven, The Netherlands — ‡Barco N.V., Kortrijk, Belgium

***Abstract*—In this work, we present a software framework which enables us to analyse the performance of medical imaging algorithms in isolation and to integrate these algorithms in a pipeline, thereby composing a medical application in a modular manner. In particular, we show how public-domain middleware can be configured in order to achieve predictable execution of a use-case application. On this use case we applied formal analysis and we validated the promised performance on a real platform.**

I. Introduction

Many safety-critical products are traditionally developed using hardware-software co-design. For example, the software of medical imaging devices is often run on dedicated hardware. However, these days custom-off-the-shelf (COTS) hardware has become an attractive alternative for the development of safety-critical devices, because the performance and programmability have significantly increased over the past decade. This trend is driven by innovations in the consumer electronics (CE) markets. Nevertheless, there are challenges that slow down the adoption of CE technology for medical devices. Firstly, the product design becomes more software oriented requiring companies to implement their existing imaging algorithms in software. Secondly, the medical application of such devices requires strict certification regarding their performance.

Just like in CE, medical imaging algorithms typically impose real-time constraints with highly transient variations in the rendering of their streams. For CE devices, however, allocating a static amount of processing resources to video applications is unsuitable [1], because it leads either to frame misses or to an over-provisioning of resources. To enable cost-effective video processing, many quality-of-service (QoS) strategies [2] have been developed. These strategies estimate the required processing resources by the processing pipeline dynamically and then allocate resources for image processing which 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 [2]. However, for medical imaging applications, as considered in the current paper, the loss of video content and quality compromises are unacceptable.

In this paper, we analyze how a framework made from COTS hardware and COTS software fits the design process of medical imaging devices. The challenge with COTS hardware is that we miss a predictable execution architecture. Moreover, COTS software is not designed to give guarantees and often lacks real-time scheduling of the imaging algorithms that we use. We know however that in practice we may have good results.

This work was supported in part by the European Union's ARTEMIS Joint Undertaking for CRYSTAL under grant agreement No. 332830.

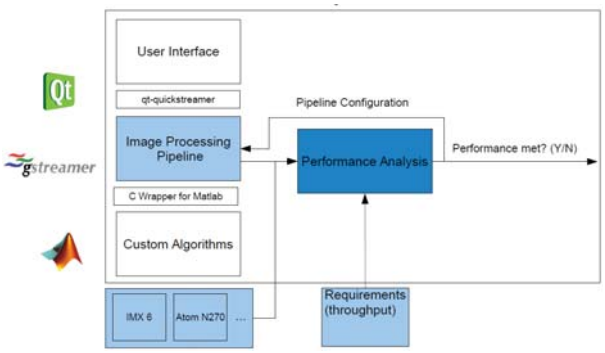

Fig. 1. Overview of tools and methods deployed in the engineering work flow in order to achieve predictable composition of medical video applications. For more details, we refer the interested reader to [3].

II. Modular software framework

In order to support modularity in the composition of a video application, we have decided to develop a flexible framework based on configurable public-domain middleware (see Figure 1), i.e., using Qt and gStreamer. The key idea behind this framework is that a video application can be decomposed into several imaging components (called plugins by gStreamer) with standard interfaces. These plugins can then be connected to each other, thereby creating a pipeline. Since Qt and gStreamer support different COTS hardware platforms, the combined framework allows for a reuse of imaging algorithms (wrapped in gStreamer's software plugins) in various setups and products.

The integration of Qt and gStreamer is work in progress. Firstly, our industrial partners are co-developing the Qt-quickstreamer plugin which extends the Qt Modeling Language (QML), so that QML can be used to compose an imaging pipeline from gStreamer plugins in an intuitive way. Secondly, Burks and Doe [4] investigated how custom imaging algorithms can be automatically imported from their development tools (Matlab Simulink) into a gStreamer plugin, i.e., an algorithm is wrapped into a plugin with a proper gStreamer interface. The integration of Qt and gStreamer is therefore expected to decouple the development of custom imaging algorithms and their integration.

Our aim is to integrate this modular software framework in the development flow of medical devices. We must therefore establish a predictable match between the execution model of gStreamer and the execution model being used at the stage of performance modeling. The remainder of the paper presents a case study in which prediction models are used to trade certain performance of an imaging application for its required processing resources during its real execution in our framework.

TABLE I
PREDICTED VERSUS EVALUATED RESOURCE USAGE FOR THE EXAMPLE PIPELINE, WITH OR WITHOUT A BACK-PRESSURED GSTREAMER IMPLEMENTATION.

Back-pressure	**Memory allocation** (number of frames per queue)	**Max. run-time memory usage** (number of frames per queue)	**Predicted throughput** (frames per second)	**Measured throughput** (frames per second)
yes	(2,1,1)	(1,1,1)	28	31.4
no	(2,2,2)	(2,2,2)	31	31.8

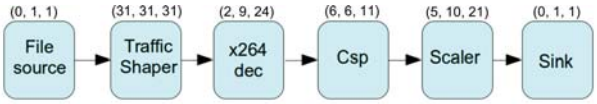

Fig. 2. An example video processing pipeline which we analysed using formal SDF analysis and which we implemented and validated in gStreamer. Each plugin in the pipeline has been benchmarked on a frame-by-frame basis, denoted by (BCET, ACET, WCET) in milliseconds.

III. USE CASE: FROM PERFORMANCE MODELS TO RESOURCE ALLOCATION AND VALIDATION

In this section, we model and implement an H264 client (see Figure 2). Since the software has to run on a medical device, we are interested in predicting, controlling and validating its execution time and memory usage. We therefore want to follow a standard design practice in which we control concurrency and memory usage to influence response times and throughput. Table I gives an overview [3] of the predicted performance and the real performance of such a controlled pipeline.

A. Experimental setup

We measure and validated the performance of our example pipeline on a X86-64 quad-core system. Each core offers two hardware threads. The example pipeline requires a number of software threads less than the number of hardware threads.

The threads are scheduled by Ubuntu 12.04 LTS (Linux kernel 3.11) and controlled by the gStreamer 0.10 and Qt 5.2 frameworks on top. The application running the pipeline is set to have the highest priority in the system and the threads get unique processor affinities (bound to separate cores). With this configuration we ensure that threads get executed as soon as possible, i.e., as mandated by our prediction models.

We fed the pipeline synthetic video sequences, generated using GStreamer's videotestsrc element (an open-source H264 encoder). They contain different patterns (white, checkers, noise and zone-plate). All sequences contain 1000 frames.

B. Constraining the data input stream

We compare two techniques to process all data in real time, i.e., without data loss and with finite sizes of queues. We therefore use a data source that reads compressed video content from a file. Some platforms (including gStreamer) support a synchronization mechanism, called *back pressure*, that suspends the data source when its output buffer is full and prevents data from getting overwritten. Alternatively, when the data source is uncontrollable, a traffic shaper can control the amount of data being pushed into the processing pipeline.

Synchronization may also be established over a network connection [5], so that the server stops sending packets when the client cannot handle more. This requires application-level streaming protocols on top of standard networking stacks, which need to be implemented and maintained. Alternatively, (without back-pressure support) the data source must constrain

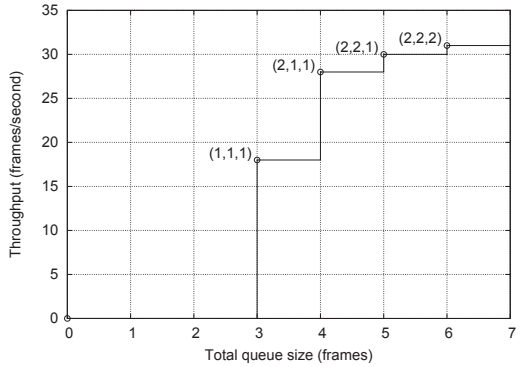

Fig. 3. Pareto optimal storage distributions of queues (AB, BC, CD) in a back-pressured example pipeline.

the amount of data being pushed into the pipeline. Some COTS network hardware is able to limit the data transmission rates by means of prioritization and buffering of specified real-time data [6]. We have implemented a traffic shaper in software as a gStreamer plugin in order to simulate streaming behaviour.

Our traffic shaper consumes and produces exactly one video frame periodically by inserting time delays between video frames. After the traffic shaper, we apply gStreamer's x264 plugin for decoding video frames, gStreamer's color-space conversion (Csp) and a synthetic spatial up scaler, which generates a random delay. These plugins all execute in a self-timed manner. Finally, the sink displays the processed video frames on the screen. For each of these gStreamer plugins, we have measured their execution times on a frame-by-frame basis for various video content; Figure 2 shows the best-case (BCET), average-case (ACET) and worst-case (WCET) execution times.

C. Concurrency control and allocation of processing resources

A gStreamer pipeline can be mapped onto several threads by explicitly placing queues between processing plugins. The plugins that are mapped upon the same thread execute in a static order, so that their execution times add up. A total of three queues, called *AB*, *BC* and *CD*, are placed after the traffic shaper, x264 decoder and Csp, respectively. With a certain positioning of queues, we can model the pipeline using the synchronous-dataflow (SDF) formalism.

An SDF graph allows us to compare the two algorithms by Stuijk et al. [7] and Salunkhe et al. [8] for computing the queue sizes and the corresponding throughput of the pipeline for setups with and without back pressure. The advantage of having a back-pressure mechanism is that waiting times of threads can be traded for throughput. Additional buffering at appropriate places in the pipeline may allow threads to work ahead and thereby increase the throughput. Figure 3 shows the Pareto optimal buffer allocations of our example pipeline obtained using the algorithm of Stuijk et al. [7].

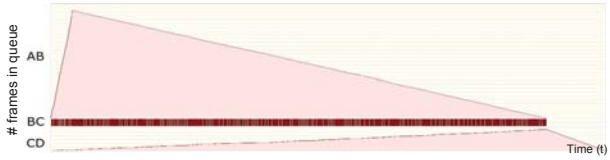

Fig. 4. Snapshot of unbounded memory usage of an unconstrained pipeline.

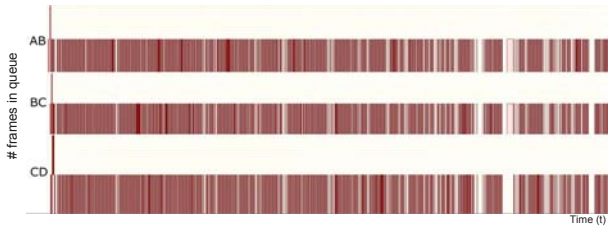

Fig. 5. Snapshot of controlled memory usage of a non-back-pressured pipeline.

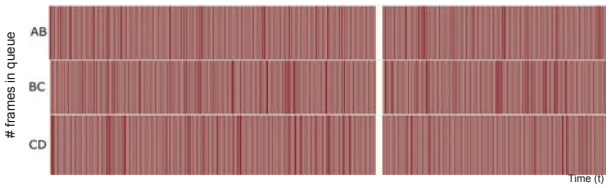

Fig. 6. Snapshot of controlled memory usage of a back-pressured pipeline.

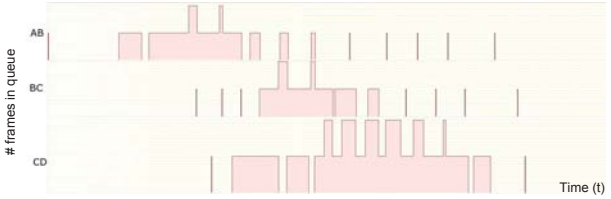

Fig. 7. Zoom in on the initialization phase of a non-back-pressured pipeline.

We recall that without back-pressure one must constrain the throughput at the input of the pipeline in order to bound the application's memory requirements. Moreover, plugins must execute in a self-timed manner, because delaying their execution may add buffer requirements to avoid data corruption. For such constrained applications, Salunkhe et al. [8] have proposed an algorithm to determine the Pareto point corresponding to the highest possible throughput. They use life-time analysis of data in the buffers based on the BCET and WCET of plugins in order to optimize the queue sizes[1]. In order to apply their algorithm, our traffic shaper limits the throughput at the input.

D. Performance validation

As shown in the methodology overview in Figure 1, the performance analysis is said to feed back configuration parameters to the application. The measured execution time parameters are the basis for a queue placement strategy, as tacitly applied in the previous subsection, and then allows us to mathematically predict trade offs in worst-case queue sizes and minimal throughput. We now validate the real-time memory usage and the real throughput of the pipeline (see Table I).

In gStreamer we log the number of buffered frames by instrumenting push and pop events of the queues in the pipeline; each buffer has the capacity of storing a video frame. Buffer access may or may not be be guarded by back pressure[2].

First, we look at a scenario of uncontrolled memory usage in which both our traffic shaper and gStreamer's back-pressure mechanism are disabled. In this scenario, the entire file is read from disk as fast as possible and stored into memory (see Figure 4). Since file readings have negligible WCETs compared to the later processing steps in the pipeline (see Figure 2), this experiment shows that, as can be expected, the memory storage requirements are proportional to the input size.

Secondly, we monitor the controlled memory usage for our pipeline (with and without back pressure). Figure 5 and Figure 6 show the number of frames $[0..2]$ stored in the queues. We confirmed that in both cases all frames in the file were actually being displayed at the output, i.e., both with and without back-pressure we report the absence of data loss. Table I reports the worst-case occupancies of the queues. Even without back pressure, the queues appear to be sized tightly and conservative.

As shown in Figure 5 and Figure 6, however, the queues in the pipeline are only occasionally fully occupied. Figure 5 also shows that initially the file reader works ahead one frame when back pressure is disabled. Figure 7 zooms in on the initial phase of the non-back-pressured pipeline. After an initialization phase, the execution pattern stabilizes and follows a repetitive order as dictated by our periodic traffic shaper.

[1]BCETs are irrelevant with back-pressure, because a delay of the earliest start time of plugins on new data can be enforced, which enables tighter life-time analysis based on just WCETs (see [8] for more details).

[2]The snapshots are created from the logged event traces with TimeDoctor [9].

IV. Conclusions

This paper presented a software framework for predictable composition of medical video applications. We configured middleware software in a way that the video pipeline is forced to execute closely in accordance with our formal application models. Formal (dataflow) analysis has been demonstrated on a case study in which we obtained optimized parallel executions of imaging algorithms by controlling execution delays and allocating memory appropriately. Since our initial experiments indicate that we can predict the performance of applications accurately, we consider our software framework a promising solution for the future design of medical streaming applications.

References

[1] D. Isović, G. Fohler, and L. Steffens, "Timing constraints of MPEG-2 decoding for high quality video: Misconceptions and realistic assumptions," in *Proc. ECRTS*, July 2003, pp. 73–82.

[2] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "QoS control strategies for high-quality video processing," *Real-Time Syst.*, vol. 30(1-2), pp. 7–29, 2005.

[3] S. C. Crăcană, "Modular composition of imaging applications on commercial-off-the-shelf programmable hardware platforms," Master's thesis, Eindhoven University of Technology, Aug. 2014.

[4] S. D. Burks and J. M. Doe, "Gstreamer as a framework for image processing applications in image fusion," *Proc. SPIE*, vol. 8064, pp. 80 640M–80 640M–7, June 2011.

[5] G.-M. Muntean and L. Murphy, "Feedback-controlled traffic shaping for multimedia transmissions in a real-time client-server system," in *Springer, LNCS, ICN Networking*, 2001, vol. 2093, pp. 540–548.

[6] E. Wandeler, A. Maxiaguine, and L. Thiele, "On the use of greedy shapers in real-time embedded systems," *ACM TECS*, vol. 11(1), pp. 1–22, 2012.

[7] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *Proc. DAC*, 2006, pp. 899–904.

[8] H. Salunkhe, O. Moreira, and K. van Berkel, "Buffer allocation for real-time streaming on a multi-processor without back-pressure," in *Proc. ESTIMedia*, Oct. 2014.

[9] M. Rutten, "TimeDoctor Version 1.4.3," May 2013. [Online]. Available: http://sourceforge.net/projects/timedoctor/