# Increasing Efficiency of data-flow based Middleware Systems by Adapting Data Generation

Herwig Guggi and Bernhard Rinner
Institute of Networked and Embedded Systems
Alpen-Adria Universität Klagenfurt and Lakeside Labs, Austria
Email: {firstname.lastname}@aau.at

*Abstract*—**Many data-flow oriented applications are based on the pipe-and-filter concept. This paper presents an improvement of the state of the art for multi-threaded pipe-and-filter processing architectures. We present a novel approach for adapting the time of data generation in the pipeline where adjacent pipeline stages exchange information about the current utilization. We compare our approach to a traditional input data driven pipeline and achieve a significant reduction of the processing delay and required memory consumption. The improvement of the presented system is based on self-adapting the data generation rate in the processing pipeline. This adaptation results in two key efficiency improvements: (i) the reduction of the time data elements spend in the pipeline and (ii) the reduction of the memory requirement for communication buffers. These improvements are of special interest for reactive and interactive multi-camera applications where short delays of the image processing pipelines is often required. The presented approach enables any data-flow based application to execute with reduced memory usage, reduced execution delay and the highest possible data-rate.**

***Keywords*—*data-flow processing; middleware system; data generation; adaptation; multi-camera systems; pipe-and-filter architecture***

## I. Overview

During the last years, more and more applications are executed within a network of devices. Therefore, design patterns are used to improve the software development for distributed systems. One of these concepts is the pipe-and-filter approach as described in [1]. In this case, a number of filters which are responsible for data processing are connected via pipes. The pipes are responsible for transferring data between filters. Based on this architecture it is possible to create a distributed system by executing some of the filters on remote devices.

The general requirements for processing systems are always a high data-rate, low memory consumption and a low processing delay. The performance that is achievable by an application mainly depends on four components. First, the hardware used to execute the code. Second, the algorithms used for processing the data. Third, the assignment of the individual filters to hardware devices. Fourth, the architecture for triggering the execution of the individual filters. In this paper, we will concentrate on the fourth component.

Finding an optimal architecture depends on a number of different factors. These factors are first the number and the possible combinations of individual filters (number of pipeline merges and splits). The architecture is further influenced by the execution time of the individual filters, the transmission time of the pipelines and the complexity of the algorithms. All these factors vary during runtime and the challenge is to find an architecture to find an optimal solution for all possible cases. Within this paper, we demonstrate that all these factors can be reduced to a few parameters that can be easily measured and enforced. By controlling these few parameters we show that it is possible to achieve a number of positive effects. First, it is possible to execute any pipe-and-filter architecture with reduced memory usage. Second, the time a data element spends in the pipeline is reduced. Third, the maximal data-rate is still provided. We further describe how the system can adapt its configuration to compensate changes in the execution of the single elements.

In this paper, we show how the reduction of the required main memory and the reduction of execution delay (time spent by data element in the pipeline) can be realised by adaption of the execution time of the data-source. The data rate of the source is adapted to the data rate of the bottleneck in the system. Therefore information about the optimal time to execute the data-source is backwarded from the bottleneck through all processing elements. The source then adapts its execution time according to the measured and the desired value. This approach is distributed and enables self-adaption with no need for a central component.

The optimisation of distributed systems is an important field to study as a lot of research currently focuses on this field. Smart cameras are one example of this trend. They combine image sensing, processing and networking. While single cameras can be used to trigger events and support a human observer in a surveillance system [2] or perform vehicle detection and speed estimation [3], distributed smart camera networks offer an even higher benefit [4]. They can be used to detect obstacles to avoid collisions [5] or perform a cooperative tracking with local image analysis [6]. Usually a middleware system is used to organise the distributed execution and the communication between devices. In cooperative scenarios, it is of special interest to use a middleware that is able to execute the application with low overhead for different system configurations. As the configurations may change during runtime, a middleware system has to support online adaption of parameters. One example of such a dynamic system is described by Esterle et al. [7] where they use a socio-economic approach for online vision graph learning and tracking handover in smart camera networks.

As more and more high-level tasks are required to be executed on these smart cameras, it is important to provide an efficient execution of these tasks.

The rest of the paper is organised as follows: The next section concentrates on the related work to the topic of data-flow based execution models. The section starts with theoretic approaches and continues with middleware systems that use a similar data-flow structure to trigger execution. Section III analyses the benefits of an ideal executed pipeline. This section further analyses the parameters that can be used to evaluate the properties of a pipeline. In Section IV, we explain the necessary modifications of the execution model to optimise the system performance. We define a system as optimal when the highest data-rate with lowest possible memory usage and lowest possible CPU usage is achieved. Section V describes the required steps of adaption to keep the system in an optimal configuration if parameters or the data-flow structure of the system changes during runtime. Section VI presents insights into the implementation of the system and shows measurement results of the proposed system and a reference implementation to demonstrate that the theoretic improvements can also be applied to real implementations. The last section concludes the paper.

## II. Related systems

The theoretical background of data-flow oriented processing frameworks is based on Kahn process networks [8]. Kahn process networks are models for distributed computing where processes communicate through unbounded unidirectional FIFO queues. The processing network provides deterministic behaviour and does not depend on computation or communication delays.

As unbounded queues are practically impossible (due to limited memory on real devices), Lee et al. [9] extend this model. They introduce bounded queues. As a consequence of bounded queues, a method to prevent message drops if a queue is full was required. Some mechanisms to overcome this problem are described. One mechanism is to block the data-providing process as soon as the queue where this process writes the data to reaches a threshold or is full. The blocking is necessary to keep deterministic behaviour of the system. In certain conditions, the blocking of processes can lead to deadlocks.

Canella et al. [10] implement and test three different approaches which implement the semantics of Khan Process networks on Network-on-Chip architectures. Also these systems provide deterministic behaviour. They block processes as soon as queues are filled higher than a threshold.

CORBA [11] provides methods for remote method invocation. Two methods, synchronous and asynchronous calls are possible. This middleware system is not comparable to data-flow concepts where data from element "a" is analysed by element "b" and then forwarded to element "c". The infrastructure provided by CORBA can be used to implement a data-flow based system, but this is not supported by the base system. The timing for method invocation and the decision between synchronous or asynchronous calls has to be made by the programmer. This requires detailed knowledge of all elements and their needs (e.g., if an algorithm needs every picture or if pictures can be dropped). Mechanisms for changing parameters such as the execution time based on the system load or the current context are missing.

Gstreamer [12] runs applications as pipelines. The pipelines are normally executed on a single device and in a single thread. Once started, pipelines will run in a separate thread until you stop them or the end of the data stream is reached. Streaming data are passed between elements in the pipeline with buffers. Buffers are created by the data provider and read by the data consumer. A reference counter is introduced to find out about the number of references currently accessing the data element. As soon as no one is using the buffer any longer, the buffer will be destroyed. This setup is good for playing or converting a video or audio file as every element from the data source is executed till the end of the queue where it is either displayed or stored.

The manual of the Gstreamer framework also argues that in some cases it makes sense to use threads. Their example is the playback of an multimedia stream where they want to visualise the images at the same time as the audio is replayed. This framework provides so called "queue" elements that force the use of threads. In this case, a classic producer / consumer model is used. This makes data throughput between threads thread safe and also acts as a buffer. These elements can be configured for specific uses. Lower and upper thresholds can be defined. If the buffer is lower than the lower threshold, it will by default block the output. If the upper threshold is reached, either the input is blocked (default) or data can be dropped. This system works good for pre-captured sources. In the case of live data, it would make sense to provide a mechanism that automatically adapts the frame-rate of the data producer to those of the consumer.

Systems more focused on a specific application are for example middleware systems for smart cameras. They usually focus on distributed processing and fusion of results [13]. Rinner et al. [14] discuss requirements on middleware for distributed smart cameras and services such a middleware has to provide. For their application - the tracking of an object with multiple cameras, they found the agent-oriented paradigm to build flexible and self-organizing applications to suite best. The following references show other examples for middleware systems built to support the developer of distributed smart camera applications. All of them are focused on data-flow oriented applications.

HIVE [15] creates pipelines called "swarms" out of single elements called "drones". Each drone is executed in its own thread. Two data transfer models are provided to transmit data from one drone to another. One is called synchronized and the other is called streaming data transfer.

In the case of streaming data transfer, a pipeline (or "swarm") is created and it is up to the programmer to set their parameters in a way that the execution speed is the same for all "drones" in the "swarm". There is no mechanism provided to react on context changes such as a change in the processing load that would reduce the speed of a single filter. If the input queue of a filter becomes full, new elements are dropped.

The other option, the synchronized data transfer, should ensure that a processing pipeline operates at its maximal capacity and does not waste unnecessary bandwidth. It works in a way that each drone only requests data from its provider as soon as this drone has finished with the processing for the last data. After receiving the data, the drone will process this data element and request the next data as soon as the processing is finished.

The problem about this mechanism is that even if the data transfer is called synchronized, the threads of the drones are not synchronized which means that it can happen, that a source drone produces more data than a filter drone can process. This would lead to frame-drops in case of a video encoding. In the case of live video-data this would mean that images are at least generated that will never be processed which means a waste of processing resources. Another drawback is the added delay that is introduced by requesting the next data after the last data has finished processing. The delay is even higher if a whole image has to be transmitted via the network.

Schriebl et al. [16] describe a system where every block has an output memory where its results can be accessed by subsequent blocks. To maintain consistency of the stored data, access to the memory is guarded by a lock that is passed between the producing and consuming block similar to a token. Blocks can form chains of arbitrary length where each pair of blocks is connected by a shared memory and a lock.

This mechanism ensures that the whole pipeline is always filled. Assume a producer (A) and a consumer (B). As soon as B finishes computation, A may provide new data to B. This data can already be pre-calculated while B processes the last values from A. With this system, the execution speed of the whole processing pipeline is automatically reduced to the speed of the slowest component and no data element will be dropped. In the case of recorded video sources, every frame will be processed with the pipeline nearly filled (only the filters after the bottleneck will be free from data for some time). This mechanism also works for live-video sources as the capturing rate is automatically adapted to the one of the processing rate of the bottleneck. It is not mentioned how filters with more than one input are handled.

Moreland [17] presents in his paper a survey about visualisation networks. The behaviour when modules get executed is described as a primary feature of visualisation pipeline systems. According to this paper, all visualisation pipelines generally fall under two execution systems: event driven and demand driven.

According to their definition [17], an event-driven pipeline launches execution as data becomes available in sources. When new data becomes available in a source, that source module must be alerted. When sources produce data, they push it to the downstream modules and trigger an event to execute them. Those downstream modules in turn may produce their own data to push to the next module. Because the method of the event-driven pipeline is to push data to the downstream modules, this method is also known as the push model. The event-driven method of execution is useful when applying a visualization pipeline to data that is expected to change over time.

A demand-driven pipeline launches execution in response to requests for data. Execution is initiated at the bottom of the pipeline in a sink. The sinks upstream modules satisfy this request by first requesting data from their upstream modules, and so on up to the sources. Once execution reaches a source, it produces data and returns execution back to its downstream modules. The execution eventually unrolls back to the originating sink. Because the method of the demand-driven pipeline is to pull data from the upstream modules, this method is also known as the pull model. The demand-driven method of execution is useful when using a visualization pipeline to provide data to an end user system. For example, the visualization could respond to render requests to update a GUI.

## III. Analysis of Pipeline Execution

This section describes the benefits that can be achieved by optimising the data-flow of a pipe-and-filter architecture. Typical requirements for a software is to provide a high data-rate with a low memory consumption. Some systems have the additional requirement of a low processing delay. Control applications for example rely on a very short reaction time which requires a low processing time. To control a crane and use a distributed smart camera application as described in [5] to detect location, orientation and dimensions of obstacles in the scene, it is necessary that the result data about the obstacles is delivered within guaranteed time bounds.

This analysis is based on a pipe-and-filter architecture. Figure 1 visualises an example of such an architecture. Pipe and filter architectures consist of two main components, first the filters, which are responsible for producing, processing and consuming data. In Figure 1, the filters are shown as circles. The numbers inside the circles represent the filter IDs. The values $t$ which are also displayed for each filter are an example of their processing time. The processing time of a filter is the period between start of processing till the result values are produced. In Figure 1, the data-inputs are visualised with arrows pointing towards the filters and the outputs are shown as arrows pointing out of the filters. Each filter may have any number of inputs but can only produce a single output type. Each input can only receive data from a single output but an output can forward data to any number of filter inputs. Source filters or data producers are filters having no input. Sinks or data consumers are filters with no output. In the current example, the filter with ID 1 is a source filter and the filter with ID 14 is a sink filter. Filters for this paper are assumed to be single rate elements meaning that they can only process data if at least one element is present on each input.

Pipes are the second component of a pipe-and-filter structure. These components are responsible for transferring data from the output of one filter to the input of another filter (represented as arrows in Figure 1). Pipe-and-filter architectures are also often used in distributed systems as the filters are self-contained and can be executed on different devices. The pipes which represent the data-communication take care of the data-transfer between devices. For communication, each filter has an input buffer to store data-elements for the next processing period and an output buffer to store results of previous calculations while they are not forwarded to the input buffers of the connected filters. Pipes transfer available data from output buffers to input buffers of other filters and delete the data from the output buffers. A combination of filters connected with pipes is called pipeline. For the current analysis, we allow all acyclic directed graph structures with a single source filter.

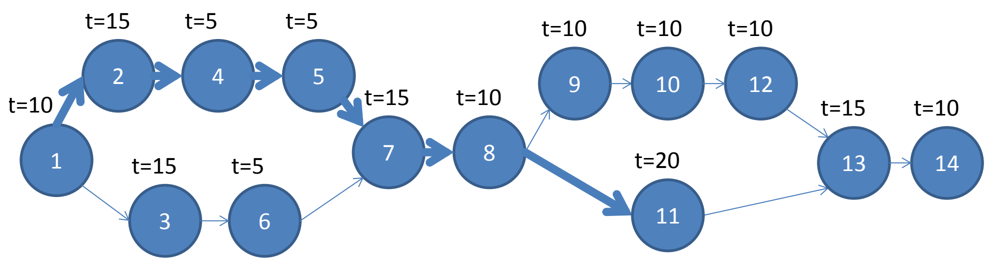

Fig. 1. A sample pipeline with a number of filters, their processing times. The filters which belong to the bottleneck-pipeline are connected with thick arrows.

The first step in preventing the system to waste resources is to ensure that no produced data-element is dropped. This requires a synchronisation between the filters in the pipeline which can be realized by blocking filter executions if buffers inside the pipeline are full. This can be easily achieved by placing a blocking buffer at the input of each filter. The filter will be automatically executed as soon as new data is present and the source of this filter can provide new data as soon as this buffer is not completely full anymore. This mechanism guarantees a maximum possible data-rate. As this architecture already provides the maximum possible data-rate for a synchronized system and is used in systems as described in [16], we use the blocking-filter structure as a reference design which we will later use to compare our system to.

Assume a set of filters $f_i$ where $i$ represents the ID of the single filter and each of these filters requires a certain processing time $t_i$ to process its task. In a synchronized pipe-and-filter architecture, the minimum delay between two data elements at the end of the pipeline equals the processing time of the slowest filter $t_b = max(t_i)\forall i$. This slowest filter is called the bottleneck-filter. This effect comes from the fact that all filters after the bottleneck have to wait for data from this filter before they can continue processing. This further means that the maximum rate at which a pipeline can produce or process data is always the data-rate of the bottleneck.

As the bottleneck is a limiting factor of the system, we extract a critical-pipeline (which we call the bottleneck-pipeline) from the whole pipeline that is defined as the path with the highest sum of processing delays from the bottleneck-filter to the source-filter. In Figure 1 the bottleneck-pipeline is marked with thick arrows.

This is the part of the whole pipeline that limits the performance. Therefore it makes sense to further analyse the different effects on this part of the pipeline as they will also be present on the rest of the pipeline.

The bottleneck-pipeline will always be a linear pipeline with a number of filters before the bottleneck and the bottleneck at the end. This linear pipeline can now be defined by the number of filters before the bottleneck $n_b$, the processing delays of the filters before the bottleneck $t_{b,i}$ and the processing delay of the bottleneck $t_b$. For the analysis, we can further replace the $t_{b,i}$ by a single average delay value $t_{avg} = (\sum t_{b,i}) \div n_b$. We can now replace the filters before the bottleneck that used to have different execution times by $n_b$ filters all having the same execution time $t_{avg}$. This reduced pipeline now consists of $n_b$ filters before the bottleneck where each filter takes $t_{avg}$ time to process a data element and the bottleneck filter at the end with $t_b = max(t_i)\forall i$.

We can now see in the bottleneck-filter that the producer filter is per definition faster than the bottleneck-filter. In the reference system, a new data-element is produced as soon as possible. This can be achieved by producing new data as soon as the output buffer is empty and new data-values can be stored. We propose a strategy that produces new data as late as possible. The realisation of this strategy is further described in Section IV. The benefits are described in this section.

The minimum delay between two data-results has already been described as $t_b$. We will now analyse the time that a single data-element takes to pass through the whole bottleneck-pipeline. Therefore we distinguish two cases: the case when the pipeline is empty and the case when the pipeline is full. In the case of an empty pipeline, the time it takes to process a single data-element is the sum of all processing times: $T_{min} = (n_b \times t_{avg} + t_b)$. This equals the minimum time it takes to process a single data element. As the producer and the filters before the bottleneck are able to produce more data-elements than the bottleneck can handle and the producer provides new data as soon as possible, the buffers before the bottleneck will start filling up to the time when all buffers before the bottleneck will be filled. This effect can later be seen in Section VI. From this moment, the time to process a data-element is automatically increased to $T_{max} = (n_b+1) \times t_b$ which means that each data-element will now remain in every filter before the bottleneck for the same time as the bottleneck-filter takes to execute a data-element. This means that there is a potential for optimisation in the part of the pipeline before the bottleneck.

Figure 2 shows the overhead that is introduced for different number of filters $n_b$ (between one and ten) and different times $t_{avg}$ where the times are expressed as fraction of the bottleneck-time $t_b$. This means that in the case of 10 filters before the bottleneck and an average execution time of these filters of half of the execution time of the bottleneck filter, it takes five times longer to process a new data element than in the optimal case. The optimal case is calculated as $n_b \times t_{avg}$.

The number of data-elements in the pipeline is also higher than the optimum. The optimal number of elements in the pipeline is the number which ensures that the bottleneck receives a new data-element to process as soon as the last processing has completed. This number can be calculated with the formula $e_{opt} = \lceil \frac{n_b \times t_{avg}}{t_b} \rceil$. The number of additional data-elements in the pipeline and therefore in the main memory is shown in Figure 3. With a relative execution time for the filters before the bottleneck of $0.4$ and $10$ filters before the bottleneck, $6$ elements more are in the bottleneck than there would be in the optimal case.

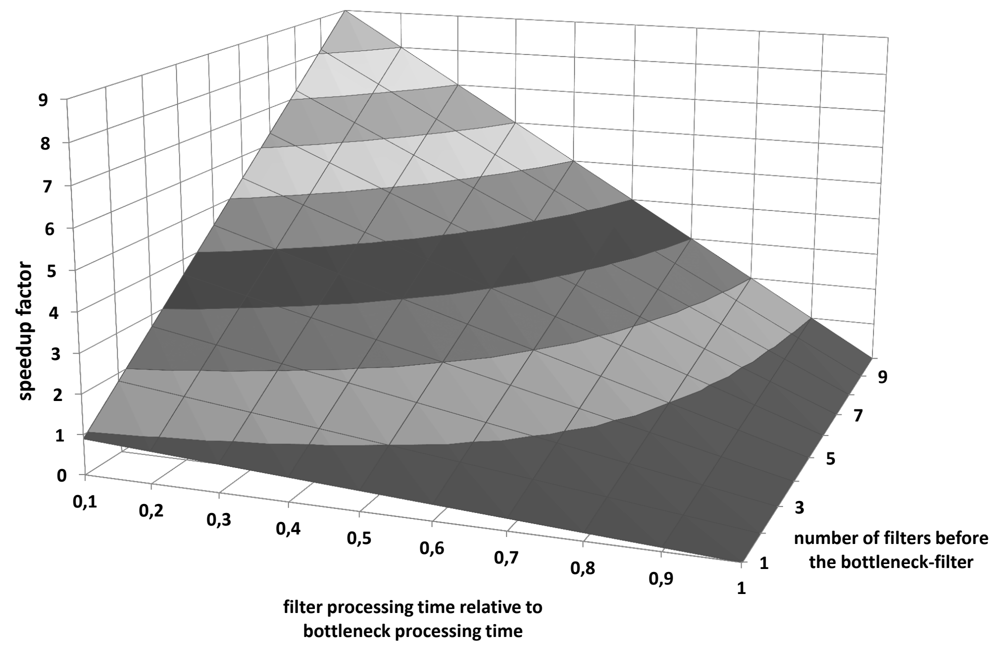

Fig. 2. The factor of which the time a data element spends in the pipeline is reduced compared to the reference architecture.

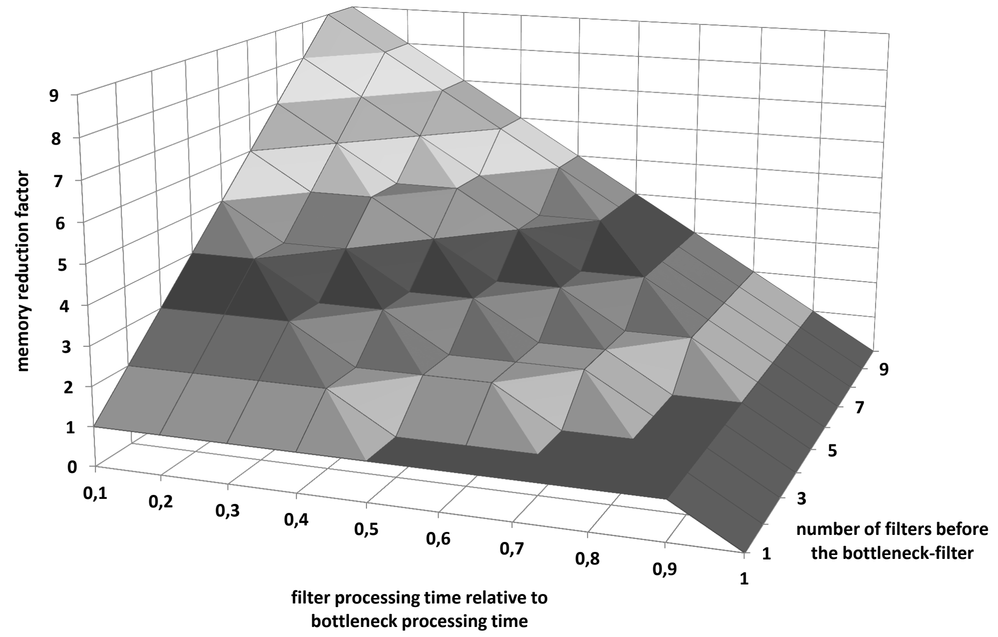

Fig. 3. The factor of which the number of data-elements in the pipeline is reduced compared to the reference architecture.

In the case of the reference system, at the moment when the bottleneck filter finishes the execution it automatically accesses the next data-element from the input buffer triggering the execution of all previous filters. This means that at this point all filters preceding the bottleneck start their execution. Figure 4 shows the number of active filters during the relative processing time of the bottleneck filter if the time of all previous filters is assumed to be $0.4 \times t_b$ for different number of filters before the bottleneck. What can be seen is that during the relative execution time before $0.4$, all filters are executed at the same time and later only one single filter is executed. This causes a high load at the beginning of the period $t_b$ and a low load at the end of the period $t_b$. In real systems, this edge will not be that sharp as not all filters before the bottleneck have the exact same processing time.

## IV. Delayed Data Generation

The drawbacks as demonstrated in the previous section can be handled by delaying the execution of the first filter (the producing filter) only. All other filters work as in the reference system meaning that they start processing as soon as all input data are available and forward the result as soon as this is present. The producing filter is delayed to have the same rate as

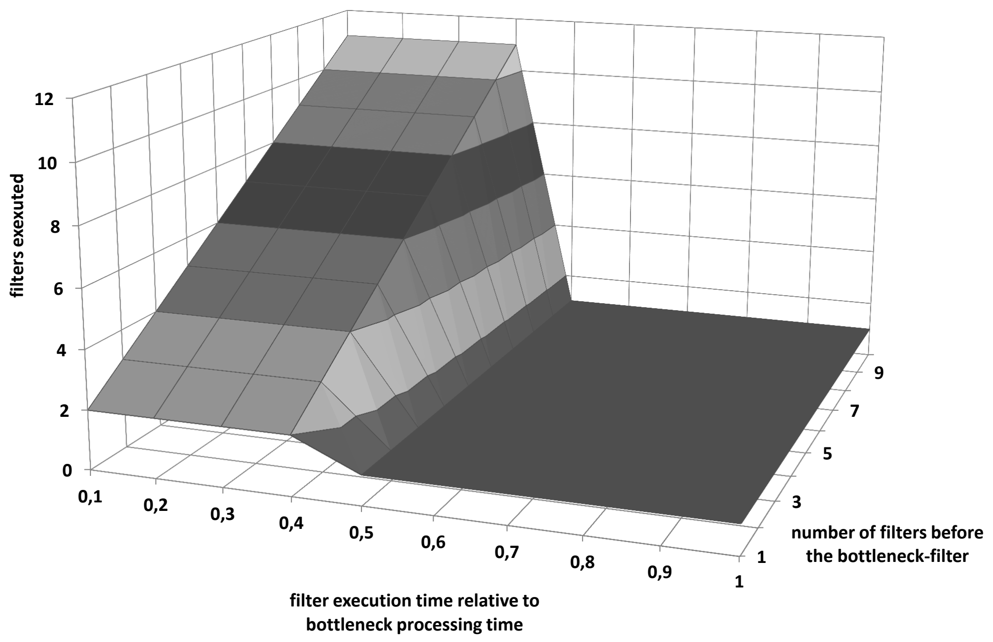

Fig. 4. The number of filters processing during the relative processing time of the bottleneck-filter ($0.1 \rightarrow 1$) where the execution time of the filters before the bottleneck-filter is $0.4 \times t_b$. The values in the graph are always bigger than one as the one represents the bottleneck-filter that is executed during the whole period. All other filters finish execution after $0.4$.

the bottleneck-filter. With this mechanism, the pipeline never fills up, the number of data-elements in the pipeline is reduced to the minimum and the time to execute a data-element equals the sum of all filter execution times which is the minimum possible execution time. The main idea of an optimal execution is to ensure that every data-element is continuously processed in one of the filters, reducing the time that a data-element is stored in the input buffers. The time that a data-element is stored in an input buffer (and not processed) should be zero in the optimal case.

By assuming an execution time of the filters before the bottleneck of $0.4 \times t_b$ and 10 filters before the bottleneck filter, the execution time overhead could be reduced by a factor of 6 as shown in Figure 2. In addition 6 data elements less would be in the main memory during execution as seen in Figure 3.

Another interesting observation is that this modification also has an influence on the number of filters being processed at a time. In the reference system, (compare Figure 4) 11 filters have been executed from the beginning of the period till relative time $0.4$. The modified version shows a more smoothed processing distribution. The distribution of the processing times of the filters for the proposed system is shown in Figure 5.

By comparing the values from Figure 4 to the values from Figure 5, the difference of the two systems can be noted. In the first case, all filters are executed at the same time resulting in a higher CPU utilisation during this period. In the second case, the execution of all filters is distributed over the whole processing period. The sum of all filter execution times is the same for both cases, the only difference is that in the second case the distribution is more smoothed over the whole period where in the first case there is always a peak at the beginning of the processing period.

In this section, we could show that by adapting the execution rate of the first filter only, we can reduce the required memory, smooth the CPU utilisation over the processing period of the bottleneck filter by still keeping the highest possible data-rate.

## V. Adaptive, Dynamic Behaviour

So far we described that with a modification of the execution-rate of the first filter, it is possible to smooth the CPU utilisation, reduce the required main memory and reduce the time that each data-element is processed within the pipeline to a minimum. All scenarios described in this paper so far had fixed, static values for the important parameters (number of filters before the bottleneck $n_b$, average execution time of the filters before the bottleneck $t_{avg}$ and execution time of the bottleneck filter $t_b$).

In real systems, these values are not constant for the entire execution period. In most modern systems, the requirements for the systems change over time or depending on events. In such scenarios, the filters are not able to process each data-element with the same processing speed. Depending on the parameters and the properties of the input data, the execution time of a filter may change over time. An example would be an image processing pipeline. By changing the resolution of the images, most filters process at different execution speeds. We now analyse the effects of dynamic changes of any of the three parameters and how the system has to adapt to come back to the optimal case.

To have a system that is able to handle all these effects, it is necessary to keep track of two pairs of values at the production filter. The first pair is the optimal configuration which the system aims to achieve. This is the optimal number of data elements in the bottleneck-pipeline $e_{opt}$ and the delay

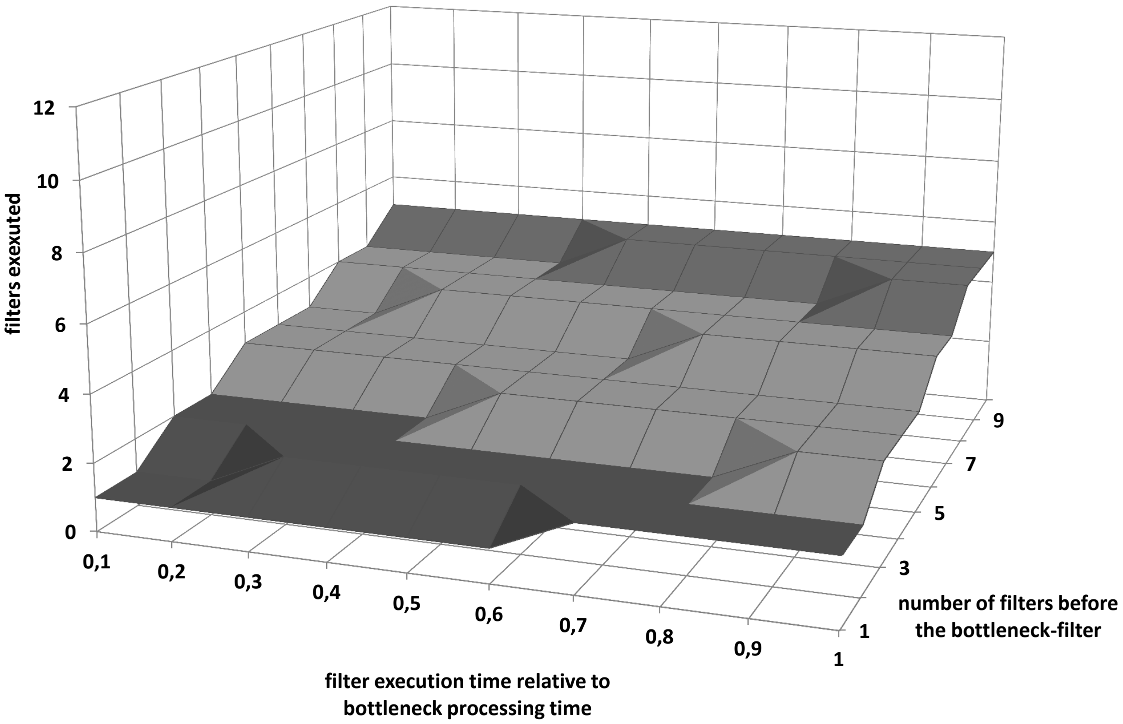

Fig. 5. The number of active filters during the relative processing time of the bottleneck-filter where the execution time of the filters before the bottleneck-filter is $0.4 \times t_{max}$. The values in the graph are always bigger than one as one represents the bottleneck-filter that is executed during the whole period. The execution of all other filters is spread over the whole processing period

of the bottleneck filter $t_b$. The second pair of variables is the current system state. First, the number of elements currently in the bottleneck-pipeline $e_b$ and the current time between two produced data elements $t_s$ (the execution period of the source filter).

Every possible change in any of the pipeline parameters $n_b$, $t_{avg}$ or $t_b$ will influence the values of $e_{opt}$ and $t_b$. This means that if there is an algorithm that can compensate changes in these two values, it will be able to handle any change in pipeline structure.

It is now sufficient to find an algorithm that can change $t_s$ and $e_b$ so that they fit the optimal values again. Changing $t_s$ is easy as this only requires the setting of the new value for $t_s$ as described in the first part of Algorithm 1. The adaption of this parameter ensures that the source produces new data elements at the same rate as the bottleneck-filter can process them. However, this parameter is not sufficient as the number of data-elements in the pipeline are not taken into account so far. The influence of a pipeline with less elements than optimal results in an additional delay at the bottleneck filter (and a reduced processing rate) as it is not be possible to provide new data to the bottleneck as soon as this finished processing. As the producer is executed at the same rate as the bottleneck consumes data, the pipeline is automatically re-filled by the producer filter in an optimal way (at the same rate as the bottleneck filter can process). If the number of elements in the pipeline is bigger than the optimal number, this results in an increased processing delay for the single data-elements as it blocks one of the filters. This does not influence the processing rate but the memory requirements are higher than necessary and the time it takes a single data-element to be processed (the processing delay for single data-elements) is increased. This is the main reason why also the number of elements in the pipeline has to be adapted to the optimal value. Adapting the number of data-elements in the pipeline can be done as described in the bottom part of Algorithm 1. If the number of elements in the pipeline $e_b$ is larger than the optimal number of elements in the pipeline $e_{opt}$, the execution will be delayed resulting in a reduction of $e_b$. This is done until the optimal number of elements in the pipeline $e_{opt}$ is reached again. If there are less elements in the pipeline than in the optimal case, no modification is needed, the source filter is executed at the current rate of $t_b$. This will fill up the pipeline in an optimal manner (time between two data-elements is the processing time of the bottleneck-filter).

This section analysed the dynamic case, how to handle the case of a different bottleneck filter and the case when the processing speed of the bottleneck-filter changes to faster or slower execution. The next section will describe the practical realisation of this mechanism and compare measurements between the reference-system and the proposed system.

## VI. Implementation

This section focuses on the practical realisation of the proposed mechanism. Later at this section we compare the proposed system to the reference system for the parameters as described in the previous section (number of data elements in the pipeline and processing delay for the data-elements).

For this comparison, a middleware system was implemented that is based on the pipe-and-filter approach. Each filter can be executed on a different hardware and the middleware handles the communication between the filters. The selected programming language was C#. The compiled binary files can be executed on Windows as well as on Linux systems (in the case of Linux systems, the help of mono is required for execution). Pipelines have been tested on heterogeneous systems with Linux and Windows operating systems. The filter logic can be implemented by the user. One requirement for the filter implementation is that the input and output data are *serializable*, otherwise these data cannot be forwarded to other devices and the application terminates. For evaluation purposes, three dummy-filters have been implemented, one producer, one consumer and one processor. Dummy filters have been chosen as they can be modified in a way that they simulate a desired behaviour. The important parameter in this case is the execution time of the filters. The chosen execution time was realized by a delay that could be set for each filter. The data that were transmitted through the pipeline were two dimensional byte arrays. We executed the reference system as well as the proposed system on two devices. The producer and consumer filter are both executed on a Linux device. The rest of the filters are equally distributed on both devices. In our case, we took a Windows pc as the second middleware device. During the execution, we stored the data that resulted in the measurements presented later in this section. The stored data include the creation time of a new data-element and the time when the data-element finally passed the pipeline. With those two times, it was possible to calculate the time that a packet is inside the pipeline. In addition, the number of data-elements inside the pipeline has been stored.

From Section V we know that it is sufficient that the source filter modifies its execution time. We further know that the correct time to trigger the next execution depends on the following values:

- The optimal number of elements in the bottleneck-pipeline $e_{opt}$
- The processing time of the bottleneck filter $t_b$
- The current processing time of the source filter $t_s$
- The current number of elements in the bottleneck-pipeline $e_b$

There are two important tasks to consider upon implementation of the middleware. First, how should the middleware react based on these values and Second, how can these required measurement data be obtained from the system.

In the real implementation, we had to compensate a lot of effects like scheduling which causes a non-constant time for processing of a single data-element. These effects were handled by using average values and standard deviations over a number of executions. These values were also considered by setting the processing time of the producer filter. In the rest of this chapter we will no longer mention the average and standard deviation of the values as these would make all algorithms too complex to describe.

Starting with the first task, Algorithm 1 shows how the source-filter decides whether to execute the filter code immediately or delay the execution. The described function is always called after a period of $t_s$ by a timer. This algorithm is based on the observations from Section V.

**Algorithm 1** Execution triggering on source filter

```
 1: function EXECTRIGGER($t_b, t_{sum}, e_b, t_s, t_{last\,time\,executed}$)
 2:     if $t_s \neq t_b$ then
 3:         $t_s = t_b$
 4:     end if
 5:     $e_{opt} = \lceil t_{sum} \div t_b \rceil$
 6:     $e_{extra} = e_b - e_{opt}$
 7:     if $e_{extra} \geq 1$ then
 8:         $d_{first} = t_b \times -(\text{NOW}() - t_{last\,time\,executed})$
 9:         $d_{rest} = t_b \times (e_{extra} - 1)$
10:         DELAY($d_{first} + d_{rest}$)
11:     end if
12:     EXECUTEFILTER()
13:     $t_{last\,time\,executed} = \text{NOW}()$
14: end function
```

Algorithm 1 works based on the values from $t_b$, $t_{sum}$, $e_b$, $t_s$ and $t_{last\,time\,executed}$. The values for $t_s$ and $t_{last\,time\,executed}$ can be directly read from the filter internal variables but the other values have to be gained from the system. As in this system, each filter only knows its direct data-sinks, an algorithm was required that is able to return the values that are required by Algorithm 1.

The implementation of the data measurement can be compared to a depth search from a tree-structure. The algorithm is shown in Algorithm 2. At the beginning, each filter assumes that he is the bottleneck filter. Therefore he checks whether he is slower in processing or in transmission. Depending on the result, a preliminary maximum delay $t_{fused,max}$ is stored. The time $t_{sum\,from\,bottleneck}$ is also initialized depending on the location of the bottleneck. Afterwards all filters that receive data from this filter are requested for their values for $t_{fused,max}$ and $t_{sum\,from\,bottleneck}$. The preliminary local results and the received results are then fused as described in the function *fuseResults*. This function stores the data for either the highest number of $t_{fused,max}$ or if they have the same value, the longest delay between the current filter and the bottleneck-filter ($t_{sum\,from\,bottleneck}$). Before returning the result, the number of data-elements in the own filter is added to the fused result ($fused$).

The practical implementation was tested for 10 filters before the bottleneck and the average time of the filters before the bottleneck was set to 10% and 40%. The test has been executed for the reference system and the proposed system. The producer and consumer were executed on a Linux host. Each second of the other filters was executed on a remote Windows host. With this system we could show that a distributed execution on heterogenious devices can be performed. The adaption process from the proposed architecture also works in this distributed scenario as expected.

In the reference architecture, the number of elements in the pipeline rises to a value of approximately 30 see Figure 6, graphs marked with *ref*. This value can be explained as each filter has an input buffer, a processing buffer and an output buffer which means that altogether each filter can store three data elements. This storage is fully used by the reference system.

The proposed system in Figure 6, graphs marked with

**Algorithm 2** Algorithm for bottleneck-queue detection

```
1:  function GETBOTTLENECKPARAMETERS
2:      if $t_{average\,execution} \geq t_{average\,transmission}$ then
3:          $t_{fused,max} = t_{average\,execution}$
4:          $t_{sum\,from\,bottleneck} = 0$
5:      else
6:          $t_{fused,max} = t_{average\,transmission}$
7:          $t_{sum\,from\,bottleneck} = t_{average\,execution}$
8:          $fused = [t_{fused,max}, t_{sum\,from\,bottleneck}, 0]$
9:      end if
10:     $notBottleneck = False$
11:     for all sink filters do
12:         $received$ =GETBOTTLENECKPARAMETERS()
13:         $newValue$ =FUSERESULTS(ref $fused, received$)
14:         if $newValue = True$ then
15:             $notBottleneck = True$
16:         end if
17:     end for
18:     if $notBottleneck == True$ then
19:         $fused.t_{sum\,from\,bottleneck} += (t_{average\,execution} + t_{average\,transmission})$
20:     end if
        $fused.num_{elements\,in\,pipeline} += num_{elements\,in\,filter}$ return $fused$
21: end function
22: function FUSERESULTS(ref $fused, received$)
23:     if $fused.t_{fused,max} > received.t_{average\,transmission}$ then return $False$
24:     else if $fused.t_{fused,max} == received.t_{average\,transmission}$ then
25:         if $fused.t_{sum\,from\,bottleneck} > received.t_{sum\,from\,bottleneck}$ then return $False$
26:         else
27:             $fused.t_{fused,max} = received.t_{fused,max}$
28:             $fused.t_{sum\,from\,bottleneck} = received.t_{sum\,from\,bottleneck}$ return $True$
29:         end if
30:     else
31:         $fused.t_{fused,max} = received.t_{fused,max}$
32:         $fused.t_{sum\,from\,bottleneck} = received.t_{sum\,from\,bottleneck}$ return $True$
33:     end if
34: end function
```

*new*, show a different utilisation of elements in the processing pipeline. The first peak can be described by the fact that at the beginning, the filters do not know how fast they can process data and therefore return a low value for their processing time. This low value corresponds to a high number of allowed data-elements in the pipeline. The source filter reacts to this request by increasing the processing speed. As soon as the first data-elements are processed by the filters, the filters return the measurements for their real execution time. As soon as the bottleneck filter returns its first measurement, the data-producing filter has to adapt to this situation. Based on these measurements, the data-producing filter decides to delay data production until a lower threshold ($e_{opt}$) is reached which depends on the parameters as described in Algorithm 1. As shown in this figure, the number of data-elements in the pipeline automatically adapts to the correct, optimal value.

Figure 7 visualises the data-element delays for the same two systems - reference (*ref*) and proposed (*new*) for an average execution time of the filters before the bottleneck at $10\%$ and $40\%$, respectably. The *ref* graphs refere to the reference architecture and the *new* refere to the proposed architecture. In the reference architecture, the delay is constantly increased till all buffer elements are full. From that point, the delay remains at a constant high level. In the proposed system, the delay also increases at the beginning. At the same time when the number of data-elements in the pipeline is reduced (see Figure 6) also the delay is reduced. Where the suggested approach reduces the number of filters in the pipeline to a minimal value, the reference system fills all available storage spaces which increases the processing delay.

## VII. Conclusion

In this paper, we showed that by adding a delay to the source-filter of a pipeline, it is possible to improve the pipeline execution in a number of ways. The required main memory is reduced, the CPU utilisation is smoothed and the processing delay of a single packet is reduced. The processing rate is not influenced by this modification bringing an overall increase in execution efficiency. The effects have been reviewed in theory, required parameters have been analysed, mechanisms to retrieve the required parameters have been invented and implemented. The final implementation has been compared to a state-of-the art reference design which proved that the proposed method also works in real systems. This mechanism can improve any multi-threaded pipe-and-filter based architecture.

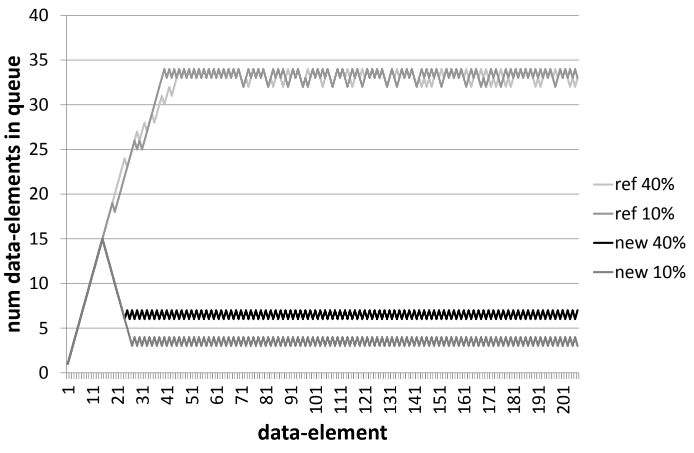

Fig. 6. Number of objects in the pipeline using the suggested (new) approach (lower two graphs) compared to the reference approach (ref) for two different average execution times for the 10 filters before the bottleneck filter. Both systems were executed on a distributed network.

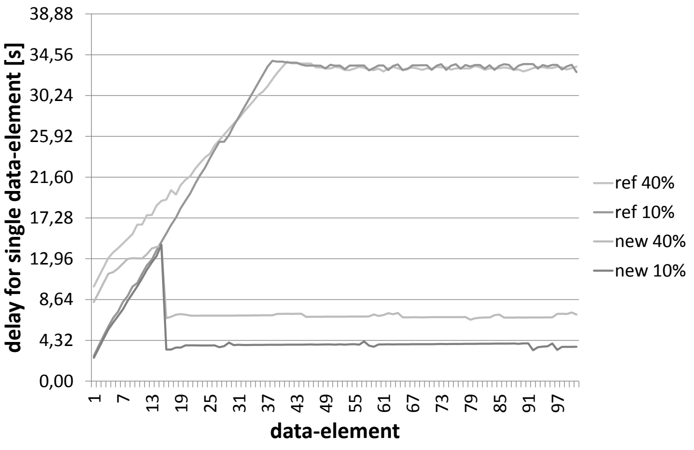

Fig. 7. The time for the single data-elements in the processing pipeline for two different average execution times for the 10 filters before the bottleneck filter using the suggested (new) approach (lower graphs) compared to the reference approach (ref). Both systems were executed on a distributed network.

## Acknowledgement

The research leading to these results was conducted in the EPiCS project (Engineering Proprioception in Computing Systems) and has received funding from the European Union Seventh Framework Programme under grant agreement no 257906 and is supported by Lakeside Labs GmbH, Klagenfurt, Austria.

## References

[1] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” Pittsburgh, PA, USA, Tech. Rep., 1994.

[2] M. Bramberger, J. Brunner, B. Rinner, and H. Schwabach, “Real-time video analysis on an embedded smart camera for traffic surveillance,” in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, may 2004, pp. 174 – 181.

[3] D. Bauer, A. N. Belbachir, N. Donath, G. Gritsch, B. Kohn, M. Litzenberger, C. Posch, P. Schön, and S. Schraml, “Embedded vehicle speed estimation system using an asynchronous temporal contrast vision sensor,” *EURASIP J. Embedded Syst.*, vol. 2007, pp. 34–34, January 2007.

[4] B. Rinner and W. Wolf, “A Bright Future for Distributed Smart Cameras,” *Proceedings of the IEEE*, vol. 96, no. 10, pp. 1562–1564, October 2008.

[5] H. Guggi and B. Rinner, “Distributed smart cameras for hard real-time obstacle detection in control applications,” in *Distributed Smart Cameras (ICDSC), 2011 Fifth ACM/IEEE International Conference on*, aug. 2011, pp. 1 –6.

[6] M. Bramberger, A. Dobl, A. Maier, and B. Rinner, “Distributed embedded smart cameras for surveillance applications,” *Computer*, vol. 39, p. 2006, 2006.

[7] L. Esterle, P. R. Lewis, M. Bogdanski, B. Rinner, and X. Yao, “A Socio-Economic Approach to Online Vision Graph Generation and Handover in Distributed Smart Camera Networks,” in *Proceedings of the Fifth ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC 2011).* IEEE Press, 2011, pp. 1–6.

[8] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” in *Information Processing ’74: Proceedings of the IFIP Congress*, J. L. Rosenfeld, Ed. New York, NY: North-Holland, 1974, pp. 471–475.

[9] E. A. Lee and T. M. Parks, “Dataflow process networks,” *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995. [Online]. Available: http://dx.doi.org/10.1109/5.381846

[10] E. Cannella, O. Derin, and T. Stefanov, “Middleware approaches for adaptivity of Kahn Process Networks on Networks-on-Chip,” in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, nov. 2011, pp. 1 –8.

[11] A. Pope, *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison Wesley, 1998.

[12] “GStreamer: open source multimedia framework,” http://gstreamer.freedesktop.org, last visited: August 2012.

[13] B. Rinner, M. Jovanovic, and M. Quaritsch, “Embedded Middleware on Distributed Smart Cameras,” in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 4, april 2007, pp. IV–1381 –IV–1384.

[14] B. Rinner and M. Quaritsch, “Embedded Middleware for Smart Camera Networks and Sensor Fusion,” *Multi-Camera Networks: Principles and Applications*, Jul. 2009.

[15] A. Afrah, G. Miller, D. Parks, M. Finke, and S. Fels, “Hive: A distributed system for vision processing,” in *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, sept. 2008, pp. 1 –9.

[16] W. Schriebl, T. Winkler, A. Starzacher, and B. Rinner, “A pervasive smart camera network architecture applied for multi-camera object classification,” in *Proceedings of the Third ACM/IEEE International Conference on Distributed Smart Cameras, 2009. ICDSC 2009.* IEEE, 2009, pp. 1–8.

[17] K. Moreland, “A Survey of Visualization Pipelines,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 19, no. 3, pp. 367–378, 2013.