Reducing Processing Delay in Dataflow-oriented Middleware Systems for Smart Camera Applications

Herwig Guggi and Bernhard Rinner
Institute of Networked and Embedded Systems
Alpen-Adria-Universität Klagenfurt
herwig.guggi@aau.at, bernhard.rinner@aau.at

ABSTRACT

Dataflow-oriented processing is an attractive paradigm for smart camera applications. In this paper we present a dataflow-oriented middleware system to support the development of such applications on resource-limited distributed systems. Our main contribution includes the reduction of the overall processing delay by delaying the data generation and by the just-in-time transfer of data elements. A side effect of this delayed data transfer is also a reduction of the memory requirement for the communication links. We introduce the basic algorithm and present experimental results of our middleware system running in a distributed environment.

Keywords

dataflow processing; middleware system; data generation; adaptation; multi-camera systems; pipe-and-filter architecture

1. INTRODUCTION

The optimisation of middleware systems for distributed applications is an important field of study as the number of computing devices is constantly increasing and the request to take advantage of these networked resources arises. Smart cameras are one example of this trend. While single cameras can be used to trigger events and support a human observer in a surveillance system [1] or perform vehicle detection and speed estimation [2], distributed smart camera networks offer an even higher benefit [3]. They can be used to detect obstacles to avoid collisions [4] or perform a cooperative tracking with local image analysis [5]. Usually a middleware system is used to coordinate the distributed execution and the communication between devices. As the system load may change during runtime, a middleware system has to support online adaption of resources. One example of such a dynamic system is described by Esterle et al. [6] where they use a socio-economic approach for online vision graph learning and tracking handover in smart camera networks.

A middleware system suitable for distributed video processing applications has to provide at least three key features. First, as these applications process data from different camera devices, it must support the processing of data streams from multiple sources and potentially to multiple sinks. Second, since the processing load is typically dependent on the content, the execution time may vary during runtime. Thus, a suitable middleware has to cope with these dynamic changes of the execution times. Finally, the overall delay of processing pipeline should be as small as possible.

Most data-flow-based applications rely on the pipe-and-filter architecture. The pipe-and-filter architecture consists of two main components. First, the filters which produce, consume or process data. Source-filters provide new data to the system. An example for a source filter is the image acquisition which generates new image data to the system. Sink filters consume the data and typically visualize them to the user or store them in a file. Process filters receive data from the input, process them and forward the result to the output. Second, pipes are responsible to forward data from the outputs of one filter to the inputs of other filters. The connections are created by system designers. Figure 1 depicts a pipeline with an ensemble of pipes and filters. The filters 1, 2 and 3 produce data. Filters 4 and 7 fuse data from different sources and filters 8 and 10 consume data (e.g., by visualizing the result on a user interface).

In this paper we describe a middleware system for dynamic smart camera applications. It supports multiple data sources and sinks, the adaption to changing execution times and the reduction of the pipeline delay. Applications represented as single-rate acyclic data-flow can be realized with this middleware system. These features are implemented in a distributed system with no central coordination unit. The required control mechanism is implemented in a distributed manner where each filter is augmented by a dedicated control unit. These control units communicate via a bidirectional control channel which is parallel to the data connections. In figure 1 the control channels are visualized as red, dotted arrows.

The pipeline delay is reduced by just-in-time delivery of data which means that new data is delivered to the input of the succeeding filter as soon as the processing of previous data element has been completed to avoid blocking. Just-in-time delivery requires the sources to be executed at the same rate as the slowest filter in the pipeline, i.e., the bottleneck filter. The max-consensus algorithm [7] is used to identify and distribute the execution time of the bottleneck filter to the pipeline. To reduce the number of data elements

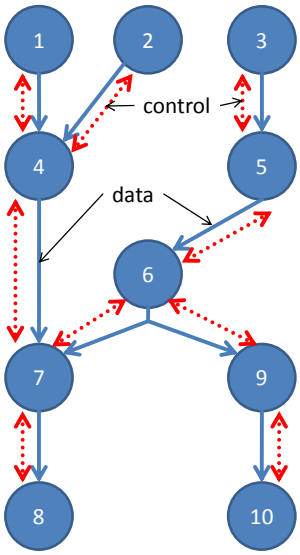

Figure 1: A sample pipeline with a number of filters. Filters with the numbers 1, 2 and 3 are source filters and filter with the numbers 8 and 10 are sink filters. Data flow (unidirectional) visualized in blue and control flow (bidirectional) visualized in red.

in the pipeline, congestions are detected by our control algorithm and distributed via the control channels. As a consequence, the source filters postpone the generation of new data elements accordingly and the pipeline delay is reduced. Our algorithm is evaluated by comparison with state-of-the-art data-flow processing. Our experiments show that the pipeline delay could be reduced from 36 s to 2.5 s.

The rest of the paper is organised as follows: Section 2 summarises related work in the area of data-flow based execution models. Section 3 describes the basic components of the system. In Section 4, we explain the necessary control mechanisms to reduce the data delay. Section 5 provides measurement results where the proposed system is compared to a reference implementation. Section 6 concludes the paper with a summary and discussion of future work.

2. RELATED WORK

A number of middleware systems for distributed smart camera applications are already available [8]. In this section we give a short overview of the most relevant related systems.

CORBA [9] one of the best known systems for distributed computing provides methods for remote method invocation. Two methods, synchronous and asynchronous calls are possible. This middleware system does not support pipe-and-filter based execution 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 [10] 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. This system works good for pre-captured sources. In the case of live data, it makes sense to provide a mechanism that automatically adapts the frame-rate of the data producer to those of the consumer.

HIVE [11] creates pipelines called "swarms" out of single elements called "drones" where each drone is executed in its own thread. Synchronised and streaming data transfer models are provided to transmit data from one drone to another. By using the streaming data transfer, a pipeline 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 pipeline. There is no mechanism provided to react on changes in the processing load. If the input queue of a filter is full, new elements are simply dropped. The synchronized data transfer ensures 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 previous 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 with this mechanism is that even if the data transfer is called synchronized, the threads of the drones are not synchronized which means that in some cases a source drone produces more data than a filter drone can process. This leads to frame-drops in case of a video encoding. Another drawback is the added delay that is introduced by requesting the next data after the last data has finished processing.

Schriebl et al. [12] describe a system where every block (representing the filter in a pipe-and-filter architecture) 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.

With this approach, the execution speed of the whole processing pipeline is automatically reduced to the rate 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. However, it is unclear how filters with more than one input are handled.

Moreland [13] present 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. An event-driven pipeline launches execution as data becomes available in sources. A demand-driven

	Multi-source	No dropped elements	Delay optimisation	Runtime adaptation	Low control overhead
CORBA [9]	○	○	○	○	○
Gstreamer [10]	●	●	○	○	○
HIVE [11]	●	○	○	○	○
Schriebl et al. [12]	●	●	○	○	●
Guggi et al. [14]	○	●	●	●	○
proposed system	●	●	●	●	●

Table 1: Classification of middleware systems. White bullets represent unsupported, gray bullets partially realised and black bullets fully covered properties.

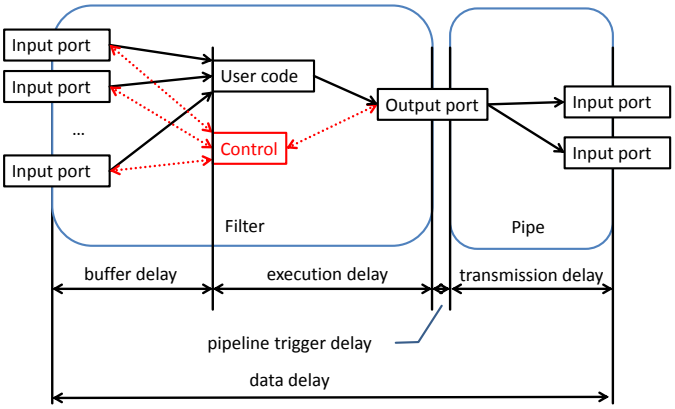

Figure 2: Visualisation of the individual components of a single block. Directed data path is shown with black, solid arrows. The bidirectional control path is shown with dotted, red arrows.

pipeline launches execution in response to requests for data.

A typical example for an event-driven pipeline could be the system as described by Schriebl et al. [12]. New data is produced as soon as possible. An example of the demand-driven pipeline could be the synchronized data transfer as described by the HIVE [11] having the already mentioned problems. Our system could be described as a combination of the two mentioned execution systems. We produce data in an event-driven manner by also regarding to the demands of the bottleneck filter.

In our previous work [14] we present a distributed pipe-and-filter middleware. In addition to the reduced memory consumption, it was shown that the pipeline delay can be reduced as compared to a state-of-the art execution model. However, this system is limited to single data source pipelines.

Table 1 compares the related middleware approaches based on the following criteria: (i) processing data from multiple sources, (ii) avoiding dropping data elements, (iii) optimisation of the pipeline delay, (iv) dynamic adaptation during runtime and (v) providing low control overhead.

3. SYSTEM DESCRIPTION

In this section, we describe the individual parts of our dataflow-oriented middleware system. An application consists of a number of filters and the pipes connecting the filters. The filter and the pipe are executed in two separate threads to ensure that processing the (input) data can be performed concurrently with transferring the (output) data of a filter. In addition to the data elements also control information such as the bottleneck time and the number of data elements in the pipeline have to be distributed. This control information is transmitted on dedicated control links along the dataflow graph.

Each filter may have any number of input ports and a single output port (cp. Figure 2). Each input port and the output port have a unique buffer to store a single data element. The input port stores arriving input data. The output port stores execution result data to be forwarded to the succeeding filters. As soon as input data is available on each input port and the buffer of the output port is empty, the execution of the user code is triggered. The result of the execution is stored in the buffer of the output port. As soon as the result is stored, the transmission thread (pipe) is triggered. The pipe copies the result of the buffer of the output port to a *transmission* buffer and empties the buffer from the output port to allow another element to be processed by this filter. The transmission thread further copies the data from the *transmission* buffer to the input buffers of the following blocks. Source filters are triggered by a timer event. The timer of all filters are set to the bottleneck time. The exact mechanism for finding the correct value for the timer is described in the next section.

The time that a single data element requires to pass through one filter-pipe combination is called the *data delay*. This delay is composed by the following individual delays (cp. Figure 2): First, the *buffer delay* which is the time that the data element is stored in the input buffer. Second, the *execution delay* which is the time that is required to execute the user code and to store the result to the buffer of the output port. Third, the *pipeline trigger delay* which is the time between storing the result to the buffer of the output port and the start of the transmission thread. Fourth, the *transmission delay* which is the time-span between copying the result from the buffer of the output port to the transmission buffer and end of storing all data elements to the input buffers of all following blocks.

In parallel to the user code, there is one control unit in each filter. The control units of the pipeline are connected via bi-directional transmission channels which are parallel to the pipes. Each control unit can receive messages downstream (from the input ports), send messages downstream (via the output port), receive messages upstream (from the output port) and send messages upstream (via the input ports).

As the main goal of this work is to reduce the processing time of any pipe-and-filter based application, it is necessary to reduce the *data delay*. This delay can be reduced by decreasing one of the following delays. The *buffer delay* depends on the time of arrival of the individual data elements. The *execution delay* depends on the processing hardware, the user algorithm and the input data. The *pipeline trigger delay* depends on the implementation of the task scheduler of the operating system. The *transmission delay* depends on the used communication channel, the amount of user data

and the number of data consumers (following filters).

For any data processing architecture in the form of a connected, directed graph, the processing rate of this system is limited by the execution time of the bottleneck filter t_b. The location of the bottleneck filer in the dataflow graph does not influence the data rate of the pipeline.

If the pipeline is empty and all source filters produce at the rate of the bottleneck filter, all data can pass through all filters without any buffer delay as each received data packet can always be processed immediately after reception. In this optimal case, the data rate is the rate of the bottleneck filter and the buffer delay is reduced to zero. In case of a congestion the total delay (time from data production to consumption) is increased by a factor of $n \times t_b$ where n is the number of data elements in all input buffers which are not currently processed.

Due to scheduling effects and variations in the execution times of each filter, it can happen, that buffers may fill up during runtime. This can be detected by the filter controller. As soon as a single buffer fills up, the corresponding source filters have to delay the next execution. The delay is calculated in a way that after the delay time, the optimal number of data elements in the pipeline is reached again.

A sync message (cp. Section 4) is generated by the filter detecting the congestion. This message is sent upstream to the source filters. As soon as the sync message reaches a source, this source is able to calculate the optimal and read the actual number of data elements in the pipeline. If the actual number is higher than the optimal one, the next execution of the source is delayed in order to reach the optimal case again.

With these two simple mechanisms (setting the processing rate of the sources to the processing rate of the bottleneck and delaying single sources if a congestion is detected) it is possible to execute any directed, multi-source, multi-sink dataflow-oriented processing architecture with reduced data delay. In addition to the reduced delay, the required memory is also minimised as the overall number of data elements in the pipeline is reduced.

We decrease the data delay by reducing the buffer delay. This can be achieved by just-in-time data delivery. Two main challenges have to be handled to ensure a low data delay. First, the bottleneck time has to be known by all filters. Second, the number of data elements in the pipeline have to be reduced. The bottleneck time has to be known by all filters for two reasons. First, if another filter detects that it processes slower than the current bottleneck, the new bottleneck has to be announced. Second, the source filters have to set the timer for triggering data generation to the value of the current bottleneck time. The reduction of the number of data elements in the pipeline helps to prevent congestion thus reducing the buffer delay.

4. CONTROL AND SYNCHRONISATION

The two challenges just explained are handled by the control unit of each filter. Two control message types are used to optimize the execution of the system. First, bottleneck update messages are used to update the information about the global bottleneck time. Second, sync messages are used to reduce the number of elements in the pipeline. Bottleneck update messages consist of the bottleneck time t_b and the filter that detected the bottleneck. Sync messages are sent if a data congestion is detected. This message consists of the current bottleneck time t_b, the sum of delays from the filter that detected the congestion to the source filter and the number of data elements from the filter that detected the congestion to the source filter.

Each filter has an internal parameter t_{ib} which represents the locally stored bottleneck time. After processing of the first data element, t_{ib} is set to the own *execution time*. This value is sent to all neighbours (upstream and downstream) in a bottleneck-update message. If a higher value than the local bottleneck is received, the local value of t_b is updated and the new value is forwarded to all neighbours. This is the implementation of the max-consensus algorithm. Nejad et al. [7] have proven that the max-consensus algorithm can be used to find the global maximum execution time in this setup.

Algorithm 1 describes the mechanism to detect a congestion at the input ports of a filter. If a new element should be transmitted to the input port while the input buffer is still occupied, the congestion is detected. The control unit detecting the congestion creates the sync message with the t_b and the sum of the own execution and transmission time as $sumDelay$. $numDataElements$ is set to the number of data elements in the current filter. The message is then sent upstream via the input ports.

Algorithm 1 Handling of new data on an individual filter input port

```
1: function handleDownstreamData(element)
2:     if inputBuffer! = EMPTY then
3:         sendCongestionMessageUpstream()
4:     else
5:         inputBuffer = element
6:     end if
7: end function
```

The processing of a received sync messages is shown in Algorithm 2. Filters receiving the sync message, add their own execution and transmission time to the value of $sumDelay$, add the local number of elements to the $numDataElements$ and send the message upstream.

Source filters will finally receive the sync message and calculate the optimal number of data elements in the pipeline with $e_{opt} = \lceil \frac{sumDelay}{t_b} \rceil$. If the counted number of data elements in this pipeline $numDataElements$ is larger than e_{opt}, the next execution of the source filter is delayed by $t_b \times (numDataElements - e_{opt})$. After this time, the optimal number of elements in the pipeline is reached and the source produces further data elements at the same rate as the bottleneck filter can process data. In an ideal system, all sources are then synchronized to the bottleneck and the minimal possible data delay can be achieved. In most real systems, scheduling effects of the operating system cause variations in the data delay which requires continuous adaption to keep the low data delay.

5. EVALUATION

The proposed algorithm is compared to dataflow processing with blocking communication which serves as reference system. In the reference system, all filters forward the data to the succeeding filters as soon as the input buffer of these filters are empty. The architecture of the reference system is also used by Schriebl et al. [12]. The reference and the

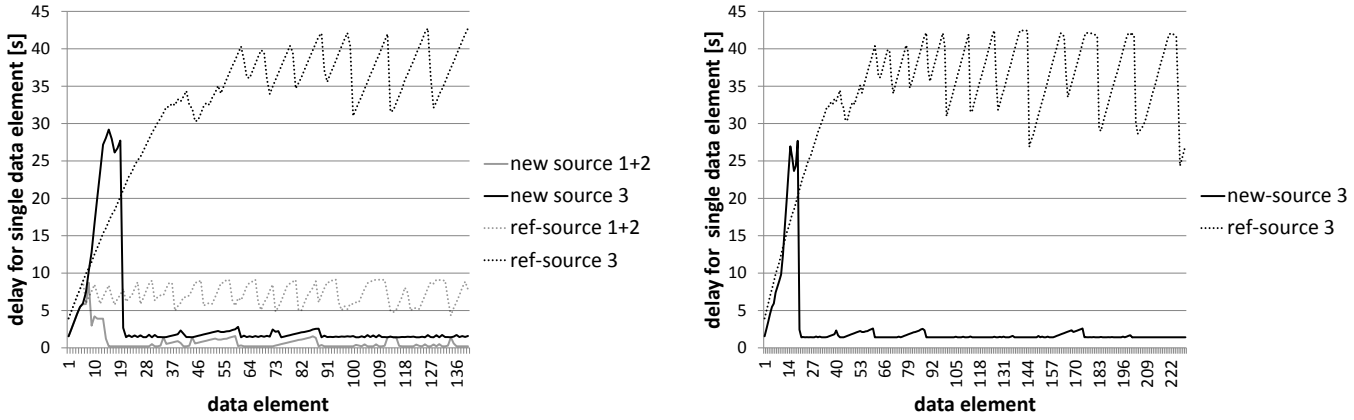

Figure 3: The data delay for single data-elements in the pipeline with 3 sources (sources one and two have the same delay) as received by sink filter 8 (left) and filter 10, respectively. Comparison between reference system (dotted graphs) and proposed system (solid graphs).

Algorithm 2 Handling of the sync messages

```
1: function handleSyncMessage(syncMessage)
2:     if isSourceFilter then
3:         t_f = syncMessage.sumDelay
4:         t_b = syncMessage.bottleneckTime
5:         e_p = syncMessage.numDataElements
6:         e_opt = ⌈t_f / t_b⌉
7:         if e_p > e_opt then
8:             delay = t_b × (e_p − e_opt)
9:             nextExecutionTimer.sleep(delay)
10:        end if
11:    else
12:        syncMessage.sumDelay+ = t_processing
13:        syncMessage.numDataElements+ = e_own
14:    end if
15: end function
```

proposed system are implemented in C# and executed on a number of different devices (including Windows and Linux PCs). A distributed middleware was implemented that supports features for downstream communication for filter-data and bidirectional communication for control-data.

The sample pipeline is the pipeline as visualized in Figure 1 with 10 additional filters between source 3 and filter 5. All filters are configured to have an execution time of 100 ms. The sample scenario has been selected in order to realize a number of challenging configurations. A number of sources with different data-delays are provided. Not all sinks depend on data from the same input filter(s). The bottleneck filter might be outside the data-path from one of the sources.

In smart camera applications, the execution time of some algorithms depend on the input data. Static scenes with only a small amount of changes between consecutive images can be processed faster than images taken in a dynamic environment. The dynamics of a scene may change during run-time. Surveillance cameras may have highly dynamic scenes during peak-time and static scenes during off-peak times. Wireless LAN is often used to mount cameras on locations independent on the availability of network cables. Therefore, important features for distributed smart camera applications are support for wireless connections and the ability to adapt to changing conditions during run-time.

Three different experiments are conducted to test if the proposed system is able to support the required features for distributed smart camera applications. The static case (to show that the system works fine while the dynamics of a scene does not change), the dynamic case (to show the capability of the system to adapt to changing environment conditions) and the distributed execution with different payload sizes (to test the execution in a distributed–wireless connected configuration). The first two experiments are executed on a single PC. The last experiment is conducted on two PCs. The two PCs are connected via a 11Mbit wireless LAN. In these experiments we measure the total pipeline delay which is defined as time difference between the generation of a data element at the source filter and its consumption at the sink filter. In case of multiple sinks, the total delay is measured for each sink individually.

5.1 Static Case

The static case compares the behaviour of the proposed system with the reference system in a static scenario. During the experiment execution, the execution times of all filters remain constant. In the static case, the sink filter with number 8 has been selected to be the bottleneck filter with an execution time of 1000 ms. All other filters have an execution time of 100 ms. Figure 3 compares the pipeline delay of the reference system with our system by visualizing the pipeline delay as received by filter 8 (left) and by filter 10 (right), respectively. As filter 10 does not receive any data from sources 1 or 2, there is also no delay measurement for these sources. Even though the bottleneck (filter 8) is not in the data path of filter 10, the proposed system is still able to adapt the processing rate of source filter 3 which ensures a low delay for data from this source. On the graphs from the reference system, it can be realized that the delay is constantly increasing up to a level where all filter buffers preceding the bottleneck are full and the maximum delay has been reached. The variations in the delays are caused by task scheduling effects of the operating system. The solid lines of the graph represent two phases of our proposed system. First, the initialisation phase where the bottleneck time has not yet been determined and distributed through

the pipeline. During this phase, all sources produce at their maximum rates which causes the pipeline to fill up (such as in the reference system). As soon as the bottleneck time has been determined and distributed among the network using max-consensus and sync messages. The sources react as described in section 4 by reducing their data production rate and delaying the next execution to a moment where the optimal number of data-elements in the pipeline is reached. The second phase is the steady state where the sources produce at the desired data rate and the delay remains at a low level. On filter 8 the average pipeline delay for our system in this scenario is 0.5 s for sources 1 and 2, and 1.7 s for source 3. On filter 8 the average pipeline delay for the reference system in this scenario are 7.2 s for sources 1 and 2 and 37.5 s for source 3. Thus, our approach is able to reduce the pipeline delay of the same code as in the reference system by an order of magnitude.

5.2 Dynamic Change of the Bottleneck Filter

This evaluation focuses on the ability of the system to react to changes of the bottleneck during run-time. We executed the sample scenario in six phases. In phases 1, 3 and 5 all filters have the same execution time of 100 ms. In the other phases, always one filter is selected a the bottleneck with an execution time of 1000 ms. In phase 2, the bottleneck is filter 8, in phase 4, filter 9 is the bottleneck and in phase 6, filter 5 is the bottleneck filter. During the execution of the six phases, we evaluate two properties of our approach. First, we demonstrate how our system adapts the data production if the bottleneck time increases or decreases. Second, we show how the sources adapt to the execution time of the bottleneck even if the sources do not provide data to the bottleneck filter. During phases 4 and 6 sources 1 and 2 do not provide data to the bottleneck filters but the system is still able to reduce the data delay for all sources.

Figure 4 visualizes the measured pipeline delays for the data elements received by filter 8 (left) and filter 10 (right), respectively. Filter 10 does not receive any data from sources 1 and 2 and therefore cannot calculate a pipeline delay for data from these sources. By analyzing the parts of the graph from phases 2, 4 and 6, it can be realized, that these sections act like in the static case. While the data delay in the reference implementation is continuously increasing (up to a limit), the proposed system reduces the data delay after a short initialisation period.

5.3 Distributed Execution and Different Payload Sizes

The third experiment evaluates the distributed execution and different payload sizes. To test the distributed execution, the sample pipeline has been executed on two devices. Two windows PCs are used for the test. These two PCs have been connected via an 11Mbit/s wireless LAN connection. The main goal of the distributed execution was not to have an optimal splitting of the sample application, but to provide a high number of data communications between the devices. Therefore every other of the ten filters between source 3 and filter 5 are executed on a different device. All sources (1,2 and 3) and sinks (8 and 10) as well as filter 6 are executed on one device and all other filters are executed on the other device. The pipeline delay values from the first sink (filter 8) is evaluated. To sum up, data from

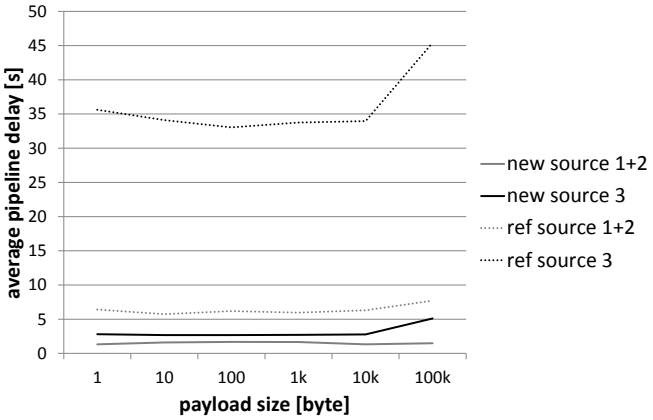

Figure 5: Average data delay for different payload sizes for sample scenario with distributed filters.

source 1 and 2 are transmitted via the network two times (once from the source to filter 4 and once from filter 7 to the sink). Data from source 3 are transmitted 13 times via the network. All in all there are 15 data connections between the two PCs and the available bandwidth is 11Mbit/s. In our experiment all connections transmit the same amount of data. Thus the maximum available bandwidth per data connection is 733kbit/s which is 92kbyte/s.

Simple filters and fusion filters have been implemented to test the system. Simple filters accept input data from a single source (cp. filters such as 5,6 and 9). After a delay that simulates the processing delay, the data element from the input is forwarded to the succeeding filters. Fusion filters merge the data from multiple inputs into a single data element. Simple filters forward the input data unmodified. Fusion filters forward the largest received payload and drop the payload data from the other inputs.

For this evaluation, filter 9 has been set to be the bottleneck with an execution time of 1000 ms. All other filters execute with an execution time of 100 ms. The visualized delay values are the average values for the steady state of the system (after initialisation). The payload size has been varied between 1byte and 100kbytes. Figure 5 visualizes the measurement results. The proposed system always works with a far lower data delay than the reference system. While the payload is lower than the calculated maximum data-size (95kbyte/s), the payload seams not to have an influence on the data delay. At a payload size of 100kbyte the data transmission becomes the bottleneck causing the data delay to increase.

6. CONCLUSION

In this paper we presented a distributed algorithm that is able to synchronise any multi-source and multi-sink pipe-and-filter based application. The improvement to state-of-the art middleware systems is on the one hand the reduced data-delay. On the other hand, our system works in a distributed environment without any central component. This is very important for current and future applications which will be executed on a number of different devices. The proposed system automatically adapts to system changes that occur during runtime.

Our experimental results show that the proposed system

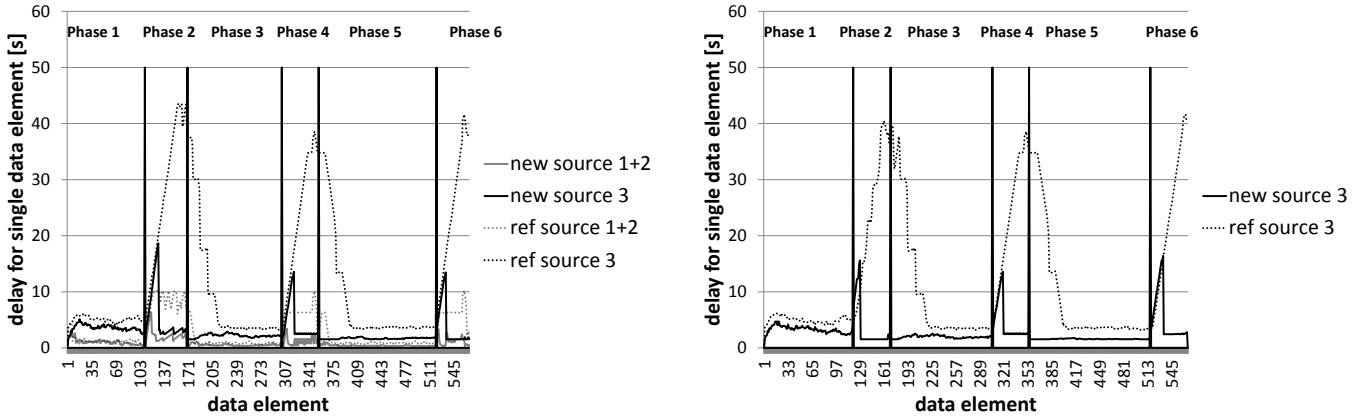

Figure 4: The data delay for single data-elements in the pipeline with 3 sources (sources one and two have the same delay). Comparison between reference system (dotted graphs) and own system (solid graphs). Data elements are received by filter 8 (left) and filter 10 (right), respectively.

manages to synchronize all sources even if the bottleneck is not in the data path of the specific source. The data delay is reduced by just-in-time data delivery. The event-based synchronisation does not require any continuous polling mechanism to be updated about the current system state. Our approach is beneficial for a number of applications. Applications working with live data as often used by smart camera systems benefit from the reduced delay and earlier visualisation of processing results. Applications working with prerecorded data experience a reduced memory requirement.

7. ACKNOWLEDGMENTS

This work was performed in the EPiCS project (Engineering Proprioception in Computing Systems) and has received funding from the European Union under grant no. 257906.

8. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] M. Bramberger, A. Doblander, A. Maier, and B. Rinner, "Distributed embedded smart cameras for surveillance applications," *Computer*, vol. 39, pp. 2006, 2006.

[6] Lukas Esterle, Peter R. Lewis, Xin Yao, and Bernhard Rinner, "Socio-Economic Vision Graph Generation and Handover in Distributed Smart Camera Networks," *ACM Transactions on Sensor Networks*, vol. 10, no. 2, pp. 24, 2014.

[7] B. Monajemi Nejad, S.A. Attia, and J. Raisch, "Max-Consensus in a Max-Plus Algebraic Setting: The Case of Fixed Communication Topologies," in *XXII International Symposium on Information, Communication and Automation Technologies*, Sarajevo, Bosnia and Herzegovina, 2009.

[8] B. Rinner and M. Quaritsch, "Embedded Middleware for Smart Camera Networks and Sensor Fusion," *Multi-Camera Networks: Principles and Applications*, July 2009.

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

[10] "GStreamer: open source multimedia framework," http://gstreamer.freedesktop.org, last visited: August 2012.

[11] 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.

[12] 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.

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

[14] H. Guggi and B. Rinner, "Increasing Efficiency of Data-flow Based Middleware Systems by Adapting Data Generation," in *Proceedings of the Seventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2013)*, 2013, pp. 189–198.