Real-Time Performance Evaluation of Media Pipeline Plug-in Architectures

V. N. Sentongo[1], K. L. Ferguson[1] and M. E. Dlodlo [2]
Council for Scientific and Industrial Research[1],
Meraka Institute, Pretoria, South Africa
Tel: +27 12 841 4266
and Department of Electrical Engineering
University of Cape Town[2]
email: {vsentongo, kferguson}@csir.co.za[1]; Mqhele.Dlodlo @uct.ac.za[2]

Abstract- Deployment of real-time applications has become ubiquitous in recent years. These applications (e.g. streaming video over the Internet) however, are processor-intensive and require fast video-processing techniques for more flexible, efficient operation. Media pipeline plug-in architectures have been introduced to provide these capabilities in form of extensible, integrated frameworks. Several architectures are in existence and many future designs are expected. This paper serves to design a set of measurement metrics on which to base the comparative performance analysis of the real-time capabilities of two of the architectures; DirectShow and GStreamer. This exemplifies the decision process through which a particular architecture can be isolated as that best suited for a specified real-time application. The metrics used are: average processing speed, plug-in scalability, threading overhead and programming complexity. Based on these metrics, effective comparison of other plug-in architectures can be made to determine their real-time performance in relation to a particular application.

Index Terms— GStreamer; measurement metrics; media pipeline plug-in architectures; Microsoft DirectShow; real-time applications

I. Introduction

Digital media is becoming more popular, with millions of people streaming various forms of media including music and videos over the Internet. In addition, real-time applications similar to video conferencing using instant messaging platforms (e.g. Skype), are now becoming a preferred means of long distance interactive communication. More recently, the increased demand for high definition television (HDTV) over Internet Protocol (IP)-based networks has led to the demand for more efficient, flexible and fast video processing techniques at the broadcasting end. In light of these developments, importance is drawn to the preparation of video for the encoding process required for such bandwidth-intensive applications. Raw captured multimedia needs to undergo various transformations and enhancements before it is transmitted to the receiver, with further quality improvements at the receiver end, before being rendered to the screen as a video image. This preparation is referred to as video processing and a few examples of this are: scaling down a large video image, compression, colour conversion after decompression etc.

Traditionally, the computing resources required to perform video processing functions were restricted to the domain of dedicated hardware or Digital Signal Processing (DSP) solutions. The current state of the art however, provides full flexibility to assemble useful video processing functions into media pipelines capable of real-time performance on standard consumer Personal Computers (PCs). Open software-based frameworks have been introduced for this purpose, the majority of which are structured as plug-in architectures. These plug-in architectures have been designed for software developers to build applications that are modular, customizable and easily extensible [1]. The basic structure of a plug-in architecture consists of a collection of filters or elements linked to form a pipeline. The basic class of objects for Microsoft DirectShow is referred to as a *'filter'* while that of GStreamer is referred to as an *'element'*. Each plug-in filter or element implements a distinct or specific video processing function [1].

Numerous media pipeline plug-in architectures are in existence today and many more will be designed in future. A few of these include: Microsoft DirectShow™, Microsoft Media Foundation®, Open source GStreamer™, RealNetworks Helix. Selecting which of the various media pipeline plug-in architectures to use for a particular real-time application is therefore of great interest. This selection can be based on various criteria. A few of these typically include: the operating system on which the application will run, the context in which the application will be used (e.g. video editing, video conferencing, live broadcasts etc.), the familiarity of the software design engineer with the architecture etc. The current selection process is often not based on the actual performance of the given application. Therefore, there is need to design a mechanism by which the most appropriate architecture can be selected that would best suit the particular application.

This work seeks to address which is the best technology or framework to use for a given application. This is performed by defining measurement metrics that generalize a fair evaluation of performance that can be applied to these and any new frameworks that may become available in the future. The two frameworks that were selected for comparison in this paper are: Microsoft DirectShow and the open source GStreamer. Microsoft DirectShow is a commercial proprietary framework and Microsoft Windows is the most popular and widely distributed operating system (OS) in the market to date [2]. Therefore due to its current widespread usage, Microsoft DirectShow has been included in this comparison. To diversify the comparison, a popular open source framework that could also be implemented on the Microsoft Windows Operating System (OS) was the criterion for selecting the GStreamer framework. However, it should be noted that our primary interest here is the measurement metrics themselves and not the specific framework. The metrics defined here can also be applied to all other frameworks.

The outline of the paper is as follows. Section II gives an overview of the operation and benefit of using media pipeline plug-in architectures. Section III describes the measurement metrics that were used to make the analysis and outlines the experiments that were carried out to investigate these metrics. In

Section IV, the experimental results are displayed and discussed. Section V presents the performance comparison and concludes on how a decision can be drawn on which media pipeline plug-in architecture best suits an application. Finally, recommendations for future work in this area are documented in Section VI.

II. Media Pipeline Plug-in Architectures

Media pipeline plug-in architectures have been designed to allow third party developers to add features to a host application that were not originally available to it without recompiling or accessing the source code of that application [3]. These features are modules referred to as *"plug-ins"* each with its own specialized task. The plug-ins are designed to be seamlessly plugged back into the host application in order to actuate the required improvements. A media pipeline consists of a succession of video processing filters (elements) such that the output of one element becomes the input of the next element based on their compatibility with one another as shown in figure 1 below.

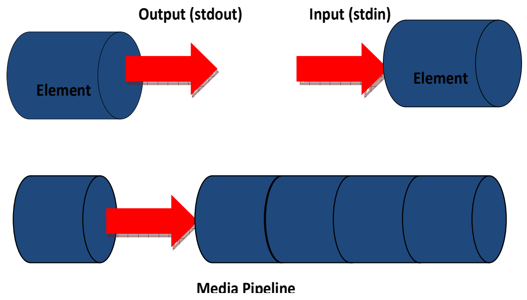

Figure 1. Output of one element forms input of the next to create a media pipeline [4]

The direct connection between the filters (elements) enables data to be transferred continuously through the pipeline rather than waiting for one video process to be terminated before the next process can begin [9].

A typical plug-in model is illustrated in figure 2 below. It consists of two basic parts, the plug-in host and the plug-in itself. The plug-in host code is designed such that well-defined areas of functionality can be provided by an external module of code (i.e. the plug-in). When the host code is executed, it uses the mechanism provided by the plug-in architecture to locate compatible plug-ins and load them dynamically. As a result, the benefit of using these plug-in architectures is that capabilities, that were not previously available, are added to the host application [3].

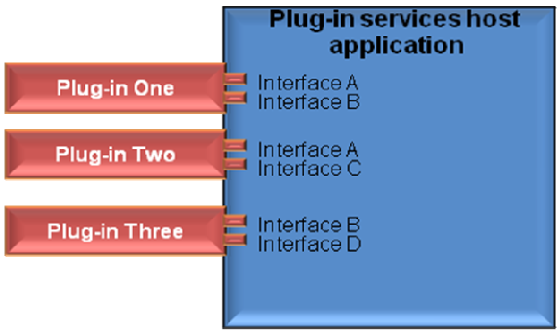

Figure 2. Plug-in host attached to three plug-ins [3]

III. Real-Time Performance Evaluation Metrics

Numerous video plug-ins exist. However, in order to maintain consistency in the tests conducted, a single video processing tool; the scaling tool, was chosen. This processing tool can be easily modified to work seamlessly with comparable efficiency on both pipeline architectures. A singe tool was also chosen to be continuously concatenated for all the test processes because our focus is on the application of the plug-in to the performance of the architecture under test and not on the functionality of the plug-in itself. Using the same base code, the scale filter was designed for DirectShow and the scale element designed for GStreamer mandating that the performance of the two processing tools remained identical across both architectures.

In order to evaluate fairly the performance of each of the media pipeline plug-in architectures involved, various performance parameters (measurement metrics) were isolated and different experiments carried out based on these metrics. The metrics used in this paper can be categorized as quantitative and qualitative as discussed below:

A. *Quantitative measurement metrics:*

These refer to the measurement metrics that exist in a range of magnitudes and can therefore be measured or quantified. Different experiments were conducted on the DirectShow and GStreamer pipelines and the general pipeline setup was shown in figure 3 below:

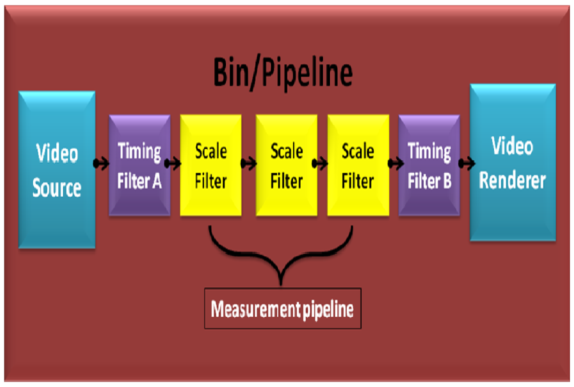

Figure 3. Simple measurement pipeline on which comparison of pipeline architectures is based

This pipeline included a video capture source (Logitech webcam that produced a live feed) and an incremental number of scale filters (elements) in the measurement pipeline that repeatedly scaled a video up and down from 320x240 to 176x144 pixels. It also consisted of two timing measurement filters at either ends of the pipeline and a video renderer to render the video to the screen. The scaling up and down of the video image was conducted with small values such that the performance of the pipeline architecture could be analysed as it performed a simple task. This ensured consistency across both platforms without making a great impact on the overall processing power utilised by the experiments.

The video from the webcam was rendered through the timing filters and the measurement pipeline to the screen. The experiments were run and measurements were taken and recorded. The quantitative metrics defined for the purpose of this comparative study include:

1) *Experiment 1: Average processing time of the 1st frame*

This is a measure of the average time in seconds that it takes a single or collection of video processing tools (e.g. scale filters/elements) within the pipeline to receive the very first frame of the video sequence, carry out the video scaling and transfer the first processed frame to the renderer. This metric can be used as a measure of the correlation between the scaling size and the performance of the architecture. It is an important metric because the average processing time of the 1st frame varies significantly between plug-in architectures and contributes to the total time taken to process all the remaining frames in the sequence. This metric would therefore determine which type of video application that the particular architecture would best serve.

For the GStreamer and DirectShow pipelines, timing tools were placed in each pipeline to measure the time at which a video frame moved through each of the two timing tools A and B located at either ends of the measurement pipeline as shown in figure 3 above. The time measured at each timing tool is referred to as the *reference time*. A single time measurement represented the time taken for a single video frame to pass through the pipeline from the first inserted timing tool A, through the measurement pipeline to the second timing tool B. The processing time was measured using the difference between the reference times recorded by the timing tools for each video frame. These measurements were averaged over 100 tests where the video was played and the processing time of the 1st frame recorded. To calculate the average processing time for the first frame for each scale filter (element) combination, the following equation was used:

$$\textit{Average Processing Time} = \frac{1}{N}\sum_{i=1}^{100}(stop\_time_i - start\_time_i) \quad (1)$$

where*:* N = Total number of measurements taken (100),
start_time = time taken from Timing Filter (Element) A,
stop_time = time taken from Timing Filter (Element) B.

The first timing measurement was taken with only one video processing tool (scale filter/element) in the measurement pipeline. Successive average timing measurements were repeated on the pipeline after incrementing the number of filters by one up until the point where the pipeline could no longer sustain any further additions.

The main limitation of this metric was the interference caused by the underlying OS processes. Despite the fact that all other applications were turned off at the time of the experiments, the underlying Central Processing Unit (CPU) processes still contributed a small amount of interference or delay.

2) Experiment 2: Average processing time of successive video frames

This metric is an extension of the previous experiment where all the subsequent frames passing through the measurement pipeline in the sequence were now included. The timing measurements evaluated in *Experiment 2* commenced with the time taken for the 2nd video frame to move through the measurement pipeline and continued to include the other subsequent video frames that were sent through the pipeline. For every filter added to the pipeline, 100 frame traversal times were averaged and the measurement was recorded. The purpose of this experiment was to determine the long term steady-state performance of the pipeline under test.

The average processing time of the successive video frames was calculated in the same way as in *Experiment 1* above and therefore had the same experimental limitations.

3) Experiment 3: Plug-in scalability

The scalability of the pipeline is a measure of how many filters (elements) that the different plug-in architectures can support while maintaining the required efficiency. The validity of this metric is based on the fact that it is important to know the maximum number of processing functions that a particular architecture can support before the system fails. The yield point at which the system fails completely is defined here as the *maximum scalability* of the plug-in architecture. This experiment was conducted by adding successive video scaling tools to the pipeline and identifying the point at which no more filters (elements) could be accommodated without rejection by the pipeline.

The limitation of this metric is that it is directly tied to the PC hardware resources available. Therefore the absolute value for the maximum scalability indicates the point at which all the CPU resources are consumed for a given PC. The relative measure between architectures on the same PC was used for the comparison and therefore still provided a good measure of relative performance.

4) Experiment 4: Plug-in (threading) overhead

This is defined as the number of active threads that are involved during the processing of the video frames. The number of threads involved in a process was measured using the *Task Manager* application of the Windows OS. This metric is important because an increased number of threads doing similar amount of work is indicative of an increase in overhead but can be balanced with a gain from multi-core CPUs.

B. Qualitative measurement metrics:

These are the measurement metrics that are based on subjective attributes that influence the software development process. The qualitative measurement metric used in this analysis was:

1) Experiment 5: Programming interface complexity

This is a measure of the degree of difficulty or complexity that was involved in designing the filter (element), creating the media pipeline, inserting the filters into the pipeline and taking the measurements. This is an important metric because it is an indirect measure of the cost of development. The comparable programming complexity of developing software for a particular plug-in architecture and also the ease with which third parties can design additional plug-ins, determines the time required to develop a given application.

The limitation of this metric though, is that it is based on the opinion and experience of the developer. However, using the same developer, designing the same task for the different architectures justifies the reasonable validity of this subjective metric.

IV. Experimental Results

In order to make a comprehensive performance comparison between the two media pipeline plug-in architectures (GStreamer and Microsoft DirectShow), this section has been sub-divided according to the measurement metrics stated above that have been used in this paper as a basis of comparison.

A. Quantitative measurement metrics:

1) Average processing time of the 1st video frame

A comparison of the average processing times of the first frame of both GStreamer and Microsoft DirectShow is presented in figure 4 below:

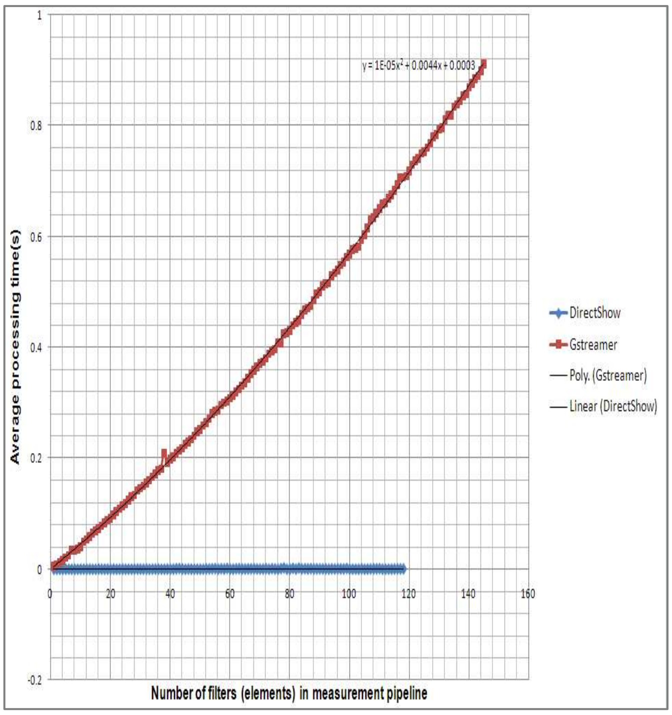

Figure 4. Average processing time vs. number of filters (elements) of the 1st video frame in the measurement pipeline for GStreamer and Microsoft DirectShow

Figure 4 shows that, with DirectShow, the average processing time for the 1st frame remained constant at a value below 0.001s regardless of how many filters were added to the pipeline. Hence the 1st video frame had no effect on the overall average processing time of the architecture. However, with GStreamer, the graph indicates a 2nd order polynomial increase in the average processing times as the number of elements in the pipeline was also increased. The equation of this curve of best fit was:

$$y = x\,(0.00005x+0.0044) + 0.0003 \qquad (2)$$

Equation (2) illustrates that the number of elements in the pipeline is a factor in the increase in overhead as more elements are added to the pipeline. The graph was seen to lose its linearity as more elements were added and hence its efficiency since the 1st frame took longer to traverse through the pipeline. As more elements were added to the pipeline, the architecture overhead also increased going up to a value of 0.705s for 118 elements. Therefore the 1st frame processing time of GStreamer had a significant impact on the overall processing time of the plug-in architecture.

This indicates a major limitation with GStreamer for large element bins and will impact the initial waiting period before the successive frames begin to flow through the pipeline.

2) Average processing time of successive frames

In figure 5 below, the processing times of both GStreamer and Microsoft DirectShow (excluding the first video frame) are presented. Figure 5 shows that as the number of filters in the DirectShow pipeline was increased (from 1 to 118 filters); there was a corresponding linear increase in average processing time (from 0.002s to 0.771s). The black trend line indicated a linear line of best fit with the following equation:

$$y = 0.0065x + 0.002 \qquad (3)$$

This implied that the average processing time of this scale filter could be predicted due to the linear relation between the average processing time and the number of filters in the measurement pipeline. The linear relationship further implied that the architecture overhead did not increase with increase in the number of filters in the pipeline.

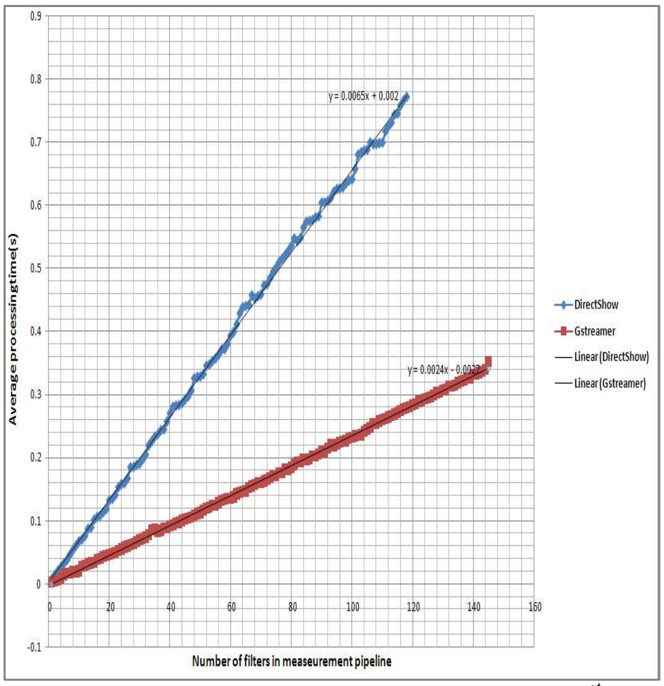

Figure 5. Average processing time excluding the 1st frame vs. number of filters (elements) in the measurement pipeline for Microsoft DirectShow and GStreamer

With regards to GStreamer, figure 5 shows that as the number of elements in the pipeline was increased (from 0 to 145 elements), the processing time values correspondingly increased linearly (from 0s to 0.352s). The trend line of best fit has an equation of best fit as shown below:

$$y = 0.0024x - 0.0027 \qquad (4)$$

Equation (4) implies a smaller constant increase in the average processing time of the GStreamer elements. Therefore, using this equation, the average processing time of the architecture could also be predicted using the number of scale elements in the pipeline. The slope of the graph derived from the equation of the line of best fit above is equal to a constant (0.0024s). Therefore the number of elements in the pipeline can be used as a direct measure of the average processing time of the successive video frames passing through the pipeline.

As a result, considering the average processing time (excluding the waiting time for the first frame) GStreamer showed better performance. The implication of this is, despite the fact that 1st frame processing time (time taken for first frame to pass through the pipeline) with GStreamer was much more than that of Microsoft DirectShow; the time taken for the successive frames to move through the pipeline was less.

Consequently, a decision can be made on which architecture to use depending on whether a long initial setup time would greatly impact the overall efficiency of the application.

Average time per filter (element)

The average time per filter, is a measure of the gradient (slope) of the graph in figure 5 above. Due to the linear nature of the relationship it is also expected to be constant. To calculate the average time per filter for each scale filter combination, the following equation was used:

$$\textit{Average Time per Filter} = \frac{\text{(Average processing time)}}{\text{(Number of filters in measurement pipeline)}} \quad (5)$$

A comparison of the average processing time per filter (element) of video frames (excluding the 1st frame) that passed through both GStreamer and Microsoft DirectShow pipelines is presented in figure 6 below.

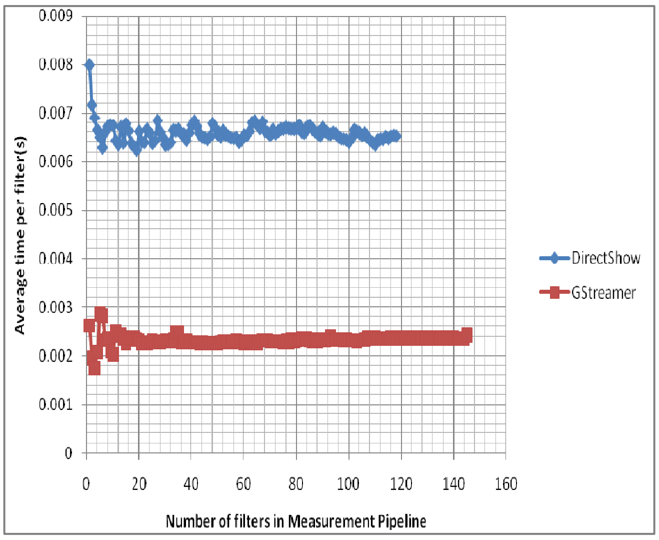

Figure 6. Average time per filter (element) vs. number of filters (elements) in the measurement pipeline for Microsoft DirectShow and GStreamer

The graph in figure 6 above shows that with DirectShow, initially when it had a few filters in the measurement pipeline, there was a high initial time per filter. This is because the first filter involved in scaling down the image performed a copious amount of work in scaling the live video image from the webcam (320x240 pixels) to the first arbitrary scale value chosen for this experiment (176x144 pixels). This work done was much greater than that required by the successive filters to scale the image from (176x144 pixels) up to (180x148pixels) then scale it down again to (176x144 pixels) and so on. Thus the influence of the first filter became less dominant on the average time per filter of the successive filters added to the pipeline due to the fact that the successive work done became relatively constant. Consequently, as illustrated in the figure 6 above, there was a marked difference in the average time per filter from 0.008s for the 1st filter to 0.0072s for the 2nd filter and then further down to 0.0069s for the 3rd filters respectively added to the pipeline.

However, with time, as more filters were added with more regular scaling value patterns, the time per filter evened out and began to show very slight variation as the number of filters added to the pipeline increased from 0 to 118. The trend fluctuated slightly above and below the value of 0.0066s. This value coincides with the constant value previously derived from the equation (3) of the graph in figure 5 above,

For GStreamer, figure 6 shows that once again initially, there was a sharp reduction in work that was done to scale the live video image received from the webcam from a value of 320x240 to a value of 176x144 pixels. The time taken to effect this large initial pixel size reduction by the first element was long (i.e. 0.0026s). With the immediate succeeding elements, this large amount of work done by the first element dominated the average time per element. The work done by the 2nd element to increase the pixel size from 176x144 to 180x148 pixels in comparison was quite small. Therefore the average time per element in turn also dropped to 0.0019s. This time further decreased to a value of 0.0018s as the image was now scaled down again to 176x144 pixels.

However, as the number of elements in the pipeline increased, the effect of the initial dominant element became less significant and the average time per element evened out to an approximate value of 0.00237s. The graph now began to show very slight variation as the number of elements added to the pipeline increased from 20 to 145. This average value of 0.00237s is approximately equal to the slope illustrated by the line of best fit of the graph in figure 5 in the GStreamer equation (4) above.

This graph therefore indicates that the average processing time per element (excluding the first frame) for GStreamer is much lower than that of a Microsoft DirectShow filter in a pipeline.

3) *Plug-in scalability*

Figures 4, 5 and 6 above illustrate that Microsoft DirectShow was only able to process a maximum of 118 filters but GStreamer was capable of processing up to 145 elements in the pipeline without rejecting them. Therefore from these experiments, when using the scale filter (element), GStreamer was more scalable than Microsoft DirectShow. More processing functions could be carried out in succession with GStreamer than they could with Microsoft DirectShow.

4) *Plug-in (threading) overhead*

The difference in the number of threads actively involved in the processing of video in a Microsoft DirectShow (11 threads) and a GStreamer pipeline (10 threads) was measured using the Task Manager of the Windows OS and was found to be constant at 1 thread. This may be considered as negligible and therefore, the plug-in threading overhead of the two media pipeline plug-in architectures was approximately the same.

B. *Qualitative measurement metrics:*

1) *Programming interface complexity*

The programming complexity is a measure of the ease of designing and inserting new plug-ins into the host application. Microsoft DirectShow is based on the Component Object Model (COM) framework [8]. Therefore, before setting up the framework on which a filter can be built, several factors need to be considered. Some of these include: the different base classes from which each filter will inherit[7], locating the libraries that contain the different properties of the filter, the settings interfaces of each base class that are required to seamlessly integrate with the host application, the property pages needed for a particular functionality etc [6]. In addition, the threading model for DirectShow filters is complex. Thus the programming complexity of Microsoft DirectShow for developers building applications for this media pipeline plug-in architecture is high.

GStreamer presented significantly lower programming interface complexity due to its simplicity of structure and encapsulation of most of the complicated setup tasks. A GStreamer element, once written and complying with all the requirements stipulated by the GStreamer Application Programmer's Interface (API), is programmatically registered as a GStreamer element and is thus ready for use. The properties of each element that GStreamer expects to observe in order to recognize and register it as a GStreamer element are the same [5]. However, in order to make the distinction between different types of GStreamer elements, the *basetransformclass* is used. A "transform class" is a GStreamer pre-made class that consists of various C language structures that may be manipulated to actuate the unique functionality of the new GStreamer element under development. Manipulating the collection of C structures and implementing the desired function was the basic requirements to develop a simple GStreamer element [5]. It was deemed significantly simpler to design with than the COM architecture of Direct Show.

V. Conclusions

The main objective of this work was to design a set of measurement metrics on which to base the comparative performance analysis of the real-time capabilities of different media pipeline plug-in architectures. Experiments were developed, compiled and executed on the two selected architectures, Microsoft DirectShow and GStreamer. The following conclusions are a summary of the real-time performance comparison results represented by Table 1 below:

TABLE I. Real-time performance comparative analysis of Microsoft DirectShow and GStreamer

Comparison parameter	Microsoft DirectShow	GStreamer
Average Processing time of the 1st frame	Remained constant at a value of 0.001s regardless of the number of filters in the pipeline.	Increased as the number of elements in the pipeline increased with the 2nd order equation: $y = x(0.00005x+0.0044)$
Average Processing time excluding the first frame	Increased linearly at a higher constant rate with the equation: $y= 0.0065x + 0.002$	It increased linearly at a lower constant rate with the equation: $y = 0.0024x-0.0027$
Average Processing time per filter (excluding the 1st frame)	Higher average processing time per filter (0.0065s)	Lower average processing time per filter (0.0024s)
Scalability	Can only hold up to 118 filters.	Can hold up to 145 elements.
Plug-in(threading) overhead	11 threads actively involved.	10 threads actively involved.
Programming interface complexity	More complex programming interface.	Less complex programming interface.

It may be concluded that each of the media pipeline plug-in architectures under test had different individual areas of better performance. Using the scaling tool, GStreamer was more scalable, exhibited less threading overhead and had less programming complexity than Microsoft DirectShow. However, Microsoft DirectShow had a lower overall average processing time than GStreamer. Selecting an appropriate media pipeline plug-in architecture would therefore depend on the trade-off that the application under development can accommodate.

VI. Recommendations

A few recommendations and suggestions of future work are listed here below.

With the advent of multi-core processors, multi-threaded programming is becoming increasingly efficient. An investigation into how this can be used to reduce the average processing time and increase overall performance of the media pipeline plug-in architectures could provide significant improvement.

A relatively simple video scaling tool was used for the purpose of this investigation. Future work needs to be conducted using a wider variety of plug-ins to investigate the effect of implementation of more complex tasks and the consequent effect on plug-in complexity and efficiency.

For GStreamer, the initial overhead required for the first video frame passing through the pipeline is high particularly for pipelines with many serial elements. An investigation into the cause of this increase, as the number of elements in the measurement pipeline increases, needs to be conducted. In addition, further investigation can consider how this overhead may be reduced. Reduction of this overhead will significantly reduce the average processing time taken by the GStreamer media pipeline plug-in architecture.

Other performance parameter metrics exist on which to base the comparison of the output video. These include memory management, processor utilisation, memory footprint, scalability of parallel filters (elements), output picture quality etc. Future work can focus on investigating these other metrics to give a more detailed overall comparison of the video processing performance of the different media pipeline plug-in architectures.

References

[1] On Plug-ins and Extensible Architectures, Dorian Birsan, Eclipse, Vol. 3, Issue 2 (March 2005), ISSN: 1542-7730, pp 40 - 46

[2] Operating system market share http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8 (Retrieved: March 25, 2010)

[3] Introduction to Plug-ins, Apple Computer Inc.2003 http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFPlugIns/CFPlugIns.pdf (Retrieved: 23 May, 2008)

[4] Pipes: A brief Introduction. http://www.linfo.org/pipes.html (Retrieved: 27 May, 2008)

[5] GStreamer Application Development Manual (0.10.23.1), Wim Taymans, Steve Baker, Andy Wingo, Ronald S. Bultje, Stefan Kost

[6] Direct Show MSDN http://msdn.microsoft.com/en-us/library/ms783323(VS.85).aspx (Retrieved :January 15, 2010)

[7] Plug-in Architectures *Jeffrey Stylos User Interface Software, Fall 2004.*

[8] Programming Microsoft DirectShow for Digital Video and Television, *Mark D. Pesce, Microsoft Press, 2003.*

[9] A Pipeline Development Toolkit in Support of Secure Information Flow Goals, *Philip Tricca, April 2009 , 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies, Article No.: 66*

Veronica Sentongo received her BSc. Electrical Engineering degree in 2007 and MSc. Electrical Engineering in 2010 from the University of Cape Town and is presently working at CSIR, Meraka Institute on the Real-time video coding project. Her research interests include real-time applications, video coding and streaming, image processing, and telecommunications applications.