A Simple Multithreaded C++ Framework for High-Performance Data Acquisition Systems

Rolando Inglés, Piotr Perek, Mariusz Orlikowski, and Andrzej Napieralski

***Abstract*—The data acquisition systems must be capable of process all the data produced by the source to ensure the highest level of accuracy, especially when it deals with hard real-time system monitoring task. However, the production of data is faster than the process to acquire and to process such a data. Using concurrency approach is an alternative to obtain the required level of performance and data processing.**

This paper presents the comparison between various *C++* frameworks that by using multithreading technology and ring-buffer data structure allow data transfer in concurrent way. The comparison is based on the time interval between the instant when data is published and the instant when the data is gathered. These latency measurements have been taken using the configuration of one producer and two consumers for all evaluated frameworks. The results show that using standard C++ libraries to develop a simple framework it is possible to achieve suitable performance in order to fulfill the requirements of the high performance data acquisition systems described.

***Index Terms*—mutithreading; ring-buffer; real-time system, data acquisition system, c++, linux.**

I. Introduction

This experimental evaluation of *C++* multithreaded frameworks has been conducted as part of a project where real-time monitoring of a power generator is required. Precisely this monitoring process implies massive data production generated for high-speed cameras. This amount of images produced by the cameras must be transferred in the fastest possible manner from the machine device to the control system. As seen in Figure 1, the process of gathering the raw data from hardware devices and propagation of them to these specialized sub-systems of control and archiving, is considered in the abovementioned project as Data Acquisition System.

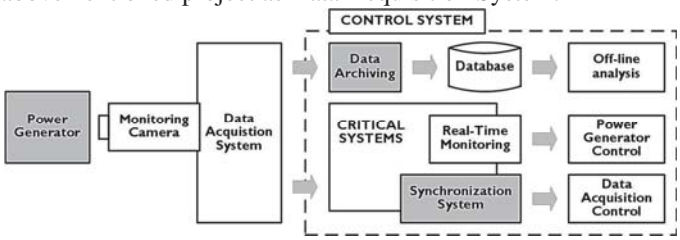

Fig. 1. Monitoring data workflow.

This work was supported by Erasmus Mundus/LAMENITEC project. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of Erasmus Mundus/LAMENITEC project.

R. Inglés, P. Perek, M. Orlikowski and A. Napieralski are with the Department of Microelectronics and Computer Science, Lodz University of Technology, ul. Wolczanska 221/223, 90-924 Lodz, Poland (e-mails: {roling,pperek,mariuszo,napier}@dmcs.pl)

In the context of the Data Acquisition System a variety of methods for transferring raw data from data sources to the processing systems has been propounded in the past. Today with the advent of multiprocessors architecture, taking advantage of this processing power gains importance. Most of the operating systems provide interfaces to use the multiprocessing resources; the most important is the multithreading technology.

In the context of the Data Acquisition System a variety of methods for transferring raw data from data sources to the processing systems has been propounded in the past. Today with the advent of multiprocessors architecture, taking advantage of this processing power gains importance. Most of the operating systems provide interfaces to use the multiprocessing resources; the most important is the multithreading technology.

The designer of *C++* programming language, Bjarne Stroustroup advices the use of threads when several tasks in a program need to progress concurrently [1] and Anthony Williams [2] defines concurrency as the parallel execution of different tasks at the same time. Ideally each of these tasks is running in its own processor.

Precisely in systems where high performance data transfer is required, the data is accessed concurrently and most of the tasks should be executed concurrently, and this can be achieved with C++ using threads.

It is important to emphasize that according to the project requirements, the software solution must be developed using *C++* programming language in a **NIX* system environment.

II. The Data Acquisition System

A. The Raw Data Source

The aforementioned power generator is monitored with *EoSens® 3CL Full CL MC3010* high-speed cameras. The Figure 2 shows the maximun frame rate for given resolution of these cameras provided by the manufacturer [8].

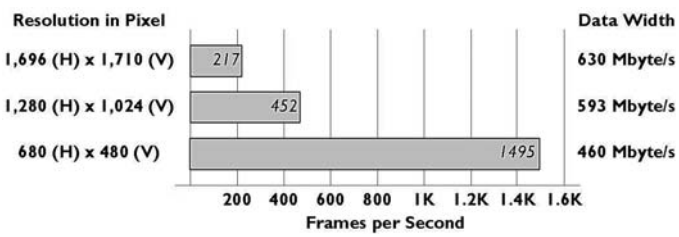

Fig. 2. Frameset factory profiles of EoSens® 3CL Full CL MC3010 high-speed camera [8].

Based on this information the cameras are able to produce *630 Mbytes* of raw data when are configured in the highest resolution *1696x1710 pixels*. According to thechnical documentation the cameras are able to produce up to 180,000 frames per second with reduced resolution [8].

B. The Control System

The Control System executes the process for controlling the appropriate operation of the power generator. These controlling tasks depend on the physical time when results are produced, hence is considered a real-time system [9]. To carry on such a task, the generated images must be sent toward two sub-systems namely, Data Archiving and Synchronization System, represented in Figure 1.

The Data Archiving subsystem is dedicated for permanent data storage. This storage task is not considered critical, because there is not restrictive time and if some failure happen is its operation does not imply some catastrophe for the whole system.

While due to their operation mode, the Data Archiving subsystem is a soft real-time system, the Synchronization System is a well-defined hard real-time system because its operation is extremely critical for the monitoring and control of the power generator [9]. First, a 100% of accuracy is required because all received raw data must be transferred to the Real-Time Monitoring subsystem to detect any failure in real-time, and secondly a 100% of reliability is demanded to evaluate permanently the quality and consistency of the transferred data. To fulfill these requirements, the Data Acquisition System must be able to transfer the frames from the data source toward the Control System in the same rate that they are produced.

C. Data Acquisition System Operation

Figure 3 presents a generic skeleton of the Data Acquisition System. The schema is made up by one producer and two consumers. The producer is responsible for taking the message from the source device, and for filling the available slot in the ring-buffer.

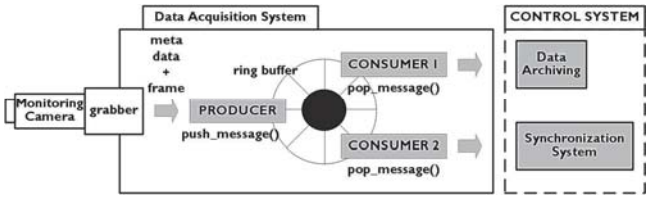

Fig. 3. Data Acquisition System block diagram.

Every consumer process is executed in its own processing thread. Specifically, the first consumer is oriented to manage complete data messages and sent them toward the archiving subsystem. And the second consumer is in charge of managing the synchronization data to verify the data accuracy of each event.

D. The Transferred Data

The data sent toward the consumers by the producer process has TIFF standardized format [11]. The message *per se* is equivalent to meta-data + raw-data. The raw-data is just the image generated by the cameras and the meta-data is the identification header of each frame.

III. Related Works

In the early stage of this research, to establish a state of the art starting point, different C++ solutions using multithreading and managing a ring-buffer data structure were tested; nevertheless only the CxxDisruptor and FastFlow were suitable to reach the appropiate level of performance and accuracy required in the preleminaries testing phases.

A. The Disruptor Framework

The Disruptor is an inter-threading communication framework based on a shared ring-buffer data structure. This framework, developed by LMAX-Exchange software development team, was designed to achieve a high performance alternative to bounded queues for exchanging data between concurrent threads [3]. This solution was originally developed using Java Technology; however there are several versions of this solution ported to C++.

In the early stage of the evaluation process, several ported versions of the Disruptor were tested, nevertheless only the *CxxDisruptor* of Henrik Baastrup [4] provided the expected accuracy levels.

This approach pre-allocates a fixed number of slots in a *C++* array and deploys a bitwise arithmetic algorithm to reuse the ring buffer slots. The *CxxDiruptor* approach is based on atomic primitives to control which thread has the current access to a specific slot in the ring-buffer.

B. The FastFlow Framework

The *FastFlow* is a *C++* parallel processing oriented programming framework advocating high-level and pattern-based parallel programming [5]. This framework has been developed to provide a parallel programming abstraction and simplify the development of multi-core platforms.

The main design philosophy of *FastFlow* is to provide application designers with key features for parallel programming (e.g. time-to-market, portability, efficiency, and performance portability) via suitable parallel programming abstractions and a carefully designed run-time support [5].

IV. The Simple C++ Framework

The relevance to make this comparison is based on the precept that it is possible to develop a multithreaded solution for high-performance data transfer with a minimalistic approach using standard *C++* libraries. To meet this premise, three versions of the *C++* simple framework have been developed. The first version has been developed using the *C++11* standard libraries. The second deployment by using the *Boost C++* library and the third approach uses the POSIX Thread API.

Only two classes form the C++ simple framework, both of them are described below.

A. The Monitor Class

The monitor is the class responsible for keeping track of each event read. Each consumer has its own read slot controller implemented with a *std::vector<int>*. The monitor receives the fill request from the producer by using the

monitor::publish method, and from this instant, the monitor transfers the request to the ring buffer and the ring buffer class increases by one the slot counter, allowing to extend the gap between the last slot read for each consumer and the last slot filled for the push request.

The monitor class receives the consumers request to gather one specific message by using the *monitor::gather* method. But this is the real work of monitor, keeping track of each read slot by each consumer. Before transferring the requested message of a specific slot to the ring buffer class, the consumer must ask the monitor class if there is a slot to be read by using the *monitor::next* method; if any, the consumer must use the method *monitor::gather* to get the event data. Basically, the *monitor::gather* is an interface to the *ring_buffer::gather* method but being controlled by the monitor for tracking reasons.

B. The Ring-Buffer Class

The ring buffer class deploys a basic ring-buffer data structure, when the ring-buffer is created through the class creator *ringbuffer<message_type>::ringbuffer(n_slots)* a new array for storing the memory addresses of the *<message_type>* data type with *n_slots* of elements is created. Thus the entire memory of the *ring-buffer* is pre-allocated.

The ring-buffer meets the required tasks by means of three simple methods described in Table II.

TABLE I.
C++STL RING-BUFFER METHODS

Method	*Description*
ringbuffer::get_slot_pointer(slot)	Returns the memory address of the specific slot number. This method is called by the producer in order to fill this slot with the new message.
ringbuffer::publish(message)	Increases the slot counter, specifically the tail of the ring buffer. This method is called by the producer to fill the slot with the specific message data.
ringbuffer::gather(slot)	Returns the address of the specific slot. This method is called by the consumer to gather the specific and available message

C. Producer & Consumer Algorithms

The Figure 4 presents the basic operation of both sides of the data processing involved in the minimalistic C++ solution. Basically the *producer* asks the *monitor* for the next free slot in the *ring-buffer*. When the free slot address is returned the *producer* fills it with the content of the new *message* and then calls the publish method to inform the *monitor* that there is a new *message* that must be sent to all *consumers*.

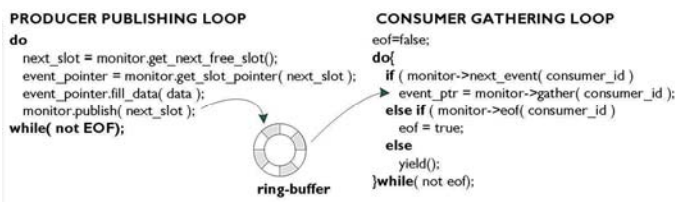

Fig. 4. Publishing and gathering algorithms.

Within the consumer algorithm there is a call to the instruction *thread::yield* in order to inform the operating system that this specific thread can be rescheduled. The call of the instrucction *thread::yield* is necessary to make a brief pause when there is no a message to be gathered or when the last message has not been reached yet.

D. Thread Affinity

The POSIX scheduling interface provides a mechanism to establish a CPU affinity mask in order to establish on which processor a particular thread can be executed. At the moment when the consumer object is created, the affinity mask is configured by calling the function *sched_setaffinity*.

E. FIFO Scheduling

In the version of the Simple C++ Framework using POSIX thread API, the consumer threads are created with the SCHED_FIFO scheduling policy giving them high priority of execution. This mechanism of scheduling has been incorporated in order to execute with more priority the consumer processes than the producer process. It is necessary to prioritize the execution of the consumers processes because in the scheme producer/consumer the rate of pushing the data in the buffer by the producer is faster than the rate in which this data is taken from the buffer and processed by the consumers.

V. EXPERIMENTAL EVALUATION

A. The Testing Enviroment

The evaluation has been done with a RedHat Enterprise Linux 7.1 (Maipo) 64bit operating system, running in a Intel® Core™i5 CPU M 520 @ 2.40GHz system with 4GB the RAM memory. The kernel version is 3.10.0-229.el7.x86_64.

This Linux distribution is provided with the g++ (GCC) 4.8.3 20140911 (Red Hat 4.8.3-9) compiler and all source code has been compiled using the flag -O3 to take advantage of all the optimization improvements provided by this compiler.

The input file used contain 2,348 frames with 680x480 pixels resolution each one, nevertheless only 1,495 have been used in order to simulate the production of frames by the camera when is configured in 680x480 pixels resolution. With these settings, the camera produces the highest frame rate per second, precisely 1,495 representing 460 Mbytes of raw data to be sent it each second.

B. The Timing Process

According to what is established by the project for which this assessment has been performed, the data should be taken from some memory buffer in order to simulate the frame production from the cameras. This is achieved through the mapping file mechanism provided by all of *NIX like operating systems. It should be pointed out that mapped memory does not only allow accessing the content of a file in shared way, but it can be applied to communication between processes as well [10].

Following the pattern presented in Figure 5, the *producer* maps the sample data file into memory by means of *mmap* call [10]. Then reads sequentially the mapped test file frame

by frame, and fills the ring-buffer with each frame. Before invoking the method to submit the event in the ring-buffer, each event is identified with an integer number. This identification number is necessary to associate every timing snapshot taken in the producer side with their counterpart in the consumer side, in order to calculate the latency of each event of both consumers.

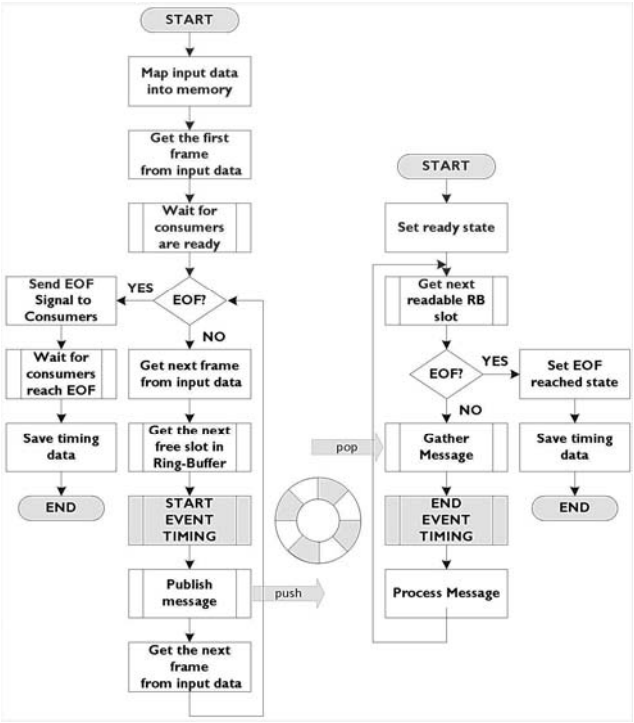

Fig. 5. Publishing and gathering algorithms.

The sub-routine START EVENT TIMING is executed precisely before the event is published to a specific slot of the ring-buffer, internally in this sub-routing, the start time measurement is taken by calling the time function *clock_gettime(CLOCK_REALTIME,&timespec)* where CLOCK_REALTIME refers to the system-wide real-time clock and &timespec to the data structure where the clock value are stored.

Concurrently to the event publishing task, each consumer checks if there is some event to be read. If any, the consumer takes the message. Immediately after the message is gathered by the consumer, the sub-routine END EVENT TIMING is called and inside of this sub-routine the stop time measurement is taken by calling the same function shown above.

After reaching the end of the data, each consumer dumps all the time measurements stored in the temporary array to a binary file. Each consumer has its own binary file with measurements. Also, the producer, after waiting for the end of each consumer, dumps all the time measurements to its own binary file.

C. The Data Processing

The output data files containing time measurements were generated after execute each C++ framework for one million of messages and with different number of slots in the ring-buffer, starting from 256 and being increased by twice up to 8192 slots. This approach has been taken because of the operation of *CxxDisruptor* and the developed *C++ frameworks* use bitwise to calculate the next wrap of the ring-buffer, and the number of the slots must be a 2^n number.

For each execution of the timing process, three files are generated, one file generated by the *producer* containing the time measurements of the instant when the message is published and two files produced by the *consumers* containing the time measurements when the messages is gathered.

In batch mode the timing data files are processed verifying the correspondence of each measurement by using the identification of each message. At the same time, the publishing time is subtracted from the gathering time. In this point, there are two time differences, one for each consumer. The greatest difference between the two is taken as the latency time taken for a specific message. The results analysis is based on this latency time calculation.

VI. Results

A. Framework Comparison

By means the latency averages reported in Table II it can be established that the *C++Boost* deployment has the lowest latency for each configuration of slots number in the ring-buffer evaluated. Nevertheless, the latency average increases directly proportional with the number of slots, in contrast with the assumption that more slots represent less latency. This behavior is also presented by *C++STL* and *CxxDisruptor* frameworks.

TABLE II.
C++ FRAMEWORKS AVERAGE LATENCY TIME*

# Slots	*C++ Boost*	*CxxDisruptor*	*FastFlow*	*Pthread*	*C++ STL*
256	201	202	829	221	239
512	174	213	823	200	251
1024	207	210	825	213	238
2048	197	4298	830	230	312
4096	207	4531	821	209	415
8192	210	9351	823	225	456

* nanoseconds.

Only the *FastFlow* framework presents a regular trend regardless of the number of slots configured in the ring-buffer. However, the latency averages for each configuration of slots are higher than the remaining frameworks. Only the *CxxDisruptor* presents more average latency in the 2018, 4096 and 8192 slots than *FastFlow* framework.

B. Latency

The Figure 6(b) reflects the latency measurements for the *CxxDisruptor* framework and reports the expected behavior for a message latency because the 97% of the messages were sent in two hundred nanoseconds or less. Furthermore, this behavior is constant for all configurations of slots in the ring-buffer evaluated.

The *FastFlow's* latency measurements are represented in the Figure 6(c). This framework is not suitable for high performance data transfer because the latency measurement

results are higher than four hundred nanoseconds and the gap goes from four hundred till five thousand nanoseconds which is large compared with the rest of the frameworks whose gaps are between two hundred and four hundred nanoseconds.

C. Affinity Mask

The comparison between the Figure 6(e) against Figure 6(f) shows that most of events have a latency around three hundred nanoseconds. This means that the usage of the POSIX scheduling affinity mask it does not have a considerable impact in performance. However, a consistent latency time is shown in the Figure 6(e). This kind of behavior is useful in real-time systems design because a prediction of latency time can be done with a high percent of certainty.

D. FIFO Scheduling

The creation of consumer threads with the *SHED_FIFO* policy in this experiment has not represented an increment in performance. The frameworks implemented with *C++ Boost Library* and *C++ STL*; Figure 6(a) and Figure 6(b) respectively, have produced similar results than the frameworks using *POSIX thread* scheduling policy FIFO.

In addition, the framework using C++ Boost Library communicates most of the events with a latency time around the two hundred nanoseconds (see Figure 6(a)), this is a better result that reported by the frameworks using POSIX thread technology.

E. CPU Usage

The experiment reports that, the CPU usage is between the 60% and 80%. Nevertheless the *FastFlow framework* makes use of the CPU above of the 80% and in some parts of the experiment reaches the 100% of CPU utilization (see Figure 7(c)). This aspect has direct relationship with the latency time reported in the Figure 6(c). The algorithm used for the *FastFlow framework* produce high levels of contention, thereby increasing the latency levels and the CPU overload.

The Figure 7(d) evinces that the solution using *C++ STL* reaches the 100% of CPU utilization in some points of the processing. This peaks of 100% CPU usage are due to that this simple solution uses the *std::this_thread::yield* function within the a *spin-lock* loop, yielding to the operating system the task of rescheduling the thread and evidently this mechanism produce a CPU overload.

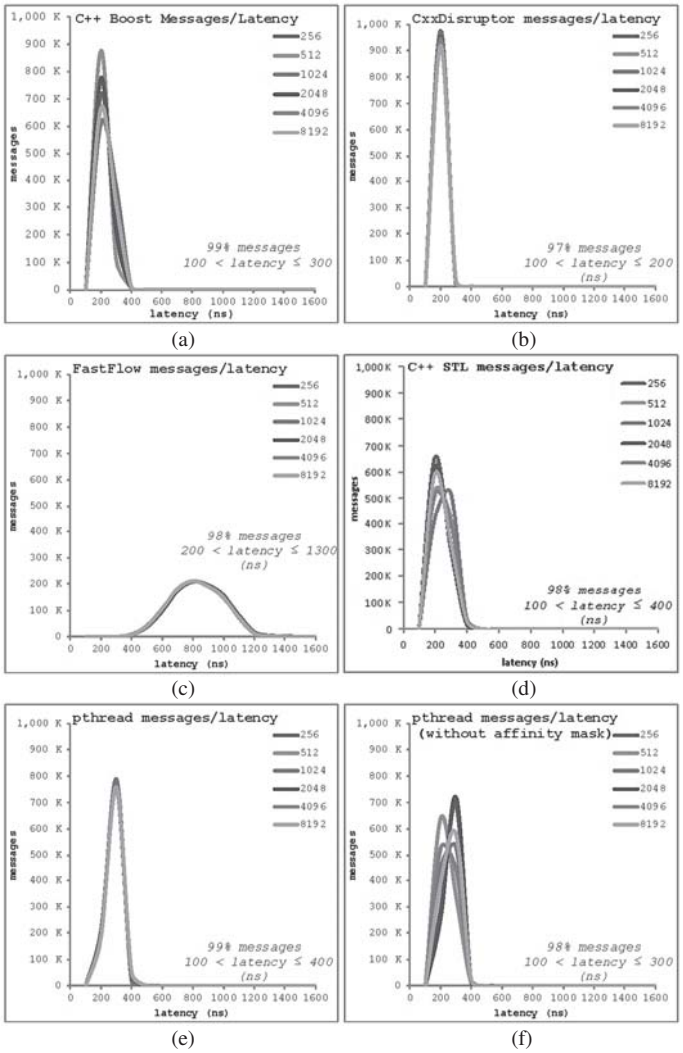

Fig. 6. C++ frameworks latency mesuarements.

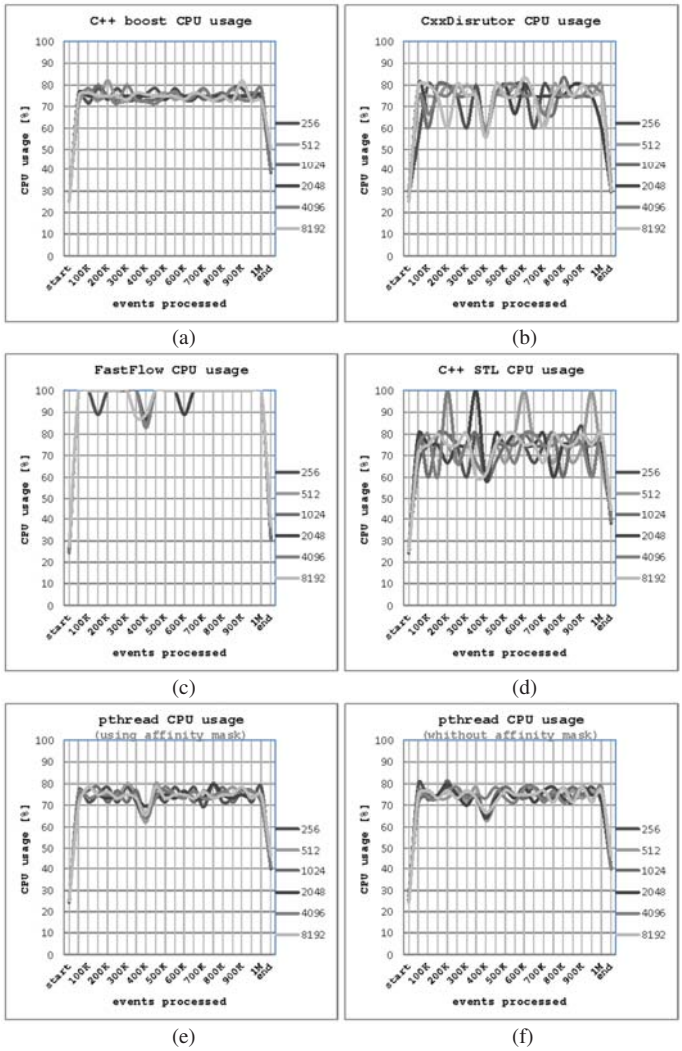

Fig. 7. C++ frameworks CPU usage percentages.

The Figure 7(b) presents the CPU usage of *CxxDisruptor* framework. As with latency measurements, the *CxxDisruptor* presents an efficient CPU usage, considering that yielding CPU resources to the operating system makes easier the rescheduling process and the threads can run more frequently.

VII. Conclusions

In this paper it has been shown that there are suitable C++ frameworks for development a High Performance Data Acquisition Systems with a 100% of accuracy. Although the *CxxDisruptor* framework has produced the best results in this empirical comparison, the results presented in this paper report that it is possible to develop a simple C++ framework using C++ standard libraries and POSIX threads API, obtaining satisfactory results related to performance and accuracy.

The addition of the SCHED_FIFO and the affinity mask did not provide the expected performance improving for the Simple C++ Framework. However, these options are well related with the operating system, which means that a tuning process should be developed to establish the better operating system configuration suitable for these options in order to achieve the expected results.

It is important underline that all the C++ frameworks evaluated can process the 90% of message in lower time than 669 microseconds which is the rate production of the camera in for the simulated frame rate, namely 1495 frames per second. Nevertheless, 10% of the messages have a variable latency and in some cases could be greater than 669 microseconds causing loss of messages which is not acceptable for the project.

There is a correlation between the latency time and the CPU usage. When the operating system has CPU resources available, the rescheduling process is faster, allowing to the threads run more frequently thereby reducing the latency time.

The next step of this research is to improve the communication between threads by means of shared-memory condition variables and kernel messaging queuing technologies. These approaches are the start point to separate each task in different process to adhere the solution to the project requirements.

References

[1] B. Stroustrup, The C++ Programming Language, Fourth Edition, Pearson Education Inc., 2013.

[2] A. Williams, C++ Concurrency in Action, Manning Publications Co, 2012.

[3] LMAX-Exchange Disruptor, Source: https://lmax-exchange.github.io/disruptor/

[4] Implementation of the LMAX Disruptor pattern in C++, Source: http://sourceforge.net/projects/cxxdisruptor/

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "FastFlow: high-level and efficient streaming on multi-core," in Programming Multi-core and Many-core Computing Systems, S. Pllana and F. Xhafa, Ed., Wiley, 2014.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multi- cores with fastflow. In Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, volume 6853 of LNCS, pages 170–181, Bordeaux, France, Aug. 2011. Springer.

[7] V. Martin, V. Moncada, J.-M. Travere. Challenges of Video Monitoring for Phenomenological Diagnostics in Present and Future Tokamaks. 2011. <hal-00609877>

[8] EoSens® 3CL Camera Manual, Rev. 1.01, Copyright © 2010 Mikrotron GmbH.

[9] H. Kopetz, Real-Time Systems, Design Principles for Distributed Embedded Applications. Second Edition, Chaper 1, Springer, 2011.

[10] M. Mitchell, J. Oldham, A. Samuel, Advanced Linux Programming, First Edition, New Riders Publishing, 2001.

[11] TIFF Revision 6.0, Adobe Developers Association, 1992.

Rolando Inglés is a Ph.D. student at Department of Microelectronics and Computer Science of the Lodz University of Technology, Poland. He received B.Eng. degree at Systems Engineering and Computing in 1996 from the University of Technology of El Salvador. In 2004, he received the M.Sc. degree at Systems Engineering from the Autonomous University of Baja California, United Mexican States. His research interests are in inter-process communication mechanisms and data structure related with high-performance data transfer in UNIX-like operating systems.

Piotr Perek received the MSc degree in the field of Electronics and Telecommunications at the Lodz University of Technology in 2010. He continues his education as a PhD student at the Department of Microelectronics and Computer Science Lodz University of Technology. His interests include embedded systems, programmable devices and software development. His recent research concerns the development of control and data acquisition systems based on ATCA, MicroTCA and AMC standards.

Mariusz Orlikowski was born in 1971. He received MSc and PhD degrees in electrical engineering from Lodz University of Technology in 1995 and 2000 respectively. He is currently an Associate Professor in the Department of Microelectronics and Computer Science Lodz University of Technology. His research interests include behavioral modelling using Hardware Description Languages, object oriented programming, distributed programming of data acquisition and processing systems.

Andrzej Napieralski received the M.Sc. and Ph.D. degrees from the Lodz University of Technology (TUL) in 1973 and 1977, respectively, and a D.Sc. degree in electronics from the Warsaw University of Technology (Poland) and in microelectronics from the Université de Paul Sabatié (France) in 1989. Since 1996 he has been the Director of the Department of Microelectronics and Computer Science. Between 2002 and 2008 he held a position of the Vice-President of TUL. He is an author or co-author of over 950 publications and editor of 21 conference proceedings and 12 scientific Journals. He supervised 48 PhD theses; six of them received the price of the Prime Minister of Poland. In 2008 he received the Degree of Honorary Doctor of Yaroslaw the Wise Novgorod State University (Russia).