Leveraging Reconfigurable Computing in Distributed Real-time Computation Systems

Apostolos Nydriotis, Pavlos Malakonakis, Nikos Pavlakis, Grigorios Chrysos, Ekaterini Ioannou, Euripides Sotiriades, Minos Garofalakis, Apostolos Dollas

ECE Department
Technical University of Crete
Chania, Greece
dollas@mhl.tuc.gr

ABSTRACT

The community of Big Data processing typically performs real-time computations on data streams with distributed systems such as the Apache Storm. Such systems offer substantial parallelism; however, the communication overhead among nodes for the distribution of the workload places an upper limit to the exploitable parallelism. The contribution of the present work is the integration of a reconfigurable platform with the Apache Storm, which is the main platform of the Big Data streaming processing community. By exploiting the internal bandwidth of FPGAs we show that the computational limits for stream processing are significantly increased vs. conventional distributed processing without compromising on the platform of choice or its seamless operation in a dynamic pipeline. The integration of a Maxeler MPC-C Series platform with the Apache Storm, as presented in detail, yields on the Hayashi-Yoshida correlation algorithm an impressive tenfold increase in real-time streaming input capacity, which corresponds to a hundred-fold computational load. Our methodology is sufficiently general to apply to any class of distributed systems or reconfigurable computers, and this work presents quantitative results of the expected I/O performance, depending on the means of network connection.

CCS Concepts

•Computer systems organization→ Distributed architectures →Cloud computing •Computer systems organization→ Heterogeneous (hybrid) systems, Reconfigurable computing, Data flow architectures

Keywords

Multi FPGA platform; streaming Big Data; Storm; distributed computational system; hybrid computational platform; high performance computing.

1. INTRODUCTION

Multi-FPGA platforms provide opportunities for system level design by tight coupling of powerful General Purpose Processors with FPGAs through fast interconnection networks or busses. FPGAs also have very fast access to external memory and direct fast connection to the internet. These systems, usually, have a "look and feel" of a conventional General Purpose Server with a Linux-based operating system, using special compilers. Thus, they improve system level performance by solving the Input/Output-related issues, i.e. the usual bottleneck in several FPGA-based computational systems.

In recent years, there exists an increasing need for real-time processing of the huge amounts of data. Processing massive amounts of data in real-time can only be achieved by distributing the workload across many computers and using distributed real-time computation systems such as the Apache Storm [7]. These systems have two inherent attributes: (i) when the parallelism is unlimited and interconnection networks are sufficient for data exchange any increase in the volume of data is processed by adding additional processing nodes, which of course imposes a financial cost (for acquisition and for operation); and (ii) the required network resources are not linear with respect to the number of nodes incorporated in the distributed real-time computation system, meaning that quite often the exploitable parallelism is limited by interconnection limitations, and even when a speedup can be achieved it is not linear with the resources. Hence, there exists a need to increase the computational capacity of a single node of a system, such as the Apache Storm. At the same time the solution cannot be a proprietary system, as this would limit its usefulness.

The purpose of this work, and thus its contribution, is to integrate a powerful FPGA-based system with the Apache Storm distributed platform, and by exploiting the internal processing and communication bandwidth of FPGAs to demonstrate that significant speedups can be achieved vs. the CPU-only approach.

More specifically, in this paper we show how we achieved the interconnection of the popular distributed real-time computation system Apache Storm with a Maxeler multi-FPGA platform. The interconnection forms the first (in our knowledge) dynamic pipeline for streaming applications, in which either the Apache Storm can be used by itself or the Maxeler reconfigurable resources can be used for the real-time processing of the input stream. This work presents the integration challenges as well as very promising initial performance results from actual experiments. More specifically, we explain how to perform the incorporation of the FPGAs within Storm without imposing modifications to the data processing inside Storm. We use a demanding streaming financial data processing scenario as a driving problem, which involves the computation of the pair-wise correlations between stocks. Based on this scenario, we experimentally demonstrate the benefits of computing the stock correlations with the hybrid distributed computational system.

The remaining paper is organized as follows. Section 2 describes the state of the art technologies that have been used in our implementation. Section 3 presents the incorporation of the FPGAs within the distributed real-time computation system. Then, Section 4 presents our proposed architecture, while Section 5 presents experimental results from a streaming finance application. Lastly, Section 6 provides conclusions and future work.

2. Technology Description

This section introduces briefly the technologies that were used and the characteristics which make these state of the art technologies useful. The platforms in Subsections 2.2 and 2.3 have been designed to process streaming data.

2.1 FPGA-Based Processing

Using FPGAs to speed-up computationally intensive processing has been done successfully for over two decades, since the early 1990s. However, major challenges towards the use of FPGAs as high-end processors are: (i) I/O problems, which prove to be a computational bottleneck, and (ii) the difficulty to program and use such systems, as their development tools and interfaces are considered exotic for the community of software applications developers.

The latest generation of FPGA devices offers significant resources in addition to the reconfigurable fabric. Special I/O transceivers, dedicated logic blocks for memory, powerful general purpose processors on chip, special modules for digital signal processing, and fast floating point operations have been added on-chip. Even the reconfigurable fabric changed, offering more logic, better routing resources and run- time reconfiguration characteristics. In addition, a large collection of functional Intellectual Property cores (IPs) is freely available to the designer through IP generator tools such as the Xilinx Core Generator, or, distributed by designers through web sites such as OpenCores. All of these available resources help designers to take up new applications, with considerable results on network systems such as network switches, network intrusion detection systems, and financial data analysis. Data streaming applications become much easier to implement due to these technological advances of FPGAs, mostly in the forms of I/O transceivers on a chip and large amount of available memory. Considering these features, the computational power of FPGA devices was exponentially increased but the problems of I/O bottleneck and ease of use where not faced for most mainstream (i.e. low cost) systems. It is very rare that a $100 FPGA-based board (regardless of vendor) will be used in large-scale production mode as a coprocessor, despite its inherent computational capabilities.

Unlike low-end systems, high-end multi FPGA platforms have been developed to offer opportunities at system level design by using powerful General Purpose Processors together with fast and large FPGAs. These systems have a "look and feel" of a conventional General Purpose Server with a Linux -based operating system, using special compilers. Application Developers for such systems usually keep the software at its original form and swap the computational intensive procedures, with hardware procedures calls which are functionally equivalent, and which are determined by profiling the applications. Such servers can offer significant computational power, which equals that of hundreds of conventional processors, while keeping the same look-and-feel for the end user (but not for the developer).

In terms of user needs, the Big Data community uses platforms that can manage dozens or even hundreds of nodes for high performance computations[1][2]. In such systems, using a reconfigurable platform as a node for specific applications can offer a significant advantage. The reconfigurable node can perform the compute intensive parts of the algorithm and the conventional nodes can perform all the other procedures which are difficult to be translated in hardware and have not significant computational load. Such a system can be considered to be a new era for reconfigurable computing which can incorporate heterogeneous computing systems providing them with powerful coprocessors.

2.2 The Maxeler System

Maxeler technologies is one of vendors that offer state-of-the-art FPGA-based platforms, such as the ones described in the previous section [3] [4] [6].

Maxeler offers Maximum Performance Computing (MPC) systems based on FPGAs. They drive MPC by using the 'Multiscale Dataflow Computing' paradigm. Dataflow computers focus on optimizing the movement of data in an application and utilize massive parallelism between thousands of tiny 'dataflow cores' in order to provide orders of magnitude benefits in performance, space and power consumption. Due to this model the Maxeler systems refer to their reconfigurable resources (FPGAs) as Dataflow Engines (DFEs). The system presented in this work is the MPC-C series, shown in Figure 1, below. It consists of 12 Intel Xeon CPU cores with 64GB of RAM and 4 DFEs (Xilinx XC6VSX475T FPGAs) with 24GB of RAM for each one. Each DFE is connected to the CPUs via PCI Express with up to 2GB/s, and DFEs are directly connected with MaxRing interconnect. Maxeler technologies also offer nodes like MPC-X and MPC-N series which support direct network connection of the DFEs.

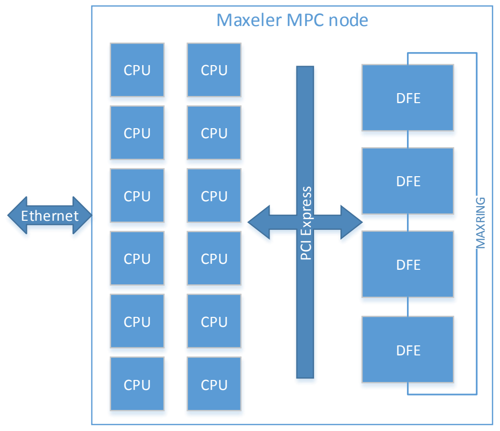


Figure 1. MPC-C series node architecture

The Maxeler DFE configuration is called through C/C++ host code running on the CPUs. The MaxCompiler translates Maxj into FPGA configuration files. Maxj, is an extended form of Java with operator overloading, and it is used as the Hardware Description Language for Maxeler systems. The Maxj language and environment offers a relatively user-friendly programming environment (vs. traditional design in, say, VHDL) but it also poses limitations as to what can be implemented, and it is not uncommon for applications which require complex data structures and synchronization to be designed with VHDL in order for the designer to have complete control of the design.

The computing kernels which handle the data-intensive part of the application and the associated manager, which orchestrates data movement between the DFE and the CPU or RAM, are written using the Maxj language. At its simplest level, DFE computation can be added to an application with a single function call, while for more fine-grained control the SLiC (Simple Live CPU) tool provides an "action"-based interface. The SLiC interface, is Maxeler's application programming interface for CPU-DFE integration.

In such an MPC system, the CPUs are in control and drive the computations on the DFEs. The data-intensive parts of the computations are typically confined to the DFE configuration and are made available through the SLiC interface.

2.3 The Apache Storm platform

The Apache Storm is a free and open source distributed real-time computation system used for processing unbounded streams of data. In order to process data with Storm [7], graphs of computation – called topologies – have to be created. The nodes of a topology encapsulate the processing logic, while connections between the nodes indicate how the data flows in the graph. Storm provides the primitives for transforming such data streams into new streams in a distributed and reliable way.

The basic primitives which Storm provides for doing stream transformations are the "spouts" and the "bolts". These are the components of a topology and the programmer uses them to implement application-specific logic. Spouts can be thought of as the entry points of data into the system. For example, a spout may connect to the Twitter API, receive a stream of tweets and emit it to the rest of the topology. A bolt, on the other hand, consumes any number of input streams, does some processing and possibly emits new streams. Networks of spouts and bolts are packaged into "topologies", which are submitted to Storm clusters for execution. A topology then, is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it actually sends the tuple to every bolt that subscribed to that stream.

Each node in a Storm topology, as the one depicted in Figure 2, executes in parallel in one or more cluster machines. The amount of parallelism of each component is subject of the topology configuration. The configured number of threads and processes will be spawned across the cluster to do the execution. Storm offers "at least once" processing guarantees as well as "no data loss" guarantees even if machines go down and messages are dropped. Any failed tasks will automatically be reassigned.

3. Adding a Maxeler node to storm

Several technical solutions have been tested in order to integrate the Storm and Maxeler platforms. These solutions are presented and discussed in this Section, with the simplest and more generic one (which was chosen as the best approach) last.

A communication framework had to be created so that the reconfigurable hardware could receive data, make any calculations needed, and then transmit the results back to the software application. This task involves the implementation of a flexible interface between the Storm and Maxeler systems, which could also be used with different tools or reconfigurable hardware platforms.

The Storm framework was installed on the Maxeler workstation and was tested as a simple Storm node. This was no difficult task, as the Maxeler systems support Linux, for which Storm does have a distribution. Following that step, the connection with the

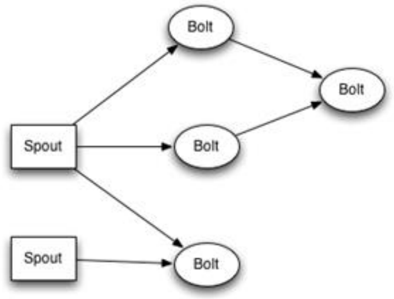


Figure 2. Storm Topology

Maxeler hardware had to be established, which proved to be impossible, as will be described below. Hence, we need to differentiate between the Maxeler system being a Storm node vs. the Maxeler system being connected to a Storm node. Obviously the former would be preferable, but as will be explained below it was not possible.

The Maxeler hardware is called by a C/C++ host code. The main problem that had to be addressed is the interface between Java (Storm) and C. Three methods that allow C/C++ and Java connection were considered: (i) SWIG (Simplified Wrapper and Interface Generator) that allows the call of C function through Java, (ii) the exec function which calls the C executable and (iii) network sockets. In the solution (iii), which was developed successfully, we have the Maxeler system connected to Storm. Although this may appear to be an issue of semantics, in reality it affects how tight the entire integration becomes as well as system performance, as will be explained below.

3.1 SWIG

In order to use SWIG to connect the Storm Java code with C, the Maxeler project was compiled as a shared library. SWIG [8] is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of target languages including common scripting languages such as Java, Perl, PHP, Python, Tcl and Ruby. SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code. After wrapping the C code with SWIG the C function can be called as a native code call from the Java. The problem was that in order to use SWIG, its data types had to be used in order to be able to exchange arguments from C to Java. This would reduce flexibility as the Java code would have to be changed to use SWIG data types. Furthermore, the default scheduler of Storm selects the cluster machines on which the various tasks will run based on load balancing features. This means that in order to pin the task that would connect with C through SWIG to a specific cluster machine – the Maxeler workstation in this case– we would have to implement a custom Storm scheduler. Such a low-level tweaking of Storm would definitely raise the complexity level of our solution to very high and undesirable levels, and deviate from the standard distributions, which was deemed undesirable.

3.2 Exec function call

Java's exec command is able to directly call the Maxeler executable or any executable file through a system call. This, of course, implies paying a performance penalty since system calls are very expensive operations. Furthermore, as with the SWIG alternative, a custom Storm scheduler would have to be implemented in order to successfully call the Maxeler executable. Although the exec function call was successfully tested locally on the workstation, when the call came through Storm, the executable would not run. The problem turned out to be related to the version and tuning of Linux which is required for Storm vs. that in which the Maxeler run-time environment, and as a result several libraries, which are needed for the Maxeler system to access the run-time environment could not become available. The combination of the implementation difficulties, described above, with the anticipated performance issues due to system calls deemed this approach as well to be undesirable for further pursuit.

3.3 Network sockets

The last method which was proposed in order to establish communication between the Java code and the C host code running on the Maxeler workstation was the use of network sockets [9]. The Maxeler node, acting as a TCP server, creates sockets on start up. The sockets, being in listening state, are waiting for connections from client applications (e.g. Storm nodes). A TCP server may serve several clients concurrently, by creating a child process for each client and establishing a TCP connection – via a unique dedicated socket – between the child process and the client. The Maxeler workstation can support up to 4 children processes with hardware calls as 4 different hardware designs can be executed simultaneously on the 4 available FPGAs. The socket server is written in C and serves as the Maxeler host code.

On the basic form of the implementation one socket client makes a call on the server. The clients are implemented in Java as Storm components (Spouts/Bolts) that implement the appropriate algorithmic family interface. When the connection is established, the clients stream the data to the server. The Maxeler server makes the hardware call and when the processing is completed it streams the results back to the clients. For example, the application presented on the following section uses two socket clients in order to make a call on the hardware server, a transmitter – that is implemented as a bolt – and a receiver – implemented as a spout. The transmitter creates a new socket in order to connect to the server and sends the configuration as well as the input data from previous Storm components. It can also request the results to be sent to the receiver. The receiver, which is connected via a different socket, receives the results and forwards them to the rest of the topology. The server stays on listening state until a connection is established by both a transmitter client and a receiver client. First, the transmitter streams configuration. Next, it transmits input data to the server. The server stores the data coming from the transmitter and performs the Maxeler hardware call to process them. After the hardware call has returned, it sends the results to the receiver client whenever a result request is received. The receiver and transmitter libraries can also be used outside Storm, increasing the flexibility even more.

The three methods were compared in terms of flexibility and functionality. The use of SWIG basically reduces the flexibility of the interface, as the Java code would have to be rewritten using SWIG's data types. The exec system call allows more flexibility as only a function call would have to be included in the Java code, but even though it worked perfectly when called by Java on the workstation, it didn't work when the function ran through Storm. Furthermore, following this method would hurt a lot the performance of the system because of the cost of the system call.

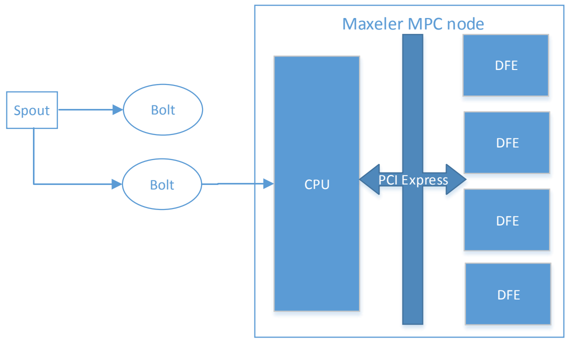


Figure 3. Storm Maxeler node Integration

The network sockets approach was chosen, as shown in Figure 3 (in which there is no bolt inside the Maxeler system) due to their flexibility and their performance. We note here that as a side-effect from the way the sockets-based system was developed, we achieve flow control and synchronization of data between the software components and the hardware ones. Any system that can implement sockets can use this approach without deteriorating performance since the expensive interface functions of SWIG as well as the system calls are avoided.

4. Platform Architecture

This section describes the architecture of the proposed platform. The proposed architecture is generic as it can combine any FPGA based platform with a stream processing framework using simple and well know communication methods, i.e. network sockets.

The FPGA-based platform consists of a combination of a reconfigurable and a general purpose processor. The general purpose processor is used for communication with the streaming framework, for parsing the incoming data and passing them to the reconfigurable platform. Last, it is used for packetizing the results, which are received back from the reconfigurable platform, and send them to the stream-based framework. On the other hand, the reconfigurable-based part is used as the main processing unit. It is used exclusively for mapping the compute intensive parts of the algorithm and process the incoming data. These parts of the hardware platform are connected with each other using PCIe links.

The second component of the proposed architecture is the stream processing platform. This platform is used for passing data to the reconfigurable platform and receiving the results back. This framework needs to use streaming formulation.

As described above, the communication scheme was the most challenging part for integrating the hardware-based platform with the software-based framework into a hybrid platform. For the presented solution, we propose the use of network sockets, which is a generic and high-throughput solution for connecting distributed systems.

5. Experiments

This section presents some experimental results on a test platform that uses FPGA devices as part of a distributed real-time computation system, i.e. Apache Storm. First, we describe the platform that was used for our experiments. The platform is generic and can be easily extended. Second, the algorithm which was used to demonstrate the proposed infrastructure is described. Finally, some initial performance results from the integration of

FPGA devices into a distributed real-time computation system are presented.

5.1 Platform

This section describes the infrastructure which was used for our experiments. The proposed solution combines the reconfigurable technology with the Apache Storm framework. The platform consists of a cluster with 7 nodes, each one with Dual-core AMD CPUs @ 2.1 GHz, 8 GB RAM, Gigabit Ethernet connection and a Maxeler MPC-C server. Storm topologies consisted of spout and bolt primitives that were mapped on the cluster, as described in previous sections. The Maxeler server was connected via TCP sockets with a bolt and a spout of the Storm topology for I/O data movement [5].

The implemented infrastructure defines a data flow from data sources through data processing components to data sinks. The same workload has been assigned to both software and hardware parts of the platform in order to compare the performance of the reconfigurable part of the cluster vs. the performance achieved by the software-only cluster solution.

5.2 Test case: Correlation on financial streaming data

The analysis and the elaboration of high data loads in real-time is crucial for the financial stock markets. The financial data arrive in a streaming fashion from various numbers of streams with high rates. The correlation metric is an industry-standard technique. One of the most well-known correlation metrics for high frequency streaming data is the Hayashi-Yoshida (HY) correlation estimator [10][11].

The HY Correlation Estimator measures the pairwise correlation of the input market stocks. It uses the transaction prices of two stocks in order to calculate their correlation. The correlation is calculated over time windows, inside of which the stock transactions take place. Figure 4 presents the equation to calculate the HY estimator for two different market stocks.

The algorithm outputs a correlation matrix that describes the correlation between all the different pairs of the incoming financial stock markets.

$$HYcor = \frac{\sum_{i,j}\left(P^1_{t_i} - P^1_{t_{i-1}}\right) * \left(P^2_{t_j} - P^2_{t_{j-1}}\right) * 1_{\{[t_i,t_{i-1}] \cap [t_j,t_{j-1}] \neq \emptyset\}}}{\sqrt{\sum_i\left(P^1_{t_i} - P^1_{t_{i-1}}\right)^2 * \sum_j\left(P^2_{t_j} - P^2_{t_{j-1}}\right)^2}}$$

$where\ P^1_{t_i} = value\ of\ stock\ 1\ at\ time\ t_i\ and\ P^2_{t_j} = value\ of\ stock\ 2\ at\ time$

Figure 4. Hayashi-Yoshida Correlation estimator

5.3 System Architecture

This section describes the system architecture of the test case algorithm on our proposed infrastructure. The implemented system takes as input a stream of the transaction prices of N stocks. The transaction data are recorded at random times, i.e. they have different timestamps, which means that we have to calculate the correlation estimator of all the pairs of the stocks during different time intervals. First, bolt units preprocess the received data in order to bring them in a proper format to be processed. The formatted data are streamed into the reconfigurable platform of the infrastructure using TCP sockets, as referred above. The data are passed to the reconfigurable part at a fixed time interval, i.e. in our tests we used time interval = 1 sec. The reconfigurable part computes the correlation metric among all the pairs of the incoming stock markets and then the results are sent via TCP socket to another storm spout unit– basically this reimports the results to the storm system. Finally the results are presented to the final user. The hardware architecture that is mapped on reconfigurable technology is presented in Figure 5.

5.4 Performance

We evaluated our system with high-volume, real-life and synthetic data streams. The used datasets consist of stock market transactions. The rate of the transactions was one transaction for each stock market per sec. We used various sized input datasets, which were streamed to the platform. The evaluated input datasets were from small loads, i.e. 250 transactions per sec, up to heavy loads, i.e. 5000 transactions per sec. As the input datasets had one transaction for each stock market, thus, the number of transactions is equal to the number of processed stock markets per sec. The data and the results were transferred over the network using the TCP sockets, as it was described above.

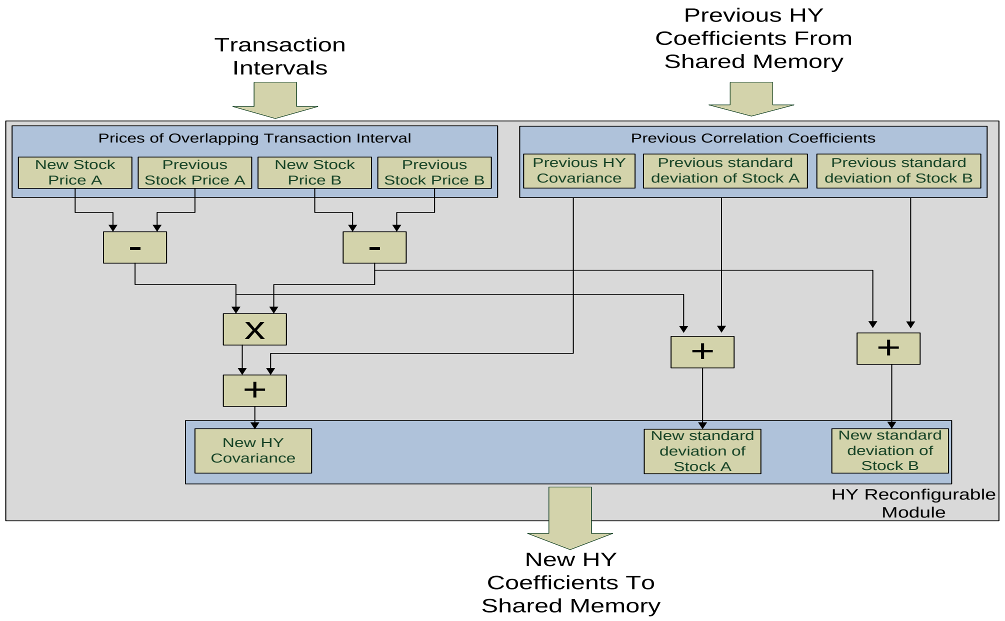
First, we tested our system with real-life dataset with 1000 stock markets (i.e. 1000 streams, as each stock market is one stream). The performance results indicated that the time needed for the calculation of the HY correlation coefficients for real-life 1000 stock markets at every timestamp was about 0.2 sec. Thus, taking into account that new transactions arrive every second, this means that our proposed solution can process in real-time the real-life input data. We also tested the proposed solution with synthetic datasets. The demonstrations showed that the correlation of about 5000 stock markets can be computed in real time, i.e. every second. On the other hand, a distributed implementation of the Hayashi-Yoshida algorithm over a 7-node cluster mapping the Storm framework achieved the calculation of maximum 500 stock markets in real time. Last, we ran the above datasets over a single thread fully optimized implementation of Hayashi-Yoshida correlation. The performance achieved, was not competitive, as the single thread solution could process up 250 stock market per sec. It is clear that a hybrid infrastructure of a distributed framework coupled with reconfigurable part can offer quite impressive performance even vs. distributed solutions for high workload streaming problems.

In order to measure the bandwidth of the integrated system several experiments were conducted. Experiments were done, using the Local and the Remote Network (Internet). Local network is the network inside the university campus which has speed of 1 Gbps. The remote tests were run using Virtual Private Network (VPN) over an ADSL (24 Mbps download and 1Mbps upload) with the TCP/IP protocol. Table 1 shows the average results received for the server of these experiments.

These results shows that over a fast network connection the system bottleneck is on processing, but it offers a sufficient bandwidth for several applications, such as the demanding financial one. On the other hand, when we have connection to the Maxeler server via VPN, the bottleneck is in the data transfer and can be a functional option for remote systems in which the data rate is not crucial or there is a higher ratio of computation vs. I/O.

Table 1. Bandwidth measurements

Network configur.	Download(Mb/s)	Upload(Mb/s)
Local Network	93.6	51.12
VPN	0.81	7.52
MPC – X Series (est.)	1000	1000



Transaction Intervals
Previous HY Coefficients From Shared Memory
Prices of Overlapping Transaction Interval
New Stock Price A
Previous Stock Price A
New Stock Price B
Previous Stock Price B
Previous Correlation Coefficients
Previous HY Covariance
Previous standard deviation of Stock A
Previous standard deviation of Stock B
-
-
X
+
+
+
New HY Covariance
New standard deviation of Stock A
New standard deviation of Stock B
HY Reconfigurable Module
New HY Coefficients To Shared Memory

Figure 5. Reconfigurable architecture that implements the HY algorithm on streaming data

The connection of another type of Maxeler server, the MPC – X Series, with on-FPGA chip network connection, shows the speed of light for this implementation. Such a connection uses fast Ethernet to send and receive data and the Infiniband protocol to transfer them to reconfigurable hardware. It hasn't been tested and the presented performance is projected. Such a performance is more than sufficient considering that for financial applications such as the complete NYSE Real-Time Reference Prices feed. That feed provides real-time last sale prices in NYSE-Traded Securities, and it needs a bandwidth of 13 Mbps for 1ms refresh rate[12].

6. Conclusions

This paper shows the potential of reconfigurable computing, used as part of standardized distributed computer systems. The proposed architecture is generic as it can be used for any streaming processing algorithmic scheme without any changes but only mapping the compute intensive parts of the algorithms on reconfigurable logic. Also, our solution is platform-independent as it combines two completely different in terms of technology computing platforms to create a powerful hybrid computer system. Both platforms were initially designed without any foresight to be connected together. The system that came up is robust, easy to use and able to achieve high performance.

Our approach has several benefits. The multi-FPGA platform is connected to a typical Storm node and thus it required no special treatment or flow modifications within Storm. It is able to offer the computing power equivalent to several conventional Storm nodes with only one Maxeler multi-FPGA platform. In addition, the suggested interconnection can work with different FPGA platforms and distributed computation systems. Finally, our approach enriches Storm with the ability to realize scalability not only with the traditional methodology of incorporating additional processing nodes but also by pushing the part of the processing on the Maxeler multi-FPGA platform.

This method can be easily applied to different computational systems as Hadoop, or with different reconfigurable platforms as Convey. Such an approach can lead to hybrid systems with different configurations depending on the nature of computations, if for example are on streaming data or not. The selected combination is for streaming data as both Storm and Maxeler systems have been designed for such calculations. In terms of ease of use this system can be fully automated by creating a function call for hardware platform

7. ACKNOWLEDGMENTS

This work has been partially funded from the European Union's Seventh Framework Programme under Grant Agreement 619525 (QualiMaster).

8. REFERENCES

[1] Pell, O., Averbukh, V. (2012). "Maximum Performance Computing with Dataflow Engines". Computing in Science & Engineering , vol.14, no.4, (pp. 98-103).

[2] Pell, O., Mencer, O., Tsoi, K. H., & Luk, W. (2013). "Maximum performance computing with dataflow engines". In High-Performance Computing Using FPGAs (pp. 747-774). Springer New York.

[3] Multiscale Dataflow Programming , Maxeler Technologies Ltd, London, UK, 2014

[4] Programming MPC Systems, Maxeler Technologies Ltd, London, UK, 2014Page 64(of 65) www.qualimaster.eu

[5] Deliverable 3.1 QualiMaster

[6] https://www.maxeler.com/

[7] https://storm.apache.org/

[8] http://www.swig.org/index.php

[9] Yadav, R. (2007). Client/Server programming with TCP/IP sockets. Technical Article, DevMentor.

[10] Hayashi, T., & Yoshida, N. (2005). "On covariance estimation of non-synchronously observed diffusion processes". Bernoulli, 11(2), (pp. 359-379).

[11] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[12] http://www.nyxdata.com/capacity