# Application-Specific Customisation of Market Data Feed Arbitration

Stewart Denholm*, Hiroaki Inoue†, Takashi Takenaka† and Wayne Luk*
*Imperial College London, UK
{swd10, wl}@doc.ic.ac.uk
†NEC Corporation, Kawasaki, Japan
{h-inoue@ce, takenaka@aj}.jp.nec.com

***Abstract*—Messages are transmitted from financial exchanges to update their members about changes in the market. As UDP packets are used for message transmission, members subscribe to two identical message feeds from the exchange to lower the risk of message loss or delay. As financial trades can be time sensitive, low latency arbitration between these market data feeds is of particular importance. Members must either provide generic arbitration for all of their financial applications, increasing latency, or arbitrate within each application which wastes resources and scales poorly. We present a reconfigurable accelerated approach for market feed arbitration operating at the network level. Multiple arbitrators can operate within a single FPGA to output customised feeds to downstream financial applications. Application-specific customisations are supported by each core, allowing different market feed messaging protocols, windowing operations and message buffering parameters. We model multiple-core arbitration and explore the scalability and performance improvements within and between cores. We demonstrate our design within a Xilinx Virtex-6 FPGA using the NASDAQ TotalView-ITCH 4.1 messaging standard. Our implementation operates at 16Gbps throughput, and with resource sharing, supports 12 independent cores, 33% more than simple core replication. A 56ns (7 clock cycles) windowing latency is achieved, 2.6 times lower than a hardware-accelerated CPU approach.**

## I. Introduction

Financial institutions require up-to-date information for risk management, the operation of algorithmic trading, and for identification of arbitrage opportunities. Such information is provided by exchanges via a market data feed. Messages describing the current state of the market, such as market events related to available and completed trades, are multicast to financial institutions which have subscribed to market data feed services.

The time-sensitive decisions that applications make based on the current state of the market are highly dependent on data feed messages being received and presented in order. Incomplete information increases financial risk and results in reduced opportunities for trade.

As message feeds are typically transmitted over Ethernet using UDP, messages may be lost or arrive out of order. To combat this, institutions subscribe to two identical message feeds from the exchange. Work is then required to merge these feeds, either using a single one-size-fits-all windowing scheme, or an application-specific approach within every application.

We accelerate market data feed arbitration on an FPGA at the network level, reducing software processing, and allow multiple, independent arbitrators per FPGA. By allowing arbitration cores to be independently customised we support multiple downstream processing platforms and applications. Sharing resources on the FPGA also permits the use of more cores compared to a simple core replication scheme.

The new contributions of our work include:

- A reconfigurable market feed arbitrator where multiple, independent arbitrators can operate within a single FPGA, sharing resources to improve scalability. Arbitration uses a timeout & message counting windowing mode, for which the thresholds can be set at runtime.
- Performance models examining the scaling of arbitration to support multiple platforms, and how application-specific customisations in arbitrators affect downstream applications and other arbitration cores.
- Implementation and evaluation of multiple market feed arbitrators within a Xilinx Virtex-6 FPGA using the NASDAQ TotalView-ITCH 4.1 message feed standard. We achieve 16Gbps throughput and a $56ns$ (7 clock cycles) windowing latency. Scalability, customisation and resource optimisations are also examined.

The paper is organised as follows. Section II reviews past market data feed processors. Section III gives the details of feed arbitration, its customisations and how we achieve it within our design. Section IV presents an analytical model for scaling and customisation. Section V describes our arbitrator's implementation within a Virtex-6 FPGA. Section VI presents the results of the experiments carried out on our implementation. Finally, Section VII draws conclusions.

## II. Related Work

Feed arbitration is performed in [1] with a Celoxica AMC board as part of an OPRA FAST trading platform. It has a 3.5M messages per second throughput and hardware latency of $4\mu s$. The arbitration latency is not given and there are no opportunities to customise arbitration or discussions on how it was configured for their design.

Some works, like the high frequency trading IP library from Lockwood et. al. [2], mention arbitration, but do not implement it. Most works, however, do not address it at all, such as the OPRA FAST feed decoder from Leber et. al. [3]. Our customised, multi-arbitration design is then ideally suited to feed into these systems.

For feed processing platforms within FPGAs, the difficulty lies in the functions they provide. It can be extremely difficult to customise or expand upon the implemented features due to the limited space within an FPGA. Such platforms are then unlikely to be deployed within financial institutions unless their design specifically matches the institution's needs, including the market data feed protocol used.

978-1-4799-2198-0/13/$31.00 ©2013 IEEE

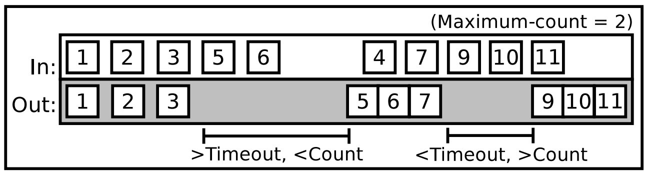

Fig. 1. An example of time & message count based arbitration.

Stand-alone arbitrators tend to be implemented within network interface cards (NICs), such as Solarflare's arbitrator [4] based on the Altera Stratix V FPGA. It supports either a low latency or maximum reliability mode, the latter being similar to our windowing approach, but it supports only one arbitration core and lacks options for application-specific customisation. Multiple message protocols are supported, but no processing latency figures are available.

Even if stand-alone designs such as [4] offered customisation options, they are not scalable for applications spanning multiple nodes. Each node would require the enhanced NIC, with each card performing the same feed arbitration work.

Our approach is to create a single customisable market feed arbitrator at the network level able to support many different applications and market data feed protocols. The arbitrator should also be small enough to facilitate multiple cores operating within one FPGA, allowing resources to be shared, improving scalability and enabling a number of downstream applications to use the same arbitrator platform.

## III. Market Data Feed Arbitration

Exchanges can transmit multiple messages within one UDP packet. Dealing with missing or out-of-order packets therefore requires we treat network packets as the smallest unit of data we process and store during arbitration. The network level, rather than within the host, is then the logical platform for arbitration. The challenge then becomes how to:

- customise arbitration for financial applications;
- provide a scalable platform to host multiple, independent arbitration cores within an FPGA.

Merging two market data feeds is facilitated via unique identifiers within each message, typically taking the form of an incrementing sequence number. Market data feed protocols either utilise sequence numbers at the message level, like OPRA [5], or the packet level, like NASDAQ TotalView ITCH [6]. As the sequence number location within our arbitrator is customisable, any protocol or financial exchange may be used. For protocols with message level sequence numbers, the packet sequence number is then that of the first message in the packet.

When we encounter a packet with a sequence number larger than the next expected sequence number, it has arrived out of order. The missing packet, or packets, may simply be late, or not arrive at all. The time we delay a packet is based on the amount of time we have stalled the output and the number of messages delayed. Within this window we accept and store any valid packets, i.e., packets we have not seen before, while waiting for the missing packets. If the missing packet or packets arrive within our window boundaries we output the new and stored packets in order. It is important to ensure we do not delay a valid, expected packet when it arrives as this is the most likely case.

By using a windowing system based on the delayed time and data feed messages we can determine an upper limit on packet delays, while also providing a time-independent means of following the incoming packet rate. Figure 1 shows an example of a windowing system with a single input, in which both the time and count thresholds are used to determine whether a stored packet should be output. Packet $P_4$ takes too long to arrive, therefore exceeding $P_5$'s timeout and resulting in $P_4$ being discarded. Later, $P_8$ is also late, but whilst we are waiting for $P_9$'s timeout, the number of messages we are buffering has exceeded maximum-count = 2 messages, and $P_8$ is discarded.

### A. Customisation

The windowing time and message thresholds can be set to benefit specific downstream applications. For example, an application that analyses price movements in the 100ms range is less willing to wait for missing packets as an application operating on the second or minute scale. With arbitration customisation we do not need to sacrifice application performance to meet generic arbitration schemes.

As we operate at the network level we must take an active role in routing non-market data packets through the network, reserving FPGA resources to process, store and route these non-market feed packets. This typically requires little space, but the level of resources provided to non-market packets can be set, letting applications communicate with each other via the network. This is processed at the lowest possible priority so as to minimise interference with market packets.

The size and number of packets stored can be configured for each arbitrator. Arbitrators can then trade-off resources to support a mix of long and short time scale applications.

The method used to compare packet sequence numbers can be altered to reduce the arbitration resolution. For example, applications that deal with long term price fluctuations may not want to be flooded with updates every few milliseconds, and so would choose to only receive new messages after a set amount of time has elapsed.

### B. Application Scaling

By default we output using Ethernet-based UDP, so a single arbitrator can multicast to multiple applications. As Ethernet is widely adopted, it is well served to communicate with a range of existing platforms and any applications they implement.

Since we utilise an Atlantic interface [7] to send packets to and from our arbitration core, additional stages can be added within the FPGA by implementing and connecting to this standard interface. This also allows input and output connections other than Ethernet to be used. Using any commercial or customised network we can then feed into a range of data processing applications, such as the data feed processing in [3] and [2], or more complex, event driven functions like [8].

Performing arbitration at the network level means downstream applications can more easily span multiple computing nodes, as each node does not need to implement the application's specific arbitration scheme. Combining this with our ability to multicast from each arbitration core, and the customisations allowing application-specific non-market packets to pass through arbitrators, our arbitration scheme is ideally suited to accommodate the arbitrary scaling of applications.

## IV. Performance Model

### A. Multi-Core Deduplication

By simply replicating arbitration cores within an FPGA we enable additional arbitration functionality with a simple linear increase in resource usage. Although the cores are independent, this approach duplicates some core functionality, such as the handling of network packets. Our design designates a single core to perform these functions, freeing resources for additional cores or to expand upon the existing ones.

This effect is more pronounced for smaller numbers of cores as the duplicate work is then a larger percentage of the utilised resources. To ensure our work is applicable to larger FPGAs we must find a platform-independent method by which to judge arbitration scaling. We measure the resource requirement scaling using:

$$R_{rep} = C * (R + D) \tag{1}$$

$$R_{dedup} = C * R + D \tag{2}$$

where $R_{rep}$ is the resource requirement for simple linear replication of cores, $R_{dedup}$ is the resource requirement for our deduplication scaling, $C$ is the number of arbitration cores in the design, $R$ is the number of unique resources needed by each core, and $D$ is the resources that are duplicated.

To determine the validity of our design in Section VI, the relative performance improvement of our deduplication scaling as we approach an infinite degree of scaling is found by:

$$\begin{aligned} P_{dedup} &= \lim_{C\to\infty} \frac{R_{rep}}{R_{dedup}} \\ &= \lim_{C\to\infty} \frac{C * (R + D)}{C * R + D} \\ &= \frac{R + D}{R} \end{aligned} \tag{3}$$

### B. Customisation Scaling

Translating our expected gains from application-specific customisation into increased performance within a multi-arbitrator system can be difficult due to platform overhead and overhead within each core. By examining two arbitrator configurations, A and B, we can isolate this overhead and find the resources required by any number of cores or customisations.

We can utilise this configuration cost to make judgements when trading-off the number of cores and resources within the FPGA. For example, to quantify the cost of a specific core implementation, or find the resources freed by removing an existing arbitration core. For this we use:

$$R_{cus} = \frac{R_A}{C} - \frac{(N * R_A) - R_B}{C * (N - 1)} \tag{4}$$

where $R_{cus}$ is the resource requirement to implement a customisable feature, such as additional memory or a wider data-path; $R_A$ and $R_B$ are the resources used by configurations A and B respectively; $C$ is the number of arbitration cores in the design; and $N$ is the expected performance improvement, for example $N = 2$ when halving the number of block RAMs.

The total number of resources used within a multi-arbitrator system is then:

$$R_i = O_p + C * (O_c + N * R_{cus}) \tag{5}$$

with $R_i$ is the total resources need for a given resource $i$; and $O_p$ and $O_c$ being the platform and core logic overhead respectively.

## V. Implementation

We implement our design within a Xilinx Virtex-6 LX365T FPGA on an Alpha Data ADM-XRC-6T1 card. As our processing rate is greater than our target 10Gbit Ethernet connection, we transfer data into and out of our design via PCI Express. As mentioned in Section III, since we utilise an Atlantic interface within our arbitrators the physical connection has no impact on performance. We configure the packet processing operations for the NASDAQ TotalView-ITCH 4.1 message format, which transmits packets using the moldUDP64 protocol [9] operating on top of UDP. TotalView-ITCH messages from 9 September 2012 are used to test the system, and packet sequence numbers are supplied by the moldUDP64 header.

The location and length of the sequence number and message number fields are all that is needed to process TotalView-ITCH packets. Changing support to another protocol requires only that we indicate where these fields—or their equivalent—are located within the new packet format.

### A. Design and Testing

Our design operates with a 128-bit data-path and at a clock frequency of 125MHz, achieving 16Gbps throughput. TotalView-ITCH sequence numbers are 64-bits wide and we utilise the full 64-bit field when comparing packets. The packet buffer can store a maximum of 8 packets and TotalView-ITCH messages range in size from 5 to 44 bytes. When taking the 9000 byte payload and all the packet headers into account, the buffer is therefore able to store upto 10224 messages.

We measure packet latency by noting the packet sequence numbers at an arbitrator core's input and output. The number of cycles between a packet entering and exiting the arbitrator is then written into the packet, allowing us to later find the latency by inspecting the packet.

## VI. Results

Simply replicating arbitrator cores within an FPGA results in an unnecessary duplication of work, wasting resources. With our approach we allocate all the duplicate work to the first arbitrator core. Non-message packet handling is the main focus of this as block RAMs—our most limited resource—will see the greatest reductions.

Figure 2 shows how block RAM usage scales with the number of arbitrator cores. On our LX365T FPGA we can operate 12 cores using deduplication, compared to 9 with simple replication. For the larger Virtex-6 SX475 FPGA this would rise to 26 cores, or 46 for the Virtex-7 X1140T.

Using Equation 3 we can find our theoretical maximum scaling gain. Block RAM allocation is simplest to quantify, for which $R = 8$, the number of packets each arbitrator can store, and $D = 2$, the number of stored non-market packets we do not need to duplicate. Our theoretical maximum scaling gain is then: $P_{dedup} = \frac{8+2}{8} = 1.25$, or 25%.

With 12 cores we experience a 33% scaling gain in our implementation. We are technically able to fit about 9.6 replicated cores into an FPGA, but as we cannot operate an incomplete core, this must be rounded down.

Figure 3 shows our improvements over simple replication, for which we see block RAMs tending towards their theoretical scaling limit as the number of arbitrator cores increases. The figure shows lower gains for slice registers and Look-Up Tables (LUTs), at 10% and 5% respectively, but block RAM storage on the FPGA is our primary concern.

### A. Customisation Performance

To find the improvements achieved by customisation we rebuilt our design for the OPRA message format. Comparing our two implementations using Equation 4, we find the resource requirements to implement a customisable feature. Essentially, this acts as a customisation co-efficient.

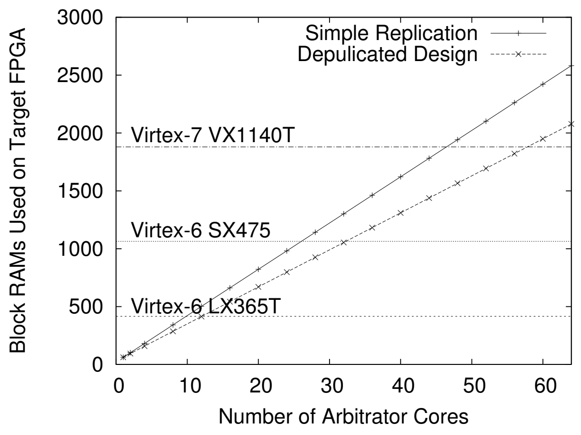

Fig. 2. The number of arbitrator cores able to fit into an FPGA.

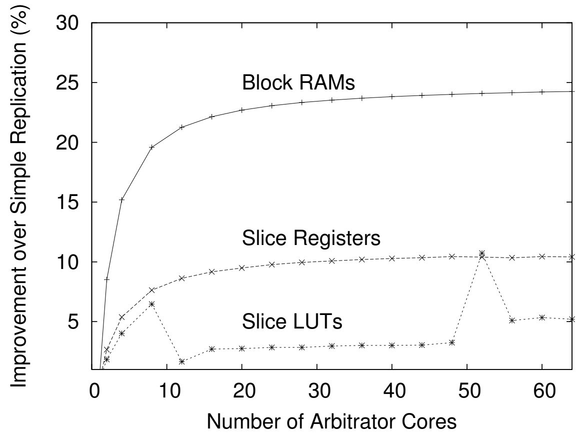

Fig. 3. Scaling improvement of our design over simple core replication.

For block RAMs—the critical factor in determining how many cores can be placed within an FPGA—we expect $N = 6$ times less buffer space to be used as we require only 1500 byte payloads, not the 9000 bytes of TotalView-ITCH. For $C = 64$ cores, $R_A = 542$ block RAMs for OPRA and $R_B = 2078$ for TotalView-ITCH. The resources requirement is then $R_{cus} = \frac{542}{64} - \frac{(6*542)-2078}{64*(6-1)} = 4.8$ block RAMs.

Similarly, applying Equation 4 to slice registers and LUTs, we get $R_{cus} = 731$ registers and $R_{cus} = 1490$ LUTs respectively. Solving for Equation 5 using our $R_{cus}$ values and our implementation results, we find the general form for our total block RAM, register and LUT usage to be:

$$\begin{aligned} R_{bRAMs} &= 30 + C * (3.2 + N * 4.8) \\ R_{reg} &= 6724 + C * (1556 + N * 731) \\ R_{LUTs} &= 9667 + C * (2325 + N * 1490) \end{aligned} \tag{6}$$

Looking at slice registers for example, since OPRA uses 32 bits for sequence numbers and TotalView-ITCH uses 64 bits, in an ideal system we expect an $N = 2$ times reduction in FPGA registers. For a $C = 12$ core system, we find that 34168 registers are needed for OPRA and 42940 for ITCH, a 20% reduction.

We can project similar calculations to an arbitrary number of cores and customisations. Resource trade-offs within and between cores are then possible without requiring a number of time-consuming, trial-and-error build processes.

### B. Software Speed-up

Finally, we compare the performance of our FPGA arbitrator with that of a software implementation. Fully-optimised software is often commercial in nature and kept confidential. For comparison, we refer to work done on market data feed processing using the cutting-edge IBM PowerEN processor [10]. Out-of-order packets are stored in L2 cache and a time-based windowing mechanism is used, rather than the combined time & message count method in this work. Arbitration takes $150ns$ compared to $56ns$ in our design. Thus, our design achieves a 2.6 times speed-up over the software. The timeout mechanism in [10] uses a 33MHz clock, so the resolution is only $30.3ns$ compared to $8ns$ in our design.

## VII. Conclusion

In this paper we outline a reconfigurable accelerated approach to market feed arbitration operating at the network level. Application-specific customisations are provided within each arbitrator allowing it to uniquely benefit feed processing platforms. As multiple arbitration cores may be present within a single FPGA we can simultaneously provide application-specific arbitration to a wide range of financial applications. Our design within a Xilinx Virtex-6 FPGA allows 12 cores to function in parallel, which would rise to 26 or 46 cores for the largest class of Virtex-6 and Virtex-7 FPGAs respectively.

Our implementation operates at 16Gbps throughput, 60% higher than the target 10Gbps Ethernet input line rate, and with resource sharing, supports 12 independent cores, 33% more than in the case of simple core replication. A 56ns (7 clock cycles) windowing latency is achieved, 2.6 times lower than a hardware-accelerated CPU approach.

## References

[1] G. Morris, D. Thomas, and W. Luk, "FPGA Accelerated Low-Latency Market Data Feed Processing," in *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, 2009.

[2] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. A. Vissers, "A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT)," in *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium*, 2012, pp. 9–16.

[3] C. Leber, B. Geib, and H. Litz, "High Frequency Trading Acceleration Using FPGAs," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, 2011, pp. 317–322.

[4] "Solarflare AOE Line Arbitration Brief," 2013. [Online]. Available: http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_AOE_Line_Arbitration_Brief.pdf

[5] "OPRA (Options Price Reporting Authority) Participant Interface Specification," Tech. Rep. [Online]. Available: http://www.opradata.com/specs/participant_interface_specification.pdf

[6] "NASDAQ TotalView-ITCH 4.1 Specification," 2013. [Online]. Available: https://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTV-ITCH-V4_1.pdf

[7] "Atlantic Interface," Altera, 2002. [Online]. Available: http://www.altera.co.uk/literature/fs/fs_atlantic.pdf

[8] T. Takenaka, M. Takagi, and H. Inoue, "A scalable complex event processing framework for combination of SQL-based continuous queries and C/C++ functions," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 237–242.

[9] "MoldUDP64 Protocol," 2009. [Online]. Available: http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/moldudp64.pdf

[10] D. Pasetto, K. Lynch, R. Tucker, B. Maguire, F. Petrini, and H. Franke, "Ultra low latency market data feed on IBM PowerEN," *Computer Science - Research and Development*, vol. 26, pp. 307–315, 2011.

**Acknowledgement**
This work is supported in part by the European Union Seventh Framework Programme under the grant agreement number 257906, 287804 and 318521, by UK EPSRC, by Maxeler, and by Xilinx.