Exploring the Potential of Reconfigurable Platforms for Order Book Update

Conghui He*, Haohuan Fu*, Wayne Luk†, Weijia Li*, and Guangen Yang*
*Tsinghua University, Email: {haohuan,ygw}@tsinghua.edu.cn, {hch13,liwj14,}@mails.tsinghua.edu.cn
†Imperial College London, Email: w.luk@imperial.ac.uk

***Abstract*—The order book update (OBU) algorithm is widely used in financial exchanges for rebuilding order books. The number of messages produced has drastically increased over time. The software solutions become more and more difficult to scale with the growing message rate and meet the requirement of low latency. This paper explores the potential of reconfigurable platforms in revolutionizing the order book architecture, and proposes a novel order book update algorithm optimized for maximal throughput and minimal latency. Our approach has three main contributions. First, we derive a fixed tick data structure for the order book that is easier to be mapped to the hardware. Second, we design a customized cache storing the top five levels of the order book to further reduce the latency. Third, we propose a hardware-friendly order book update algorithm based on the data structures we proposed. In the experiment, our FPGA-based solution can process 1.2-1.5 million messages per second with the throughput of 10Gb/s and the latency of 132-288 nanoseconds, which is 90-157 times faster than a CPU-based solution, and 5.2-6.6 times faster than an existing FPGA-based solution.**

***Keywords*-FPGA; finance; algorithm; latency; order book**

I. Introduction

The financial markets with significant participation of algorithmic trading strategies have an increasing demand for low latency access, therefore brokers, banks and funds have continued to invest in both faster algorithms and low latency networks. As software solutions have long and random latency due to the operating system and event driven interrupts, more and more trading modules such as market data feed arbitrators, option pricing algorithms and high frequency trading libraries, are based on reconfigurable hardware platforms that have the potential of revolutionizing electronic trading, by providing significantly improvements in deterministic results, lower processing latency and higher energy efficiency [1] [2] [3].

Even though the brokers, banks and funds keep pushing their hardware solutions to the limit, the systems of their upstream or financial exchanges are mainly still based on the software solutions, which becomes the bottleneck for improving the overall trading efficiency. Compared with the systems of exchanges' clients, it is more challenging for exchanges to design algorithms on hardware platforms owning to the logic complexities and data volume requirements, one of which being the order book update algorithm that needs to maintain a large volume of data in a low-latency way.

The order book update (OBU) algorithm is employed by exchanges to update order books with hundreds of millions of requests every day. The order books are implemented by an electronic list of buy and sell orders for a specific security or financial instrument, organized by price levels. It lists the number of shares being bid or offered at each price point [4]. By collecting the statistical information from order books, the OBU algorithm provides the traders with the changing status of the marketplace, called market data feeds, which enable the traders to reconstruct the marketplace, monitor market conditions, and perform algorithmic trading [5].

However, when the number of messages the financial exchanges need to process drastically increases, the evaluation of the OBU algorithm can become the main bottleneck. This situation would take place when, for example, a financial exchange provides a larger range of products with higher resolution monitoring, or when traders perform algorithmic trading that generates and cancels trades at a much higher rate than expected. The software solutions become increasingly difficult to scale with the growing message rate and meet the requirement of low latency, especially during large bursts of trading activities.

This paper explores the potential of reconfigurable platforms and proposes a hardware-friendly order book update (HFOBU) algorithm that is effectively optimized for achieving maximal throughput and lowest latency on reconfigurable platforms such as those based on field programmable gate arrays (FPGAs). The contributions of our work include a fixed tick data structure for storing the order book optimized for our hardware platform (Section III); a customized cache that stores the top five levels of the order book as well as a clipped reduction tree (Section IV-B), and most importantly a hardware friendly order book update (HFOBU) algorithm based on the data structures proposed above which reduces latency by at least 90 fold (Section V).

II. Background

A. Motivation

Financial exchanges such as the China Financial Futures Exchange (CFFEX) have unified data bus containing and streaming different categories of trading messages. Different modules in exchanges as shown in 1 get input by subscribing to one or multiple categories of messages from the unified data bus and send their results back to the unified data bus, which will be subscribed by other modules. For each type of message, the exchange provides a corresponding data structure and related interfaces that can be used among

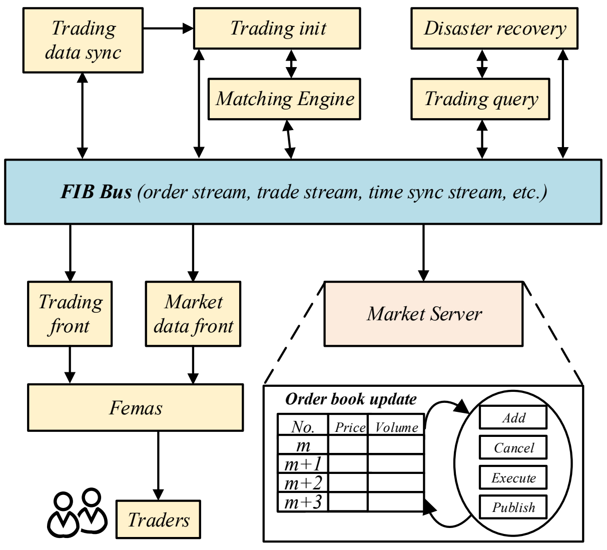

Figure 1. The architecture and modules in CFFEX

different modules. This approach greatly improves design modularity and reduces code redundancy.

One category of messages can be subscribed by multiple modules. For instance, the order stream is subscribed by the matching engine, market server and trading query module. One module only uses a small portion of information from one category of message in most cases. It is unnecessary to parse and maintain the entire message in every module, which not only wastes the memory resources but also also greatly decreases the performance. In addition, it is especially challenging to map the existing object-oriented paradigms and multi-level encapsulated data structures to reconfigurable platforms such as FPGAs. This paper aims to explore the potential of FPGAs and customize the workflow of the marker server module for achieving maximal throughput and lowest latency.

B. Data Structures of Order Books

The market server module maintains and updates order books according to millions of requests from traders in a very low latency. From the point of view of financial trading, an order book is updated and modified mainly from three actions, adding an order, canceling an order and executing a trade. An order book is also frequently accessed when the exchange broadcasts the top of the order book as market data feeds to traders. Table I summarizes the actions and their frequencies of accessing the order book. Good data structures that maintain order books must have low penalties for processing the actions in Table I.

An order book stores all the bid and ask orders of an instrument, identified by an instrument ID. An instrument ID is a string, thus different instrument IDs are organized by a hash table with the perfect hash scheme [6]. In the order book, orders are categorized by different price levels. The price are often sorted so that it is convenient to access the optimal price, which is the lowest bid price or the highest ask price. At each specific price, different orders are queued in a FIFO. An order is identified by the order ID (OID), with additional information such as the price, quantity, direction (bid/ask) and etc.

Table I
DESCRIPTIONS AND FREQUENCIES OF DIFFERENT ACTIONS TO THE ORDER BOOK

Action	Description	Frequency
Add orders	Add a new order at a specific price level to the order book	10,000+/s
Cancel orders	Remove an existing order at a specific price level from the order book	5,000+/s
Execute trades	Match trades when the bid on a particular instrument meets or exceeds the ask price	10,000+/s
Publish feeds	Collect the top 5 levels of price and quantity of each order book and broadcast	2-10/s

Figure 2 presents a typical data structure that maintains order books with a hierarchy of multiple levels. A hash table is used to track different instruments. For each instrument, an AVL tree is employed to organize the price levels of an order book while different orders at the same price are chained by a linked list. A balanced tree has the benefit of inserting, deleting and traversing a node at the complexity of $O(\log(n))$ while the linked list eases the process of adding or removing an order.

C. The Order Book Update (OBU) Algorithm

The order book update algorithm (OBU) is a set of routines to update the order book when the events in Table I are triggered. There are four routines in the OBU corresponding to the four actions in Table I. The OBU algorithm is summarized in Algorithm 1.

1) Add an order: The procedure *Add* illustrates the process of adding a new order to the order book. The order book is got by hashing the instrument ID (line 1). If the price of the new order exists in the order book, we append the new order directly. Otherwise, a node for the new price is created and followed by enqueuing the new order (line 3 to 9).

2) Cancel an order: The procedure *Cancel* in Algorithm 1 shows how to cancel an order. It is guaranteed that the order exists in the order book. To remove an order from the order book, we first traverse the order book to find the price of the order and remove it from the linked list (line 11 to line 14).

3) Execute a trade: The exchanges match trades when the bid on a particular instrument meets or exceeds the ask price. The procedure *Trade* illustrates the process that the quantity of the trade packet is subtracted from an order at the head of the queue. If the remaining quantity of the order

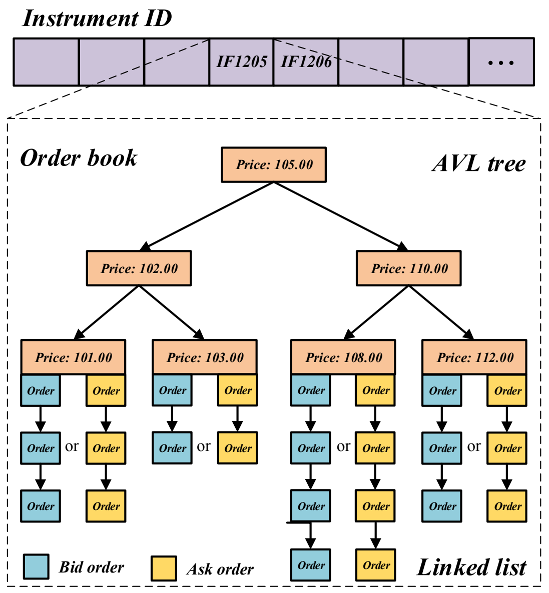

Figure 2. A typical data structure that maintains order books with a hierarchy of multiple levels. A hash table is used to track different instruments. For each instrument, an AVL tree is employed to organize the price levels of an order book while different orders at the same price are chained by a linked list.

in the queue is less than zero, meaning that the order is not valid any longer, the order is dequeued (line 17 to line 25).

4) Publish feeds: The procedure *Publish* illustrates the process of publishing market data feeds (line 27 to line 36). For each order book, we need to collect five highest-ask-price levels and five lowest-bid-price levels. For each price level, the quantity of each order also needs to be summed up.

The computing complexity of the *Add* and *Trade* routines is $O(\log(n))$ where n is the number of price levels, while the complexity of the *Cancel* and *Publish* routines is $O(\log(n)+m)$ where m is the number of order at a specific price level.

Algorithm 1 Original Order Book Update (OBU)

```
1:  book ← getbook(hash(instrumentID))
2:  procedure ADD(OID, price, quantity, dir)
3:      if book.not_found(price) then
4:          book.insert(price)
5:      order = new Order(OID, price, quantity, dir)
6:      if is_bid(dir) then
7:          book.at(price).bidlist.enqueue(order)
8:      else
9:          book.at(price).asklist.enqueue(order)
10: procedure CANCEL(OID, dir, price)
11:     if is_bid(dir) then
12:         book.at(price).bidlist.remove(OID)
13:     else
14:         book.at(price).asklist.remove(OID)
15: procedure TRADE(dir, price, quantity)
16:     if is_bid(dir) then
17:         order = book.at(price).asklist.head()
18:         order.quantity -= quantity
19:         if order.quantity ≤ 0 then
20:             book.at(price).asklist.dequeue()
21:     else
22:         order = book.at(price).bidlist.head()
23:         order.quantity -= quantity
24:         if order.quantity ≤ 0 then
25:             book.at(price).bidlist.dequeue()
26: procedure PUBLISH(orderbooks)
27:     for book ∈ orderbooks do
28:         $N \leftarrow 1$
29:         askodr ← book.head()
30:         bidodr ← book.tail()
31:         while $N \leq 5$ do
32:             bidfd.add(price, sum_qty(bidodr.bidlist))
33:             askfd.add(price, sum_qty(askodr.asklist))
34:             $N \leftarrow N + 1$
35:             bidodr ← bidodr.back()
36:             askodr ← askodr.next()
    return bidfd, askfd
```

III. A Fixed Tick Order Book

A. Data Organization Scheme

The original data structure and the OBU algorithm do not fit well into a pipelined hardware design. The AVL tree and the linked list fit general purpose processors, however, mapping them into hardware will need a great deal of FPGA resources while making the design complex and error-prone.

After carefully analyzing the OBU, we find that the output of the order book is the tuple of the price and the total quantity shown in the *Publish* procedure in Algorithm 1. Thus it is not necessary to chain each order by a linked list. Instead, we can use only one variable to accumulate the total quantity for a specific price level. The reasons are as follows.

- Although keeping the whole information of an order makes it convenient to modify each order, it takes $O(\log(m))$ time to sum up the total quantity for each price level.
- Increasing the total quantity when adding an order and decreasing the total quantity when canceling an order can result in identical market data feeds.
- Tracking the total quantity directly reduces the computing complexity as well as the resource usage.

An AVL tree has the advantage of inserting an order at any price level. In practice, however, the price of an order must be in a specific range in a day, which is $[limitdown, limitup]$. For instance, in China, the

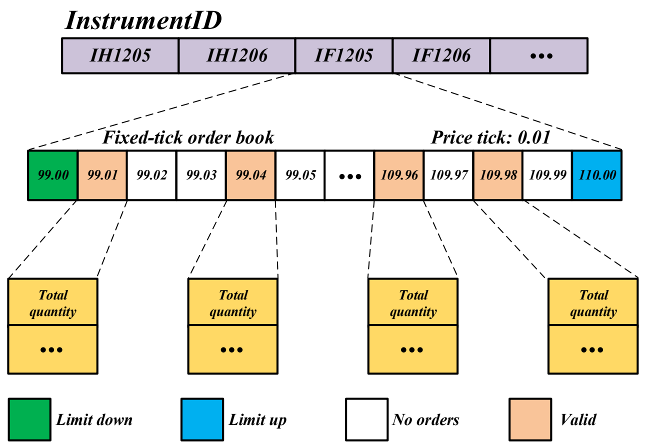

Figure 3. The data organization of the fixed tick order book. The order book contains all possible price levels, with the price tick of 0.01. The price in green and blue is the limit down price and the limit up price respectively. The pink cells represent the valid price. For each price, we store the total quantity.

$limitdown$ is $0.9OP$ and the $limitup$ is $1.1OP$ where OP is the open price of an instrument. Furthermore, the price within the range $[limitdown, limitup]$ are distributed uniformly. The minimum difference between any two price levels is called the price tick, which is a parameter of an instrument.

Taking the factors above into consideration, we derive a fixed tick data structure for the order book that is easier to be mapped to the hardware in less latency, which is shown in Figure 3. The order book contains all possible price levels, with the price tick being 0.01. The price in green and blue is the limit down price and the limit up price respectively. The pink cells represent the valid price. For each price, we store the total quantity instead of keeping each order.

In the fixed tick order book, we can add/cancel an order by directly increase/decrease the quantity of the order to/from the total quantity. The address of the total quantity of a price level is calculated by Equation 1.

$$A_p = A_{ld} + \frac{p - p_{ld}}{p_t} \tag{1}$$

where p is the price of an order; p_{ld} is the limit down price; p_t is the price tick; A_{ld} is the address of p_{ld}.

B. Tree Reduction

The fixed tick order book makes it extremely cheap to add/cancel an order by simple arithmetics. However, current design does not support executing trades and publishing feeds because the top of the order book is needed in both procedures. We employ the parallel tree reduction algorithm to keep track of the top of the book (the highest ask price and the lowest bid price).

Figure 4 presents how the algorithm searches for the highest valid price, which is 99.14. The price levels with

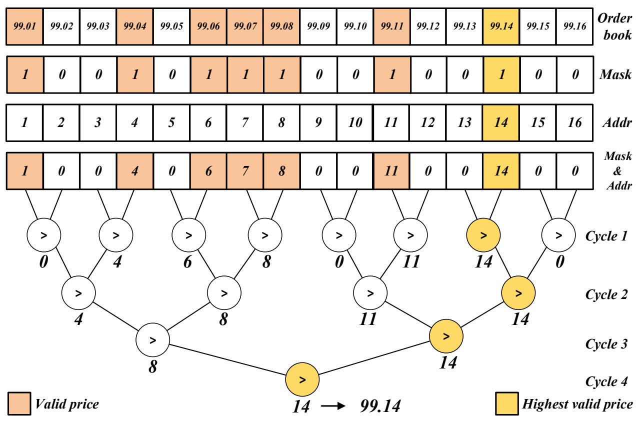

Figure 4. Find the highest price by parallel reduction. A mask table is used to identify the valid price level. The parallel tree reduction is applied to *(mask & addr)*, resulting the address of the highest price.

nonzero quantities are denoted by orange squares. A mask table is used to identify the price with nonzero quantities. Performing the logical *AND* operation to each element in the mask table and the address table generates the *mask & addr* table, which is the input of the parallel reduction algorithm. The tree reduction algorithm parallelizes in space. The pipeline depth of reducing an array of n elements is $\log(n)$. In this case, it takes four cycles to find out the address of the highest price. Then we can have the quantity of the highest price level by reading the resulted address.

If we need the top five price levels, we can mask the highest price after the reduction and perform a second run, and so forth. With the tree reduction algorithm, now we have a complete set of hardware-based OBU routines.

IV. Customized Cache with a Clipped Tree

A. Customized Cache

When executing a trade or publishing the feeds, the top of the book needs to be accessed. Instead of performing a tree reduction every time we need to access the top of the book, we design a customized cache storing the top five levels of the order book, as shown in Figure 5.

The cache is maintained by the bitonic sorting routine [7] [8] [9] to guarantee that the price levels are always sorted when a new order arrives. So the first element of the cache is the top of the order book for executing the trade, and the first five elements of the cache are all we need for publishing the feeds. The *Trade* and *Publish* only need $O(1)$ latency with the cache.

When a new order is added, the cache and the fixed tick order book are updated simultaneously. If the new order whose price belongs to the top five levels of the order book, the new order is added to the end of the cache as well as the fixed tick order book. The price levels in the cache will be in order automatically with the bitonic sort before the next order arrives.

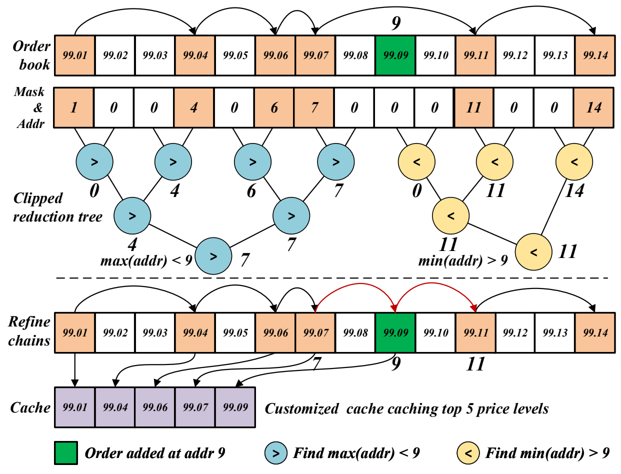

Figure 5. A clipped reduction tree is used to refined the chains of valid price levels when updating the order book. The top five price levels are then stored in the customized cache.

B. Chain Price Levels in Order

The customized cache greatly reduce the latency of the *Trade* and the *Publish* routines, from $O(\log(n))$ to $O(1)$. However, if the price levels are less than five after canceling or executing trades, we need the scheme to load the sub-optimal price levels, which are the valid price levels after the top five levels, from the fixed tick order book to the cache. To achieve that, we refine our fixed tick order book by adding more information to each price level. For each price level, a link is added pointing to the next valid price level, as shown in Figure 5. The link enables the cache to load the sub-optimal price level when necessary.

The links are maintained and updated when adding, canceling orders or executing trades via a clipped reduction tree. We call it a clipped tree because we clip the last level of the tree, resulting two nodes in the last layer instead of one. Figure 5 shows an example of updating the links when adding a new order at the price of 99.09 at address 9 as follows.

- Instead of finding the highest valid price shown in Figure 4, the reduction in Figure 5 searches for the maximal valid address less than 9 and the minimal valid address greater than 9. In this case, they are 7 and 11 respectively.
- Refine the chains according to the output. In this case, the price level 99.07 at address 7 points to the newly added price level 99.09 at address 9, followed by pointing the price level 99.11 at address 11.

The process of refining the chains can also be applied when canceling orders or executing trades with just minor modification.

C. Comparisons and Discussions

Compared with the fixed tick order book without the cache discussed in Section III, current order book design with the cache and the clipped reduction tree further reduce the overall latency by a factor of 5, as shown in Table II. Without the cache, the latency of *Add* and *Cancel* routines are extremely low. They only need to calculate the address of the price level being updated according to Equation 1. However, the overall latency of the design is $5\log(n)$ because the depth of one FPGA design is determined by the longest path, which is $5\log(n)$ in the *Publish* routine. A good FPGA design for low-latency applications tends to balance the depth of different routines as much as possible.

Table II
LATENCY COMPARISONS OF DIFFERENT ROUTINES BETWEEN THE FIXED TICK ORDER BOOK WITH AND WITHOUT THE CACHE.

Scheme	Add	Cancel	Trade	Publish	Overall
fixed tick order book	1	1	$\log(n)$	$5\log(n)$	$5\log(n)$
With cache and clipped tree	$\log(n)-1$	$\log(n)-1$	$\log(n)-1$	1	$\log(n)-1$

The fixed tick order book with the cache and clipped reduction tree is a far better design in terms of latency balancing. First, the latency is amortized to *Add*, *Cancel* and *Trade* routines. The longest depth is reduced from $5\log(n)$ to $\log(n)-1$. Secondly, the clipped tree also reduces both the latency and resource by clipping the last level of the tree.

V. A Hardware-friendly Order Book Update (HFOBU) Algorithm

Data structures and algorithms are essentially and inherently complementary. In most cases, data structures help algorithms achieve their goals in the most efficient way. That is why we propose and update the data structures iteratively to make it best suited on reconfigurable platforms before discussing the algorithms. In this section, we propose the hardware-friendly order book update (HFOBU) algorithm based on the fixed tick order book with the customized cache. We are targeting at an extremely high-throughput and low-latency algorithm for updating order books on reconfigurable platforms such as FPGAs.

A. Hardware-friendly Add

The features of the fixed tick order book greatly simplify the procedure of adding an order. Procedure *HWAdd* in Algorithm 2 shows how to add an order. The core idea is to add the quantity of the new order to the total quantity at the same price in the order book (line 9).

Besides, we also need to maintain the cache and the chains for other routines. If the price of the new order also belongs to the cache (the top 5 levels), the total quantity is accumulated if the price level of the new order is in the cache (line 3 to 5), or replace the last element of the cache because the price of the new order is smaller than that of

$cache[5]$ (line 7). A bitonic sort is then applied to keep the elements in the cache in order (line 8).

To refine the chains of the fixed tick order book, we search the maximum/minimum valid address less/greater than the address of new order, and update the links (line 10 to 12). No matter whether the links exist or not, resetting the links can always guarantee the correctness.

B. Hardware-friendly Cancel

Canceling an order decreases the total quantity from the order book (line 16), which is the opposite of adding an order, as shown in the *HWCancel* procedure in Algorithm 2. We need to be careful if the quantity of the price level becomes zero after canceling the order (line 17), which means that the price level is not valid any longer. We not only need to refine the chains (line 18 to 19), but also need to load the sixth price level to the cache (line 20). A bitonic sort is then applied to keep the cache sorted (line 21).

C. Hardware-friendly Trade

The process of executing a trade is shown in procedure *HFTrade* in Algorithm 2, which is similar to *HFCancel*. The only difference is that the *HFTrade* always subtract the quantity from the top of the book (line 23) in the cache while the *HFCancel* can subtract the quantity of any levels in the cache (line 15).

D. Hardware-friendly Publish

The publish routine is the most time-consuming routine in the OBU algorithm because it needs to find out the top five price levels of the order book and sum up their quantities. The customized cache makes it the most efficient routine, as shown in procedure *HFPublish* in Algorithm 2. As the top five price levels and their quantities are already and always in the cache, collecting them enables us to get all information we need for publishing feeds (line 31 to 32).

E. Discussions

The customized data structure and hardware-friendly order book update (HFOBU) algorithm change the way for storing the information of orders. Compared with the original data structure that utilizes the uniformed interface among different modules in financial exchanges, our new design aggregates the information the market servers needs as well as keeping the same input and output message format while discarding the redundancy so that it can achieve maximal throughput and minimal latency.

The most time and resource consuming operation of HFOBU is the clipped tree reduction. The depth is $O(\log(n))$ and it needs $n-1$ multiplexers and comparators. Note that the logic block of bitonic sort is inexpensive in terms of both time and logic resources because it only needs to sort five elements in the cache table.

Algorithm 2 Hardware-friendly Order Book Update (HFOBU)

```
 1: $addr \leftarrow A_{td} + (price - p_{td})/p_t$          ▷ Equation 1
 2: procedure HFADD(price, quantity)
 3:     if price $\leq$ cache[5].price then
 4:         if cache.find(price) then
 5:             cache.at(price).total += quantity
 6:         else
 7:             cache[5] = (price, quantity)
 8:     sort(cache)
 9:     orderbook[addr] += quantity
10:     (max, min) = orderbook.reduce(addr)
11:     max.next = addr
12:     addr.next = min
13: procedure HWCANCEL(price, quantity)
14:     if price $\leq$ cache[5].price then
15:         cache.at(price).total -= quantity
16:     orderbook[addr] -= quantity
17:     if orderbook[addr] == 0 then
18:         (max, min) = orderbook.reduce(addr)
19:         max.next = min
20:         cache.at(price) = cache[5].next
21:     sort(cache)
22: procedure HFTRADE(price, quantity)
23:     cache[1].total -= quantity
24:     orderbook[addr] -= quantity
25:     if orderbook[addr] == 0 then
26:         (max, min) = orderbook.reduce(addr)
27:         max.next = min
28:         cache[1] = cache[5].next
29:     sort(cache)
30: procedure HFPUBLISH(cache)
31:     for $i \in [1..5]$ do
32:         feeds.add(cache[i])
        return feeds
```

VI. Experiments

A. Experiment Setups

We run our design in the environment of the China Financial Futures Exchanges (CFFEX), which also provides us three data sets, each containing all the packets of one trading day.

We test the accuracy, throughput and latency of four implementations: the CPU implementations of the OBU and the HFOBU algorithms; the FPGA implementation of HFOBU without and with customized cache. The four implementation are named 'CPU1', 'CPU2', 'FPGA1', 'FPGA2' for short in our experimental records. The two CPU implementations are fully optimized and deployed in a PC with Intel Core i7 CPUs. The two FPGA design are mapped to the Maxeler MAX4 MAIA acceleration card with a Stratix-V

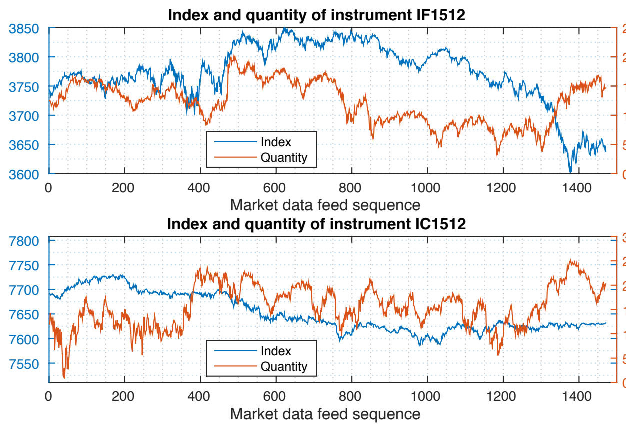

Figure 6. The top of the order book of instrument IF1512 and IC1512 containing the index and quantity from a portion of market data feed sequence.

5SGSD8 FPGA running at 200MHz with 48GB DDR3 on-board memory [10].

We also compare our design with an FPGA implementation from NovaSparks [11] which also rebuilds the order book. However, the targeted exchanges, the protocols as well as the data set we used are different, the latency and comparison we provide here are only for reference. This configuration will be named ‘NovaSparks’ for short in our experiment records.

B. *Accuracy Results*

Using the same data set and initial parameters, we compare the market data feeds generated by the four implementations. Figure 6 presents the top of the order book of instrument IF1512 and IC1512 containing the index and quantity from a portion of market data feed sequence. As the four implementations can generate identical market data feeds, they are plotted as a single curve at each plot.

C. *Throughput Results*

We describe the throughput by the number of messages processed in every second. The throughput of different actions of orders is recorded in Table III. The last column in the table records the speedup value of our FPGA-based HFOBU algorithm with customized cache over the CPU-based solution (the original OBU algorithm).

Table III
THROUGHPUT RESULT (MESSAGES PER SECOND)

Action	CPU1	CPU2	FPGA1	FPGA2	SPEEDUP
Add	$4.52E4$	$5.31E4$	$1.31E6$	$1.31E6$	28.98×
Cancel	$4.81E4$	$5.44E4$	$1.31E6$	$1.31E6$	27.23×
Trade	$4.72E4$	$5.22E4$	$1.25E6$	$1.25E6$	26.48×
Publish	$3.41E4$	$1.52E5$	$1.53E6$	$1.53E6$	44.87×

The throughput of the FPGA-based implementation is highly related to the message size of each packet. Multiplying the number of messages processed by the packet size, our FPGA design can sustain the throughput of 10Gb/s. Compared with the CPU-based implementation of OBU, our FPGA-based HFOBU design can provide 26-44 times more throughputs. Also note that the throughput of the CPU-based HFOBU implementation is larger than the CPU-based OBU implementation. We consider it reasonable because we reduce the complexity of the data structure and eliminate the need to traverse the linked list.

D. *Latency Results*

We measure the latency by subtracting the latency of the network stack from the end-to-end latency of our design. Table IV presents the average latency results of different actions of different implementations. In the fixed tick order book design without the customized cache (FPGA1), we can achieve extremely low latencies for *Add* and *Cancel* routines. But this design suffers long latencies for *Publish* routine. The HFOBU design with the customized cache (FPGA2) achieves more balanced latency between 132-288 nanoseconds, which is 90 to 157 times lower than the CPU-based OBU implementations (CPU1).

Table IV
AVERAGE LATENCY RESULTS (NANOSECONDS)

Action	CPU1	CPU2	FPGA1	FPGA2	SPEEDUP
Add	26544	25693	96	288	92.16×
Cancel	24658	23896	96	276	89.34×
Trade	25451	18695	272	260	97.88×
Publish	20753	19898	1186	132	157.2×

We also compare the latency of our different implementation with the one from NovaSparks [11], which is summarized in Table V. The last two columns record the speedup values of our FPGA-based solution over the CPU-based solution and the NovaSparks’ solution. Compared with the NovaSparks’ FPGA solution, our design can achieve 5.2 to 6.6 times speedup in terms of latency.

Table V
LATENCY COMPARISON WITH NOVASPARKS

CPU1	NovaSparks	FPGA2	SU_{cpu}	SU_{ns}
20.7-26.5us	880-1500ns	132-287ns	90-157x	5.2-6.6x

VII. RELATED WORK

There has been growing interest in using dedicated acceleration logic to accelerate financial applications, specifically using FPGAs as a high-throughput and low-latency solution [2] [12]. Pottathuparambil et al. described an FPGA-based ITCH feed handler to handle a peak data rate of 420 Mbps with a deterministic latency of 2.7μs [13]. Subramoni et al. presented a prototype of an on-line OPRA data feed decoder [14], achieving a latency of less than 4μs. Lockwood et al. presented an FPGA IP library with networking, I/O, memory interfaces and financial protocol parsers [3]. These applications demonstrate FPGAs as an ideal choice for low-latency financial packet processing, however, they mainly focus on parsing, decompressing, filtering and forwarding the market data feeds without the need of maintaining a large volume of data items.

Most reconfigurable designs for rebuilding order books are commercial and their implementation details are usually not presented. The Algo-Logic System provides a full order book with a maximum processing latency of fewer than 230 nanoseconds on a single FPGA Platform [15]. Miles et al. utilized a ternary tree to rebuild the market data from the Nasdaq stock exchange with a latency between 180-205ns [16]. Both of them rebuild the order books with the market data feeds from the exchanges while our design is targeting building order books in the exchanges, which involves more complex operations such as publishing market data feeds. The NovaSparks announces that its FPGA-based order book capability for global cash equities achieves latencies between 800-1500ns [11], which is much worse than our design.

VIII. Conclusion

This paper presents our FPGA-based solution for the order book update (OBU) algorithm for financial exchanges, which is a widely used but latency sensitive algorithm. We derive a fixed tick data structure for the order book to better fit the FPGA architecture. We also design a customized cache with a clipped tree to further reduce latency. Based on these data structures, we propose the hardware-friendly order book update algorithm that is highly efficient on FPGAs. In the experiments, the FPGA-based solution is shown to provide identical results as CPU-based solutions. It achieves the throughput of 10Gb/s with latency of 132-288ns, which is 90-157 times faster than a CPU-based solution, and 5.2-6.6 times faster than an existing FPGA-based solution.

Current and future work includes optimizing the proposed approach to minimize latency while enhancing security for the entire system, and exploring the automation of building block optimization.

Acknowledgment

This work was supported in part by the National Key & D Program of China (Grant No. 2016YFA0602200), the National Natural Science Foundation of China (Grant No. 4137411, 91530323), the China Postdoctoral Science Foundation (No. 2016M601031), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, UK EPSRC (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), Maxeler and Intel Programmable Solutions Group.

References

[1] S. Denholm, H. Inoue, T. Takenaka, T. Becker, and W. Luk, "Network-Level FPGA Acceleration of Low Latency Market Data Feed Arbitration," *IEICE Transactions on Information and Systems*, 2015.

[2] S. Wray, W. Luk, and P. Pietzuch, "Exploring algorithmic trading in reconfigurable hardware." in *ASAP*, 2010.

[3] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers, "A low-latency library in FPGA hardware for high-frequency trading (hft)," in *20th Annual Symposium on High-Performance Interconnects*, 2012.

[4] Investopedia, "Order Book Definition." [Online]. Available: http://www.investopedia.com/terms/o/order-book.asp

[5] G. W. Morris, D. B. Thomas, and W. Luk, "FPGA accelerated low-latency market data feed processing," in *IEEE Symposium on High Performance Interconnects*, 2009.

[6] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *International Conference on Field Programmable Logic and Applications*, 2005.

[7] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *The VLDB JournalThe International Journal on Very Large Data Bases*, 2012.

[8] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on FPGAs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

[9] K. E. Batcher, "Sorting networks and their applications," in *Proceedings Spring Joint Computer Conference*, 1968.

[10] H. Fu, L. Gan, R. G. Clapp, H. Ruan, O. Pell, O. Mencer, M. Flynn, X. Huang, and G. Yang, "Scaling reverse time migration performance through reconfigurable dataflow engines," *IEEE Micro*, 2014.

[11] NovaSparks, "Novasparks announces FPGA-based order book capability for global cash equities." [Online]. Available: http://www.novasparks.com/news-and-events/press-releases/novasparks-announces-fpga-based-order-book-capability-for-global-cash-equities.html

[12] N. A. Woods and T. VanCourt, "FPGA acceleration of quasi-monte carlo in finance," in *International Conference on Field Programmable Logic and Applications*. IEEE, 2008.

[13] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch, and V. Natoli, "Low-latency FPGA based financial data feed handler," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE.*

[14] H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto, "Streaming, low-latency communication in on-line trading systems," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[15] "Full Order Book." [Online]. Available: http://algo-logic.com/orderbook

[16] "A Full-Hardware Nasdaq Itch Ticker Plant on Solarflares AoE FPGA Board." [Online]. Available: http://www.cs.columbia.edu/ sedwards/classes/2013/4840/reports/Itch.pdf