Low Latency Book Handling in FPGA for High Frequency Trading

Milan Dvořák, Jan Kořenek
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno, 612 66, Czech Republic
idvorakmilan@fit.vutbr.cz, korenek@fit.vutbr.cz

***Abstract*—Recent growth in algorithmic trading has caused a demand for lowering the latency of systems for electronic trading. FPGA cards are widely used to reduce latency and accelerate market data processing. To create a low latency trading system, it is crucial to effectively build a representation of the market state (*book*) in hardware. Thus, we have designed a new hardware architecture, which updates the *book* with the best bid/offer prices based on the incoming messages from the exchange. For each message a corresponding financial instrument needs to be looked up and its record needs to be updated. Proposed architecture is utilizing cuckoo hashing for the book handling, which enables low latency symbol lookup and high memory utilization. In this paper we discuss a trade-off between lookup latency and memory utilization. With average latency of 253 ns the proposed architecture is able to handle 119 275 instruments while using only 144 Mbit QDR SRAM.**

I. Introduction

Financial exchanges today prefer electronic trading, which currently accounts for more that 90 % of all trades. This development gave rise to algorithmic trading and high frequency trading (HFT). Traders no longer focus on specific trades, but rather on tweaking parameters of algorithm, which is responsible for the trading itself.

Important aspect of the HFT systems is the latency of reaction on the change in the market, because the exchange executes only the first incoming order. Moreover, the trading algorithms require the most recent market data. Thus, it is important to achieve very low latency of the system reaction for every message received from the exchange.

Pure software solutions became obsolete in this area because of high and nondeterministic latency, which is in the order of tens of microseconds [1]. First efforts were focused on latency of data transfers from network interface to the processor using designated acceleration cards. The transfers were bypassing the kernel to directly access the memory space of user application.

Further reduction of the latency was achieved by accelerating the decoding of messages coming from the exchange. To speed up this time critical operation, a high-performance FPGA card is typically used (e.g. Celoxica AMDC [2]). The ASIC technology is not suitable in this area, as the format of incoming messages is often changing. Majority of published approaches to decoding are limited to a design of application specific processor implemented in the FPGA [3] [4]. Overview of different existing approaches and a proposal of complete application for trading in FPGA is presented by Lockwood [1]. For the algorithmic trading, however, it is important not only to decode the messages, but also to create a representation of the state of the market (*book*).

Therefore we present a hardware architecture, which enables book handling with latency only 253 ns while processing a large number of financial instruments. The proposed architecture is utilizing cuckoo hashing to address a trade-off between memory utilization and lookup latency. The architecture was synthesized for Virtex-5 technology running at 150 MHz. With two modules of QDR SRAM with total capacity of 144 Mbit, it is possible to store the book for 119 275 instruments, which is sufficient for majority of trading systems.

II. Problem Statement

Financial exchanges provide information about the state of the market in multicast data streams, which are often called market data feeds. The feeds consist of messages, each message is describing a single financial instrument. There are several types of messages providing information about a specific event on the market, for example price changes, executed trades, status information etc.

The number of instruments in a feed depends on the type of traded assets. The stock exchange, for example, contains few thousands of instruments. The number of instruments in options market can reach up to hundreds of thousands. The whole options universe is usually divided into several feeds and size of each feed does not exceed 100 000 instruments.

Market data feeds can be used to create the *book*. The book can be described as a large table, which contains information about orders on each instrument, especially the prices and the quantities at which each instrument can be traded. The book always has two sides, the bid (buy) side and the offer (sell) side.

The complete order book contains all orders that were put on each instrument. More orders can have the same price from different traders. Processing these orders (for example, determining the best bid and ask price) is a complex and resource consuming task. Therefore, the complete order book is usually used only for stock.

Information about options is provided in a simpler way. Exchange groups orders with the same price, thereby creating an aggregated book. The aggregated book consists of several price levels, each of which contains the number of aggregated orders and the total number of shares. The number of price levels can be, in theory, unlimited. Thus, exchanges usually provide only several of the best price levels. A book, which is limited to a fixed number of price levels, is called the Depth of Book. The number of price levels in the Depth of Book is usually five, but it can vary slightly between exchanges. If only the best price level is provided, we get the Top of Book.

Different types of books are illustrated in Figure 1. The aggregated book with n instruments is represented by the dotted box. Note that each instrument (dashed line) can have a different number of price levels. The Depth of Book is highlighted in light gray color, the Top of Book in dark gray color.

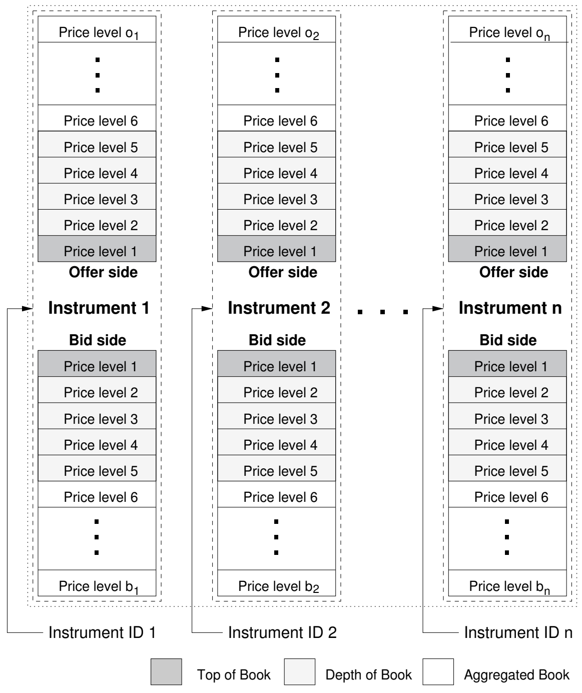

Fig. 1. Aggregated book

If any change occurs in the top five price levels, the exchange generates an update message to inform the user. Typical update messages are insert new price level, delete a price level or change existing price level. For example, processing of message “insert new price level 4 on the bid side” requires to create new price level 4, move previous price level 4 to price level 5 and delete previous price level 5.

In this paper, we focus on the book handling and its implementation in FPGA. The main task is to process the incoming decoded messages and create the book. Specifically, we focus on the Depth of Book for options. Thus, we need to process a large number of instruments – up to 100 000 instruments per feed. These instruments are identified by unique ID, which is usually 32 bits or 64 bits large.

The Depth of Book record for each instrument is memory consuming; it contains prices and quantities for 10 price levels (5 for both sides) and some additional information, like sequence numbers, timestamps, flags etc. Storing such record requires approximately 1100 bits. That is about 105 Mbit for 100 00 instruments. The problem we need to address is how to store this data while keeping the lookup operation as fast as possible. Insert and delete operations are not required as all instrument IDs are broadcast at the beginning of the day and do not change during the processing. To achieve low and deterministic latency, the lookup has to be done quickly, preferable in constant time.

III. Analysis

Hash functions enable fast lookups in large data sets. Main shortcoming of hash functions are collisions, which increase the number of necessary memory accesses and thus the time required for a lookup. Collisions are usually compensated by a larger size of the hash table. Algorithms for hash table implementation differ in the lookup time and maximum achievable utilization of the table. These two requirements are contradictory and it is necessary to find a solution, which is suitable for the book handling. Thus we have conducted an analysis of several algorithms. Comparison of time performance of different hashing schemes running on a PC is presented in [5]. We will focus on the achievable memory utilization.

The first algorithm is a *naive* approach, which uses only one hash function. If the the given index is already occupied, the algorithm fails. This approach can be improved if we allow to store the key in the index given by the hash function or in the N following addresses (*Linear Probing*). Another modification is to compute the N following addresses by adding a value given by a second hash function (*Double Hashing*). We do not consider *Chained Hashing* and *Two-way Chaining* as these algorithms require dynamic memory allocation.

Better results are provided by cuckoo hashing [5], which utilizes two hash functions. These functions determine the addresses, where the key can be stored. It is possible that neither one of the two addresses is free. In that case, new record will replace one of the current occupants. Then, the misplaced record is inserted in its other possible address. This process is repeated until the misplaced record is stored in a free space or until a predefined number of iterations is reached. In the second case, the insertion procedure fails. There are two variants of the cuckoo hashing with different types of memory access. Both hash functions either address the whole memory (*Cuckoo-common*), or the memory is divided into separate parts and each function addresses one part (*Cuckoo-divide*).

The above described algorithms were implemented using the Jenkins function [6]. We conducted tests based on real instrument identificators from the exchange, which were inserted to a table of size 131 072. When the insertion procedure failed, we stored the current memory utilization. Then, the table was cleared and the test was repeated for new randomly generated seeds of hash functions. For each algorithm, 100 000 iterations were run with five different sets of identificators.

Measured values are shown in the graph in figure 2. The x axis shows the memory utilization, the y axis shows the number of successful runs with the given utilization. We set $N = 4$ for *Linear Probing* and *Double Hashing*. Increasing this number would enable higher memory utilization, however, it would also increase the latency, which is not acceptable in

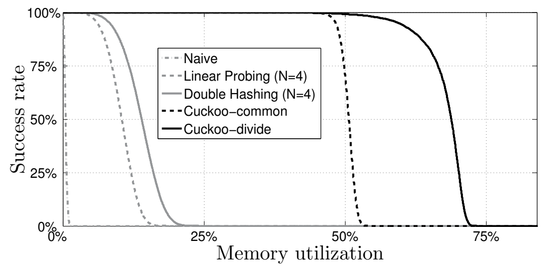

Fig. 2. Achievable memory utilization for different hashing schemes

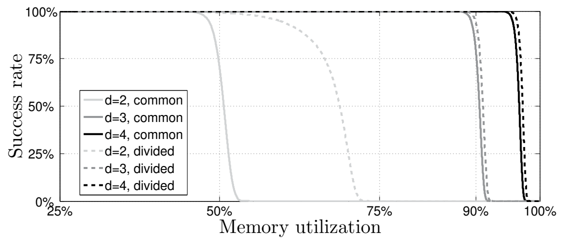

Fig. 3. Achievable memory utilization for d-ary cuckoo hashing

this case. It can be seen in the graph that the best utilization is achieved by cuckoo hashing with separated memory blocks.

Memory utilization of cuckoo hashing can be further improved by using more hash functions. This approach is called d*-ary* cuckoo hashing [7] and it is a generalization of basic cuckoo hashing. Each key can be stored in one of the d slots determined by the hash functions. Graph in figure 3 shows achieved memory utilization for d-ary cuckoo hashing with 2, 3 and 4 hash functions for both variants of the algorithm. The measured values correspond with the theoretical bounds in [8]: 50 % for $d = 2$, 91 % for $d = 3$, 97 % for $d = 4$. The d-ary cuckoo hashing with 4 functions achieves the best memory utilization. However, with more hash functions, the number of memory accesses and thus the latency increases. This trade-off is discussed in more detail in section V. We can also see that it is more beneficial to use the variant with divided memory.

IV. Architecture

In the case of book handling, the input of hashing algorithm is a set of instrument identificators. The set is known at the beginning of each trading day. Thus, we can prepare the hash table in software (insert all identificators to the table) and focus hardware implementation just on the lookup operation.

Each item in the hash table prepared in software contains identificator of assigned instrument or value 0 (we suppose 0 is not a valid identificator). This table and the generated seeds of hash functions are loaded to the hardware as a configuration. The table cannot be stored in the on-chip memory in the FPGA due to its size (approximately 105 Mbit is required for 100 000 instruments). Considering the required capacity and latency, a suitable option is QDR SRAM memory.

Architecture for symbol lookup using d-ary cuckoo hashing with 3 hash functions and divided memory is shown in figure 4. As the external memory provides only one read interface, we do not need to compute all hash functions in parallel. Instead, only one block for hash computation is used. Correct seed for the hash function is provided by controlling *FSM*. Resulting values are then transformed to addresses for the separate memory parts in the *Offset* block. The first value is used directly as an address, other two values needs to be incremented by corresponding offset. The computed addresses are used to read the data records from the *Book Memory*. Keys in the records are compared with the input identificator in the *Validate* block. Valid record is selected according to the comparison result. If neither of the keys is equal to the identificator, the lookup fails and the instrument is ignored. The *Pre-validate* block in the dashed box is described later.

The external memory usually has a high latency. Thus, a whole data record for each instrument is always read from the memory. If a memory with lower latency is available, it would be more effective to first read only the key from each address, then determine the correct one and read only the relevant data record. Also, the keys can be stored in the on-chip memory while the rest of the record remains in the external memory. In such case, a single clock cycle is required to determine the correct address of the instrument and only one record needs to be read from the external memory.

Capacity of the on-chip memory in the FPGA is limited and it is also used for buffering and other parts of the design. Therefore, it is not always possible to store the keys in the on-chip memory. However, we can store only a hash value of the key in the memory. To validate a key, we compare its stored hash value with a hash value of the input identificator (see the *Pre-validate* block in figure 4). If hash values are different, the key can be ignored. Only the valid record has to be read from the memory. Collisions can occur if two different identifiers have the same hash value. Then, two or more records have to be read from the external memory to validate the full value of the key. Our preliminary experiments show that collisions can be avoided if hash size is at least $\lceil log_2 M \rceil$, where M is the number of instruments.

V. Experimental results

The cuckoo hashing architecture described in the previous section was implemented in VHDL. As a testing platform, we used COMBO-LXT card, which is equipped with a Virtex-5 XC5VLX155T chip and two QDR-II SRAM CY7C1513AV18 modules with a total capacity of 144 Mbit.

The data bus interface of the memory modules is 72 bits wide. We use the modules in parallel in order to double the memory throughput and the data word width. One record in the book (1 100 bits) is thus split into 8 words and up to 131 072 instruments can be stored in the memory. The other possibility is to use the two modules as a separate blocks of memory. This enables to read data for two instruments at once, but only at a half speed. Because of that, the minimal latency (i.e. time to read one record) would double when compared to the previous case. The average latency would worsen as well. Thus, it is more beneficial to use the modules in parallel.

We synthesized our implementation using the Xilinx ISE tool version 14.4. Maximum achievable frequency is 159.5 MHz, the real design was running at 150 MHz. The latency of

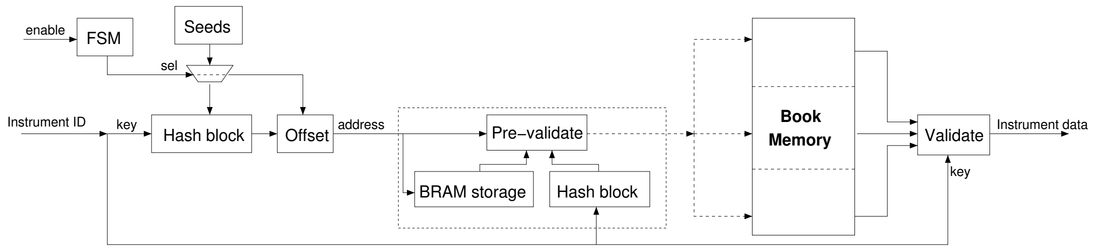

Fig. 4. Cuckoo hash architecture

memory modules was determined by analysis in simulation and measurements in hardware. One read operation has a latency of 19 clock cycles, the following words are available with every clock cycle. Delay of the hash function computation is 4 cycles.

The comparison of d-ary cuckoo hashing with 2, 3 and 4 hash functions in terms of latency and maximum number of stored symbols is presented in table I.

The second column shows minimal, average and maximum latency required for instrument lookup and reading of the data record. The actual value of the latency depends on which part of the memory is the record stored in. For example, if the record is stored in the first part of the table, the latency will be 30 cycles (4 cycles hash computation, 19 cycles memory latency, 7 cycles to read the rest of the record). If, however, the record is stored in the third part of the table, the latency will increase by 16 cycles (the first two records were read unnecessarily).

Comparison of the resource consumption is in the third column. As the block for hash function computation is always shared, the required resources do not differ significantly. The memory controllers are not accounted for in these numbers.

The forth column contains the maximum number of instruments that can be stored in the memory. These values were determined based on the analysis of achievable memory utilization in section III.

The last row of each section of the table shows the theoretical optimal solution, which would have 100 % memory utilization and a latency equal to the time required to read one record from the memory.

The effect of adding more memory modules to the card is illustrated by the last two sections in the table. We can see that the difference between the minimum and maximum latency is decreasing with increasing size of memory data bus. The minimum latency, however, decreases only slightly as the latency of memory controller becomes dominant.

The table shows that the memory utilization increases with the number of used hash functions, however, the average and maximum latency of lookup increases as well. Suitable solution needs to be selected based on the requirements for the trading application and the available memory size.

Algorithm	**Latency [ns]**			**Resources**		**Number of**
	min	avg	max	Flip-Flops	LUT	**instruments**
2 memory modules						
Cuckoo $d = 2$	200	227	253	390	552	65 536
Cuckoo $d = 3$	200	253	307	422	585	119 275
Cuckoo $d = 4$	200	280	360	454	586	127 139
Optimal	173	173	173	—	—	131 072
4 memory modules						
Cuckoo $d = 4$	173	213	253	454	586	254 279
Optimal	147	147	147	—	—	262 144
8 memory modules						
Cuckoo $d = 4$	160	180	200	454	586	508 559
Optimal	133	133	133	—	—	524 288

TABLE I. COMPARISON OF LATENCY, RESOURCE CONSUMPTION AND MAXIMUM NUMBER OF INSTRUMENTS

VI. CONCLUSIONS

The paper proposes book handling hardware architecture for low latency trading on financial exchanges. According to our knowledge, this is the first hardware realization of the book handling. The architecture utilizes external memory to store large book of financial instruments and uses cuckoo hashing for fast lookup in the book. The architecture was tested on COMBO-LXT card with average lookup latency 227–280 ns. The latency was decreased by two orders of magnitude in comparison to recent software implementations [1]. Moreover, the proposed hardware architecture allows to store large book with 65 536–127 139 instruments.

ACKNOWLEDGMENT

This work was supported by the BUT project FIT-S-14-2297.

REFERENCES

[1] 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)," *High-Performance Interconnects, Symposium on*, vol. 0, pp. 9–16, 2012.

[2] G. W. Morris, D. B. Thomas, and W. Luk, "Fpga accelerated low-latency market data feed processing," *High-Performance Interconnects, Symposium on*, vol. 0, pp. 83–89, 2009.

[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] 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 19th Annual International Symposium on*, 2011, pp. 93–96.

[5] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Journal of Algorithms*, 2001, p. 2004.

[6] B. Jenkins, "Algorithm alley: Hash functions." *Dr. Dobb's J. of Software Tools*, vol. 22, no. 9, 1997.

[7] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," in *In STACS*, 2003.

[8] N. Fountoulakis, K. Panagiotou, and A. Steger, "On the insertion time of cuckoo hashing," *CoRR*, vol. abs/1006.1231, 2010.