DSL Programmable Engine for High Frequency Trading Acceleration

Heiner Litz, Christian Leber and Benjamin Geib
University of Heidelberg
B6, 26, 68131 Mannheim, Germany
{heiner.litz, christian.leber, benjamin.geib}@ziti.uni-heidelberg.de

ABSTRACT
In High Frequency Trading systems, a large number of orders needs to be processed with minimal latency at very high data rates. We propose an FPGA based accelerator for High Frequency Trading that is able to decrease latency by an order of magnitude and increase the data rate by the same rate compared to software based CPU approaches. In particular, we focus on the acceleration of FAST, the most commonly used protocol for distributing pricing information of stock and options over the network. As FPGAs are hard to program, we present a novel Domain Specific Language that enables our engine to be programmed via software. The code is compiled by our own compiler into binary microcode that is then executed on a microcode engine. In this paper we provide detailed insights into our hardware structure and the optimizations we applied to increase the data rate and the overall processing performance.

Categories and Subject Descriptors
B.1.5 [**Control Structures and Microprogramming**]: Microcode Applications – *Direct data manipulation, Special-purpose*

General Terms
Performance, Design.

Keywords
FAST, FIX, low latency, high throughput, FPGA, decoder, DSL, domain specific language, stock, trading.

1. INTRODUCTION
In today's major stock exchanges well beyond 90% of all trades are executed electronically by exchanging billions of messages per second. The automation of trading services through electronic means is motivated by significant advantages in terms of increased speed and reduced cost of transactions. Recently, not only the trading process but also the trading decision making process has been automated in the form of Algorithmic or High Frequency Trading (HFT). According to the Aite Group, the impact of HFT on the financial markets is substantial, accounting for more than 50% of all trades in 2010 on the US-equity market with a growth rate of 70% in 2009 [3]. HFT describes a set of techniques within electronic trading of stocks and derivatives, where a large number of orders are injected into the market at sub-millisecond round-trip execution times [2]. HFT is particularly interesting for so-called market makers whose responsibility is to provide liquidity to the markets by providing quotes for buying and selling stock. Thereby, they enable valuation of stock, at all times, even if there are currently no interested buyers or sellers, by providing liquidity in times of low volatility. High frequency traders utilize several strategies to generate revenue, by buying and selling stock at very high speed. Utilized strategies include liquidity-providing, statistical arbitrage and liquidity detection strategies [2].

In liquidity-providing strategies, for example, high frequency traders try to earn the bid-ask spread which represents the difference of what buyers are willing to pay and sellers are willing to accept for trading stock. High volatility and large bid-ask spreads can be turned into profits for the high frequency trader while in return he provides liquidity to the market and lowers the bid-ask spread for other participants, adopting the role of a market maker. Liquidity and low bid-ask spreads are desirable as they reduce trading costs and improve the informational efficiency of asset pricing [4]. All strategies have in common that they require absolute lowest round-trip latencies as only the fastest HFT firm will be able to benefit from an existing opportunity.

Electronic trading of stocks is conducted by sending orders in electronic form to a stock exchange. Bid and ask orders are then matched by the exchange to execute a trade. Outstanding orders are made visible to the market participants through so-called feeds. A feed carries pricing information of stocks and is multicasted to the market participants using standardized protocols like the Financial Information Exchange (FIX) protocol Adapted for Streaming (FAST) which is used by most stock exchanges to distribute their market data [13]. Therefore, every high frequency trading solution must focus on optimizing the decoding process of FAST and the underlying Ethernet, IP and UDP protocols as much as possible. To achieve this, we propose to offload the complete protocol decoding process into an FPGA based hardware accelerator. While accelerators enjoy the reputation of being hard to program and to maintain, they provide the opportunity of significantly speeding up the process. For

WHPCF'11, November 13, 2011, Seattle, Washington, USA.

example, offloading Ethernet, UDP and IP decoding from software to hardware, bypassing the complete operating system's software stack, can improve latency by an order of magnitude, from 20us to 2us. To address the disadvantage of accelerators in terms of programmability we propose a novel approach that utilizes a Domain Specific Language (DSL) programmable microcode engine to accelerate processing of FAST messages. This approach offers easy adaptability to stock exchanges and feed handlers that employ their own version of the FAST protocol. A detailed description of the microcode engine and the utilized DSL will be presented in this paper. Furthermore, we provide an analysis of the FAST data rates we are currently facing in trading systems and we propose optimization techniques that are required to cope with these data rates.

2. BACKGROUND

The following paragraphs will provide background information about the basic concepts of trading and the trading infrastructure. In addition, the basic properties of the FAST protocol will be introduced, defining the requirements of the trading accelerator we present in this paper.

2.1 Trading Infrastructure

A typical trading infrastructure consists of three different entities which are the stock exchange, the market data feed handlers and the market participants or traders. The stock exchange receives buy and sell orders from the market participants through so-called gateway servers. The stock exchange then tries to match these orders using its matching engines. The market participant is informed in the case of a successful trade while pending orders which are not fulfilled can be canceled by the market participants. Current bid and ask prices are made transparent to the market participants through feed handlers. The feed handlers receive their data from the stock exchange and transmit it with minimum latency to the interested market participants. In many cases, the FAST protocol is utilized to transport the market data feeds. The described trading infrastructure is shown in Figure 1.

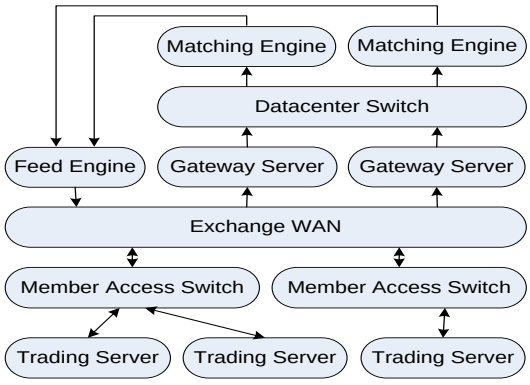

Figure 1. Trading Infrastructure

2.2 Utilized Protocol Stack

In current electronic trading deployments a number of protocol layers need to be traversed to be able to execute trades.

2.2.1 TCP/IP, UDP and Ethernet

The basic communication protocol utilized by stock exchanges and all other market participants is TCP/IP or UDP over Ethernet. Non critical information like market data feeds are generally transferred using UDP to decrease the latency and overhead, critical data including buy and sell orders are generally transmitted using the reliable protocol TCP/IP. Optimized processing of the UDP stack represents a basic requirement for effective high frequency trading systems. For our trading accelerator we decided to offload UDP processing completely into hardware.

2.2.2 FAST

The FAST protocol, which runs on top of UDP, has been specified to transmit market data, from exchanges or feed handlers to market participants. FAST messages contain different fields and operators that are used to transport meta- as well as payload data. FAST, has been optimized for minimal bandwidth usage and, therefore, exploits several compression techniques. The first important technique is to use delta updates, which instead of continuously transmitting all stocks and their corresponding data, only updates, for instance the pricing difference between the current and the previous validation of a stock, are provided. The second technique implements variable length encoding for each data word to compress the raw data as much as possible. While these techniques enable to keep up with the increasing data rates that are provided by the feed handlers, it increases processing complexity significantly. To transform the compressed FAST data stream into processable data, the complete stream needs to be decoded and interpreted in real time. If at any time the processing system cannot keep up with the data rate, critical information is lost. Furthermore, by decompressing the data stream, the bandwidth that needs to be processed effectively increases. Hence, to develop a high performance trading accelerator, two different goals need to be achieved. First, the decoding of the different protocols need to be performed with the lowest possible latency. Second, it must be guaranteed that the data rate can be processed at all times. To analyze the data rates of current trading systems, a closer look into the FAST protocol is required.

FAST messages are transferred using the UDP protocol. To reduce UDP overhead, multiple of these FAST messages are encapsulated in a single UDP frame. FAST messages do not contain any size information nor do they define a framing. Instead, each message is defined by a template which needs to be known in advance to be able to decode the stream. Most feed handlers define their own FAST protocol by providing independent template specifications. Care has to be taken as a single decoding mistake requires dropping the entire UDP frame. Templates define a set of fields, sequences and groups, where groups are a set of fields that can only occur once and sequences are a set of fields that can occur multiple times.

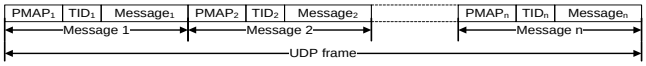

Figure 2. UDP Frame Including Multiple FAST Messages

Each message starts with a presence map (PMAP) and a template identifier (TID) as it is shown in Figure 2. The PMAP is a mask and used to specify which of the defined fields, sequences or groups are actually present in the current stream. Fields can either be mandatory or optional and can in addition have an operator assigned to it. It depends on the presence attribute (mandatory or optional) and the assigned operator if a field uses a bit in the PMAP. This adds additional complexity, as it has to be determined in advance whether the PMAP needs to be interpreted or not.

The TID is used to identify the template needed to decode the message. Templates are specified using XML; an example template definition is given below.

```
<template name="This_is_a_template" id="1">
  <uInt32 name="first_field"/>
  <group name="some_group" presence="optional">
    <sint64 name="second_field"/>
  </group>
  <string name="third_field" presence="optional"/>
</template>
```

In this example the TID equals 1 and the template consists of two fields and one group. A field can be either a string, an integer, signed or unsigned, 32 or 64 bit wide, or a decimal, which is a set of a 32 bit wide signed integer for the exponent and 64 bit wide signed integer for the mantissa.

Data compression is achieved by removing leading zeros within each data word. For example only one byte is transmitted for a sint64 (64bit signed integer) with value '1' even if the actual value is 64bit wide.

To enable decoding of compressed data, the so-called stop bits are used. In each byte, only the first seven bits are used to encode actual data, the eighth bit is used as a stop bit in order to be able to separate the fields. For decompression, the stop bit needs to be removed and the remaining seven bits need to be shifted and concatenated in all cases where a field is larger than a single byte.

Consider the following incoming binary stream:

1̲0000111 0̲0101010 1̲0111111

These three bytes are two fields as it can be seen at the underlined stop bits. In order to receive the actual value of the first field and assuming it is an integer, it is sufficient to replace the eighth bit with a 0. The result is:

Binary value: 00111111

Hex value: 0x3F

The second field spans over two bytes. To get the actual transmitted value of this field, the first seven bits of each of the two bytes need to be concatenated and padded with two 0 bits. The result is:

Binary value: 00000011 10101010

Hex value: 0x03 0xAA

To aggravate decoding, fields also have to be decoded differently depending on their presence attribute. An optional integer for example needs to be decremented by one, a mandatory integer, however, does not. In case of ASCII encoded fields it is sufficient to replace the stop bits with '0' bits.

2.2.3 Decision Making

Decision making, for example the decision to buy or sell a stock, can be a very complex and resource consuming task depending on the applied algorithm. Essentially there are a variety of given parameters and incoming variables that are compared using mathematical and statistical approaches. A detailed analysis of the various algorithms that can be applied is out of the scope of this paper. Due to its complexity, the decision making process is not offloaded to the FPGA but kept in software.

3. BASELINE IMPLEMENTATION

In our previous works [14], we have presented an FPGA based accelerator for high frequency trading. Our proposed approach offloads Ethernet, IP, UDP and decoding of the FAST protocol from software into hardware. The amount of parallelism and the high operating frequencies provided by modern FPGAs enable to significantly increase FAST decoding throughput compared to software based solutions. In addition, by integrating the network interface itself as well as the network stack into the same chip, latency can be reduced significantly. The proposed architecture, as shown in Figure 3, which we describe herein only briefly will act as the baseline for the detailed analysis and the novel optimizations we are introducing in this paper.

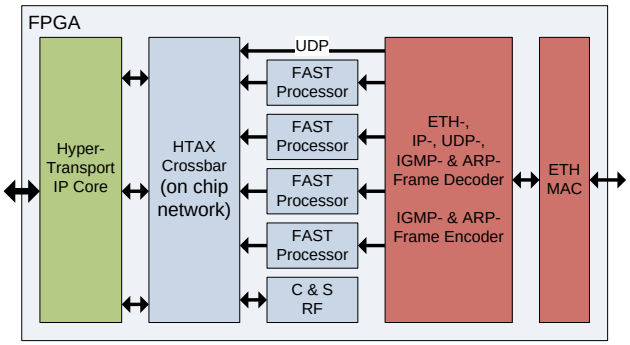

Figure 3. Trading Accelerator Architecture

3.1 IP, UDP and ETH decoding in Hardware

To reduce the latency of market data processing we opted to move decoding of the complete IP, UDP and Ethernet stack into hardware. Our utilized FPGA, a Xilinx Virtex-6, provides high speed serial transceivers which can act as a PHY to directly access the network switch using 10Gbit Ethernet. The Ethernet MAC as well as decoding of the different protocol layers is handled by a number of internal modules. All elements are implemented in Register Transfer Level (RTL) code and utilize hand-optimized finite state machines (FSMs) to provide minimum latency and high throughput. The hardware modules operate at 156 MHz and provide a 64 bit data bus resulting in an on-chip bandwidth of 1.25GByte/s which matches the incoming data rate of 10Gbit/s.

3.2 FAST decoding in Hardware

To further improve the performance of our proposed trading accelerator, also FAST decoding has been offloaded into hardware. However, while UDP decoding is basically identical for each application, the FAST protocol is much more generic and flexible. In particular, each feed handler can define own templates and PMAPs, which makes a comprehensive hardware implementation using FSM based logic unfeasible. Therefore, we opted for a different approach to implement the FAST decoder. On the hardware side we developed a microcode engine that supports a set of instructions which enables it to decode any variation of the FAST protocol. To support a new template, only a specific microcode instruction stream needs to be developed, with the advantage that no modifications to the hardware are required.

The microcode itself is derived from a specification written in a domain specific language (DSL) which we specifically developed for this project. The DSL description is then passed to a compiler which transforms it into binary microcode that can be executed by the microcode engine. The rational for using a microcode engine instead of a general purpose processor is performance. Due to the limited set of instructions which are specifically tailored to operate on FAST data, processing can be accelerated by several orders of magnitude. While our engine utilizes only 2-3

instructions to process a field in the FAST data stream, software based CPU implementations might require several 100'ds of instructions per field. Nevertheless, as it will be seen in the next paragraphs, the FAST processing engine represents the limiting factor for performance.

Figure 4 provides an overview of the hardware implementation of the FAST decoder. All stages are fully pipelined and utilize FIFO buffers to compensate for different and non deterministic latencies of the different units. All three stages operate at 200 MHz and utilize a 64 bit data path. In our utilized FPGA technology, the resource consumption of the FAST processing engine is 5630 Lookup Tables (LUTs).

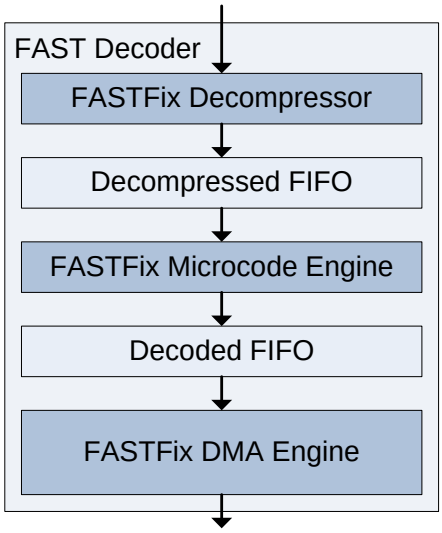

Figure 4. FAST Decoder

3.2.1 FAST Decompressor

The FAST Decompressor detects stop bits and aligns all incoming fields to a multiple of 64 bit. Having fixed size fields is required to alleviate decoding of FAST for the successive units.

3.2.2 FAST Microcode Engine

The microcode engine runs a program that is loaded into the FPGA on startup with an individual subroutine for each template. A jump table provides the pointers that enable to jump to the right subroutine depending on the template ID that is defined at the start of each FAST message. All fields in the FAST messages are decoded according to the corresponding subroutine. Depending on the subroutine, the content of the fields is either discarded, further processed by the microcode engine or forwarded to the DMA unit.

3.2.3 FAST DMA Engine

To provide the trading software with the decoded FAST stream, a DMA engine has been developed. Each field in the different templates is tagged with an eight bit tag to allow efficient and unique identification by the trading software. The DMA engine forwards the received data into a ring buffer that can be directly accessed by the trading software. Relying on user level communication to bypass the operating system and utilizing polling instead of interrupts reduces latency to the minimum.

4. DOMAIN SPECIFIC LANGUAGE

The binary code for the Microcode Engine is produced by an assembler from a simple domain specific language that has been designed for this purpose. The assembler code for the template presented in the introduction above would look like the following.

Table 1. DSL Code Snippet

NOP		STORE_PRES	
SET_TID	0	STORE_TEMP	
NOP		JUMP_TEMP	
CON_U32_MAN	1	INCR_PC_DATA	
NOP		JUMP_PRES	1
CON_S64_MAN	2	INCR_PC_DATA	
CON_ASCII	3	INCR_PC_PM_DATA	

As can be seen, the proposed domain specific language defines four columns. The two left most columns describe the data value which is processed in that specific time step; while the two right most columns specify the command that shall be executed by the microcode engine. In particular, the first column defines the field with its presence attribute, while the second column maps the field to a unique identifier such that it can be later interpreted by software. The third column defines the control command, which increments the program pointer, jumps over some commands, shifts out data from the data FIFO or checks the PMAP. The last column is used to specify the jump target. Not shown here are NOPs that are required to accommodate the branch penalty. Using an assembler in collaboration with a microcode engine makes it easy to adapt the FPGA to template changes of an exchange or even to support additional exchanges without any knowledge of developing FPGA designs. This speeds up the adaption of the FPGA to protocol modifications significantly.

5. FAST PROTOCOL ANALYSIS

In [14], we showed that our baseline implementation provides an aggregated latency of only 2.6 us to decode the Ethernet, UDP and FAST data stream and transfer it to user space where it can be accessed by software. However, with increasing bandwidth of market data feeds it gets increasingly challenging to cope with the ascending data rates. To build a system that can operate under all market conditions, specifically in periods of very high volatility, the FAST protocol and its data rates need to be analyzed in more detail. In particular, the maximum data rates that can occur in realistic trading systems need to be understood. Therefore, we developed a theoretical model that describes an upper bound for the data rate, taking into account the different compression mechanisms and processing overhead of FAST. Second, we present a more realistic model derived from measurements of real market feed data. Subsequently, we present a number of optimization techniques to increase the throughput of our trading accelerator. Formula 1 shows the sustained FAST data rate that is provided by the feed engine. Note, that although, peak data rates can be higher, only the sustained data rate is of interest as peak data rates can be absorbed with a buffer of sufficient size between the Ethernet and the FAST processor:

$$D_{fast} = D_{feed} \cdot \frac{1}{overhead} \cdot compression \qquad (1)$$

wherein *compression* is the compression factor of the FAST protocol, and *overhead* the overhead of the Ethernet and UDP protocol. Accordingly, we can denote the maximum data rate that can be supported by our trading accelerator as:

$$D_{max} = cf \cdot \frac{1}{CPI} \cdot fieldsize \quad (2)$$

wherein *cf* is the clock frequency, *CPI* denotes cycles per instructions and *fieldsize* represent the number of bits per field/instruction.

We can give an upper bound for D_{fast} by choosing the theoretical maximum for the values D_{feed} which is currently 10Gbit/s, and for the *compression* rate which is eight as a maximum of 64 bits can be compressed to the size of a byte. To provide a realistic estimation of the *overhead* we performed an analysis of several GBytes of market data feeds. While the overhead of UDP and Ethernet depends on the size of the packets and various other factors we could determine a median overhead of about 10% for the Ethernet, IP and UDP headers and framing.

$$D_{fast,upper} \approx 10Gbit/s * \frac{1}{1.1} * 8 \approx 72Gbit/s$$

Measurements of real market feed data, however, shows that current data rates are closer to 2Gbit/s and that the median *compression* rate is approximitely 4. Note that in our case we are decompressing every field to 64 bits, even if the original data was only 32 bits wide. Hence, we can give

$$D_{fast,realistic} \approx 2Gbit/s * \frac{1}{1.1} * 4 \approx 7.2Gbits$$

In the same way, we can calculate the sustained data rate Dmax that can be processed by our trading accelerator. In the current implementation fieldsize equals 64 bits and *cf* equals 200 MHz. One might suggest to increase one or both of these values to increase throughput, however, both appears to be difficult. As FAST is a truly sequential protocol, processing of a field depends on the previous field, which renders it impossible to increase the data width by processing multiple fields in parallel. The operating frequency already represents the maximum that can be achieved with the current three pipeline stage design in the deployed FPGA technology. While a deeper pipeline could allow a further increase of the clock rate, the sequential nature of FAST ensures that there are only a very limited amount of instructions that can be executed independently. Furthermore, as will be seen in the next paragraph, performance is mainly limited due to branches in the microcode. A deeper pipeline, therefore, could be even counterproductive as it would increase the branch penalty. The last parameter which we can analyze is *CPI*. In particular, the analysis of our baseline architecture revealed that in average our engine requires 2.66 cycles to process a field, which leads to a maximum data rate of the FAST processor of 4.8Gbit/s:

$$D_{max} = 200MHz \cdot \frac{1}{2.66} \cdot 64 = 4.8Gbit/s$$

As $D_{max} < D_{fast}$ it is obvious that in periods of high volatility our engine might not be able to cope with the data rate. As discussed, the only parameter that can be improved is *CPI*. Therefore, we propose three different optimizations for improving the *CPI* of our approach.

6. OPTIMIZATION TECHNIQUES

As outlined in the previous paragraph, the optimization of the *CPI* parameter represents the only possibility to increase the data rate of the design. Therefore, it is necessary to determine the instructions which consume multiple clock cycles. An analysis of the generated microcode instruction stream reveals that in all cases branch instructions are responsible for reducing the amount of fields that can be processed per cycle whereas the main reasons are control flow branches and processing of variable sized fields. In our design the cost of a branch instruction is effectively three cycles, one cycle for the compare instruction and two cycles to fetch the instruction pointer from the instruction memory. As it can be seen in Table 2, each time the program counter (PC) is changed due to an instruction, the execution logic will see the effect not until three cycles later. In contrast to sequential code where the PC is simply incremented, branches require the introduction of two NOP instructions into the pipeline. The table shows a typical program flow, where row 1 defines the control instruction, in this case it implements processing of an ASCII field. The issue with ASCII fields is that they have an arbitrary length, therefore it is necessary to take a branch depending on whether the end of the field has been detected. In this case, JUMP_EOFIELD_DATA will simple increment the PC otherwise it will set the PC back to the original program counter that points to the ASCII handling sequence.

Table 2. Branch Penalty

CON_ASCII_MAN	42	JUMP_EOFIELD_DATA	-3
NOP		INCR_PC	
NOP		INCR_PC	
CON_U32_MAN	43	INCR_PC_DATA	

Due to its size, the instruction code needs to be stored in block RAMs. Providing an instruction cache was considered, but dismissed due to the relatively low temporal and spatial locality provided by the microcode and the hardware complexity which limits the size of such an Icache to only a few entries. Therefore, we followed a different direction and developed a set of branch avoidance techniques which we will present in the following.

6.1 Custom Instruction Segments

Decoding of FAST messages follows certain rules which are explicitly defined by the TID and PMAP. Therefore, for each incoming message our microcode engine first performs a lookup of the TID to determine the instruction code that needs to be executed for this template. Within that instruction stream, further branches are necessary depending on the presence of individual fields defined by the PMAP. An obvious solution that significantly reduces the number of branches is to offer discrete microcode segments, not only for each template but also for each {TID, PMAP} tuple as this tuple uniquely predetermines all branches of a message. The disadvantage of this approach, however, is that it increases the development overhead substantially as for each combination a microcode segment needs to be provided. Furthermore, the size of the instruction code would increase significantly. Nevertheless, an analysis of large traces of FAST data regarding the utilized {TID, PMAP} tuples, provides an interesting observation. As it can be seen in Figure 5, only five tuples represent a 92% probability of occurrence. Therefore, it might be beneficial to provide discrete microcode segments for these tuples while processing all other tuples with the generic microcode segments. This approach has been implemented and indeed reduces the number of branches significantly. The evaluation shows that *CPI* could be reduced from 2.66 to 2.35 which equals an improvement of 13%. The size

of the microcode was increased only marginally and can be stored in the same block RAM capacity. The development efforts to implement the additional microcode sequences were modest due to our efficient DSL based approach.

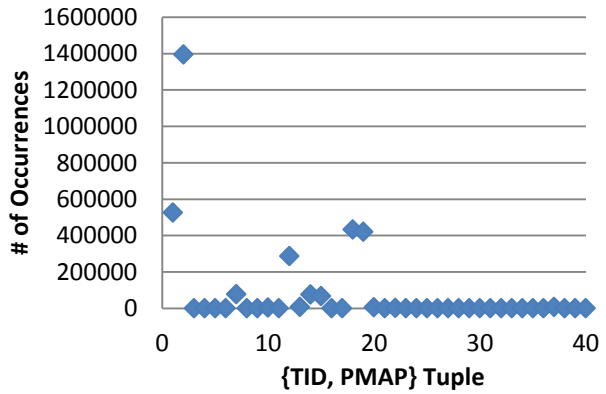

Figure 5. Occurrences of all supported TID, PMAP Tuples

6.2 Branch Precalculation

As outlined, each branch instruction requires three clock cycles, and effectively causes two bubbles or NOPs in the processing pipeline. The main reason for this delay is the latency of the block RAM that is utilized to store the microcode. In our architecture, there exist two different types of branch instructions. The first one uses the TID to lookup the instruction pointer segment, the other instruction is used to branch within the microcode segment. The template segment lookup latency can be completely hidden by performing a speculative lookup of the incoming fields and forwarding that information to the template memory. As UDP frames contain a large number of templates, the TID lookup occurs often providing a good opportunity for optimization.

This approach will not reduce the startup latency for the very first FAST message of an UDP packet, because the first lookup will still imply the three cycle penalty. However, as the decompressor can always process one field per cycle and is hence much faster than the microcode engine, the FIFO in between the two units will already contain the new branch address as soon as the microcode engine is finished with the first FAST message. Forwarding of branch information in this way can, thereby, reduce the TID branch penalty completely and provided a *CPI* reduction from 2.35 to 2.1.

6.3 Variable Length Field Processing

Another source of branch instructions is the processing of variable length fields. Due to the stop bit compression scheme integers within the raw FAST stream can have a size of in between 8 – 72 bits (64 bit payload + 8 stop bits). As our datapath is only 64 bits wide, integers can either span one or two clock cycles which needs to be checked for each field and can lead to a branch. In both cases the compare instruction introduces additional NOP cycles. We addressed this issue by increasing the datapath to 72 bits and by speculatively decompressing the field as an integer. In the case where the field, in fact, represents a multi cycle integer it can be processed directly; in all other cases the additional 8 bits are simply ignored. This approach avoids all branch instructions for integer processing completely. Implementing the variable length field processing optimization resulted in an improvement of *CPI* from 2.1 to 1.7.

7. EVALUATION

We have evaluated the performance of our design by comparing the different optimization techniques to our baseline approach. Figure 6 presents the improvements of our techniques in terms of the maximum data rate. In total, our baseline approach that offers a D_{max} of 4.8Gbit/s could be improved by over 56% to a new $D_{max,\ optimized}$ of 7.5Gbit/s.

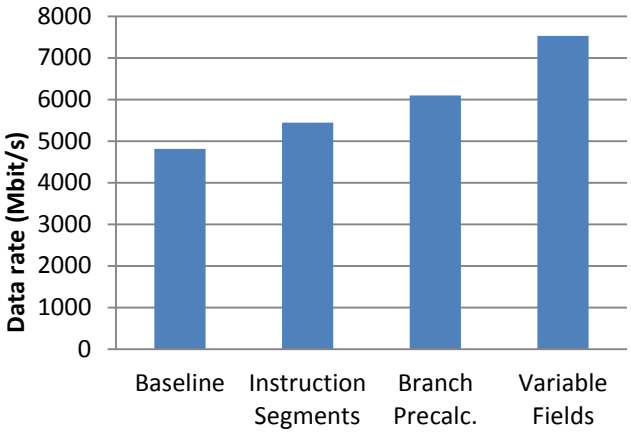

Figure 6. Optimization Techniques

To put our results in perspective we compare our FPGA based trading accelerator to conventional software based CPU approaches. Therefore, we reference performance results [7] that have been measured on an Intel Xeon 5472 and an IBM Power6 using a software based FAST decoder. According to the paper we assume that the average size of a message is 21 bytes. Figure 7 shows the maximum data rates that can be achieved on the three different architectures. The left column shows the single-thread performance while the right column depicts the multi-thread performance of the different architectures. It can be seen that the single-thread performance of our design of 7.5Gbit/s is about one order of magnitude higher than the performance of the CPU approaches (320Mbit/s and 640Mbit/s respectively). In addition, we present multi-threaded performance results, although of limited use, as individual messages can only be processed by a single thread concurrently. As a result, multi-thread implementations can only process multiple independent streams in

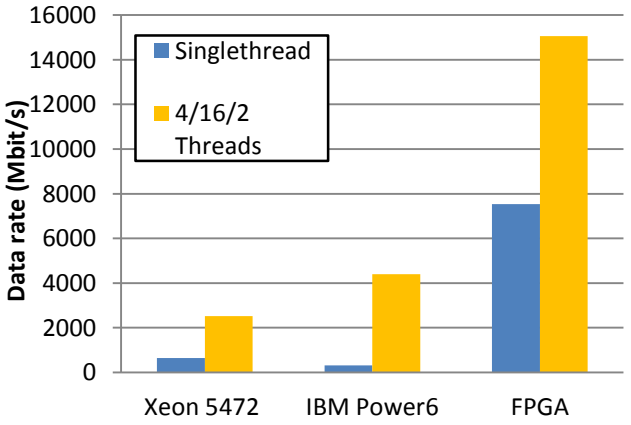

Figure 7. FAST Data Rate

parallel. The Intel implementation uses a maximum of 4 threads, IBM uses 16 threads and our engine uses 2 threads. Although, as shown in Figure 3 we were able to fit up to four FAST processors into our design, we have limited our approach to a maximum of two units as in this case the host interface as well as the ethernet connection already becomes the performance limiting bottleneck. Nevertheless, our approach is well prepared for even higher upcoming data rates.

Another reason for offloading FAST decoding to an FPGA is the latency reduction that can be achieved. We have compared the latency of processing an average sized FAST message (21 bytes) of the different implementations. As can be seen in Figure 8, the Intel Xeon implementation requires 261 ns, the IBM implementation 476 ns while our FPGA based microcode engine only requires about 40 ns to decode a 21 byte FAST message.

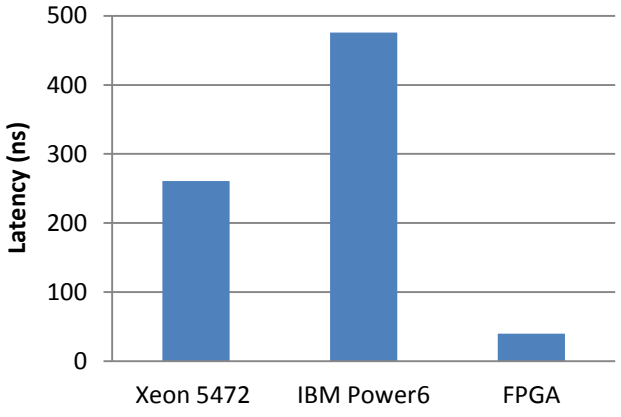

Figure 8. FAST Decoding Latency

8. RELATED WORK

A good introduction to HFT is given in [1]. An introduction to FAST processing and its acceleration is provided in [6], while the previously mentioned "faster FAST" processing engine using a multi-threaded approach is presented in [7]. While both approaches focus on accelerating FAST decoding, to the best of our knowledge our approach is the first one that deploys FPGA hardware for this purpose. Morris [8] presents an FPGA assisted HFT engine that accelerates UDP/IP Stream handling similar to [9]. Sadoghi [10] proposes an FPGA based mechanism for efficient event handling for algorithmic trading. Mittal proposes a FAST software decoder that is executed on a PowerPC 405 embedded in a Xilinx FPGA [12]. Finally, Tandon has presented "A Programmable Architecture for Real-time Derivative Trading" [11].

9. CONCLUSION

In this paper we have presented a novel FPGA based accelerator for High Frequency Trading. Our design integrates the Ethernet MAC as well as IP, UDP and FAST protocol encoding to provide lowest possible latencies and highest data rates. While the application independent Ethernet and UDP decoders are implemented using optimized finite state machines, we developed a unique microcode engine for decoding FAST messages to increase the flexibility. Following this approach, arbitrary FAST templates and protocols can be adapted to support a wide range of stock exchanges and feed handlers. The microcode engine can be programmed via software using a domain specific language we specifically developed for this purpose. Our DSL programmable engine combines high programmability with high performance, providing data rates of up to 941MByte/s and a latency as low as 40ns for decoding FAST messages which is one order of magnitude faster than traditional software based CPU approaches.

10. REFERENCES

[1] J.A. Brogaard, "High Frequency Trading and its Impact on Market Quality," 5th Annual Conference on Empirical Legal Studies, 2010.

[2] M. Chlistalla, "High-frequency trading Better than its reputation?," Deutsche Bank research report, 2011.

[3] A. Group, New World Order: The High Frequency Trading Community and Its Impact on Market Structure, 2009.

[4] K.H. Chung and Y. Kim, "Volatility, Market Structure, and the Bid-Ask Spread," Asia-Pacific Journal of Financial Studies, vol. 38, Feb. 2009.

[5] J. Chiu, D. Lukman, K. Modarresi, and A. Velayutham, "High-frequency trading," Stanford University Research Report, 2011.

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

[7] V. Agarwal, D. a Bader, L. Dan, L.-K. Liu, D. Pasetto, M. Perrone, and F. Petrini, "Faster FAST: multicore acceleration of streaming financial data," Computer Science - Research and Development, vol. 23, May. 2009.

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

[9] F. Herrmann and G. Perin, "An UDP/IP Network Stack in FPGA," Electronics, Circuits, and Systems (ICECS), 2009.

[10] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-arno Jacobsen, "Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading," Journal Proceedings of the VLDB Endowment, 2010.

[11] S. Tandon, "A Programmable Architecture for Real-time Derivative Trading," Master Thesis, University of Edinburgh, 2003.

[12] G. Mittal, D.C Zaretsky, P. Banerjee, "Streaming implementation of a sequential decompression algorithm on an FPGA," International Symposium on Field Programmable Gate Arrays – FPGA09, 2009.

[13] FIX adapted for Streaming, www.fixprotocol.org/fast

[14] C. Leber, B. Geib, H. Litz, "High Frequency Trading Acceleration using FPGAs," 21st International Conference on Field Programmable Logic and Applications (FPL 2011), September 5-7, 2011, Chania, Greece.