Implementing a High-Volume, Low-Latency Market Data Processing System on Commodity Hardware Using IBM Middleware

Xiaolan J. Zhang∗, Henrique Andrade, Buğra Gedik, Richard King, John Morar, Senthil Nathan, Yoonho Park†, Raju Pavuluri, Edward Pring, Randall Schnier, Philippe Selo, Michael Spicer, Volkmar Uhlig, Chitra Venkatramani
IBM Watson Research Center and IBM Software Group

ABSTRACT
A stock market data processing system that can handle high data volumes at low latencies is critical to market makers. Such systems play a critical role in algorithmic trading, risk analysis, market surveillance, and many other related areas. We show that such a system can be built with general-purpose middleware and run on commodity hardware. The middleware we use is IBM System S, which has been augmented with transport technology from IBM WebSphere MQ Low Latency Messaging. Using eight commodity x86 blades connected with Ethernet and Infiniband, this system can achieve 80 µsec average latency at 3 times the February 2008 options market data rate and 206 µsec average latency at 15 times the February 2008 rate.

Categories and Subject Descriptors
C.3 [Performance of Systems]: Performance Attributes

General Terms
Design, Implementation, Performance

Keywords
Market Data Processing, Commodity Hardware, IBM Middleware

1. INTRODUCTION

Figure 1 shows a typical market data processing system. Market data streams from stock and options exchanges are decoded and normalized by feed handlers continuously throughout the trading day. The full normalized data is routed to decision engines, and a reduced-bandwidth summary feed is also sent to theoretical price servers and risk servers. Decision engines process normalized data and generate quotes that are routed back to the exchanges. The theoretical price and risk servers implement analytical models that determine the behavior of the decision engines. The feed handlers and decision engines are part of the low-latency zone. Market data from stock and options exchanges must be processed as quickly as possible in the low-latency zone in order to take advantage of market opportunities.

Current market data processing systems tend to use specialized software such as in-house custom code and sometimes even custom processors constructed from field-programmable gate arrays (FPGAs) [16, 22]. While such specialized systems achieve good performance for current market data rates, they will be difficult to maintain and adapt to the ever-increasing volumes of market data.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions: c©2009 ACM 978-1-60558-716-5/09/11... $10.00

Figure 1: A typical market data processing system.
In this paper we describe and evaluate a high-performance market data processing system built with general-purpose, stream-processing middleware running on commodity hardware. This approach provides the appropriate high-level abstractions for scalable, continuous, pipelined processing while retaining the flexibility and low cost of commodity hardware.

The general-purpose middleware is System S from IBM Research [13, 20, 21, 23, 24] which has been augmented with transport technology from IBM WebSphere MQ Low Latency Messaging [6]. For the remainder of this paper, we will refer to IBM WebSphere MQ Low Latency Messaging as “LLM.” System S is a distributed stream-processing programming and execution platform. LLM is a high-throughput, low-latency messaging transport. The application is developed in a stream-oriented, operator-based language called SPADE [15, 18, 19].

To evaluate our system, we used OPRA market data [9]. OPRA is the Options Price Reporting Authority. OPRA consolidates options quote and trade information from all US options markets and calculates the “National Best Bid and Offer” for each option. OPRA market data is quite challenging because it is by far the largest source of market data in terms of volume, and OPRA market data is expected to grow rapidly due to the growth of algorithmic and electronic trading.

Figure 2 illustrates the rapid growth of OPRA data between 2000 and July 2008 [13]. The graph shows one minute peaks during this time period. In 2000 the peak was 4,000 messages/sec. In July 2008 the peak was 907,000 messages/sec. If we fit this data to an exponential function, we can see that the peak rate has been increasing by approximately 1.9 times each year. In the past nine years. The fitted exponential function is

\[
y = 4000 \times 1.9^{(x-2000)}
\]

with a starting rate of 4,000 messages/sec.

The paper is organized as follows. In Section 2 we describe System S and LLM. Section 3 describes the characteristics of OPRA market data. In Section 4 we describe our market data processing system including the hardware and network.

2. MIDDLEWARE BACKGROUND

We utilize System S which has been augmented with LLM to implement our market data processing system. System S provides distributed run-time support as well as a programming language called SPADE. LLM provides a high-throughput, low-latency messaging transport on both Ethernet and Infiniband [8].

2.1 System S

System S provides an execution platform and services for applications that need to ingest and process massive volumes of continuous multi-modal data streams. For this paper, it is run on commodity x86 blades running Linux connected with Ethernet or Infiniband. System S has also been run on many different platforms including IBM Blue Gene/P [6], IBM Cell Broadband Engine [4], and IBM z/OS [7], in the context of prototypical applications within Research.

A stream application is composed of one or more jobs, each of which is organized as a data flow graph. Processing operators are nodes in the graph, and data streams between the operators are directed edges in the graph. Each stream carries data in the form of typed tuples. Data may be exchanged between operators within an application or between operators in different applications. An operator may have zero or more input ports that receive tuple streams from upstream operators and zero or more output ports that produce message streams. Source operators create tuple streams from external origins, and sink operators send message streams to external destinations.

System S provides a programming language called SPADE which allows developers to quickly construct applications for processing data streams [19]. The SPADE programming model consists of a stream-oriented operator-based language, a compiler, and a toolkit. The SPADE language provides a set of type-generic, built-in operators (BIOPs) and also allows users to define their own operators (UDOPs). The language allows flexible composition of these operators into the logical data flow graphs representing the desired computation. The SPADE compiler generates C++ operators and operator data flow graphs that can be deployed to cluster nodes at runtime. The generation of operators and transport to use between operators can be specified at compile time.

These operators are deployed onto physical nodes as threads within a System S container process. The cluster can be shared among multiple applications and managed by the System S runtime. Tuples can be sent through a variety of transport types such as TCP, UDP, and Infiniband. Besides managing the deployment of the operators to physical nodes, the System S runtime also manages the stream interconnections, the operator lifecycle, and failure recovery.
2.2 IBM Websphere MQ Low Latency Messaging

LLM is a high-throughput, low-latency messaging transport. It provides multicast and unicast support on UDP, TCP, and Infiniband transport. It also provides higher-level functionality such as high availability, congestion control, and message filtering. For this work we used the LLM Reliable Unicast Messaging (RUM) protocol over TCP (called RUM-TCP) and Infiniband (called RUM-IB).

LLM RUM is a messaging technology above the TCP and Infiniband primitives. It provides a message send and receive API with channel definitions and a push model of message delivery. The developer does not need to handle sockets and streaming directly. RUM enhances the basic TCP and Infiniband capabilities to achieve high throughput and low latency through threading and buffering techniques. There are distinct services that deal with the network transmission and the efficient packetization of messages. The mapping of these services to physical threads is configurable to maximize system performance. A number of separate threads can be working on their tasks of message transmission, receipt, and application processing in parallel while maintaining stream ordering. A number of buffering techniques are used to efficiently store, batch and transmit messages for maximum throughput and minimum latency.

3. OPRA MARKET DATA

Three types of data are distributed by OPRA: quotes from a particular exchange, National Best Bid/Offer (NBBO) quotes, and sales. Quote messages contain bid and/or offer information for a stock option from interested buyers and/or sellers, respectively, on a particular exchange. NBBO data, which is appended to quote messages, identifies the exchange with the best bid and/or offer price for a stock option. Sale messages contain information about the sale of a stock option on a particular exchange. OPRA messages range in size from 60 bytes to 100 bytes.

OPRA messages are delivered by a central provider through UDP multicast to subscribers for a fee. The central provider is the Security Industry Automation Corporation (SIAC) [12]. The messages are grouped into 24 multicast channels based on their stock symbol. Multiple messages are encoded in a packet using FAST (FIX Adapted for STreaming) compression [1]. FAST achieves a compression ratio of about three to one using techniques such as implicit tagging, field encoding, stop bits, and binary encoding. An OPRA packet cannot exceed 1000 bytes.

The OPRA market data we used in our experiments was collected on Monday, February 4, 2008. All 24 channels were collected along with Ethernet, IP, and UDP headers and packet data. Each packet was time-stamped with a nanosecond resolution time stamp. One-minute recordings were taken at 9:31 am, 10:07 am, and 12:47 pm. This represents a total of 1,807,800 packets containing a total of 25,662,619 messages. The packets in our recording average about 350 bytes, and each packet contains an average of 14.2 messages. 90 percent of the messages are quote messages. 40 percent of the quote messages contain NBBO messages. 0.02% percent of all the messages are sale messages.

The feed senders are C++ programs. To evaluate the system at different data volumes, these feed senders can be configured to send the recorded data with a rate multiplier at constant or variable rates. The feed handlers, decision engines, and metric collectors are SPADE C++ UDOPs.

Figures 1(a) shows the route of a quote message for IBM on Exchange 1. Figure 1(b) shows the route of an NBBO message for IBM. Sale messages take the same route as NBBO messages. Feed senders replay the OPRA market data recordings by sending stored packets over IP multicast repeatedly. The quote message for IBM on Exchange 1 is
only routed to the decision engine for IBM on Exchange 1. The NBBO message for IBM is routed to all decision engines that process IBM. In this case there are three decision engines.

Each feed handler ingests a single UDP multicast channel with a highly optimized OPRA FAST decoder written specifically for this system. Our feed handlers only forwards messages from three out of seven exchanges and approximately one-half of the stocks. This represents about 90% of the total message volume. Each decision engine processes all received messages and forwards 1 out of 1,600 messages to the metrics collectors.

The feed handlers time-stamp each packet as it arrives. This time-stamp is copied to each message during the packet decoding and normalization process. Another time-stamp is added when the message is sent to a decision engine. The metrics collector in the third ply adds a final time-stamp, gathering the resulting time-stamps and using them to calculate operator processing times, time between operators, and end-to-end latencies.

Calculating latencies in this way is possible because the clocks in our blades are synchronized with IBM Coordinated Cluster Time (CCT) technology [17, 29]. CCT provides synchronization within 1 μsec for machines connected via Infiniband.

The mapping of operators to hosts is shown in Figure 5. The hosts are IBM HS21 and IBM HS22 blades running Linux running on the IBM HS21 blades sees eight cores. Linux running on the IBM HS22 blades sees 16 cores due to Simultaneous MultiThreading (SMT).

5. SYSTEM OVERHEADS

There are three sources of overhead in our system: operator processing, transport overhead, and middleware/operating system noise.

5.1 Operator Processing

The most significant amount of processing is performed by the feed handler. It must decode and normalize OPRA packets as well as copy NBBO and sale messages going to multiple decision engines. The feed handlers developed for this system implement only the function required to measure the low-latency path. For instance, they only consume the OPRA A channel (OPRA transmits duplicate A and B feeds for redundancy), though they do provide all the function required to measure the performance of an options processing system.

Each feed handler ingests a single UDP multicast channel with a highly optimized OPRA FAST decoder (and normalize) written specifically for this system. To examine

Figure 5: The mapping of SPADE operators to hosts. IBM HS22 blades host the feed handlers, decision engines, and metrics collector. IBM HS21 blades host the feed senders.

Red Hat Enterprise Linux 5.3 [10] maintained at the IBM Wall Street Center of Excellence [6]. The two blades running the feed senders are IBM HS21 blades. IBM HS21 blades contain two Intel Quad Core Xeon E5450 processors running at 3.0 GHz and 16 GB of RAM. The eight blades running the feed handlers, decision engines, and metrics collector are IBM HS22 blades. IBM HS22 blades contain two Intel Quad Core Xeon X5570 (Nehalem) processors running at 2.93 GHz and 14 GB of RAM. All the blades are connected with 1 Gb Ethernet and Double Data Rate (DDR) Infiniband. DDR Infiniband gives us a theoretical limit of 16 Gb/sec, but the attainable throughput is about 12 to 13 Gb/sec.

We settled on the mapping shown in Figure 5 through extensive experimentation. System S and SPADE gives us the ability to easily reconfigure the number of operators in each ply, and the operator placement.

Figure 4: Example message routes.

(a) Quote message for IBM on Exchange 1.

(b) IBM NBBO message.
the overhead of the decoder, we measured its performance with the recordings preloaded into memory. On a 2.3 GHz Intel Xeon processor, the decoder can ingest over 430,000 packets/sec which translates to more than 6,000,000 messages/sec. By comparison, Agarwal [13] reports that their decoder operates at 3,400,000 messages/sec on a 3 GHz Xeon processor. At these speeds, decoding does not introduce a significant amount of latency.

To mitigate the effects of the copying overhead from the NBBO and sale messages, the feed handler employs a pipeline strategy with two threads. A receiver thread decodes packets and places messages on a queue. A sender thread takes messages off the queue and routes them accordingly, possibly copying NBBO and sale messages.

5.2 Transport Overhead

In SPADE, messages that flow between operators are called “tuples.” SPADE-generated code contains classes for manipulating tuples in general-purpose programming languages such as C++. These tuple classes include methods for accessing, inspecting, and setting the values of attributes as well as methods for marshalling, unmarshalling, encoding, and decoding the tuple contents according to a specific network protocol. The serialization and de-serialization methods in a SPADE tuple are capable of converting the C++ object into a byte stream message and converting a byte stream back into a C++ object, respectively.

Tuple processing can be quite expensive due to the cost of marshalling and unmarshalling. In order to avoid these costs, SPADE has implemented façade tuples. Façade tuples eliminate copying overhead by eliminating the transformation between the transmitted data packet and the tuple object. In order to eliminate the transformation, façade tuples are limited to fixed size attributes. Our application can use façade tuples because it does not utilize variable length fields.

In order to measure the effectiveness of façade tuples, SPADE has also implemented raw UDOPs. Raw UDOPs operate on byte blobs, and the programmer is responsible for overlaying structure on these blobs. Implementing our application with both raw UDOPs and façade tuples provides a measure of the cost of tuple functionality such as data stream typing.

5.3 Middleware and Operating System Noise

In our experiments, we found that it was important to quiesce both System S and Linux. To quiesce System S, we disabled all unnecessary services such as system monitoring. We treated Linux similarly, turning off daemons such as gpfs, autofs, postfix, irqbalance, cron, and ipmi.

In order to support high feed sender rates, we increased the UDP socket buffer size to 128 KB on all the feed sender and feed handler blades.

6. PERFORMANCE EVALUATION

In this section, we present results from microbenchmarks and then results from experiments with our market data processing system. All of the microbenchmarks and system experiments were run with three transports: RUM-IB, RUM-TCP, and TCP. RUM-IB is LLM Reliable Unicast Messaging over Infiniband. RUM-IB uses native Infiniband primitives to transmit and receive data. RUM-TCP is LLM Reliable Unicast Messaging over Ethernet. RUM-TCP uses TCP primitives. TCP is plain TCP over Ethernet.

6.1 Transport Microbenchmarks

We measured the throughput and latency of the transports using simple SPADE applications on the IBM HS22 blades. Throughput was measured with a SPADE application that consists of sender and receiver operators. The sender and receiver are placed on two different blades. Messages are transmitted as quickly as possible and the throughput is calculated over a three minute period.

We measured latency with a SPADE application that consists of a sender, a receiver, and a reflector. Messages from the sender to the receiver are routed through the reflector. The sender and receiver run on the same blade. The reflector runs on a different blade. The sender time-stamps each message and the receiver is able to use this time stamp to calculate the sender-to-receiver latency.

Figure 6(a) shows the throughput measured for the three transports. We only show message sizes between 20 and 120 bytes because OPRA messages fall within this range. RUM-IB slightly outperforms TCP. For some message sizes,
<table>
<thead>
<tr>
<th></th>
<th>RUM-IB</th>
<th>RUM-TCP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw UDOPs</td>
<td>15x</td>
<td>5x</td>
<td>3x</td>
</tr>
<tr>
<td>Façade tuples</td>
<td>15x</td>
<td>5x</td>
<td>3x</td>
</tr>
</tbody>
</table>

Table 1: Maximum feed sender rates for the three transports.

RUM-TCP outperforms RUM-IB and TCP. For other message sizes, it does not. However, these relationships do not hold true for latency and overall system performance.

Figure 6(b) shows the sender to receiver latency for the three transports using a message of 84 bytes. The messages are sent at rates ranging from 25,000 messages/sec to 250,000 messages/sec, which is approximately the range the feed handlers and decision engines must handle as discussed in the next section. As we can see from Figure 6(a), these rates are well below the maximum sustainable rates. At these rates, RUM-IB clearly outperforms RUM-TCP, which clearly outperforms TCP.

6.2 System Experiments

To examine the performance of our system we varied the feed sender rate and examined the latencies of 65,535 output messages. Output messages are emitted from decision engines at the rate of one for every 1600 quote messages so this covers a period during which the system ingests approximately 105 million messages. The 65,535 output messages were captured during normal operations. We configured the system to use both raw UDOPs and façade tuples as well as the three transports: RUM-IB, RUM-TCP, and TCP.

6.2.1 Throughput

We define throughput as a rate multiplier of the feed sender. In our experiments we varied the feed sender rate multiplier between 3x and 15x. By “15x”, we mean we replay the OPRA recordings at 15 times the original speed, maintaining the relative inter-arrival times between packets.

Due to the lack of industry standards and the bursty nature of OPRA traffic, it is difficult to translate the feed sender rate multiplier to a message rate. From the recordings, we calculated the 15x peak rates with various time bins. With 1 minute bins, the peak rate is 3 million messages/sec. With 5 second bins, the peak rate is 4.35 million messages/sec. With 1 second bins, the peak rate is 5 million messages/sec. With 1 millisecond bins, the peak rate is over 10 million messages/sec. Note that these are peak rates before the feed handlers. While the feed handler only retains 90% of the messages, 40% of the retained messages are NBBO messages, which are sent to three decision engines. This means the message volume and peak rates after the feed handler are slightly higher than those before it.

These peak rates are aggregated across 24 feed handlers. If we assume a uniform distribution of traffic, each feed handler will generate 125,000 messages/sec if we consider 1 minute binning and up to 209,000 messages/sec for 1 second binning.

While raw UDOPs impose less overhead than façade tuples, we found no significant difference in the feed sender rates that both can support. Table 1 shows the maximum throughput for the three transports with both raw UDOPs and façade tuples. We found that RUM-IB can handle feed sender rates up to 15x. Above 15x, the latencies increase significantly. RUM-TCP can handle rates up to 5x. TCP can only handle rates up to 3x. The differences between raw UDOPs and façade tuples are found in message latencies, which we discuss next.

6.2.2 Latency

To examine latency in detail, each operator adds a time stamp when it receives a message and when it sends a message. The feed handler adds a time stamp to each packet it receives, which is copied to all the resulting messages from the packet. The metrics collector adds a final time stamp to each packet it receives and processes all the time stamps. The resulting data is placed in memory.

Figure 7(a) shows the distribution of message processing latencies for raw UDOPs and façade tuples with three different transports at various feed sender rates. Complete distributions are not shown due to long tails.

Figure 7: Distribution of message processing latencies for raw UDOPs and façade tuples with three different transports at various feed sender rates. Complete distributions are not shown due to long tails.
Table 2: Latencies in µsec and outliers for raw UDOPs with three different transports and various feed sender rates. An outlier is a latency greater than or equal to 1 msec.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Outliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUM-IB 3x</td>
<td>68</td>
<td>28</td>
<td>663</td>
<td>0</td>
</tr>
<tr>
<td>RUM-IB 15x</td>
<td>169</td>
<td>25</td>
<td>3,311</td>
<td>0.66%</td>
</tr>
<tr>
<td>RUM-TCP 3x</td>
<td>303</td>
<td>91</td>
<td>10,994</td>
<td>1.18%</td>
</tr>
<tr>
<td>RUM-TCP 5x</td>
<td>1,276</td>
<td>97</td>
<td>684,553</td>
<td>6.32%</td>
</tr>
<tr>
<td>TCP 3x</td>
<td>345</td>
<td>85</td>
<td>38,314</td>
<td>1.54%</td>
</tr>
</tbody>
</table>

Table 3: Latencies in µsec and outliers for façade tuples with three different transports and various feed sender rates. An outlier is a latency greater than or equal to 1 msec.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Outliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUM-IB 3x</td>
<td>80</td>
<td>32</td>
<td>773</td>
<td>0%</td>
</tr>
<tr>
<td>RUM-IB 15x</td>
<td>206</td>
<td>25</td>
<td>8,445</td>
<td>1.93%</td>
</tr>
<tr>
<td>RUM-TCP 3x</td>
<td>328</td>
<td>88</td>
<td>206,700</td>
<td>1.37%</td>
</tr>
<tr>
<td>RUM-TCP 5x</td>
<td>27,811</td>
<td>105</td>
<td>1,398,228</td>
<td>29.02%</td>
</tr>
<tr>
<td>TCP 3x</td>
<td>44,882</td>
<td>83</td>
<td>44,882</td>
<td>1.96%</td>
</tr>
</tbody>
</table>

Figure 7(b) compares the system using raw UDOPs and façade tuples. Looking at 100 µsec latency again, façade tuples are close to raw UDOPs. With RUM-IB and a 3x feed sender rate, the system with façade tuples processes 86% of the messages with 100 µsec (versus 93% for raw UDOPs). With RUM-IB at 15x, the system with façade tuples processes 41% of the messages within 100 µsec (versus 43% for raw UDOPs). With RUM-TCP at 5x, there is a significant gap between raw UDOPs and façade tuples. We believe this gap is due to the system being very close to saturation. RUM-TCP cannot support a rate multiplier beyond 5x, and the extra overhead of façade tuples over raw UDOPs is magnified at 5x.

Table 4: Latencies in µsec of each stage of the message processing route for raw UDOPs with RUM-IB and 15x feed sender rate.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in feed handler</td>
<td>7.93</td>
<td>1</td>
<td>146</td>
</tr>
<tr>
<td>Time to decision engine</td>
<td>119</td>
<td>3</td>
<td>3,244</td>
</tr>
<tr>
<td>Time in decision engine</td>
<td>0.779</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>Time to metrics collector</td>
<td>41.1</td>
<td>5</td>
<td>962</td>
</tr>
</tbody>
</table>

Table 5: Latencies in µsec of each stage of the message processing route for façade tuples with RUM-IB and 15x feed sender rate.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in feed handler</td>
<td>10.64</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>Time to decision engine</td>
<td>160</td>
<td>3</td>
<td>8,410</td>
</tr>
<tr>
<td>Time in decision engine</td>
<td>0.406</td>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>Time to metrics collector</td>
<td>34.7</td>
<td>5</td>
<td>1,468</td>
</tr>
</tbody>
</table>

While the minimum hop latencies in Tables 4 and 5 are less than 5 µsec, the maximum hop latencies range from 962 µsec to 8.4 msec. Figure 8 shows the distribution of the hop latencies for raw UDOPs and façade tuples with RUM-IB at 15x. The distributions show a significant number of outliers.

Experimentation with smaller versions of the market data processing system using a single OPRA channel show that LLM RUM tuning, thread pinning, and interrupt shielding have a large affect on latency and outliers. Unfortunately, we were not able to complete our experimentation before the publication date.

Figure 8: Distribution of “Time to decision engine” (DE) and “Time to metrics collector” (MC) latencies for raw UDOPs and façade tuples with RUM-IB at 15x. Complete distributions are not shown due to long tails.
7. CONCLUSIONS
We have demonstrated the feasibility of building a high-performance market data processing system using general-purpose middleware on commodity hardware. Using IBM System S and IBM LLM on commodity x86 blades connected with Infiniband, this system can achieve 80 µsec average latency at 3x the February 2008 options market data rate and 206 µsec average latency at 15x the February 2008 rate. We eliminated additional messaging overhead by employing raw UDOPs. With raw UDOPs this system can achieve 69 µsec average latency at 3x and 169 µsec average latency at 15x.

With Ethernet this system can only achieve 328 µsec average latency at 3x and 27,811 µsec at 5x. With raw UDOPs this system can achieve 303 µsec average latency at 3x and 1,276 µsec at 5x.

One difficulty in our study was the lack of standards. For example, there are no standard metrics to discuss performance in a broader context. We introduced some in this paper, but they may not be applicable in all cases and when using different technologies.

While the Securities Technology Analysis Center (STAC) provides standard benchmarking services, the services are not open nor are they free [11]. We hope that the research community can begin to address some of these issues in the near future.

Acknowledgements
We would like to thank the LLM team members, Nir Naaman and Avraham Harpaz, for their assistance during the System S-LLM integration. We would like to thank Shu-Ping Chang and Wesley Most for their efforts in maintaining the cluster in Hawthorne, NY and servicing our constant stream of requests and complaints. We would also like to thank David Weber, the Program Director of the IBM Wall Street Center of Excellence, for giving us access to the HS22 and HS21 blades.

8. REFERENCES