Adding Stream Processing System Flexibility to Exploit Low-overhead Communication Systems

Philippe Selo, Yoonho Park, Sujay Parekh, Chitra Venkatramani, Hari K. Pyla, and Fang Zheng
IBM Watson Research Center
Hawthorne, NY 10532
{*pselo, yoonho, sujay, chitrav*}*@us.ibm.com, harip@vt.edu, fzheng@cc.gatech.edu*

***Abstract*—Previously, we demonstrated that we can build a real-world financial application using a stream processing system running on commodity hardware. In this paper, we propose making stream processing systems more flexible and demonstrate how this flexibility can be used to exploit low-overhead communication systems to speed up streaming applications. With our prototype, we now have an options market data processing system that can achieve less than 30 μsec average latency at 30x the February 2008 OPRA rate on a cluster of blades using InfiniBand. Across shared memory, this system can achieve less than 20 μsec average latency at 25x the February 2008 OPRA rate on a single machine.**

***Keywords*-Stream Processing; InfiniBand; Shared Memory; Stock Market Data**

I. Introduction

In this paper, we improve on our previous work [1] wherein we described a streaming application (called the Options Market Data Processing application or OMDP) for processing high volumes of OPRA [2] data using System S. System S is a high performance middleware platform supporting streaming analytics. In the System S programming model, applications are specified as directed dataflow graphs. The graph nodes are operators where computation is performed. The graph edges are streams over which data is transported in typed tuples.

When operators reside in separate address spaces, the stream data transport requires an inter-process communication (IPC) mechanism. The System S runtime provides several built-in data transports over different mediums including Ethernet, InfiniBand (IB), and shared memory. While the built-in transports allow the application developer to focus on developing the analytics, there is no easy way to plug in their own customized transport mechanism or to exploit other technologies such as MPI.

The first contribution of this paper is to increase the runtime system flexibility by allowing a custom data transport to be provided as two operators, a sender and a receiver. The original application specification is then automatically augmented to insert these senders and receivers appropriately. Finally, the application is compiled and deployed as usual. This approach is very powerful, since it adds significant flexibility *without* any changes to the System S runtime system and applications.

The second contribution of this paper is to use this flexible mechanism to explore and build an efficient transport for System S using IB and shared memory. We call this transport Vega.[1] We were also able to partially implement MPI as a System S data transport. Using microbenchmarks and the OMDP application, we compare the performance of Vega to MPI, Reliable Unicast Messaging (RUM) across IB, RUM across shared memory, and operator fusion. Operator fusion is a special mode in System S which improves performance by placing multiple operators in the same process. This allows data transfer by directly invoking the downstream operator logic through function calls instead of using IPC.

We show that Vega outperforms existing System S transports such as RUM and is similar in performance to fused operators and MPI. With our prototype, the OMDP application is now able to achieve less than 30 μsec average latency at 30x the February 2008 OPRA rate on a cluster of blades using IB. Across shared memory, this system can achieve less than 20 μsec average latency at 25x the February 2008 OPRA rate on a single machine.

The techniques used in Vega are largely taken from prior work in optimizing performance with low-overhead communication systems. Our principal inspiration is from Liu [3] but other work in this area includes Subramoni's study of RDMA over Ethernet [4] and Narravula's study of MPI over iWARP [5]. In a recent study, Mancini [6] proposed a hybrid MPI-stream programming model where MPI is used to program parallel operators which are connected with streams. We propose to extend this concept by using MPI (among other techniques) as the fundamental transport mechanism for all operators, not just the parallelized ones.

The rest of the paper is organized as follows. In Section II we provide background on System S transports, MPI, and IB. In Section III, we describe how we extended System S with transport operators. In Section IV we describe the design of Vega transport. In Section V we evaluate and compare Vega to MPI, RUM, and operator fusion across IB and shared memory using microbenchmarks and the OMDP

[1] Vega in Sanskrit means rush or speed.

application. Finally, we provide conclusions in Section VI.

II. Background

A. System S

System S is a high-performance middleware platform for writing applications that process large amounts of streaming data [7], [8], [9].[2] Other stream processing systems include STREAM [10], Borealis [11], StreamBase [12], TelegraphCQ [13], and Infopipes [14]. Each job in the System S programming model may be viewed as a data flow graph with processing operators as nodes and the data streams connecting these operators as directed edges within a graph. The data in each stream is represented by typed tuples. An operator may have arbitrary number of input ports that receive tuple streams and an arbitrary number of output ports that produce tuple streams.

System S applications are written in the SPADE language which enables programmers to compose a set of operators into the data flow graph representing the desired application [15]. In addition to a set of built-in streaming operators, such as for streaming relational algebra, the programming language provides a flexible interface to allow user-defined operators written in C++ or Java. The SPADE compiler translates the Stream application to C++ classes and the resulting binaries are deployed to the cluster nodes at runtime. In our work, we heavily rely on the user-defined operators capability to implement the transport operators.

B. System S Transports

The communication mechanism used for a stream between two ports depends on how the operators are mapped to runtime containers (called *partitions* or *processing elements*). For efficiency purposes, SPADE allows *fusion*, wherein multiple logical operators are physically combined into one runtime process [16]. By default, a single thread executes the code for the subgraph that is contained in the PE, executing each operator in a depth-first manner. For sending a tuple to another operator in the same partition, it (effectively) invokes the downstream operator through function calls, with the tuple being passed by reference (to avoid copy overhead).

For operators contained in different PEs, System S utilizes an inter-process transport called RUM (Reliable Unicast Messaging). This is a configurable high-throughput, low-latency messaging transport. In addition to basic data transport, it provides other features such as high availability, congestion control, and message filtering.

RUM provides a message send and receive API with channel definitions and a push model of message delivery. RUM can be configured to use TCP, IB, or shared memory (within a single host), and it achieves high throughput and low latency by adding threading and buffering. For sending data, an additional thread per System S PE collects messages from different application level senders and is responsible for the actual transmission of data over the medium. On the receiving side, an (optional) thread receives data off the medium on behalf of multiple receivers and then delivers it to the application level threads. Thus, there are usually one and possibly multiple additional threads in the transmission path. Separate threads can be working on their tasks of message transmission, receipt, and application processing in parallel while maintaining stream ordering.

[2]System S is a research project that was productized as IBM InfoSphere Streams in 2009. The System S modifications we describe are not available in the product version at this time.

C. MPI

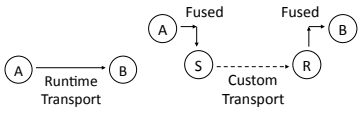
MPI has become the de facto standard for writing HPC applications [17], [18]. Due to its popularity, MPI has been ported to many different processor architectures and can take advantage of many different types of physical networks. MPI-1 provides point-to-point communication operations, collective operations, synchronization, and topology information among other features. MPI-2 adds features such as parallel I/O and dynamic process management. For this work, we used MVAPICH2 [19] and OpenMPI [20]. The numbers reported in the paper were obtained with MVAPICH2.

D. InfiniBand

IB is a high-performance switched network, used as an alternative to standard Ethernet-based networks within a clustered environment. IB owes much of its performance advantages to the network adapter or host channel adapter (HCA) which provides a work queue abstraction, kernel bypass, and remote direct memory access (RDMA). When an application wants to send data, it posts a request on the HCA work queue. The request contains the address of the data and a length. The sending HCA takes care of fragmentation and reliable delivery. The receiving HCA takes care of reassembly. Kernel bypass means an application does not need to make a system call to post a request to or collect completion events from the work queue. RDMA means the HCA will place messages in their final destinations. The receiver does not waste time copying data.

III. Transport Operators

To allow streaming application writers to take advantage of technologies such as MPI or IB, we propose transport operators. An application writer provides custom transport as a sender operator and a receiver operator. In System S these operators would be implemented as SPADE user-defined operators. The sender receives tuples from an operator on its input port and sends them out as binary blobs to the receiver. The receiver converts the binary blobs into tuples and sends them to an operator on its output port. Suppose we want an operator A to communicate with an operator B as shown in Figure 1(a). To use custom transport operators, we first add a sender operator (S) and a receiver operator (R)



(a) Before (b) After

Figure 1. Adding transport operators to a source-sink application.

between A and B as shown in Figure 1(b). The dotted line between S and R represents the new transport connection. After adding S and R, we fuse A with S and B with R. Thus tuples are passed from A to S (and R to B) with function calls.

Another aspect of customizing transports is automating the process of inserting the sender and receiver operators into an application. Inserting these operators manually can be time consuming and error prone. For example, the OMDP application uses several dozen operators, and we have other streaming applications that use thousands of operators. We have developed a SPADE preprocessor tool to automatically add transport operators to applications. This tool parses the application specification into a syntax tree. When it detects that operator A's output port is connected to operator B's input port in a different partition, it breaks this connection and inserts a transport sender/receiver pair.

High-performance transports such as Vega, MPI, and RUM require spinning threads. While spinning threads ensure the best response time, they reduce CPU available to applications. When multiple output ports are connected to a single input port, the tool groups the connections to a single source operator to avoid creating multiple transport operators and unnecessary spinning threads.

Once the application has been augmented, we convert the syntax tree back to an application specification and submit it to the SPADE compiler.

In our first attempt to create custom transport operators, we tried to exploit MPI which runs well on many HPC platforms. However, we faced several challenges which we have not been able to completely overcome. First, we encountered difficulties with both MVAPICH2 and OpenMPI because System S and and MPI have their own job management mechanisms. The dynamic process management support in MPI should help resolve these differences, but unfortunately this feature is not yet well supported. We are working with the developers of OpenMPI to tackle the remaining challenges. Second, we also encountered issues with threading support that prevent us from reporting performance numbers for MPI over shared memory at this time; thus, we are only able to evaluate MPI in the internode case. Finally, the most efficient MPI primitives like *MPI_Recv* are not interruptible, which makes it difficult to write operators that shutdown properly on request.

Given our difficulties with MPI, we decided to explore a new option for writing transport operators using IB called

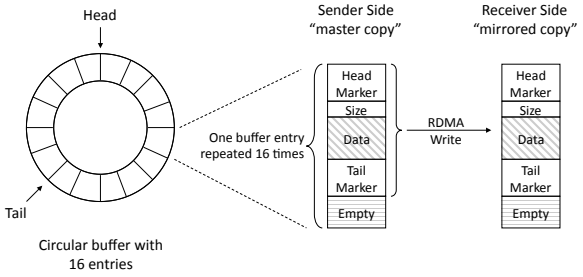


Figure 2. Circular buffer queues used by Vega

Vega. The next section provides a detailed discussion of this new transport.

IV. Vega Transport

Vega is our name for a low-overhead, high-throughput, lock-free transport mechanism that uses a unidirectional point-to-point communication design. To achieve the best performance, it avoids locks and memory copies. Vega borrows heavily from prior work on efficient RDMA based transports over IB by Liu [3].

The Vega API is minimalistic. It consists of five calls for data movement plus a few more for connection establishment. Since it uses point to point communication, we will refer to the sender and receiver as the connection endpoints. Most functions are inline for performance. The sender uses the following calls:

- *ready_to_send(sender)*: Check if Vega has room to send a new buffer.
- *put(sender, data, size)*: Save the data in the internal queue and send a copy to the receiver.

The receiver uses the following calls:

- *available(receiver)*: Check if Vega has new data available.
- *get(receiver, chunk, size)*: Obtain the next data buffer and return the *slot_id* of the next buffer.
- *release_buffer(receiver, slot_id)*: Release a buffer previously obtained by the *get* call. This allows Vega to reuse the buffer to receive new data.

To avoid kernel overhead, the API calls are non-blocking and there is no mechanism to wait for new data to arrive. This means the receiver must loop until data arrives. Similarly, the sender must call *ready_to_send* in a loop until there is space to send more data. Busy waiting will most likely occur but this is not unusual in high-performance transports; it exists in MPI and RUM as well.

Internally, Vega is implemented around a pair of circular buffer queues, one at the sender and one at the receiver, as shown in Figure 2. The receiver's queue is a mirror copy of the sender's queue. We use IB RDMA writes to push updates between the two queues. The queues are allocated entirely at the beginning of the execution.

When the sender posts new data to the queue, the data is first copied into the local buffer. A begin marker, the size, and an end marker are added. Then we issue an IB RDMA write to copy the data along with the markers and size in one operation. Since IB guarantees reliable, in-order delivery of data, we can rely on memory changes to detect the arrival of new data. The receiver simply polls the head of its own queue until the begin marker appears. It then uses the size of the buffer to locate the end marker and starts polling the memory for the arrival of the end marker. The arrival of both markers indicates that a complete buffer has been received. The sender picks the end of buffer marker appropriately to avoid conflicts with data previously sent to the same location. When the receiver releases the new buffer, it uses an IB RDMA write to signal the sender that the buffer is free. This signaling mechanism was first used by the IB transport of MPICH2 [3].

Over shared memory the protocol is almost identical except the IB calls are skipped. Instead the sender and receiver share the same queue via the Linux shared memory system. Any changes made by the sender is immediately visible to the receiver and vice versa.

Both the Vega sender and receiver transport operators expose the maximum message size and queue size to the System S application writer. The sender operator also exposes an additional parameter which controls whether calls to the Vega transport should be protected with a mutex. In System S, operators can be multi-threaded (especially user-defined operators) and when combined with fusion, this can result in multiple threads invoking the same operator code concurrently. It is safer to always use the mutex to protect the operator, and this is the default behavior.

At this point, Vega is a prototype and has several limitations compared to RUM and MPI. The Vega message size and queue size are not dynamic. While we were able to determine good sizes for the OMDP application, this required a good understanding of the application. Vega does not support fragmentation and reassembly of messages. The maximum message size must accommodate the largest message the application will generate. If very large messages are sent, Vega will suffer from the initial memory copy since it does not have a copy-free mechanism like the MPI Rendezvous protocol [21]. Finally, Vega lacks features such as multi-rail support that can be found in RUM and MPI.

V. Evaluation

In this section we evaluate Vega transport using microbenchmarks and a real-world financial application, for both an internode cluster-based scenario (using IB) as well as an intranode scenario (using shared memory). In the internode case, we compare Vega to MPI and RUM with microbenchmarks. Due to the difficulties with MPI which we discussed in Section III, we only compare Vega to RUM with the real-world financial application. In the intranode case, we compare Vega to operator fusion (or Fused), MPI, and RUM with microbenchmarks. Due to the poor performance of RUM, we only compare Vega to Fused and MPI with the real-world application. We treat the internode and intranode scenarios independently, and we do not attempt to make comparisons between them.

We chose the real-world financial application because its topology includes both fan-in and fan-out which is found in many streaming applications. In addition, the performance of the application is measured in terms of throughput (input load) and latency (processing time) which are important transport metrics. While the performance of the application can vary depending on the topology, the topology is fixed in order to focus on transport performance.

A. Benchmarks

1) Microbenchmarks: To measure latency across IB, we used a *sender-reflector-receiver* SPADE application. The sender and receiver are placed on the same node while the reflector is placed on a different node. The sender is configured to output tuples at a desired rate to the reflector which forwards each tuple to the receiver. Each tuple contains a time stamp. The receiver calculates the round-trip time for each tuple, and we report the one-way latency as half the round-trip time. To measure latency across shared memory, we dropped the reflector and report the one-way latency as calculated by the receiver.

To measure throughput across IB and shared memory, we used a *sender-receiver* SPADE application. The sender outputs tuples to the receiver as fast as possible. The receiver records the time between the arrival of the first and last tuples. After the last tuple, the receiver calculates the throughput by dividing the data volume it received (total number of tuples multiplied by the size of each tuple) with the elapsed time.

Both the latency and throughput microbenchmarks were run for ten seconds, and we used *rdtsc* to obtain time stamps. For all of our runs, we turned off all unnecessary OS daemons and pinned threads appropriately.

2) Options Market Data Processing Application: Figure 3 shows the logical view of our Options Market Data Processing (OMDP) application. Some number of feed senders multicast prerecorded OPRA packets to the feed handlers. OPRA is the Options Price Reporting Authority which consolidates options quote and trade information from all US options markets and deliver. Our OPRA data was collected on Monday, February 4, 2008. While just two feed senders are shown in the figure, we can use up to 24 feed senders, one for each OPRA channel.

Each feed handler is assigned to one OPRA channel. The feed handlers decode each packet and forward the resulting messages to appropriate decision engines. The number of decision engines is configurable. Each decision engine is assigned a group of stocks in a particular exchange.

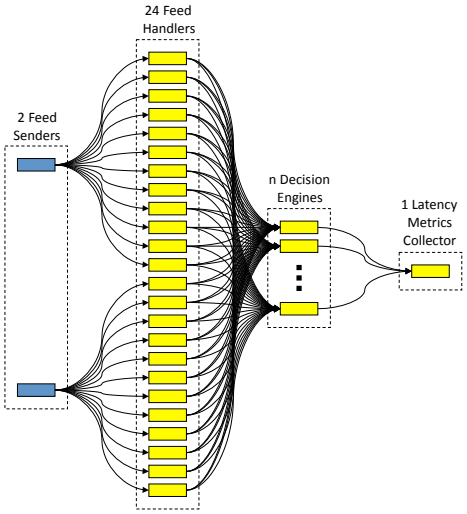


Figure 3. Logical view of the OMDP application.

This assignment is based on the symbol distribution in the recoded data. Each decision engine processes all received messages and forwards 1 out of 1,600 messages to the metrics collector.

Messages are time-stamped upon arrival and exit at each operator in the processing chain, and the time stamps are carried along with the message. The metrics collector in the third ply adds a final time-stamp, gathers the resulting time stamps and uses them to calculate operator processing times, time between operators, and end-to-end latencies. Additionally, there is an ingest metrics collector not shown in Figure 3 which receives data from each feed handler and calculates missed messages.

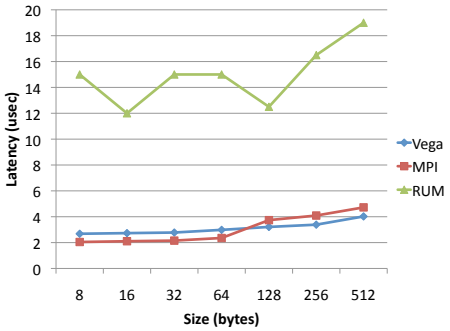
When the OMDP application is running on multiple machines, we use IBM Coordinated Cluster Time (CCT) to synchronize the clocks and to obtain time stamps [22], [23]. CCT provides synchronization within 1 μsec for machines connected via DDR IB. When the OMDP application is running on a single machine, we use *gettimeofday()* to obtain time stamps.

We collected 150,000 latency samples from the metrics collector, and used the middle 100,000 for our results. If the number of missed messages was greater than 0.01% for a run, we considered the application incapable of handling that message throughput. For all of our runs, we turned off all unnecessary OS daemons and pinned threads appropriately.

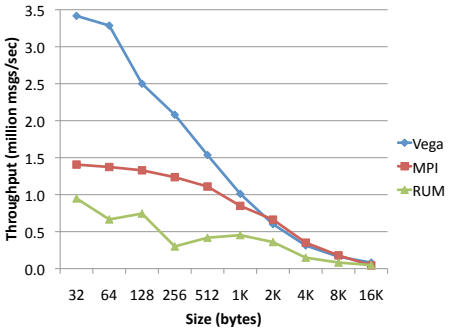
In our previous paper [1], we used multi-threaded feed handlers and decision engines to obtain the best performance. In this paper, the feed handler and decision engines are single threaded to highlight transport performance.

B. InfiniBand (Internode) Results

Our IB testbed consists of IBM HS21 blades running 64-bit Red Hat Enterprise Linux 5.3 [24]. The blades contain two Intel Xeon X5365 processors running at 3.0 GHz. Each



(a) Latency



(b) Throughput

Figure 4. Vega, MPI, and RUM performance over IB.

processor has 4 cores with an L1 cache of 32 KB and an L2 cache of 4 MB. Each blade contains 16 GB of RAM. All the blades are connected with 1 Gb/sec Ethernet and DDR IB which provides a theoretical throughput of 16 Gb/sec. The DDR switch is a Cisco SFS-7012 and the HCA are Mellanox ConnectX IB DDR MT25418.

1) Microbenchmarks: Figure 4(a) shows the latencies of Vega, MPI, and RUM for tuples between 8 and 512 bytes. We chose this range because OPRA messages are between 60 and 100 bytes long. While MPI is faster than Vega for tuples between 8 and 64 bytes, Vega is faster than MPI for tuples between 128 and 512 bytes. Vega is faster than RUM throughout the tuple range.

Figure 4(b) shows the throughput of Vega, MPI, and RUM for tuples between 32 bytes and 8 KB. Vega provides higher throughput than MPI for tuples between 32 bytes and 1 KB. In the 60 to 100 byte range, Vega is 1.9x to 2.4x faster than MPI. However, MPI provides higher throughput for tuples between 2 KB and 16 KB, Vega provides higher throughput than RUM throughout the tuple range.

While MPI provides higher throughput for tuples between 2 KB and 16 KB, we believe that Vega can provide higher throughput than MPI across the tuple range with further tuning. In the next section, we show the effect of Vega tuning on the OMDP application.

2) OMDP Application: We used a total of 39 HS21 blades for the OMDP application. 12 blades were used for

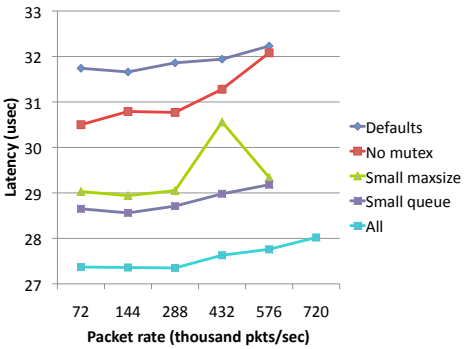


Figure 5. Vega over IB tuning.

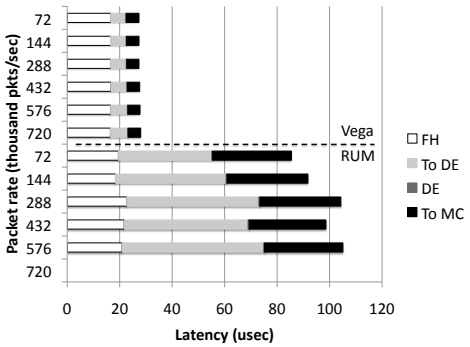


Figure 6. OMDP latency over IB with Vega and RUM.

24 feed senders. 12 blades were used for 24 feed handlers. 14 blades were used for 28 decision engines. 1 blade was used for the latency metrics collector and the ingest metrics collector.

First, we tuned Vega for the OMDP application. Figure 5 shows the latencies of the OMDP application with the default Vega settings when the aggregate feed sender rate is varied from 72,000 packets/sec to 720,000 packets/sec. The figure also shows the effect of removing the mutex protecting the HCA work queue, reducing the maximum message size to 256 bytes, reducing the queue size to 32, and finally the effect of all three combined. The default maximum message size is 64 KB, and the default queue size is 128. The combination of all three not only reduces the latency by about 15%, it also allows the application to process packets at a higher rate. The combination of all three allows the application to process up to 720,000 packets/sec versus 576,000 packets/sec. Without all three optimizations, the number of missed messages is greater than 0.01% at 720,000 packets/sec.

Figure 6 shows the latency breakdown of the OMDP application with Vega (tuned) and RUM. The overall latency with Vega is lower than the overall latency with RUM. With Vega the overall latency ranges from 27.4 μsec to 27.8 μsec. With RUM the overall latency ranges from 85.5 μsec to 105.1 μsec. Also, RUM is not able to support 720,000 packets/sec.

The overall latency consists of the time in the feed handlers ("FH"), the time between the feed handlers and the decision engines ("To DE"), the time in the decision engines ("DE"), and the time between the decision engines and the metrics collector ("To MC"). With both Vega and RUM, the OMDP application spends approximately the same amount of time computing in the feed handlers and the decision engines. However, with Vega the OMDP application spends much less time transporting data than with RUM. With Vega the OMDP application spends 5.7 μsec to 6.0 μsec in "To DE" and 4.7 μsec to 4.9 μsec in "To MC". This is less than 40% of the overall latency. In contrast, with RUM the OMDP application spends 35.6 μsec to 54.0 μsec in "To DE" and 29.3 μsec to 30.9 μsec in "To MC". This is greater than 75% of the overall latency.

C. Shared Memory (Intranode) Results

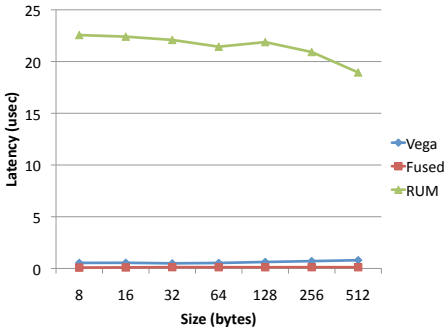
Our intranode testbed consists of an IBM x3950 X5 running 64-bit Red Hat Enterprise Linux 5.5 [24]. The IBM x3950 X5 is a 32-core NUMA machine with four Intel Xeon X7560 processors running at 2.26 GHz. Each processor has 8 cores, an L1 cache of 32 KB, an L2 cache of 256 KB, and an L3 cache of 24 MB. The machine has 256 GB of RAM. For the experiment we disabled DVFS (Dynamic Voltage/Frequency Scaling) to ensure that the processors were running at maximum frequency.[3]

1) Microbenchmarks: Figure 7(a) shows the one-way latency with a sender-receiver application for Vega, Fused, and RUM. In the Fused case, since the communication between sender and receiver is through function invocations, the latency is extremely small – about 0.1 μsec. With Vega, the latencies range from 0.5 μsec to 0.7 μsec, while the RUM latencies range from 18 μsec to 22 μsec.

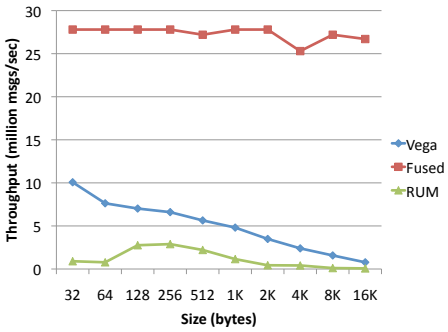
Figure 7(b) shows the throughput result for Vega, Fused, and RUM. With Fused, the throughput is quite high and ranges from 25.3 million messages/sec to 27.8 million messages/sec. With Vega the throughput ranges from 10.1 million messages/sec for 32 bytes tuples to 0.8 million messages/sec for 16 KB tuples. With RUM the throughput ranges from 2.8 millions messages/sec for 256 byte tuples to 70,000 messages/sec for 16 KB tuples. The fusion throughput is stable across the tuple sizes because there is no copying involved, whereas in the other cases the copy overhead causes a dropoff with tuple size.

2) OMDP Application: To fit the OMDP application on 32 cores available on our IBM x3950 X5, we reduced it (relative to the internode case) to 2 feed senders, 20 feed handlers, 8 decision engines, 1 latency metrics collector, and 1 ingest metrics collector. The Vega version has 30 separate processes, while the Fused version has a single

[3]We did not mention DVFS in our IB testbed discussion because DVFS is not available on HS21 blades.



(a) Latency



(b) Throughput

Figure 7. Vega, Fused, and RUM performance over shared memory.

process which contains the 20 feed handlers, 8 decisions engines, and the 2 metrics collectors. The single process has 20 threads, one for each feed handler. When a thread sends a tuple to a downstream operator (a decision engine in this case) it continues until it reaches a sink (a metrics collector). In all cases, the feed senders run as separate processes outside the application.

Figure 8 shows the latencies of the OMDP application with Vega and Fused when the aggregate feed sender packet rate is varied from 60,000 packets/sec to 600,000 packets/sec. While the microbenchmarks show that Fused is faster than Vega, the OMDP application latencies are smaller with Vega than Fused above 120,000 packets/sec. Most of this is due to the increase in the time spent in the feed handlers, which is caused by two effects. First, a single OPRA FAST [25] packet contains multiple messages, and second, after a packet is decoded all the resulting messages are processed in sequence by the single thread corresponding to that feed handler. While the first message is being processed, the subsequent messages are queued. Since the time stamp for all the messages in the packet reflect the arrival of the packet, the latency of the subsequent messages includes the latency of the earlier message in that packet.[4] Compared to this, the Vega configuration can exploit more

[4]The increase in the decision engine time is due to contention at mutexes protecting counters. However, this increase is much smaller compared to the feed handlers.

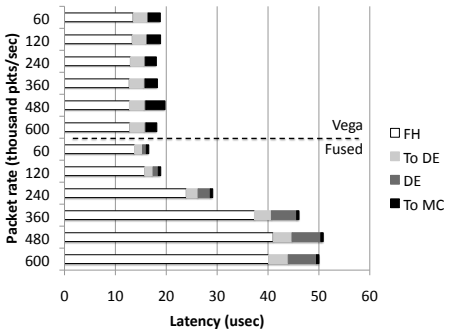


Figure 8. Latency over shared memory of OMDP application with Vega and Fused.

pipelining because the feed handler and decision engines execute in independent threads.

VI. Conclusions

In this paper, we proposed a method that provides more flexibility to System S application writers, by inserting custom transport as application-level operators. We demonstrated that this method can be used for leveraging an existing data transport such as MPI as well as a custom data transport such as Vega. While we were not able to fully implement MPI as a data transport, we feel that it will be possible in the future as MPI dynamic process management support matures.

We showed that Vega is a promising data transport for System S. Vega outperforms RUM significantly over IB and shared memory. Vega outperforms MPI for small tuples over IB. While Vega did not outperform operator fusion in the microbenchmarks, it provides an attractive alternative to operator fusion for applications running on multi-core machines, especially those applications that want to take advantage of the fault tolerance and security provided by separate address spaces.

In the future, we plan to mature Vega and extend our study to more transports that can be used in stream processing applications. We also plan on using the flexibility mechanism to dynamically fuse operators and make applications adaptive to resources at runtime.

Acknowledgments

The authors would like to thank Senthil Nathan for his help with the OMDP application and Wesley Most for his help with the experimental testbeds.

References

[1] X. J. Zhang, H. Andrade, B. Gedik, R. King, J. Morar, S. Nathan, Y. Park, R. Pavuluri, E. Pring, R. Schnier, P. Selo, M. Spicer, V. Uhlig, and C. Venkatramani, "Implementing a high-volume, low-latency market data processing system on commodity hardware using IBM middleware," in *Proceedings of the Workshop on High Performance Computational Finance (WHPCF)*, 2009.

[2] "Options Price Reporting Authority (OPRA)," http://www.opradata.com.

[3] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. K. P, D. Ashton, W. Gropp, D. Buntinas, and B. Toonen, "Design and implementation of MPICH2 over InfiniBand with RDMA support," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[4] H. Subramoni, P. Lai, M. Luo, and D. K. Panda, "RDMA over Ethernet - A preliminary study," in *IEEE International Conference on Cluster Computing (Cluster)*, 2009.

[5] S. Narravula, A. R. Mamidata, A. Vishnu, G. Santhanaraman, and D. K. Panda, "High performance MPI over iWARP," in *International Conference on Parallel Processing (ICPP)*, 2007.

[6] E. P. Mancini, G. Marsh, and D. K. Panda, "An MPI-stream hybrid programming model for computational clusters," in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2010.

[7] G. Jacques-Silva, J. Challenger, L. Degenaro, J. Giles, and R. Wagle, "Towards autonomic fault recovery in System S," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2007.

[8] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the Linear Road Benchmark on the Stream Processing Core," in *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[9] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, and K.-L. Wu, "SODA: An optimizing scheduler for large-scale stream-based distributed computer systems," in *ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2008.

[10] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, no. 1, 2003.

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Conference on Innovative Data Systems Research Conference (CIDR)*, 2005.

[12] "StreamBase Systems," http://www.streambase.com.

[13] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[14] A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu, "Infopipes: An abstraction for multimedia streaming," *Springer-Verlag Multimedia Systems*, vol. 8, no. 5, 2002.

[15] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soulé, and K.-L. Wu, "SPADE language specification," IBM, Tech. Rep. RC24830, 2009.

[16] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach to optimizing high-performance distributed data stream processing," in *ACM Conference on Information and Knowledge Management (CIKM)*, 2009.

[17] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.

[18] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI–The Complete Reference: Volume 2, The MPI-2 Extensions*. Cambridge, MA, USA: MIT Press, 1998.

[19] N.-B. C. Laboratory, "MVAPICH/MVAPICH2: MPI-1/MPI-2 for InfiniBand and iWARP with OpenFabrics," http://mvapich.cse.ohio-state.edu, 2008.

[20] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[21] S. Sur, H. W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand: Design alternatives and benefits," in *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[22] S. Froehlich, M. Hack, X. Meng, and L. Zhang, "Achieving precise coordinated cluster time in a cluster environment," in *IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, 2008.

[23] L. Zhang, Z. Liu, and C. H. Xia, "Clock synchronization algorithms for network measurements," in *IEEE Conference on Computer Communications (Infocom)*, 2002.

[24] "Red Hat Enterprise Linux 5," http://www.redhat.com/rhel.

[25] "FAST Protocol," http://www.fixprotocol.org/fast.

[26] "IBM Cell Broadband Engine," http://www-03.ibm.com/technology/cell.

[27] "InfiniBand Trade Association," http://www.infinibandta.org.

[28] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," in *ACM International Conference on Supercomputing (ICS)*, 2003.

[29] "IBM WebSphere MQ Low Latency Messaging," http://www-01.ibm.com/software/integration/wmq/llm.

[30] M. Schlansker, Nagabhushan, E. Oertli, P. M. S. Jr., L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi, "High-performance Ethernet-based communications for future multi-core processors," in *ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2007.