Understanding and Improving the Cost of Scaling Distributed Event Processing

Shoaib Akram, Manolis Marazakis, and Angelos Bilas†
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {shbakram,maraz,bilas}@ics.forth.gr

ABSTRACT

Building scalable back-end infrastructures for data-centric applications is becoming important. Applications used in data-centres have complex, multilayer software stacks and are required to scale to a large number of nodes. Today, there is increased interest in improving the efficiency of such software stacks. In this paper, we examine the efficiency of such a stack used for distributed stream processing, an important application domain. We use a specific streaming system, Borealis [10], and extensively hand-tune the end-to-end data path. We focus on parts of the stack that are related to intra- and inter-node communication and data exchange, a central component of many software stacks. We find that application-independent code in stream processing middleware employs operations for communication that consume significant amount of CPU cycles and are not strictly necessary. We first categorize these operations based on the protocol function they support. We then proceed to remove these operations by producing a functionally equivalent software stack in terms of application processing. Our results show that restructuring the data path achieves up to 5x higher throughput, reduces energy consumption by up to 60% and saves infrastructure cost by up to 40%. Finally, we project that with 1024-core processors per node, stream processing applications will demand up to 2 TBits/s/node of networking throughput.

Categories and Subject Descriptors

H.2.4 Systems [Query processing]

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Distributed Event Processing, Stream Processing Engines, Data-centric Infrastructures

DEBS .12, July 16–20, 2012, Berlin, Germany.

1. INTRODUCTION AND MOTIVATION

The emergence of "Big Data" [20] has provided modern enterprises with new opportunities to better serve and expand their market. However, to gain a competitive edge, enterprises need to rapidly process large amount of unstructured data in short periods of time. This requires new and powerful tools beyond traditional database systems [18]. Stream processing [18, 32] is an emerging paradigm that has the potential to process massive records of information in real time. With the trend towards building large-scale data-centres, the software stack of data-processing applications is attracting significant attention due to the inefficiencies that are being revealed in various application domains [23, 22].

In this paper, we examine the factors that influence the efficiency of middleware for scalable distributed stream processing. We examine and optimize an event-processing stack based on the Borealis streaming system [10]. Our initial experiments with Borealis show a high CPU-overhead for processing a single event. This implies inefficiency in terms of both energy and infrastructure [12, 27]. Modern distributed application stacks typically include the operating system (OS) that is heavily used for networking and storage I/O purposes, one or more middleware layers that provide a convenient, domain-specific abstraction, and the application itself that provides the desired service. In this work, we mainly examine the flow of data within and across nodes which is the main function of streaming systems. We propose alternatives and optimizations that are able to improve intra- and inter-node communication throughput. We focus on application-independent issues related to data communication in the end-to-end path between two nodes and we examine overheads both in the middleware and the OS. We find that the middleware performs operations that are not strictly necessary but result in significant overhead in performance. We first categorize these operations and then proceed to remove them and produce hand-tuned versions of the software stack.

We use the hand-tuned versions to quantify the impact of our optimizations. We evaluate their efficiency using metrics that are relevant to modern data-centric infrastructures. As a starting point, we define the CPU cycles spent for processing a single byte (or event) as a metric for characterizing application behavior [4]. We then use this metric to analyze how efficiently the processor and network resources are utilized, the energy required to process a given amount of data, and the server machines required to maintain a given streaming rate. Finally, we project future demands of streaming systems in terms of network throughput.

The main results of this paper are:

1. We show that restructuring the data flow within servers results in up to 5x improvement in throughput, 60% improvement in energy consumption, and 40% in infrastructure cost, reducing by half both the power and capital expenses of streaming platforms and infrastructures.

2. We find that user-level communication systems such as MX can improve throughput by 20% for small messages and reduce system time by 84% over an optimized TCP/IP based middleware stack. Thus, we expect that high-speed Ethernet-based interconnects running TCP/IP or similar protocols can serve well this domain of applications.

3. We find that the optimized streaming stack requires up to 2 GBits/s of network throughput per core, and thus, when the number of cores in server systems reaches the 1024-range, each server will demand up to 2 TBits/s of network bandwidth. This provides us specific guidelines for dimensioning server systems in the future.

The rest of this paper is organized as follows. Section 2 discusses relevant background related to stream processing and Borealis. Section 3 presents our categorization of operations for networking-related functions in Borealis, the benefits of their use and the limitations imposed by their elimination. Section 4 presents a methodology for evaluating the efficiency and projecting the future infrastructure demands of stream processing. We also describe our experimental platforms in the same section. Section 5 discusses our experimental results. We present related work in Section 6 and draw our conclusions in Section 7.

2. BACKGROUND

In this section, we provide the necessary background on stream processing engines and describe Borealis. We describe all layers related to exchanging data between nodes from application memory to the network interface.

Distributed stream processing is an upcoming and important class of applications for data centric computing [28]. In this class of applications, stream processing engines (SPEs) operate on streams of incoming information. Conceptually, a stream serves the same purpose as does a schema in traditional database systems. Each SPE is composed of generic, stream processing middleware, such as Infosphere [29] or Borealis [10] that runs on top of a traditional OS. Unlike traditional database systems, the queries in stream processing systems are defined statically while data is continuously (re)defined [18].

To process a specific data stream, the user needs to provide the application query that describes the required logic and the schema that describes the data streams. An example is to filter a data stream (online transactions performed by users) based upon the financial value of the transaction.

Each query consists of a set of operators that process tuples. Data streams merely consist of continuous tuples of information that conform to the application schema. Although tuples can contain arbitrary information, they typically always include a header with a monotonically increasing timestamp and additional fields that relate to the data being processed.

In single-node setups, all operators of a query are deployed on a single machine where all input streams need to be fed and where results are generated. In distributed setups, each operator of the query can be deployed on one or more machines, depending on the capabilities of the middleware system. In this case, each SPE runs only a subset of the query operators on all or a subset of the input streams. As a result, all SPEs implementing the query are connected in data-flow type graph, with each SPE receiving data from and forwarding results to pre-defined SPEs. The shape of the graph can either be determined by the users themselves or by the system, depending on the middleware capabilities.

Unlike traditional warehousing, stream processing systems in general operate on *windows* of incoming streams. Thus, although stream operators have similar functionality as typical database operators, e.g. filtering, sorting, unions, aggregation, their exact semantics differ to conform to deal with the fact that only a window of input data is visible at any point in time [1, 10]. Individual operators may require less or more CPU cycles per tuple, depending on the processing they perform.

For the rest of this work, the following aspects of stream processing systems are particularly important:

- First, all stream processing systems are required to scale to hundreds of nodes due to the increasing application requirements for data processing [33]. Note that different middleware systems may scale in different ways, depending on their ability to distribute individual query operators.

- Second, all stream processing systems use heavy internode communication when large amount of data is involved. Typically, streaming systems today use TCP/IP over Ethernet-type networks, although low-latency networks are starting to be considered as well [29].

- Third, the intrinsics of how operators are implemented in each system can be decoupled by the communication protocol stack used to communicate data across SPEs. This allows us to argue about overheads in protocol stacks for streaming systems without having to consider how individual operators work.

In the rest of this section, we describe the design of the Borealis [10] SPE that we use in our work.

2.1 The Borealis Streaming Middleware System

Borealis [10] is based on an event-driven architecture. The unit of transfer between two nodes is an event, shown in Figure 1. A single event represents a batch of one or more tuples. Each tuple consists of a number of fields. Alternatively, it is possible to define tuples that contain a pointer (as one of their fields) to an additional array of data that is stored within the event but outside the tuple itself. Typically, the fields of a schema that are used to execute query consist of a few bytes and are part of the tuple header, while fields that are simply passed to downstream nodes are larger arrays and are part of tuple arrays. This division helps with the memory management of large arrays of bytes.

Borealis is divided into four distinct tasks that are assigned to separate threads and execute asynchronously (Figure 2):

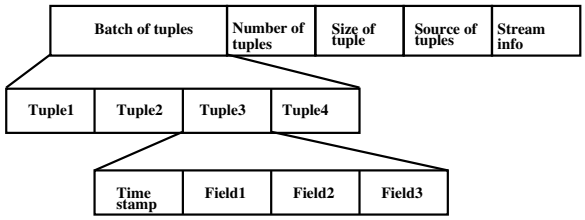


Figure 1: Event structure in Borealis.

The *receive* task is responsible for receiving raw data from the network, reconstructing events from the raw network data, extracting all tuples within an event, and enqueuing tuples to global data structures. Upon receiving data from the network into the application space, data is first appended to a read buffer. Events are extracted from the read buffer through a deserialization operation and a callback handles the event in the same context as the receive task. The handler of the event extracts the metadata from the event, enqueues the tuples into an input queue (stream), and stores the array data associated with the tuples in a large and statically assigned global buffer (array memory). The event handler also informs the process task of the availability of a new event.

The *process* task (also called aurora task) is responsible for dequeuing tuples from the stream and applying query operators to tuples. The process task dequeues tuples from the stream, evaluates the query on the tuples, and inserts the processed tuples into an output queue (tuple queue). In the current implementation of the process task, the array data associated with tuples is not touched at all. In essence, Borealis is scaling glue for aurora stream processing engine. In this paper, we are mainly interested in understanding the cost of scaling and we do not modify aurora. Our optimizations are thus relevant for multi-node deployments of streaming systems.

The *prepare* task is responsible for preparing events and pushing them to a list of pending events that are ready to be sent. This task continuously checks for the availability of processed tuples in the tuple queue. When tuples are available, a new event is constructed by collecting tuples from the tuple queue and the arrays associated with the tuples. The newly constructed event is placed in contiguous memory for the send task (described below). After creating the complete event structure, the event is inserted into a list of pending events ready to be sent over the network.

The *send* task is responsible for removing events from the list of ready events and attempting to send them to the next downstream SPE. This task dequeues events from the pending list of events, serializes them, and tries to send them out. If this is not possible, and the corresponding socket may block, the remaining event bytes are appended to a write buffer for sending at a later time. Note that events must be sent out in order, which imposes certain restrictions on how the ready event list and write buffer are processed. The bytes that are appended to the write buffer are sent in the context of the receive task. A common design practice for backend applications that use TCP sockets is to use the select() call to pick globally, for the full application, the event socket that has pending items. Instead in our work, we allow the send and receive contexts to separately monitor their sockets. We show that our improved design increases overall throughput of Borealis.

The four Borealis threads communicate with each other through shared memory within an SPE. The process task operates on individual tuples, whereas the rest of the tasks can operate on batches of tuples. Batching affects communication behavior significantly as it results in larger packet sizes. On the other hand, it can also affect response time for individual tuples as the tuples are buffered in the process of creating larger batches. Although it is possible to consider adaptive batching techniques, we merely assume a static batching scheme in this work for clarity purposes and present results for different batch sizes. Communication among nodes is done via standard socket send/receive calls. Finally, Borealis is implemented in a mix of C++ and Java and exhibits complex control flow, typical of backend processing applications.

In the next section, we describe our new communication stack for Borealis.

3. END-TO-END DATAPATH EFFICIENCY

In this section, we identify the most important sources of overhead in the communication protocol stack of Borealis and the functions to which they are associated. We then propose restructuring certain aspects of communication for efficiency purposes. We show the impact of several optimizations that eliminate certain overheads, resulting in a more efficient communication subsystem. Next, we discuss in more detail the main operations that occur in the communication protocol stack.

3.1 Inter-thread and Inter-node Flow Control

We see in Figure 2 that internally in each SPE, threads need to pass data from one to the next in a pipeline fashion. To avoid buffer overflows and dynamic memory allocation errors, there is a need for inter-thread flow control.

In its original form, Borealis does not perform any flow control and either assumes that allocated buffers never overflow or that eventually dynamic memory allocation can never return an error. Although this seems unconventional, it is not an uncommon practice in complex systems. Typically, buffer sizes are set to conservatively large values and dynamic memory allocation is assumed to always work, as virtual memory is backed up by swap space on disk. This approach can work reasonably well when some other system resource, e.g. network speed, throttles memory. With current technology trends, there are two problems: First, conservative values for buffers can waste valuable memory resources, especially on today's virtualized platforms. Second, it may not be possible to estimate these values properly as network speed increase, occasionally leading to errors during execution.

In this respect, we extend Borealis to support proper inter-thread flow control. The receive thread stops receiving events as soon as the (static) buffer allocated between the receive and process thread fills up. The receive thread will be restarted as soon as the process threads starts consuming tuples from the static buffer. The prepare thread removes tuples form the tuple queue and the static buffer, repackages them into events, and places the events in a list for sending. Thus, the prepare thread needs to dynamically allocate memory for events, which may fail. To eliminate this possibility, we modify the prepare process to not allocate memory

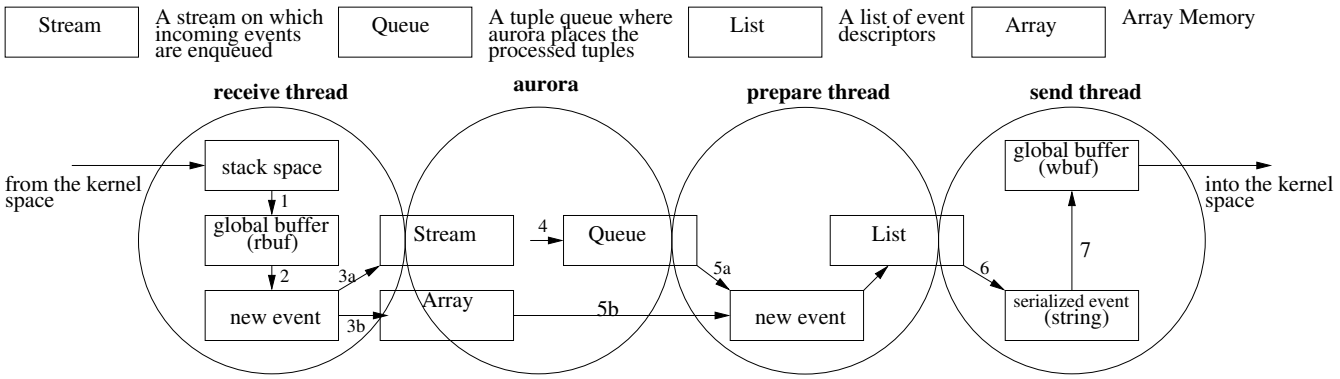


Figure 2: Main threads and data structures involved in end-to-end flow of events in Borealis. The data structures shared between threads are listed at the top.

for tuple data, but rather for event descriptors only, dramatically reducing the amount of memory required at this stage by at least one order of magnitude. In this approach event descriptors merely contain pointers to tuple data in the static buffer. Finally, the send thread removes event descriptors from the list, serializes events and tuple data and sends them to the next node. This approach closely follows the actual memory required by each SPE and is able to dramatically reduce memory requirements, as there is no need for conservatively estimating the size of the static buffer and the number of outstanding events in each node.

3.2 Buffer Management for Portability and Convenience

Typically, when preparing objects for sending them over the network, data is already placed in a buffer of a specific size that must be passed over to the sockets layer. Similarly, when a message arrives, data is already in a buffer and need to be passed to another part of the system. This handover could be between two different software modules or between two threads.

Data can be passed in two ways: (a) as a (buffer_ptr, size) pair that will require buffer management across threads (or modules) or (b) by creating a new buffer, copying the data and passing it to the event handler. Passing a new buffer has the advantage that it decouples buffer management across threads (or modules) and is the method used in Borealis. Borealis creates a self-contained object (string) that contains both the buffer and the size information. This transformation occurs in two places: (i) when an event needs to be sent or received over the network and (ii) when tuples are packed in events. Packing the buffer and its size in a new object has the advantage that buffer management is isolated across modules: buffer allocated by the process thread can be freed as soon as the event is created, whereas otherwise the buffer would need to be freed by the send thread, after the event has been sent.

We modify Borealis to implement an alternative approach, where objects (events) are exchanged in the form of (buffer_ptr, size) and buffer management happens across threads, removing all overheads related to packing and unpacking data from one buffer to the next.

3.3 Serialization and De-serialization for Heterogeneity and Portability

Serialization and deserialization has been discussed in previous research [17] and is typically necessary for two reasons: First, when communicating data across heterogeneous node (processor) architectures. In this case it is necessary to create an architecture-independent representation of the data. Then, each node participating in a communication operation needs to convert data back to the appropriate form for placing in memory and processing. The conversion of data to an architecture independent format may be necessary at the application-level but this is not required in many cases today, because back-end applications tend to run on processors of the same architectural family.

Second, serialization is used to pack the data structures that are spread out in non-contiguous areas of memory into a single buffer. This buffer can then be transmitted over a network. At the receiving side, the de-serialization process recreates the data structure by placing data in memory and reconstructing pointers. Alternatively one can pass pointers to the different fields of the data structure to the layer responsible for sending the events (send thread). Current trends in high-speed networking advocate this second approach, since the per-message cost is reducing.

We remove serialization and deserialization from Borealis, using separate send operations for each sub-object of an event and without converting events to an architecture-independent format. To achieve this, we modify the event structure to include the size of each sub-object and then use separate network send/recv operations to transmit/receive the event structure and its sub-objects. This introduces additional complexity for determining the size of an event at the receive side, which however can be dealt with appropriate placement of size and marker fields within the event structure.

3.4 Message Queuing for Asynchronous Operation

The original Borealis design assumes that the send thread might block on a slow network. To cope with this, first the prepare thread provides a descriptor of the newly created event to the send thread via a list of descriptors shared by both threads. The send thread tries to send each event and when this is not possible, it copies the rest of the event bytes in a buffer. When the network becomes idle, the send thread first transmits the contents of this buffer and then proceeds with subsequent events. This approach decouples

the prepare and send threads, allowing the prepare thread to drain all tuples from aurora asynchronously from the send thread.

This decoupling of the prepare thread from the network by using an additional thread can only help when adequate buffering is available to compensate for temporary backlogs in network performance. However, with high-speed networks, such as 10 GBits/s Ethernet, used in back-end infrastructures [13, 25], this asynchronous operation introduces significant overhead. Moreover, if the system is ultimately bound by (outgoing) network throughput, temporarily buffering outgoing events will not prevent processing threads from eventually blocking.

We modify Borealis to simply perform the send operation from the prepare thread, without the use of the intermediate buffer, and block when the network cannot service a specific send operation. From now on, we refer to the prepare thread as the send thread.

3.5 Avoiding Socket-based Network Communication

Most streaming systems today use TCP/IP sockets as the communication abstraction. Although TCP/IP is widely used in back-end applications, it introduces significant CPU overheads due to kernel and protocol processing, especially for smaller messages. Alternatively, user-level communication systems such as Infiniband [19] and the latest generations of Myrinet [9] can reduce the amount of cycles required per message as well as achieve higher throughput (and lower latency). Such networks are already being used today in communication intensive, back-end applications. We explore replacing TCP/IP with the native Myrinet communication protocol MyrinetMX [24].

We replace TCP/IP sockets with the Myrinet MX API over a myrinet network. The main issue to address is the need for application-level buffer management between the sender and the receiver, which originally was done by TCP/IP. The basic operation of an MX receiver is shown in Figure 3. During the initialization phase, the receiver allocates and posts a number of buffers where data could be received directly from the network interface controller (NIC). Later, the receiver can poll each location to find out if data has arrived in the buffer. If so, data is picked up from the buffer and the buffer is posted again as ready for receiving new data. Similarly, after preparing and placing data in a buffer, the sender marks the buffer as ready to be sent.

There are two issues when using the Myrinet MX communication protocol that need further consideration.

First, the sender can reuse a send buffer only after the DMA of the NIC has been performed and the data is stored on the NIC. This means that the sender needs to either (a) synchronously spin after each send for the DMA completion or (b) asynchronously proceed with a next send, but then, at a later time, check that the DMA has completed and reclaim the send buffer. Although the synchronous policy is typically used because it does not require send buffer management, it incurs spinning after each send operation. To further reduce the impact of streaming communication on CPU resources, we explore the asynchronous approach and quantify its benefits. We use a small send queue to hold events that are ready to be sent. This allows us to collect and prepare new events while the previous events are being sent to the next node. We will refer to this queue as a send-side queue in our results.

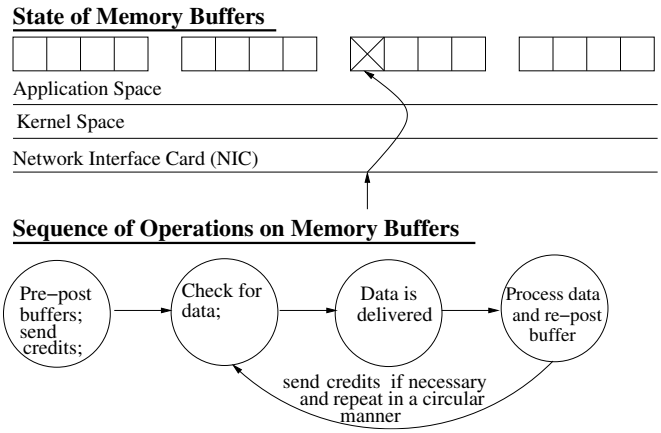


Figure 3: MX-based communication in Borealis. 'X' indicates that the buffer is holding valid data and is not available for re-use. Arrows indicate the sequence in which operations are performed.

Second, there is a need to deal with "unexpected" messages [24]. User-level communication systems operate without copies when receive buffers for messages have already been pre-posted. In case a message arrives at the receiver and there is no receive buffer specified by the application the system will deliver the message to a library-level, internal buffer and then, later copy the message to the application buffer, when this becomes known via a receive operation.

To allow Myrinet-MX to operate without data copies on the receive path, there is a need to ensure that receive buffers are pre-posted and arriving messages are always delivered directly to these application buffers. For this purpose we use an application-level flow control mechanism for buffer management purposes between the sender and the receiver. The receiver (Figure 3) pre-posts an agreed number of buffers for the sender. Then, the receiver updates the sender with new credits as it frees receive buffers after sending events to the process thread.

The number of pre-posted buffers depends on the relative speed of processing and communication in each streaming application. Although this number can be adjusted at runtime by the receiver [3], we only explore a static scheme. We over-provision the receive buffers, but still this is a small amount of memory compared to the rest of the SPE. An adaptive scheme for dynamically regulating the use of memory on the receive path, based on the relative speed of communication vs. processing in each SPE is an interesting problem for future work.

Overall, replacing TCP/IP with a user-level communication system removes the kernel crossings from the send and receive path, reduces the number of interrupts on the receive side, and removes three copy operations (one on the send and two on the receive side). We note that user-level communication requires custom NIC hardware support, which however, eventually might migrate to more commodity NICs.

3.6 Removing Spin Loops

Spin loops are often found in scalable middleware systems. In Borealis, we encounter two type of loops that hurt per-

formance. We qualify the spin loops as tight loops and loose loops. Tight loops repeatedly spin on the same variable. These loops are easy to identify and remove.

An example of a tight loop in Borealis is the prepare task waiting for tuples from the process task as it spins on the next location in the tuple queue (Figure 2). Unlike spinning on a variable in a tight loop, a loose loop sleeps for a pre-specified time, attempts to perform useful work by checking a variable, and blocks again. If it is unable to perform useful work and the time-out is zero, it results in a tight loop.

Although harder to identify, loose loops eventually hurt performance significantly. The most critical loops that hurt performance in original Borealis are discussed below.

In Figure 2, the process task (aurora) and the prepare task communicate via a tuple queue. When the batching factor is large, the thread responsible for making the event has to collect the complete batch from the tuple queue. In original Borealis, if the tuples are not available, the prepare thread spins until a new tuple is available. This seemingly unusual design is common among other applications, because, sleep-wakeup mechanisms are known to suffer from spurious wakeup(s) and in general do not work as precisely as expected. We convert the spin loop by using a sleep-wakeup mechanism. We also remove a spin loop from the send thread by using a blocking socket.

Next, the receive task has a loose loop. In original Borealis, sockets used for both receive and send tasks are monitored from a central location in the code (receive task). After blocking for a time-out specified in the select() system call, if data is available on the receive-side socket, a function call is made to collect the data (4 Kbytes). At the same time, if the send-side socket is able to send data, a separate function picks up data from the write buffer (Figure 2) and sends it to the network. After sending, the receive task monitors the sockets again for activity. Our decoupling of send and receive-side socket handling and our removal of serialization operation allows us to remove this loop. We use a blocking receive operation to receive the entire event rather than collecting smaller portions and buffering in a read buffer (Figure 2).

3.7 Summary

Table 1 summarize the above discussion by listing the most critical operations and their cost in terms of event size. We use the term "tuple" for the fields associated with the event and "data" as the array structure associated with the tuple. These operations are numbered in Figure 2. We describe the operation, analyze the cost of the operation and summarize the need for the operation in original Borealis.

4. EVALUATION METHODOLOGY AND EXPERIMENTAL PLATFORMS

In this section, we describe our evaluation methodology and experimental platforms.

4.1 Evaluating Performance

Our main goal is to understand the impact of restructuring the communication protocol stack of Borealis on performance. We measure performance in terms of tuples processed per second and the observed network throughput. We also quantify the impact of important parameters such as tuple size and batching factor. In addition we explore the following parameters:

- Number of Borealis instances per node: We perform experiments with different number of instances of Borealis on each machine. A different stream is associated with each instance of Borealis. There is a separate load-generator and receiver for each instance. Note that today a single instance of Borealis and similar systems can not fully utilize the entire bandwidth of a 10 Gigabit Ethernet NIC. Therefore, we believe that running multiple instances is a realistic mode of operation for streaming systems.
- Tuple size: We perform experiments for different tuple sizes. Although current applications of streaming systems tend to use smaller tuples, more advanced applications will demand larger tuple sizes.
- Batch (event) size: Batch size is an important parameter to better utilize the available network bandwidth. We will show results for a large range of event sizes to understand the full potential of our optimizations.

4.2 Evaluating Efficiency and Projecting Future Requirements

To capture the amount of work done by the application (and the OS) for each unit of data stored or communicated, we use the cycles per byte (CPB) and cycles per event (CPE) metric. We measure cycles by running the application and measuring the average execution time as reported by the OS and consisting of user, system, idle, and iowait time. We also measure the number of bytes and events transmitted and received during the same interval. We further introduce processing cycles per event (CPE_p), user cycles per event (CPE_u), system cycles per event (CPE_s), and idle cycles per event (CPE_i) using only the user, system or idle cycles in their calculation. Note that CPE_p is equal to the sum of CPE_u and CPE_s. We then use the following equations to calculate the network bandwidth required by an application on a system with N cores, and F as the frequency of each core :

$$GB/s = (\alpha.N.F)/CPB_p \quad (1)$$

$$Events/s = (\alpha.N.F)/CPE_p \quad (2)$$

α is the CPU utilization or the ratio of processing cycles to total physical cycles available on chip. We use an analytical model for estimating the energy consumption. The inputs to the model is CPE_p and CPE_i. We use the following model to calculate energy per event (EPE):

$$EPE = \{P * (1 * CPE_p + 0.7 * CPE_i)\}/(N * F) \quad (3)$$

We assume that under full utilization the machine draws peak power while during idle times, about 70% of peak power is used. Note that in Equation 3, we divide by $N*F$ to convert cycles to seconds and get Joules (or Watt*Seconds).

4.3 Configurations

We use three configurations that include different levels of optimizations:

- *original* is the original version of Borealis extended as follows: 1) We introduce flow-control as described in

Table 1: Summary of operations in Borealis during the flow of events within and across nodes.

No	Thread	Description	Cost of operation	Necessity
1	Receive	Data is transferred from the local stack space of receive thread to a global application buffer (rbuf).	Buffer management for convenience	sizeof(event)
2	Receive	Form a new event from the raw data using a deserialization operation.	Heterogeneity and portability	sizeof(event)
3	Receive	Transfer tuples and data to the input queue of the process thread.	Asynchronous operation of two threads	sizeof(tuples + data) * batching factor
4	Prepare	Create a new output tuple and enqueue to a structure called tuple queue.		sizeof(tuples) * batching factor
5	Prepare	Prepare outgoing event from tuples and data.	Contiguous memory allocation for serialization	sizeof(tuples + data) * batching factor
6	Send	Serialize event for sending.	Heterogeneity and portability	sizeof(event)
7	Send	Append serialized event to an output buffer (wbuf).	Message queuing for output buffer (wbuf) asynchronous operation	sizeof(event)

Section 2. 2) Since we are experimenting with a fast network, we removed the use of send thread along with the associated structures i.e., the list of pending events and the write buffer in the send path. 3) We remove the spin loops described in Section 3.6.

- *tcp-opt* is the original version with all optimizations described in Section 3 but still using TCP sockets as the communication abstraction.
- *mx-opt* is the original version with all optimizations including replacement of TCP sockets with Myrinet MX.

4.4 Experimental Platforms

We use two different experimental setups for evaluation purposes. Each of the test environments consist of four server-type systems. Setup-A consists of low-end servers with one Intel Xeon Quadcore chipset (X3220) and 8 GB of DRAM. Setup-B consists of high-end servers with two Intel Xeon Quadcore chipset (E5620) and 12 GB of DRAM. The servers in both setups have a 10 Gigabit Ethernet NIC from Myricom that is capable of operating both in TCP/IP and Myrinet MX mode. The four high-end servers of setup-B are physically connected via 10 GBits/s Ethernet HP ProCurve 3400cl switch and thus cannot run mx-opt. The first node in the pipeline runs a load-generator that generates a batch of tuples (events). The next two nodes in the pipeline run Borealis servers. The last node runs a sink that receives tuples and consumes them internally. To send and receive events using Myrinet MX, we use an interrupt-based approach which results in higher throughput compared to polling. Finally, the peak power (P) of machines in Setup-A when fully utilized is 200 Watts.

In our evaluations, we mainly use a simple pass-through filter graph. This is important because it shows the cost and inefficiency of the base infrastructure that is required for moving tuples and events. We use a query consisting of two filter operators in a chain. The parameters of the filter are set to pass each incoming tuple down the pipeline. Since a filter is the most light-weight operator in terms of processing overhead, this query is used to stress the network.

We report the network throughput in terms of tuples/s and GBits/s observed at the final node (receiver).

5. EXPERIMENTAL RESULTS

In this section, we discuss the performance and efficiency of the three configurations of Borealis using a range of parameters. We also project future requirements and quantify the overhead of query operator alone compared to the overhead of entire application.

5.1 Performance

Figure 4 and 5 shows the throughput of Borealis on Setup-A and Setup-B respectively. The index n prefixed with original and tcp-opt indicates the number of Borealis instances. On both Setup-A and Setup-B, tcp-opt achieves up to 50% improvement for small tuple sizes (128 bytes) and up to 200% for large tuple sizes (4096 bytes). mx-opt further improves throughput by up to 30% for small tuples and up to 34% for large tuples. We observe two trends as events become large. First, tcp-opt provides greater improvement because the high, per-event, overhead in original Borealis is eliminated. Second, as events become large and more tuples are packed in a single event, the improvement due to mx-opt is not significant because most available CPU cycles are consumed by Borealis for packing/unpacking tuples. However, for small event sizes, where there are spare CPU cycles, mx-opt is able to take advantage of them.

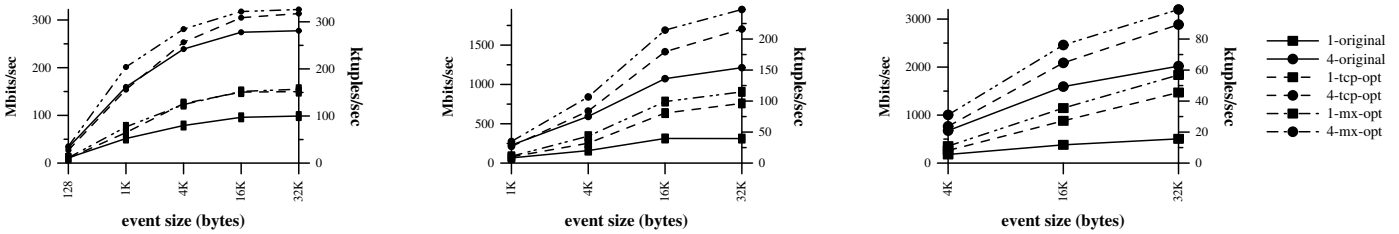
Finally, in Figure 6, we show the results for tcp-opt with and without the spin loops described in Section 3.6. The results show that for 8 instances of Borealis and using the tcp-opt configuration, there is a 44% improvement in throughput by removing the spin loops.

5.2 Efficiency

We analyze the efficiency of the three software stacks of Borealis in terms of resource utilization, infrastructure cost, and energy consumption. We also discuss the impact of deploying different operators on performance and infrastructure cost. All measurements in this section are reported for a single node in our four-node Setup-B running four instances of Borealis. We count both incoming and outgoing events to calculate various metrics. Unless mentioned otherwise, the reported results are for a filter operator.

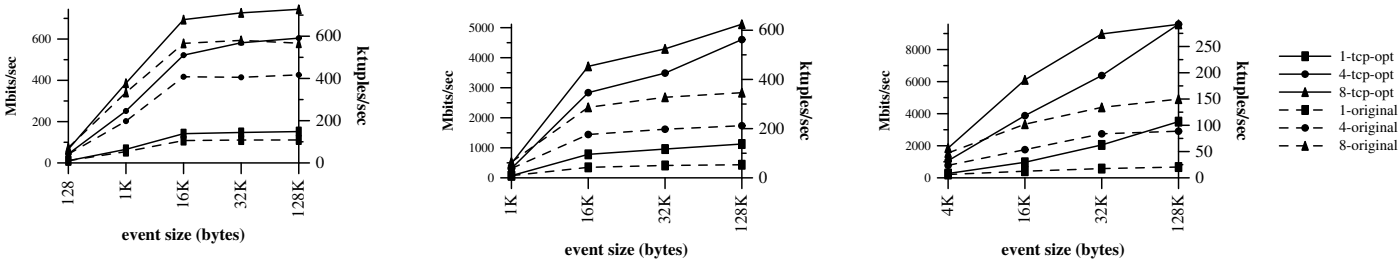
5.2.1 Resource Utilization

We are mainly interested in how efficiently the processor and the network is utilized. We observe that all three versions of Borealis have approximately the same processor utilization for the duration of the experiment. However, in terms of efficiency, tcp-opt consumes less CPU cycles for



(a) Tuple size=128 bytes (b) Tuple size=1 KB (c) Tuple size=4 KB

Figure 4: Network throughput of Borealis (in MBits/s and tuples/s) for different tuple and batch (event) sizes on Setup-A.



(a) Tuple size=128 bytes (b) Tuple size=1 KB (c) Tuple size=4 KB

Figure 5: Network throughput of Borealis (in MBits/s and tuples/s) for different tuple and batch (event) sizes on Setup-B.

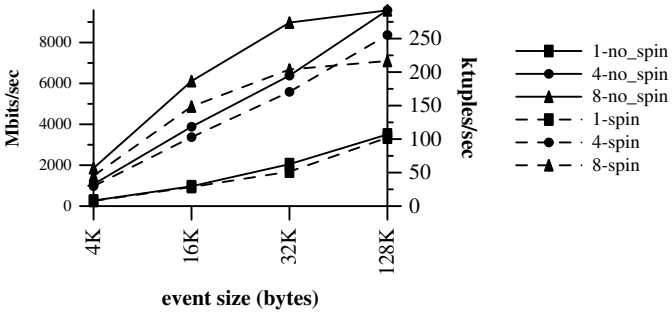


Figure 6: Throughput of tcp-opt with and without spin loops.

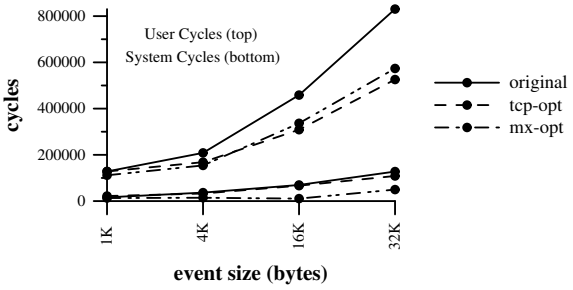
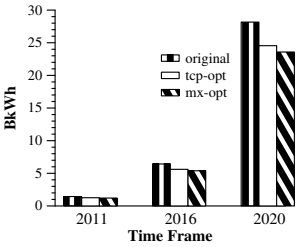


Figure 7: User cycles per event and system cycles per event for 4 KB tuples (4 instances).

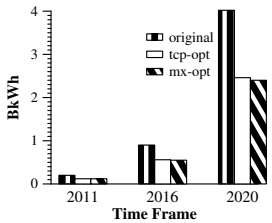
processing one event. In particular, CPE_u is reduced by up to 37%. Further, mx-opt reduces the cycles consumed by the OS by eliminating the kernel crossings; CPE_s is reduced by up to 84%. Figure 7 shows the CPE_u and CPE_s for 1 KB tuples and different batching factors. In terms of network utilization, we observe that tcp-opt and mx-opt utilize the network resources better compared to original for large events. In particular, we note that the maximum network throughput observed with the optimized stacks on Setup-A and Setup-B is 3 GBits/s and 8 GBits/s respectively as opposed to approximately 1.5 GBits/s and 4 GBits/s for original stack.

5.2.2 Energy Consumption

We now examine the energy consumption to process a given dataset using stream processing. We use projections for data growth from the Digital Universe study [11]. We calculate the energy required to process the entire data produced in years 2010, 2015, and 2020. The projected data is respectively 1.8, 8, and 35 Zeta Bytes. To calculate the energy consumption, we use Equation 3 to measure the energy for one event and assume that the entire dataset will use 32 KB events. Although not realistic for many-core processors of future, we use a per-core power consumption of 25 Watts to project trends based on today's consumption levels. We show results for both small and large tuples in Figure 8. We observe that, for large events, tcp-opt can reduce up to 61.18%, the energy required for stream processing. mx-opt provides only a 3% reduction in energy consumption compared to tcp-opt for large events. Note that, in the 2020 timeframe, the energy consumption using original to do a pass over all available data and using small tuples is approximately 30 billion kWh, which is 20% of the projected

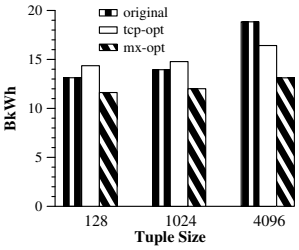


(a) Tuple size=128 bytes

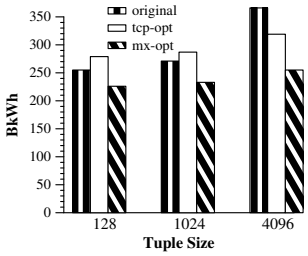


(b) Tuple size=4 KB

Figure 8: Billions of kWh to process all data produced in one year time frame using streaming systems.



(a) Year=2010



(b) Year=2020

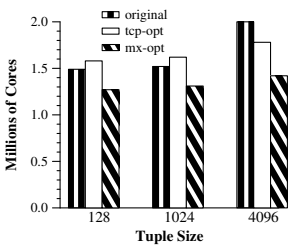
Figure 9: Benefits of mx-opt in terms of energy consumption for small events (batching factor=1).

energy consumption of all data-centres in the United States in 2020 [2].

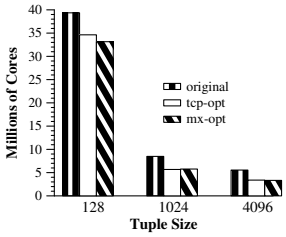
Figure 9 compares the energy consumption of the three configurations for small events. We show the energy consumption to process the entire data produced in year 2010 and 2020 for three tuple sizes. Each event consists of only one tuple. We observe that mx-opt can further reduce energy consumption by 20% for small events compared to tcp-opt. However, tcp-opt does not improve over original for small events because the per-event overhead in terms of the number of bytes communicated is greater for tcp-opt compared to original. The cost of communicating additional bytes is amortized only for large events.

5.2.3 Infrastructure Cost

We now show the implications of our optimizations for server infrastructure in data-centres that host streaming systems. At the rate at which the world produces new data [11], it will become imperative for streaming systems to deal with millions of events per second. We take the extreme case of an infrastructure required to sustain a rate of ten billion events per second at a global scale (counting both incoming and outgoing events). Figure 10 shows the number of cores required to stream ten billions events per second. We assume an event size of 32 KB and show results for three tuple sizes. We first use Equation 2 to calculate the events per second that a single node (and thus four cores) is able to stream today. We always assume 100% CPU utilization and thus fix α at 1. We also assume that doubling the cores will double the events streamed per second. We note that when streaming data at such high rates, tcp-opt can save from 12% to 40% in server infrastructure cost. In general,

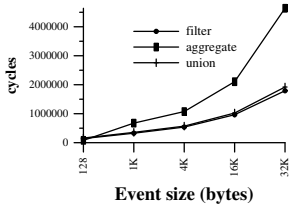


(a) Event size=32 KB

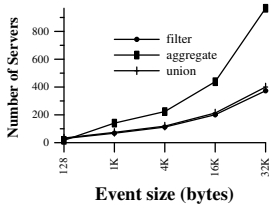


(b) Event size=Tuple size

Figure 10: Millions of cores to stream 10 billion events per second for small and large events.



(a) Cycles per event



(b) Server machines

Figure 11: Infrastructure overhead of three different operators.

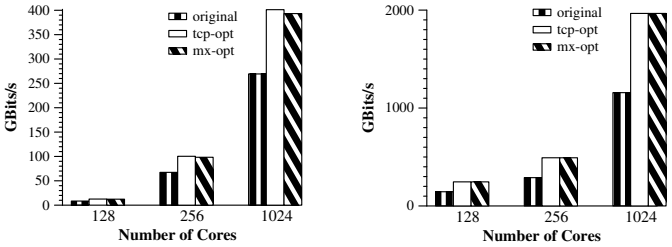
small tuple sizes require an order of magnitude more servers than larger tuple sizes.

5.2.4 Impact of Different Operators

Finally, we discuss the efficiency of different streaming operators. We use tuples of 12 bytes and run four instances of Borealis on each node using three operators: filter, union, and aggregate. The filter operator passes tuples coming from a stream based on the true evaluation of a predicate and drop the rest. The union operator merges two streams with the same schema into a single stream. We use the union operator to combine two streams. Aggregate is a stateful operator that computes a function over a window of tuples. We deploy the aggregate operator using a tuple schema with three fields namely event id, transaction id, and a timestamp. The aggregate collects four tuples in a window. The four tuples have the same transaction id and event ids are in an increasing order. The aggregate then verifies a pre-specified timing constraint on the arrival of four tuples. Figure 11(a) shows that CPE_p is the highest for the (stateful) aggregate operator while filter and union have the same CPE_p. In Figure 11(b), we show the number of 4-core server machines required to sustain a rate of one million events per second using various operators using the same methodology as in Section 5.2.3. We note that as events become large, the cost of deploying an aggregate operator grows very large compared to the other two operators: The aggregate operator requires up to 119% more servers compared to the filter operator.

5.3 Future Projections

With the trend towards increasing number of cores, there is an expectation that by 2020, processors with 1000s of cores will become available. We now quantify the network bandwidth that will be required by stream processing. In



(a) Tuple size=128 bytes (b) Tuple size=4 KB

Figure 12: Projection of network bandwidth required by streaming systems in the many-core era.

Figure 12, we use Equation 1 to show the network throughput demands required by a single server node, equipped with a many-core processor, consisting of 128, 256, or 1024 cores. We show the results for 32 KB events and for both small tuples (128 bytes) and large tuples (4096 bytes). Using small tuples will only require a few 10 GBits/s links. However, with 1000-core processors and large tuple sizes, the demands for networking throughput could go up to 1 TBits/s. We also note that the optimized stacks of Borealis will demand twice the network bandwidth of (2 TBits/s) per server node.

5.4 Operator Overhead

To calibrate communication overheads, we examine the impact of processing on overall throughput. We use a simple pass-through operator instead of the filter operator we use in our experiments. We find that, for most configurations, the overhead of filter operator is up to 30%. This implies that the overheads related to enqueuing and dequeuing of events still dominate the overall system overheads in Borealis.

6. RELATED WORK AND DISCUSSION

Data streaming has recently been gaining importance due to the large number of emerging applications that need to process data [18]. Research both in academia [10, 16, 6] and industry [29] aims at building scalable distributed stream processing systems. Efforts span the space from designing and implementing efficient relational operators for streaming databases [1], to proposing high-level query languages for specifying streaming workloads [15, 7], and to mapping different applications to streaming systems [31, 21, 1].

Less attention has so far been paid to understanding the performance implications of communication protocols and infrastructure for this communication-intensive class of applications. The authors in [29] present an evaluation of System S, a commercial data streaming system built by IBM. They discuss, at a high level, the impact of communication on streaming performance. In contrast, in our work, we not only quantify the performance of an existing stream processing system on a cluster consisting of modern server machines but also discuss a number of specific issues related to the communication protocol stack and related optimizations. We also evaluate, in detail, the impact of these aspects and show how future streaming systems can benefit from careful design.

In addition, there is a lack of standard benchmarks for evaluating streaming systems.

Recently, there has been increased interest in research on issues related to (in)efficiency of software stacks in data-centric applications. The authors in [22] examine trends in building data-centric applications from existing components that lead to large inefficiencies. In addition, recent work has been pointing out that inefficiencies in software stacks have an impact on the energy efficiency of data-centric infrastructures [5]. Our observations and results about the flow of data over the intra- and inter-node communication path in Borealis are in line with these trends. In our work we discuss these issues specifically for streaming systems, we quantify their impact and, propose optimizations and restructuring that mitigate many related costs. We show the detailed measurements required to understand the overheads introduced by providing such features as heterogeneity or communicating serialized objects in a specific class of data-intensive applications.

The dominant methodology so far on the power reduction side has been to construct power models by first taking measurements for a subset of the design space. Using this approach, there is strong evidence by [12] and [30] that directly relates the power consumption in clusters running typical data-centric workloads to the CPU utilization and physical memory usage. The measurements in [12] further suggest that CPU and physical memory are major contributors to power. The actual modeling techniques are developed and discussed extensively in [26]. In this work, we show that a given throughput could be achieved by running fewer instances of our optimized communication stack compared to our baseline. This implies reduction in CPU utilization and thus power for the same system throughput. We further reduce the memory utilization at various places in the application space that also leads directly to saving power. Thus, our results and analysis can be use further to deduce potential benefits in energy efficiency of backend infrastructures by improving software stacks.

Projection studies for data-centric infrastructures are usually done based on the growth rate of data or processing capabilities. We discuss a methodology for projecting networking and I/O demands of distributed applications in future based on real measurements. Various studies [19, 8] project networking throughput need in future applications. Our results indicate that streaming applications will require TBits/s throughput. Earlier prototypes have laid the roadmap towards TBits/s networks [14].

7. CONCLUSIONS

In this work, we examine the overheads associated with a modern software stack for processing data streams. We find that such stacks perform operations that are not related to the specific application or service they offer but consume significant amount of CPU cycles. We categorize these operations and quantify the combined impact of these operations on network throughput. Our optimizations improve network throughput by up to 5x. We also define and use new metrics for analyzing the efficiency of software stacks and projecting future requirements. Finally, results show that by carefully designing middleware systems, it is possible to reduce the overall number of nodes with benefits in performance, energy consumption, and management.

8. ACKNOWLEDGEMENTS

We thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs

through the STREAM (FP7-ICT-216181), HiPEAC2 (FP7-ICT-217068), IOLANES (FP7-ICT-248615) and the SCALUS (FP7-PEOPLE-ITN-2008-238808) projects. We are thankful to Stavros Passas for producing the 64-bit version of Borealis and to Vincenzo Massimiliano Gulisano at Polytechnic University of Madrid (UPM) for his help with running some of the experiments.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] U. E. P. Agency. Report to congress on server and data center energy efficiency.

[3] S. Akram and A. Bilas. A sleep-based communication mechanism to save processor utilization in distributed streaming systems. In *Proceedings of the Second Workshop on Computer Architecture and Operating System co-design*, CAOS'11, 2011.

[4] S. Akram, M. Marazakis, and A. Bilas. Understanding scalability and performance requirements of i/o intensive applications on future multicore servers. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS'12. IEEE, 2012.

[5] E. Anderson and J. Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44(1):40–45, 2010.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[7] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[8] A. Benner, P. Pepeljugoski, and R. Recio. A roadmap to 100g ethernet at the enterprise data center. *Communications Magazine, IEEE*, 45(11):10 –17, november 2007.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the 2003 CIDR Conference*, CIDR'03, 2003.

[11] EMC^2. Extracting value from chaos.

[12] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA'07.

[13] W.-c. Feng, J. G. Hurwitz, H. Newman, S. Ravot, R. L. Cottrell, O. Martin, F. Coccetti, C. Jin, X. D. Wei, and S. Low. Optimizing 10-gigabit ethernet for networks of workstations, clusters, and grids: A case study. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC'03, pages 50–, Washington, DC, USA, 2003. IEEE Computer Society.

[14] M. Galili, J. Xu, H. C. Mulvad, L. K. Oxenløwe, A. T. Clausen, P. Jeppesen, B. Luther-Davies, S. Madden, A. Rode, D.-Y. Choi, M. Pelusi, F. Luan, and B. J. Eggleton. Breakthrough switching speed with an all-optical chalcogenide glass chip: 640 gbit/s demultiplexing. *Opt. Express*, 17(4):2182–2187, Feb 2009.

[15] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD'08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[16] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *Proceedings of the 30th International Conference on Distributed Computing Systems*, ICDCS'10, pages 126–137. IEEE Computer Society, 2010.

[17] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic. Object serialization analysis and comparison in java and .net. *SIGPLAN Notes*, 38:44–54, Aug 2003.

[18] J. Hyde. Data in flight. *ACM Queue*, 7(11):20–26, 2009.

[19] InfiniBand Trade Association. Infiniband Architecture Specification, Version 1.0, Oct. 2000.

[20] A. Jacobs. The pathologies of big data. *Commun. ACM*, 52:36–44, Aug. 2009.

[21] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: optimizing stream processing applications via graph partitioning. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware'09, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[22] N. Mitchell. The big pileup. In *Proceedings of the International Symposium on Performance Analysis of Systems Software*, ISPASS'10, page 1, march 2010.

[23] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *SIGPLAN Notes*, 42:245–260, October 2007.

[24] Myrinet Inc. Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, Version 1.2., October 01, 2006.

[25] R. J. Recio. Server i/o networks past, present, and future. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, NICELI '03, pages 163–178, New York, NY, USA, 2003. ACM.

[26] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A comparison of high-level full-system power models. In *Proceedings of the conference on Power aware computing and systems*, HotPower'08, pages 3–3, Berkeley, CA, USA, 2008. USENIX Association.

[27] S. Rivoire, M. Shah, P. Ranganatban, C. Kozyrakis, and J. Meza. Models and metrics to enable energy-efficiency optimizations. *Computer*, 40(12):39 –48, dec. 2007.

[28] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[29] T. Suzumura, T. Yasue, and T. Onodera. Scalable performance of system s for extract-transform-load processing. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR'10, pages 1–14, New York, NY, USA, 2010. ACM.

[30] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. Worth their watts? - an empirical study of datacenter servers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, HPCA'10, pages 1 –10, Jan. 2010.

[31] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference*, pages 306–325, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] A. Wright. Data streaming 2.0. *Commun. ACM*, 53:13–14, Apr. 2010.

[33] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB'06, pages 775–786. VLDB Endowment, 2006.