# A Sleep-based Communication Mechanism to Save Processor Utilization in Distributed Streaming Systems

Shoaib Akram 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,bilas}@ics.forth.gr

**Abstract.** Energy-efficiency of applications deployed in data-centers is becoming increasingly important. Techniques that reduce CPU utilization for specific workloads can help improve energy consumption. An application domain that has been studied in the past extensively and is lately gaining importance in data-centers is distributed stream processing. In this work we examine an existing stream processing system, Borealis [6], and we identify significant sources of overhead in the communication stack. Specifically, we examine the inter-node communication path in a distributed setup and the overheads associated as streams flow from node to node. We find that the send and receive tasks in Borealis take up significant CPU resources. We redesign the send and receive paths of Borealis by replacing TCP with a user-level protocol based on Myrinet MX. We then evaluate the CPU utilization and network throughput on a 10 Gbits/s network using both polling and interrupts for communicating data. Finally, we propose a sleep mechanism that avoids the CPU overheads associated with both interrupts and polling. We use a real setup consisting of four eight-core nodes equipped with 10 Gbits/s Ethernet and native Myrinet MX communication subsystems to examine the impact of our approach. Our results show that our approach saves CPU utilization for a range of workload conditions and is able to achieve better throughput compared to TCP with lower CPU utilization (upto 40%).

## 1 Introduction

Modern data-center applications tend to employ complex software stacks that are processor hungry. Recent work [7, 18] shows that CPU utilization can be directly correlated to total system energy consumption.

Stream processing is a workload that has been studied extensively in the past and has recently been gaining attention due the data-oriented nature of many

†Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR-71409, Greece.

modern applications. In this work we examine the impact of the communication protocol on CPU utilization of distributed stream processing applications over 10 Gbits/s networks. We use an existing stream processing system, Borealis [6], to investigate how improved communication mechanisms can reduce CPU utilization.

Traditionally, in communication-based applications, senders and receivers need to detect the arrival of new (data or control) messages. Two approaches to detecting these events are using interrupts or polling [17]. Neither of these approaches is satisfactory. On one hand, interrupts incur high overhead at high-network speeds and large packet rates. Polling, on the other hand, wastes CPU at low packet rates. In addition, previous work has also examined adaptive techniques that switch between interrupts and polling [13].

An alternative approach is to release the CPU by sleeping for a specific amount of time when required events are still pending. Compared to interrupts, releasing the CPU can be implemented synchronously thus avoiding overheads related to asynchronous events. However, this approach can result in waking up long before or after the event is due. Compared to polling, sleeping can release the CPU to other threads but results in a system call. Ultimately, sleeping has the potential to adjust CPU utilization to the required processing rate. The main obstacle is regulating the sleep interval.

One approach to address this problem is to try and predict the amount of sleep that is appropriate at any given point. However, in modern operating systems it is not possible to exactly regulate sleep time at fine grain, because it depends on a number of coarse grain events in the operating system kernel. Thus, a more robust approach is to use a notion of doing work in "waves". Consider two nodes connected in a pipeline fashion with the first node being the sender and the second node being the receiver. The receiver informs the sender of the available memory buffers that it has reserved for receiving data and then checks the buffers for the arrival of data. If the test fails, the receiver sleeps for a fixed amount of time. In essence, the receiver accumulates work while taking no extra CPU cycles. Similarly, the sender distributes the work and then waits for a message from the receiver. This message informs the sender if the receiver is ready to receive more data. If the test for the message fails, the sender sleeps for a fixed amount of time. Thus, the sender accumulates work for the receiver, which will process this during the next "wave". The relationship between network throughput and processing rate determines the exact shape of these "waves".

Normally in applications such as borealis, there is a separate task that process the incoming data. Thus one way to look at our approach is that the sleep gives the opportunity to conserve energy by not consuming excess energy polling or processing interrupts. This more of the available CPU time is spent doing useful work.

Implementing such a "wave" approach requires dealing with three issues. First, we need to convert blocking send and receive operations to non-blocking combined with a sleep operation. Second, we need to introduce an appropriate

buffering mechanism at each of these points to allow the rest of the system to operate during the sleep. Finally, we need to ensure that the amount of buffering available at any point is appropriate for tolerating the inaccuracy of the corresponding sleep operation.

We investigate the impact of this approach on Borealis. The original version of Borealis uses TCP/IP for inter-node communication. It is possible to evaluate our sleep-based approach with a communication stack based on TCP. However, we port the communication stack of borealis to user-level Myrinet MX for two reasons. First, as we will show in Section 3, user-level communication protocols result in higher throughput compared to TCP/IP. This helps to accomodate the drop in throughput due to the difficulty of implementing a precise sleep interval for an entire range of system parameters. Further, a user-level communication protocol allows a fine-grained control of buffering resources in the send and receive paths since all the memory buffers for sending and receiving are allocated at the user-layer.

In this work, we create a version of Borealis that replaces kernel-based TCP/IP with user-level Myrinet MX communication [5]. Our modified version of Borealis uses a more light-weight communication mechanism and is able to achieve higher throughput than the original, TCP/IP-based version. We implement all three approaches (interrupts, polling, sleeping) in this version of Borealis and examine the impact on CPU utilization. We also compare the results with the original Borealis with TCP. We use a setup of four, eight-core systems (with each one being 2-way hyperthreaded) connected with a 10 Gbits/s Myrinet network using simple streaming workloads.

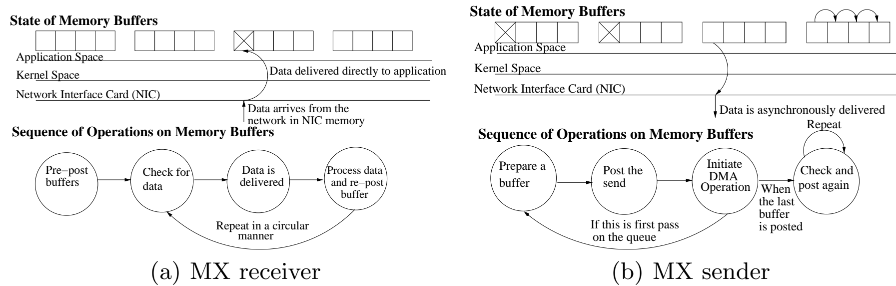
The rest of this paper is organized as follows. Section 2 presents our design of the new communication subsystem of Borealis using Myrinet MX and the new sleeping mechanism that saves CPU utilization for a range of workload conditions. Section 3 presents our experimental platform and evaluation methodology and discusses our experimental results. In Section 4 we present related work. Finally, we draw our conclusions in Section 5.

## 2 Communication Subsystem Design

In this section, we describe our modifications to the communication subsystem of Borealis and our proposed algorithm for sleeping in the send and receive threads.

### 2.1 Communication using Myrinet MX

Figure 1 shows our implementation of Borealis using Myrinet MX. 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 reception of new data. Similarly, the sender operates on a queue of buffers. First, the sender fills up and post all buffers of the queue. Then, the




(a) MX receiver

(b) MX sender

**Fig. 1.** 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.

sender scans the queue one by one, checks for the completion of DMA and fills up and posts the buffer for sending data.

In Myrinet MX there is a need to deal with "unexpected" messages [15]. Myrinet MX, similar to other user-level communication systems, operates 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. Later, when the message is successfully detected via a receive operation, it is copied to the application buffer. To ensure that Myrinet MX will operate without data copies in 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 1(a)) pre-posts an agreed number of buffers for sender to send events. Then, the receiver updates the sender with new credits as it frees receive buffers after sending events to another thread that processes these events.

### 2.2 Sleep-based communication algorithms

We use non-blocking send and receive Myrinet MX calls combined with sleep system calls to replace blocking send and receive calls. This in turn requires introducing buffering at certain points in the path to ensure that other parts of the system continue processing when a specific thread sleeps. Figure 1 and Algorithms 1 show our receive and send paths.

In the receive path, we check each buffer for arrival of data. If the test fails, the receive thread sleeps for a fixed amount of time. In the send path we check for completion of a DMA operation on a buffer posted earlier. If the operation is not complete yet, the send thread sleeps for a fixed amount of time. Also, before posting a send buffer, the sender checks to see if credits are available for sending data. If there are no credits to send data, the send thread sleeps for a

**Algorithm 1** Receive (left) and send (right) path with sleeping enabled.

n=size of circular queue that holds incoming or outgoing events
f=frequency with which to send credits to upstream node
t=time interval for which to sleep
m=number of credits to send to the upstream node

**Receiver:**

```
// initialize receive buffers and credits
post_buffers(n)
send_credits(n);
i = 0
// wait until next receive buffer contains
new event
while TRUE do
   while poll_buffer(i) do
      sleep(t)
   end while
   process_event(i)
   i = (i + 1)%n
   // if above threshold replenish sender
   if (i%f) = (f − 1) then
      send_credits(m)
   end if
end while
```

**Sender:**

```
credits=waitfor_credits()
i = 0
while TRUE do
   //wait for next send buffer
   while poll_buffer(i) do
      sleep(t)
   end while
   prepare_event(i)
   //replenish in case you run out
   credits += collect_credits()
   while credits = 0 do
      sleep(t)
      credits += collect_credits()
   end while
   send_event(i)
   i = (i + 1)%n
end while
```

fixed amount of time. The sleep in the receive thread and the sleep when there are no credits available are related and work together to reduce CPU utilization without reducing overall throughput.

The rational behind sleeping in the receive thread is to accumulate work while consuming no CPU cycles. While the receive thread sleeps it has already posted buffers for receiving data. When the receiver wakes up, it takes some time to process buffers and re-post them. Therefore, the credits to receive more data are sent to the upstream node after some time which depends upon the frequency with which credits are sent. The send thread of the node upstream sleeps during this time interval to conserve energy. If the sleeping intervals are approximately correct, when the send thread wakes up, it will receive credits to send data to the downstream node.

The minimum sleep time in our systems is 2 ms. However, we use a sleep interval of 10 ms in our sleeping algorithms. We experimentally find this to be an appropriate time interval for our specific systems. Note that the sleep interval can change based upon the system load. An algorithm to adjust sleep interval according to system load is left as future work.

## 3 Experimental results

In this section, we describe our evaluation methodology, the experimental platform, and our experimental results.

### 3.1 Experimental platform and methodology

Our test environment consists of four, server-type systems running the Linux operating system (CentOS release 5.4). Each system has two Intel Xeon Quad-core chipsets. The cores have two-way hyper-threaded capability. Therefore, each machine can potentially execute sixteen threads simultaneously. Each machine has 14 Gbytes of DRAM physical memory and a 10 Gbits/s Ethernet NIC from Myricom that is capable of operating both in TCP/IP and Myrinet MX mode. The four machines are physically connected via 10 Gbits/s Ethernet HP ProCurve 3400cl switch. 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. The last node runs a light-weight receiver that receives tuples and consumes them internally without storing them.

We use an in-house micro-benchmark that consists of two filter operators in a chain. The parameters of the filter are set to pass each incoming tuple down the pipeline.

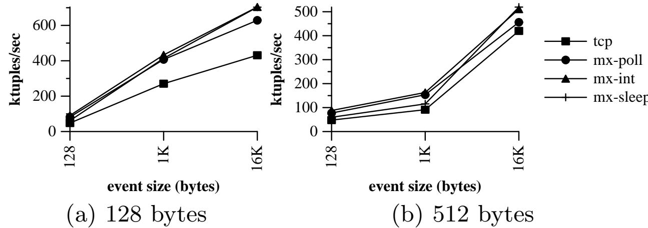
Our main goal is to understand the impact on performance and CPU utilization of using spinning, interrupts, and sleeping in the send and receive threads of Borealis with Myrinet MX API vs. TCP/IP. We build and use the four configurations of Borealis, *tcp, mx-poll, mx-int, mx-sleep*, that use TCP, Myrinet MX with interrupts, polling, and our sleeping mechanism, respectively.

We perform experiments with 8 instances of Borealis running on each node. A different stream is associated with each instance of Borealis. There is a separate load-generator (source) and receiver (sink) for each instance. A given instance of Borealis can not utilize the entire bandwidth of 10 Gbits/s Ethernet because processing time becomes the limiting factor. For this reason, we use multiple instances of Borealis on each node to exploit the maximum bandwidth available from the underlying 10 Gbits/s network. We believe that this is a realistic mode of operation for data streaming systems. Finally, we show results for different tuple sizes and different batching factors.

We perform experiments for different tuple sizes and event sizes. Note that a larger event size is obtained by using a large batching factor. We use a buffer queue of 100 entries on the receive side and send credits to the nodes upstream after processing every 10 buffers. The size of queue on the send side is 10 entries. The sender posts a send operation for receiving credits when 5 credits are left.

### 3.2 Impact on Network Throughput

Figure 2 shows the network throughput of Borealis for TCP and Myrinet MX (interrupts, polling, and sleeping). In all our experiments, the processing thread is the bottleneck. For this reason the use of polling when running 8 instances of Borealis results in a lower throughput compared to that of interrupts. Using Myrinet MX with interrupts instead of TCP improves the throughput of Borealis by upto 22% for a tuple size of 512 bytes. Further note that mx-sleep always results in better throughput compared to TCP between $23 - 63\%$.




(a) 128 bytes (b) 512 bytes

**Fig. 2.** Network throughput of Borealis in tuples/s for TCP/IP and Myrinet MX for different tuple and batch (event) sizes running a filter query.

### 3.3 Impact on CPU Utilization

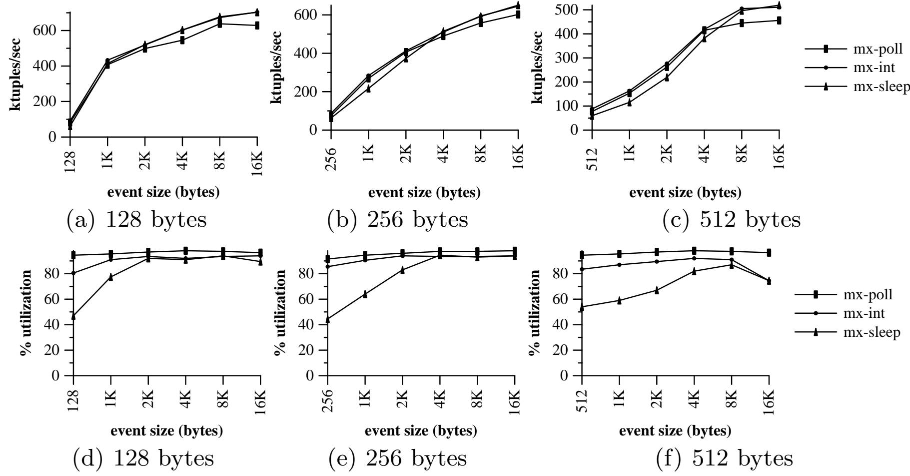
Figure 3(d-e) shows the CPU utilization of Borealis for Myrinet MX (polling, interrupts, and sleeping). CPU utilization is reported as the average utilization across the two nodes that run Borealis. A utilization of 100% in Figure 3(d-e) means that all 16 hardware threads in both nodes are fully utilized. We note that in all cases CPU utilization exceeds 90% for both TCP (not shown) and Myrinet MX with polling. Using Myrinet MX with interrupts reduces the CPU utilization slightly. Note that at small event sizes, the receive thread does not have enough work to perform as it receives fewer tuples per event, quickly enqueues them, and waits for the next batch. Therefore, mx-sleep results in significant reduction in CPU utilization compared to both mx-poll and mx-int. For an event size of 1024 bytes ( batching factor of 8), mx-sleep approaches exactly the throughput of mx-int, with saving in CPU utilization upto 15% for a tuple size of 128. Note that since data streaming is an upcoming application, the typical range of parameters such as batching factor is not obvious. However, since our sleep-based mechanisms do not result in significant loss in throughput compared to mx-int, we believe it to be a reasonable approach.

Next, we note that we do not observe any saving in CPU utilization beyond event sizes of 1024 bytes, 2048 bytes and 4096 bytes respectively for tuple size of 128 bytes, 256 bytes and 512 bytes. This is because events larger than these result in a large batching factor. A large batching factor implies more tuples in a single event and thus high overhead relating to enqueuing of tuples compared to communication overhead of sending or receiving a single event. Therefore, when the receive thread enqueues one event, it finds the next event readily available.

In summary, an implementation based upon Myrinet MX with sleep-based send and receive paths, we are able to achieve better throughput compared to TCP and reduce CPU utilization upto 40%.

## 4 Related Work

Data streaming has recently been gaining importance due to the large number of existing and emerging applications that need to process data [10]. Research both in academia [6, 9, 3] and industry [16] aims at building scalable distributed




(a) 128 bytes (b) 256 bytes (c) 512 bytes

(d) 128 bytes (e) 256 bytes (f) 512 bytes

**Fig. 3.** Network throughput in tuples/s and % utilization for different tuple and batch (event) sizes for a filter query.

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 [8, 4], and to mapping different applications to streaming systems [20, 11, 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 [16] presents an evaluation of SystemS, a 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.

Recently there has also been increased interest in research on issues related to (in)efficiency of software stacks in data center applications. The authors in [14] examine trends in building data center applications from existing components that lead to large inefficiencies. In addition, recent work has been pointing out that software stack inefficiencies have an impact on the energy efficiency of data center infrastructures [2, 12]. These approaches propose architectural techniques to adaptively manage power subject to changing workload conditions or policies above the hardware layer to exploit techniques and have pointed out inefficiencies in the software stack of existing middleware systems.

The authors in [19] have proposed hardware mechanisms to avoid OS spin overheads. Their techniques addresses the problem of extra spin overhead in over-committed virtual machines running operating systems that do not use

gang scheduling. In this paper, we quantify the negative impact of using spinning in the send and receive paths of the communication stack of Borealis when the system and underlying network is heavily-loaded.

## 5 Conclusions

In this work we examine how the communication stack of a distributed streaming system, Borealis, can use a sleep-based mechanism to avoid both interrupt and polling overheads. We identify subtle reasons for excessive CPU utilization in this class of applications and propose mechanisms to improve the efficiency of individual nodes. We propose a sleeping mechanism that saves CPU utilization for a range of workload conditions. In comparison with TCP, using Myrinet MX and our sleep-based send and receive operations achieves better throughput and reduces CPU utilization upto 40% for a range of parameters. We expect that the importance of our approach will increase as systems become more heterogeneous by combining networks and processors of different speeds in consolidated data center environments that execute multiple distributed applications at the same time..

## 6 Acknowledgments

We would like to thank Manolis Marazakis for his help with designing and implementing the base Myrinet MX version of Borealis. We thankfully acknowledge the support of the European Commission under the 6th and 7th Framework Programs through the STREAM (FP7-ICT-216181), HiPEAC (NoE-004408), HiPEAC2 (FP7-ICT-217068), and SCALUS (FP7-PEOPLE-ITN-2008-238808) projects. We thank Jim Holt for useful feedback on an initial version of this paper.

## References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

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

[3] 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.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[5] 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, February 1995.

[6] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur etintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003.

[7] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 13–23, New York, NY, USA, 2007. ACM.

[8] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[9] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Patrick Valduriez. Streamcloud: A large scale data streaming system. In *ICDCS*, pages 126–137. IEEE Computer Society, 2010.

[10] Julian Hyde. Data in flight. *Queue*, 7(11):20–26, 2009.

[11] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: optimizing stream processing applications via graph partitioning. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[12] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, 2010.

[13] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling watchdog: combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 179–188, New York, NY, USA, 1996. ACM.

[14] N. Mitchell. The big pileup. pages 1 –1, mar. 2010.

[15] Myrinet Inc. Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, Version 1.2. http://www.myri.com/scs/MX/doc/mx.pdf, October 01, 2006.

[16] Toyotaro Suzumura, Toshiaki Yasue, and Tamiya Onodera. Scalable performance of system s for extract-transform-load processing. In *SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–14, New York, NY, USA, 2010. ACM.

[17] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[18] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. Worth their watts? - an empirical study of datacenter servers. pages 1 –10, jan. 2010.

[19] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 124–133, New York, NY, USA, 2006. ACM.

[20] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa 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.