Tutorial: Stream Processing Optimizations

Scott Schneider
IBM Watson Research Center,
Yorktown Heights, NY, USA
scott.a.s@us.ibm.com

Martin Hirzel
IBM Watson Research Center,
Yorktown Heights, NY, USA
hirzel@us.ibm.com

Buğra Gedik
Computer Engineering Dept.,
Bilkent University, Turkey
bgedik@cs.bilkent.edu.tr

ABSTRACT

This tutorial starts with a survey of optimizations for streaming applications. The survey is organized as a catalog that introduces uniform terminology and a common categorization of optimizations across disciplines, such as data management, programming languages, and operating systems. After this survey, the tutorial continues with a deep-dive into the fission optimization, which automatically transforms streaming applications for data-parallelism. Fission helps an application improve its throughput by taking advantage of multiple cores in a machine, or, in the case of a distributed streaming engine, multiple machines in a cluster. While the survey of optimizations covers a wide range of work from the literature, the in-depth discussion of fission relies more heavily on the presenters' own research and experience in the area. The tutorial concludes with a discussion of open research challenges in the field of stream processing optimizations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*optimization*; H.2.4 [**Database Management**]: Systems—*query processing*; D.4.8 [**Operating Systems**]: Performance—*operational analysis*

Keywords

Stream processing, optimizaition, data parallelism, fission

1 Introduction

We are living in an increasingly connected and instrumented world, where a large number and variety of data sources are available from various software and hardware sensors. These data sources often take the form of continuous data streams. Examples can be found in several domains, such as live stock ticker data in financial markets, call detail records in telecommunications, video streams in surveillance, production line status feeds in manufacturing, and vital body signals in health-care. In all of these domains there is a need to gather, process, and analyze data streams, detect emerging patterns and outliers, extract valuable insights, and generate actionable results. Most importantly, this analysis often needs to happen in near real-time.

Stream processing is a computational paradigm that enables carrying out these tasks in an efficient and scalable manner. Streaming applications are programs that process continuous data streams on-the-fly, as the data flows through the system. Various research communities have independently developed programming models and systems for streaming. While there are differences both at the language level and at the system level, each of these communities ultimately represents streaming applications as a *graph* of streams and operators. Since operators run concurrently, stream graphs inherently expose parallelism. At the same time, many streaming applications require high performance, and as a result each community has developed optimizations that go beyond this inherent parallelism.

Unfortunately, while there is plenty of literature on streaming optimizations, the literature uses inconsistent terminology. Furthermore, different communities have different assumptions that are often taken for granted. To address the terminology issue, this tutorial includes a survey of streaming optimizations using a uniform terminology. To address the diverse assumptions, the survey clarifies conditions that specify when the optimizations can be applied without changing the semantics of the applications, as well as when they are expected to improve the performance. This part of the tutorial is based on a survey paper by the authors [18].

Handling large volumes of live data in short periods of time is a major characteristic of streaming applications. Thus, supporting high throughput processing is a critical requirement for streaming systems. It necessitates taking advantage of multiple cores and/or host machines to achieve scale. This requires language and system level techniques that can effectively locate and efficiently exploit data parallelism in streaming applications. This latter aspect, called *fission*, is the focus of the second part of the tutorial.

Many streaming optimizations are limited in terms of the speedup they can bring, due to their strong dependence on application characteristics such as pipeline depth or filter selectivity. In contrast, the main limiting factor for data parallelism is the number of available cores, which can be easily scaled by providing additional hardware. As such, fission is a fundamental optimization that can provide good scalability as long as resources are available and the application is free of non-parallelizable bottlenecks.

This tutorial formalizes the problem of fission and provides details on how to apply it safely (no impact on appli-

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.
Copyright 2013 ACM 978-1-4503-1758-0/13/06 ...$15.00.

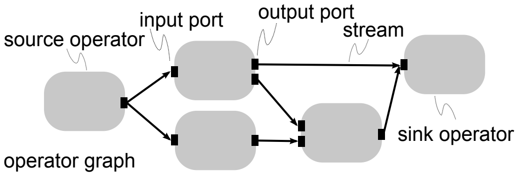


Figure 1: Basic concepts related to streaming applications.

cation semantics), transparently (no or minimal intervention from the application developers), and elastically (adaptive to run-time dynamics). Additional details on these topics can be found in our recent work [30].

This tutorial concludes by discussing open research challenges. The discussion includes both broad challenges valid for multiple optimizations, and in-depth challenges specific to fission. For example, the programming model design plays a big role in making many optimizations easier or harder to apply. Specifically, for fission, the programming model can help by providing well-defined interfaces for state.

The optimization techniques covered in this tutorial will help application developers to better understand performance trade-offs, compiler and run-time designers to implement safe and profitable optimizations, and researchers to explore new areas in streaming optimizations that are in need of technical innovations.

2 Background

This section provides a brief overview of fundamental concepts related to stream processing applications.

Operator Graphs

A stream processing application is organized as a *graph*, formed by a set of *operator*s connected to each other by *stream*s. A stream is a series of *data item*s, where each data item consists of a set of *attribute*s. Operators are generic data manipulators. They can have input and output ports. An operator fires when a data item is delivered to one of its input ports. During its firing, an operator can perform processing and produce data items on its output ports. Streams connect output ports of operators to input ports of other operators using FIFO semantics. A *source* operator does not have any input ports. It performs edge adaptation to receive data from an external source and converts it into a stream. Similarly, a *sink* operator does not have any output ports. It performs edge adaptation to deliver data from a stream to an external sink. Figure 1 illustrates these concepts.

State in Operators

A streaming operator that does not maintain state across firings is called *stateless*. For instance, a projection operator that drops some of the attributes of each data item is a stateless operator. Operators that maintain state across firings are called *stateful* operators. For instance, an operator that computes the maximum value of an attribute over the last 10 data items is stateful.

A special case of stateful operators is *partitioned stateful* operators. Such operators maintain independent state for non-overlapping sub-streams defined by a *partitioning attribute*. A typical example is the computation of volume weighted average price for each stock symbol in a financial trading stream, independently over the last 10 transactions involving each stock symbol. In this case, the partitioning attribute is the stock symbol and the data items with a specific stock symbol value constitute a sub-stream. Independent state, which takes the form a window containing the last 10 data items, is maintained for each sub-stream.

Selectivity of Operators

Streaming operators with a single input and a single output port have a notion of *selectivity* associated with them. Selectivity is the number of data items produced per data item consumed. For example, a selectivity value of 0.1 means 1 data items is produced for every 10 consumed. Selectivity is an important property, as it is used in establishing safety and profitability in many optimizations. Many streaming operators have *dynamic selectivity*, where the selectivity value is not known at development time, and can change at runtime (such as data-dependent filtering, compression, or time-based windows).

Operators can be categorized based on their selectivities. Operators that always produce one data item for each data item consumed are said to have a selectivity of *exactly-one*. Operators that produce zero or one data items for each data item consumed have a selectivity of *at-most-one*. Finally, all other operators have *unknown* selectivity. An example of unknown selectivity is *prolific* operators, which produce more than one data items for each data item consumed.

We assume a programming model that does not restrict the selectivity of operators, even though we categorize the operators based on their selectivity and use this information for safety and profitability analysis. In contrast, *synchronous data flow* (SDF) languages [24] assume that the selectivity of each operator is fixed and known at compile time. While this provides an opportunity for the compiler to create static execution schedules, the resulting inflexibility reduces the set of applications that can be expressed in this model mostly to the signal processing domain.

Flavors of Parallelism

There are three main forms of parallelism that can be found in streaming applications. The first is *pipeline* parallelism, where an operator processes a data item at the same time its upstream operator processes the next data item. Since different operators in an operator graph can be executed on different cores, processors, or machines, this kind of parallelism is inherently present in streaming applications.

The second is *task* parallelism, where different operators process a data item in parallel. Task parallelism takes place when the data items produced by an operator are consumed by more than one downstream operator. For instance, in a video processing application, a frame generated by an operator can be used to perform face detection and background detection in parallel. Again, this kind of parallelism is inherent in the operator graph.

The third is *data* parallelism, where different data items are processed by the same operator in parallel. This is typically achieved by replicating the operator in question and routing data items to different replicas. There are two aspects of data parallelism that stand out. First, it needs to be extracted from the streaming application, as the operator graph needs to be modified to include new operator instances, a splitter needs to be included to route data items to replicas, and a merger is needed at the end to bring the results back. Second, it requires additional mechanisms to preserve the application semantics. For instance, the merger should reorder the data items so that the original order before the split is reestablished.

Safety and Profitability

An optimization is *safe* if the programs generated by applying it are guaranteed to maintain the semantics of the original program. Data parallelism is safe if the operators that are replicated are stateless or partitioned stateful. In the latter case, the routing needs to be done according to the partitioning key, so that each sub-stream is routed to a single replica. Equally importantly, safe data parallelism requires a reordering at the merger, details of which depend on the selectivity of the replicated operators.

Safety alone is not enough to make an optimization useful in practice. For that, we also need to make sure that the optimization applied increases the throughput. In the case of data parallelism, the optimization has a configuration option: the number of replicas. Determining the best setting that maximizes the throughput is the *profitability* problem.

The fission optimization aims at performing safe data parallelism that is profitable. It also aims at performing this *transparently*, such that the application developers do not need to explicitly deal with parallelizing their application.

Adaptive Optimization

The profitability of many optimizations depends not only on application characteristics (such as where the bottleneck is), but also on system dynamics (such as the workload and resource availability). As a result, ideally, the profitability decisions should be *adaptive*. For instance, when there is an increase in the workload availability, the number of replicas in fission would need to be increased.

An important challenge in making optimization profitability decisions adaptive is to satisfy the *SASO* properties of control systems: stability (do not oscillate wildly), accuracy (eventually find the most profitable operating point), settling (quickly settle on an operating point), and overshoot (steer away from disastrous settings).

3 Optimization Catalog

With the definitions from the previous section in place, this section surveys 11 common optimizations for streaming applications. The survey is presented in the form of a catalog, where each optimization has a subsection of its own, and all subsections follow a similar structure. This presentation format is inspired by catalogs for other concepts in computer science, such as design patterns or refactorings. For a more detailed version of this catalog, see our prior work [18]. Each subsection is structured as follows:

Name: For optimizations known under multiple names, we picked what we believe should be the definitive term.

Tag line: Brief summary of what the optimization does.

Figure: Before-and-after picture for the optimization.

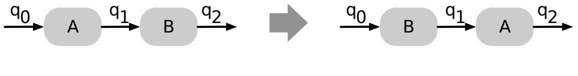
Profitability: When and how the optimization is expected to improve performance.

Safety: Conditions to be checked to establish that the optimization preserves semantic equivalence.

Literature: Pointers to the most influential or unique work in the area (for a more thorough literature review see [18]).

Operator Reordering

Change the order in which operators appear in the graph.



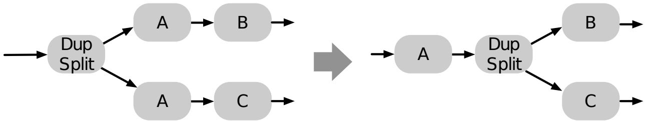
Profitability: The core idea of operator reordering is to hoist selective operators upstream so they can eliminate some data items early. That way, expensive operators downstream can spend less time by not processing those data items. If operators A and B are equally selective, it is more profitable to put the less expensive one first. If operators A and B are equally expensive, it is more profitable to put the more selective one first.

Safety: Operator reordering is a common optimization in the relational domain. In that domain, safety is established via algebraic equivalence: $\mathsf{A}(\mathsf{B}(S)) \equiv \mathsf{B}(\mathsf{A}(S))$. However, in practice, many streaming operators are not simply relational operators. In that case, one way to establish safety from first principle is as follows. Reordering is safe if both operators are stateless, operator A reads only portions of data items that B forwards unmodified, and vice versa.

Literature: Graefe identified a special case where reordering is particularly profitable: when the merger at the end of a data-parallel region is immediately followed by the splitter at the beginning of the next data-parallel region, swapping them avoids a choke-point [16]. Eddies are a dynamic technique for finding the most profitable ordering of operators with independent selectivities [6]. Rather than literally rewriting the graph, Eddies instead change the data-item routing. Hueske et al. present a static analysis for Java that establishes reordering safety from first principle [20].

Redundancy Elimination

Eliminate operators that are redundant in the graph.



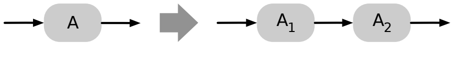
Profitability: Eliminating redundant operators is profitable if resources are limited. For example, if a redundant task takes time away on a core that could be put to better use, eliminating that task improves overall performance. A common cause for redundancy is compilation based on instantiating simple templates. In some cases, redundancy is not immediately obvious, and instead needs to be exposed by other optimizations. Another common cause for redundancy is multi-tenancy, where many users independently launch similar applications that can share subgraphs.

Safety: By definition, finding redundancy requires identifying equivalent computations. While this is undecidable in general, it can often be trivially established based on identical code, or more generally based on algebraic equivalences. One thing to look out for in redundancy elimination is that the state of the operators needs to be combinable as well.

Literature: The Rete algorithm is a seminal example for detecting and eliminating redundancies in a massively multitenant system, where applications are frequently launched and retracted [11]. NiagaraCQ applied similar ideas in the context of streaming XML processing [9]. Pietzuch et al. also eliminate redundancy at application launch time, while performing distributed placement [29].

Operator Separation

Separate an operator into multiple constituent operators.



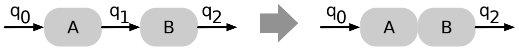
Profitability: Operator separation can be profitable in and of itself via pipeline parallelism. If A_1 and A_2 each have roughly half of the cost of A, and their cost exceeds the communication overhead, then they can exploit an additional core to improve overall throughput. But often, operator separation is profitable by enabling other optimizations, such as fission or operator reordering. For example, MapReduce applications often separate the Reduce operator to extract a Combine operator, which they then reorder and piggy-back on the Map operator [10]. This optimization is valid in streaming systems as well.

Safety: Operator separation is among the more difficult optimizations to establish safety for. To do this from first principles, one must analyze the low-level code and establish all data dependencies. But there are several special cases where operator separation is easier. For example, an idempotent operator such as a Select or an associative Aggregate can be simply repeated. A Select operator can also be separated to filter one conjunct at a time. A Project operator can be separated to map one attribute at a time.

Literature: Algebraic equivalences for separating Select, Project, and other operators can be found in standard database text books [12]. Yu et al. present a compiler analysis that separates Aggregate operators with the help of user annotations [37]. Decoupled software pipelining separates general code by analyzing data dependencies from first principle [28].

Fusion

Fuse multiple separate operators into a single operator.



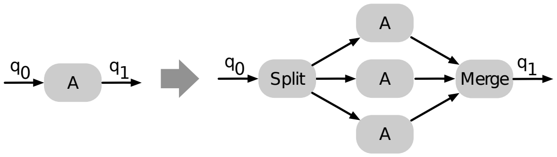
Profitability: Fusion is the dual of operator separation. Its main performance advantage comes from reduced communication overhead, and from enabling traditional (non-streaming) compiler optimizations on the fused operator. However, fusion requires sharing the same machine and potentially the same thread, thus using fewer available resources. In other words, with fusion, there is less opportunity for task or pipeline parallelism.

Safety: Fusion is among the easiest optimizations to establish safety for. It is usually safe, except when there are conflicts with placement constraints. For example, the user may request colocation, isolation, or exlocation of operators based on scarce resources such as FPGAs or network cards.

Literature: Fusion is a central optimization for the StreamIt programming language, because applications in that language tend to consist of a large number of fine-grained operators [15]. In Aurora, fusion is called superbox scheduling [8]. The COLA fusion optimizer for System S takes other placement safety constraints into account while striving for the most profitable solution [22].

Fission

Replicate an operator for data-parallel execution.



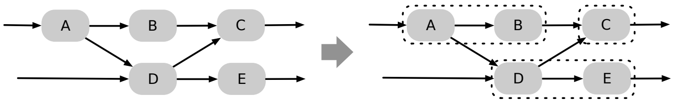
Profitability: Fission is most profitable when applied to an operator with a high processing cost per data item, and when the overhead of parallelization is low. In the ideal case, fission has the potential of improving throughput by a factor of N on N cores. In practice, speedups of 8× or even 16× are not uncommon, but the speedup is rarely ideal and usually tops out eventually. Besides parallelization overhead, load imbalance can also often get in the way, and is the subject of a separate optimization later in this section.

Safety: To be safe, fission must address state and ordering, and avoid deadlocks. In terms of state, fission is easiest if there is either no mutable state or if the state can be partitioned such that each data-parallel operator replica owns a disjoint subset. Otherwise, stateful fission requires synchronization. In terms of ordering, fission is trivially safe when no ordering is required; otherwise, ordering must be enforced by the runtime system, for example, via sequence numbers. Fission can cause deadlocks if there is a circular wait condition, where the splitter waits for buffers to be drained before sending data, but the merger waits for a data item with a particular sequence number before draining buffers.

Literature: The StreamIt compiler derives large benefits from fission of stateless operators with static selectivity [14]. Schneider et al. explored how to make fission safe in the more general case of partitioned-stateful and selective operators [30]; that work is the topic of the deep-dive in Section 4. Finally, Brito et al. propose using transactional memory to make fission safe in the case of arbitrary operator state [7].

Placement

Place the logical graph onto physical machines and cores.



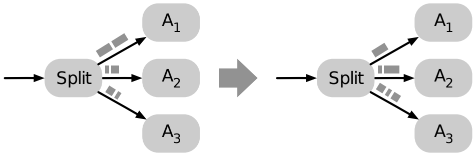
Profitability: Placement is profitable if it maximizes resource utilization while minimizing communication. Assume a distributed streaming system. On the one hand, colocating operators on the same machine can cause resource contention, for instance, on cores or the disk. On the other hand, spreading operators around too much can cause unnecessary cross-machine communication. A good placement finds profitable middle ground between these extremes.

Safety: The safety of placement is easy to establish, unless there are special resource constraints. For instance, a particular operator may only work on a GPU, which may be available only on certain machines. Another safety challenge consists in dynamically changing the placement of a stateful operator, because the state must be migrated transparently.

Literature: An early version of the StreamIt compiler explored placement on a multi-core with non-uniform memory access [15]. Pietzuch et al. explored placement in a stream-based overlay network [29]. And SODA combines placement with job admission in a distributed streaming engine that runs on a cluster [36].

Load Balancing

Avoid bottleneck operators by spreading the work evenly.



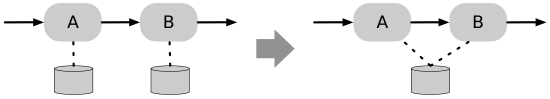
Profitability: The throughput of a stream graph is usually bounded by its slowest operator. Load balancing strives to spread the work around evenly so that all nodes in the system can operate near capacity. Thus, the profitability of load balancing depends on how imbalanced the load was to begin with, and how well it can be balanced. In the case of data parallelism (fission), load balancing can be accomplished at the splitter by routing data items to each operator replica that add up to roughly the same amount of work.

Safety: Balancing load by routing data items to data-parallel replicas assumes that all replicas are qualified to handle all data items they receive. This is easy if they are stateless, but more challenging when they have state. Besides data-item routing, another approach to load balancing is operator placement, which of course is subject to its own safety constraints explained in the previous subsection.

Literature: A good resource for load balancing via routing data items is the River work [5]. On the other hand, examples for load balancing via operator placement include the StreamIt compiler [15] as well as Amini et al.'s work [2].

State Sharing

Share identical data stored in multiple places in the graph.



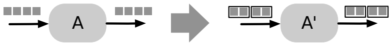
Profitability: On general-purpose hardware, applications are unlikely to just flat run out of memory. Rather, they would experience throughput and latency degradation due to exceeding the L1 cache, L2 cache, or even main memory. Therefore, state sharing is profitable if it helps keep data closer to the processor, thus avoiding the slower layers of the memory hierarchy. Another performance advantage of state sharing is that it can help avoid data copies, allocation, and serialization, all of which cost time.

Safety: State sharing is typically combined with fusion, because it is easier to share state when running in the same process. If each operator still has its own thread, safe state sharing must avoid race conditions by properly handling mutability, synchronization, and scheduling. Another concern with state sharing is avoiding memory leaks by releasing the shared state when none of the co-owners need it anymore.

Literature: Brito et al. tackle general state sharing between data-parallel operator replicas using transactional memory [7]. A more restrictive case is sharing window state only, which both StreamIt [14] and CQL [3] support. Finally, an even more restrictive, but common and profitable, case shares the state of a queue between two pipelined operators [31].

Batching

Communicate or compute over multiple data items as a unit.



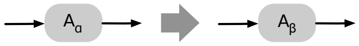
Profitability: Batching improves throughput by amortizing some fixed overheads over multiple data items, such as communication, indirect calls through layers of the stack, and bringing code and auxiliary data into the cache or into registers. Batching trades throughput against latency: individual data items have longer latency because they wait for a batch to fill. Therefore, in systems where latency matters, batching must ensure data items are still processed within their deadlines. Batching creates inner loops that traditional (non-streaming) compilers are good at optimizing.

Safety: In latency-critical systems, users may view the adherence to deadlines as a safety issue rather than a profitability issue. Aside from that, batching poses few safety challenges. One thing to look out for is potential deadlocks in cyclical graphs, if an operator waits for a batch to form at its input, but that batch does not fill up because of missing output from the same operator.

Literature: The SEDA architecture relies on a dynamic batching controller for picking a profitable batch size [35]. Aurora refers to batching as train scheduling [8]. StreamIt performs batching statically, calling it execution scaling [31].

Algorithm Selection

Replace an operator by a different operator.



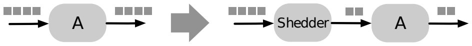
Profitability: The idea of operator selection is, of course, to pick a less expensive operator. In some cases, there is a choice between multiple operators with equivalent functionality, and it depends on the data characteristics which one is less expensive. In other cases, the default operator is more general, and the other operator is less general but faster.

Safety: Algorithm selection poses a safety question when the operators differ not just in performance, but also in functionality. In other words, if the faster operator is less general, we must establish that it is applicable based on the configuration. There are even cases where strict semantic equivalence can be sacrificed for performance. An example for that is using an approximation algorithm. This usage of algorithm selection is a variant of load shedding, discussed below.

Literature: The SEDA architecture enables an operator to pick a different algorithm to provide degraded service [35]. Borealis enables an operator to switch to a different algorithm based on a control input [1]. And the SODA optimizer offers algorithm selection at the granularity of entire jobs, to run a variant of a job that is cheaper but lower-quality [36].

Load Shedding

Degrade gracefully during overload situations.



Profitability: The core idea of load shedding is to sacrifice some accuracy so requests do not pile up when the offered load exceeds the processing capacity. This is often a latency issue: by dropping some data items, the remaining ones that are not dropped get processed fast enough to satisfy their deadlines. Sometimes, there are also priorities involved: by dropping less important data items, the more critical ones need not get dropped.

Safety: Given that Section 2 defines safety as semantics preservation, load shedding is by definition unsafe. However, the alternative (not shedding load) is also unsafe if it

```
composite Main {
 type
  Entry = tuple<uint32 uid, rstring server, rstring msg>;
  Summary = tuple<uint32 uid, int32 total>;
 graph
  stream<Entry> Messages = ParSrc() {
   param    servers:      "logs.*.com";
            partitionBy:  server;
  }
  stream<Summary> Summaries = Aggregate(Messages) {
   window  Messages:   tumbling, time(5), partitioned;
   param    partitionBy:  uid;
   output   Summaries: uid = Any(uid), total = Count();
  }
  stream<Summary> Suspects = Filter(Summaries) {
   param    filter:          total > 100;
  }
  () as Sink = FileSink(Suspects) {
   param    file:            "suspects.csv";
            format:        csv;
  }
}
```

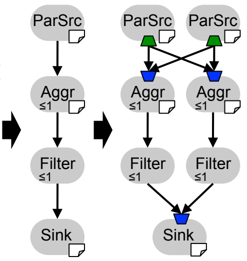


Figure 2: Example SPL program (left), its stream graph (middle), and the parallel transformation of that graph (right). The paper icons in the lower right of an operator indicate state, and the numbers in the lower left indicate selectivity.

means that the system crashes or otherwise fails to live up to its specification, such as quality of service. Therefore, the goal cannot be safety, but rather maximizing accuracy within the constraints of load and resources.

Literature: The Scout operating system uses a data-flow model for its network layer, among other things to enable informed load-shedding decisions [26]. The Aurora streaming engine implements priority-based load shedding [33]. And Compact Shedding Filters ship the task of load shedding from the server to data-generating sensors to avoid unnecessary network communication [13].

4 Fission

This section presents a deep-dive into the fission optimizations, which was briefly mentioned in the optimization catalog (Section 3). The catalog is platform and language agnostic. However, in order to study an optimization in practice, we must look at a specific platform and language. We will first introduce the System S platform and the SPL language [17], and then discuss applying fission there.

System S and SPL

The programming model behind SPL is asynchronous, as it allows operators to have dynamic selectivity. The System S platform allows for distributed execution.

Figure 2 presents a sample SPL program [17] on the left. The program is a simplified version of a common streaming application: network monitoring. The application continually reads server logs, aggregates the logs based on user IDs, looks for unusual behavior, and writes the results to a file.

The types Entry and Summary describe the structure of the *tuples* in this application. A tuple is a data item consisting of attributes, where each attribute has a type (such as uint32) and a name (such as uid). The stream graph consists of operator invocations, where operators transform streams of a particular tuple type.

The first operator invocation, ParSrc, is a source that produces an output stream called Messages, and all tuples on that stream are of type Entry. The ParSrc operator takes two parameters. The partitionBy parameter indicates that the data is *partitioned* on the server attribute from the tuple type Entry. Thus, {server} is this operator's partitioning *key*.

The Aggregate operator invocation consumes the Messages stream, indicated by being "passed in" to the Aggregate operator. The window clause specifies the tuples to operate on, and the output clause describes how to aggregate input tuples (of type Entry) into output tuples (of type Summary). This operator is also partitioned, but this time the key is the uid attribute of the Entry tuples. Because the Aggregate operator is stateful, we consider this operator invocation to have *partitioned state*. The Aggregate operator maintains separate aggregations for each instance of the partitioning key ({uid} in this case).

The Filter operator invocation drops all tuples from the aggregation that have no more than 100 entries. Finally, the FileSink operator invocation writes all of the tuples that represent anomalous behavior to a file.

The middle of Figure 2 shows the stream graph that programmers reason about. In general, SPL programs can specify arbitrary graphs, but the example consists of just a simple pipeline of operators. We consider the stream graph from the SPL source code the *sequential semantics*, and the fission optimization seeks to preserve such semantics.

The right of Figure 2 shows the stream graph that the runtime will actually execute. First, the compiler determines that the first three operators can have data parallelism, and it allows the runtime to replicate those operators. The operator instances ParSrc and Aggregate are partitioned on different keys. Because the keys are incompatible, the compiler instructs the runtime to perform a *shuffle* between them, so the correct tuples are routed to the correct operator replica. A shuffle is a bipartite graph between the end of one parallel region and the beginning of the next. The Filter operator instances are stateless and can accept any tuple. Hence, tuples can flow directly from the Aggregate replicas to the Filter replicas, without another shuffle. Finally, the FileSink operator instance is not parallelizable, which implies that there must be a merge before it to ensure it sees tuples in the same order as in the sequential semantics.

Safety

In the context of fission, *safety* means preserving an application's sequential semantics. Doing so requires support in both the compiler and the runtime.

Compiler

The compiler's task is to decide which operator instances belong to which parallel regions. Furthermore, the compiler picks implementation strategies for each parallel region, but not the degree of parallelism. One can think of the compiler as being in charge of safety while avoiding platform-dependent profitability decisions.

As usual in compiler optimization, the approach is conservative: the conditions may not always be necessary, but they imply safety. The conditions for parallelizing an individual operator instance are:

- ***No state or partitioned state:*** The operator instance must be either stateless, or its state must be a map where the *key* is a set of attributes from the input tuple. Each time the operator instance fires, it only accesses its state for the given key. Safe fission gives each operator replica a disjoint partition of the key domain.
- ***At most one predecessor and successor:*** The operator instance must have fan-in and fan-out ≤ 1. This means parallel regions have a single entry and exit where the runtime can implement ordering.

The conditions for forming larger parallel regions with multiple operator instances are:

- ***Compatible keys:*** If there are multiple stateful operator instances in the region, their keys must be compatible. A key is a set of attributes, and keys are compatible if their intersection is non-empty. Using the intersection as the region key ensures that the splitter is at most as fine-grained as any individual operator's key.
- ***Forwarded keys:*** Care must be taken that the region key as seen by a stateful operator instance o indeed has the same value as at the start of the parallel region. This is because the split at the start of the region uses the key to route tuples, whereas o uses the key to access its partitioned state map. All operator instances along the way from the split to o must forward the key unchanged.
- ***No shuffle after prolific regions:*** A prolific region is a region with prolific operators. Prolificacy causes tuples with identical sequence numbers. Within a single stream, such tuples are still naturally ordered. But after a shuffle, this ordering could be lost. Therefore, the compiler does not allow a shuffle at the end of a prolific region.

The compiler must establish the previously described safety conditions. We must first distinguish an operator definition from an operator invocation. The *operator definition* is a template, such as an Aggregate operator. It provides different configuration options, such as what window to aggregate over or which function (Count, Avg, etc.) to use. Since SPL users have domain-specific code written in C++ or Java, we support user-defined operators that encapsulate such code. Each operator definition comes with an *operator model* describing its configuration options to the compiler. The *operator invocation* is written in SPL and configures a specific instance of the operator, as shown in Figure 2. The operator instance is a vertex in the stream graph.

We take a two-pronged approach to establishing safety: *properties* in the operator model for operator definitions and *program analysis* for operator invocations in SPL. This is pragmatic and requires some trust: if the author of the operator deceives the compiler by using the wrong properties in the operator model, the optimization may be unsafe. Operator models must specify whether or not they have *state*, how *selective* they are, and whether or not they *forward* all attributes on input tuples.

In most cases, analyzing an SPL operator invocation is straightforward given its operator model. However, operator invocations can also contain imperative code, which may affect safety conditions. State can be affected by mutating expressions. Selectivity can be affected if the operator invocation calls submit to send tuples to output streams. Our compiler uses data-flow analysis to count submit-calls. If submit-calls appear inside of if-statements, the analysis computes the minimum and maximum selectivity along each path. If submit-calls appear in loops, the analysis assumes that selectivity is Unknown.

Runtime

The System S runtime has a concept of *Processing Elements* (PEs), which are a group of operators that the fusion optimization has been applied to. They execute inside of a single operating system process. The runtime support for fission is mostly concerned with PEs, as parallel regions are composed of PEs.

The runtime has two primary tasks: routing tuples to parallel channels, and enforcing tuple ordering. Parallel regions should be semantically equivalent to their sequential counterparts. In a streaming context, that equivalence is maintained by ensuring that the same tuples leave parallel regions in the same order regardless of the number of parallel channels.

Routing and ordering are achieved through the same mechanisms: *splitters* and *mergers* in the PEs at the edges of parallel regions. Splitters exist on the output ports of the last PE before the parallel region. Their job is to route tuples to the appropriate parallel channel, and add any information needed to maintain proper tuple ordering. Mergers exist on the input ports of the first PE after the parallel region. Their job is to take the streams from each parallel channel and merge their tuples into one, well-ordered output stream. The splitter and merger must perform their jobs invisibly to the operators both inside and outside the parallel region.

When parallel regions only have stateless operators, the splitter routes tuples in round-robin fashion, regardless of the ordering strategy. When parallel regions have partitioned state, the splitter uses the attributes that define the partition key to compute a hash value. It then uses that hash to route the tuple, ensuring that the same attribute values are always routed to the same operators.

There are two classes of ordering strategies: implicit and sequence number based. Implicit ordering strategies can be applied to parallel regions that contain only stateless, non-selective operators. In such cases, the splitter and merger can conspire on the order in which tuples are split to, and merged out of, a parallel region. In these cases, no extra information is needed on the tuples themselves to maintain the correct order.

Sequence number based ordering strategies are required when operators in a parallel region are stateful, selective, or prolific. While there are various kinds of specializations, the general idea is that the splitter attaches sequence numbers to all tuples, and the merger uses those sequence numbers to put the tuples back in order.

Profitability

The previous section is concerned with how to discover opportunities to safely extract data parallelism, and how to safely execute it in a runtime system. However, discovering where fission can be safely applied does not answer the basic question: is it profitable? In the context of fission in a streaming system, solving profitability means finding how many parallel channels to use in a parallel region.

In synchronous streaming systems, it may be possible to answer the profitability question statically, at compile time. But in asynchronous streaming systems with user-defined operators, it is impractical to determine profitability statically, which means the decision must happen at runtime.

The following sections describe what problems a solution to dynamic profitability for fission must solve.

Control Algorithm

Dynamically determining the number of parallel channels in a parallel region means that there must be a runtime control algorithm. The input to the control algorithm must be a runtime metric that system implementers wish to use to determine profitability. Fission in particular, and data parallelism in general, tend to pay attention to throughput. However, one could devise an objective function which also uses latency, so as to cap the potential harm to latency while improving throughput. The control algorithm's job is to determine the number of parallel channels that maximizes this objective function.

However, observing *only* throughput and latency may not be enough for a control algorithm to obey all of the SASO properties described in Section 2. Instead, the control algorithm needs a metric to tell it whether operators in a parallel channel have reached their capacity limit. Detecting capacity limitations drives the algorithm, and observing the effect on the objective function checks the accuracy of that capacity detection.

The general approach for the control algorithm is to detect when all of the active parallel channels can not handle any more capacity. In such a case, there is evidence that adding parallel channels will increase throughput. The control algorithm then increase the number of channels, and after the next period, determines if throughput increased. If throughput increased, then it will stay at at least this number of channels, and then evaluate capacity again. If throughput decreased, then it will backtrack by decreasing the number of channels.

State Management

Operators in a parallel region may have partitioned state. As the control algorithm changes the degree of parallelism, operators that were maintaining state for a partition may become dormant, and the tuples they would have handled will be routed elsewhere. To maintain sequential consistency, the operators that remain active in the parallel region must adopt the partitions from the dormant operators. The same issue arises when parallel channels are added: in order to maintain an even spread, the new channels must adopt some partitions that existing channels are responsible for. Adopting partitions means that state must be migrated across operators.

State migration across operators in a streaming system involves several challenges. First, in normal operation, the tuples are always flowing. If an operator migrates a partition's state, then receives a tuple in that partition, sequential consistency will be violated. Hence, in order to maintain sequential consistency, tuples must not flow in the parallel region while partition state is in flux. In order to ensure that tuples do not flow during state migration, the splitter and all of the operators in the parallel region must obey a protocol that stops their flow while state is moving, and starts it again once state has settled.

A second challenge is avoiding too many state transfers while maintaining an even partition distribution among the operators. As the number of parallel channels expands and contracts, operators donate and adopt partitions. However, any given donation or adoption phase should transfer only the minimal amount of state. Balancing minimal state transfers with maintaining an even distribution requires collusion between the operators and the splitter. All of the operators must be able to deterministically decide which partitions they must donate or adopt, and the splitter must agree with these decisions. The splitter's goal is to use a hashing algorithm that is likely to produce a balanced distribution of partitions while minimizing state movement. In practice, consistent hashing schemes can solve these problems [21].

5 Open Research Problems

So far this paper was mostly about existing work. The preceding sections aimed to help users either hand-optimize their code, or understand automatic optimizations applied to their code. They also aimed to help implementers build more efficient streaming systems. In contrast, this section is about what is missing in existing work. By exploring which challenges are still open, and have not been fully solved yet, it aims to help researchers come up with new ideas. These open challenges are grouped into subsections. Each subsection first describes a broad spectrum of high-level research opportunities that apply across the range of optimizations from Section 3. Following that, each subsection makes the discussion more concrete for the fission optimization from Section 4. That way, each subsection highlights both high-level longer-term and specific shorter-term opportunities.

Programming Model Challenges

A *programming model* for stream processing is either a stream programming language (such as StreamIt [34], CQL [3], or SPL [17]), or a library that exposes the functionality of a streaming system as a framework (such as SVM [23], S4 [27], or Storm [32]). Programming model design is an exercise in juggling several, sometimes conflicting, goals. The programming model needs to be expressive enough so the domain of applications it works for is not too narrow. At the same time, it needs to be amenable to static analysis and have clear semantics to facilitate optimizations. A new programming model needs a foreign-code interface for incorporating legacy code written in other languages. And the more familiar a new programming model looks and feels, the easier it is to adopt by a broad community.

Our own work with fission taught us several programming model lessons. One is that even when the programming model is a new language, optimizations must also take libraries in existing languages into account. For fission, that meant providing operator models that assert properties about state, selectivity, etc.; and providing an API for state to be called from C++ but handled by the streaming runtime system. Another lesson was that since partitioning plays such a central role in fission, it should be a first-class concept in the language as well.

Optimization Combination

If a streaming system supports two or more of the optimizations described in Section 3, one question is how to combine them. One approach is to just apply them one by one. The order of optimizations matters. For instance, performing operator separation early opens up opportunities for other optimizations such as operator reordering. Conversely, fusion should happen late, as it makes other optimizations more difficult. But rather than performing optimizations separately one by one, another option is to truly combine them, making a unified profitability or safety assessment. Since different optimizations have their own cost models, combining them leads to new research challenges.

Being a particularly profitable optimization, fission is a prime candidate for combining with other optimizations, such as fusion, load balancing, placement, and batching. Of course, fission can be extended not just with other optimizations, but also with transformations for different purposes than optimization. For instance, both fission and fault tolerance can be accomplished by replicating operators.

Interaction with Traditional Compilers

We use the word *traditional* compiler to refer to a compiler for a non-streaming language such as Fortran, C, or Java. Traditional compilers play a role at both ends of compiling a streaming language. First, besides new features for supporting stream processing, a streaming language usually also has features in common with traditional languages, such as expressions with function calls, arithmetic, variable accesses, and so on. Compiler analysis on such expressions is used to establish safety properties for optimizations. Second, many compilers for streaming languages generate source code for non-streaming languages, which must then still be compiled by a traditional compiler. The traditional compiler comes with its own traditional optimizations, such as function inlining or loop unrolling. Overall performance is best if the code generated by the streaming compiler is easy to optimize by the non-streaming compiler.

Fission is a prime example for an optimization that can benefit from traditional compiler analysis. The analysis can discover information about state, ordering, selectivity, and attribute forwarding that drives the safety decisions for fission. One challenge with this is to analyze general or even legacy code. At the other end, examples of streaming optimizations that interact with downstream traditional compiler optimizations include batching and fusion more than fission. Batching gives rise to loops that a traditional compiler can unroll and optimize. And fusion gives rise to function calls that a traditional compiler can inline. The challenge is to ensure that generated code does not obscure these opportunities. For example, calls are easier to inline if they are monomorphic and part of the same compilation unit.

Dynamic Optimization

A *dynamic* optimization is performed at runtime, as opposed to static optimizations that are performed before the application starts. Some optimizations, such as load balancing or load shedding, are dynamic by nature. But many optimizations, in particular those that modify the operator graph such as operator reordering and fusion, are more typically static. Dynamic optimizations have the advantage that they can use profiling information with statistics about the currently ongoing run, and can even adapt to changes in load or resources. The challenge for dynamic optimizations is to satisfy the SASO properties of control systems outlined earlier in Section 2. Eddies are an example for operator reordering at runtime without actually changing the graph [6], whereas Flextream is an example of pausing the application to rewrite the graph at runtime [19]

In the case of fission, an important dynamic optimization is *elasticity*, which means dynamically changing the degree of parallelism. The main challenge here is profitability: picking the degree of parallelism that yields the best performance. This is complicated when there are multiple parallel regions, each of them with its own degree of parallelism, or when parallel regions nest. Another challenge with dynamic fission is state migration for stateful operators. While the authors have done some work along those lines, there are opportunities for further improvement by minimizing the disruption while the degree of parallelism is being changed.

Benchmarks

Demonstrating that an optimization indeed improves performance requires benchmarks. Everyone can make up a micro-benchmark, but the question is how representative that is of real applications. Realistic benchmarks are required to evaluate both generality (does the optimization work for real cases?) and profitability (is the effect large enough to matter in practice?). The LinearRoad benchmark is an advanced streaming application in the transportation domain [4]. The BiCEP micro-benchmarks demonstrate whether or not relational streaming engines implement certain optimizations [25]. And the StreamIt benchmarks encompass 65 audio, video, and digital signal processing workloads [34]. Together, they offer different viewpoints from different communities on the rapidly evolving field of stream computing.

Fission nicely illustrates the challenges in curating a benchmark suite. On the one hand, the StreamIt benchmark suite is the most comprehensive set of streaming applications available, totaling 33,000 lines of code, including 30 realistic applications and 35 micro-benchmarks. On the other hand, all of these applications are in the media-processing domain. Only 25% of StreamIt benchmarks have any stateful operators, which would simplify both the profitability and safety problems for fission. However, stateful operators appear to be more prevalent in commercial applications for transportation, communication, finance, science, and health-care.

Generality of Optimizations

The approach to establishing safety is almost always conservative: optimizers err on the side of caution by finding sufficient conditions, not necessary conditions. The idea is to support the common case, and not optimize uncommon cases when their safety conditions are too difficult to prove. Unfortunately, as discussed above under benchmarks, it is often not known what the common cases are in practice. Hence, generalizing an optimizer to make its safety conditions more liberal is fertile ground for intricate research challenges. Such work needs to be motivated with workload characterization to demonstrate practical relevance.

To make the discussion concrete, consider the safety conditions for fission. Fission can be more or less conservative when it comes to state, ordering, topology, and user code. In each case, there is a spectrum from more restrictive and easier to handle cases to more liberal but harder to handle cases. For state, the spectrum ranges from stateless to partitioned stateful to arbitrary stateful operators. Ordering is easier to handle for static selectivity than for dynamic selectivity. The internal topology of a data-parallel region can range from a single operator to a simple pipeline of operators to a general subgraph. And in terms of user code, the spectrum ranges from built-in operators only to code that is user-defined in the streaming language to user-defined legacy code in a foreign language to code-generation.

6 Conclusion

This tutorial aims at helping users understand streaming optimizations and performance trade-offs, helping implementers optimize their streaming systems and languages, and helping researchers select relevant and original problems. The tutorial starts with a survey of stream processing optimizations, in the form of a catalog for easy cross-referencing. Following the broad survey comes a deep-dive into fission, a particularly effective optimization that introduces data parallelism. The tutorial concludes with a discussion of open research questions, both for streaming optimizations in general and for fission in particular.

Acknowledgments

We thank our co-authors from prior work: Kun-Lung Wu worked on the fission optimization with us [30], and Robert Soulé and Robert Grimm worked on the optimization catalog with us [18]. The optimization catalog is currently under submission for a journal article.

7 References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A distributed, scalable platform for data mining. In *Workshop on Data Mining Standards, Services and Platforms (DM-SSP)*, pages 27–37, 2006.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 15(2):121–142, 2006.

[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Very Large Data Bases (VLDB)*, pages 480–491, 2004.

[5] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 10–22, 1999.

[6] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *International Conference on Management of Data (SIGMOD)*, pages 261–272, 2000.

[7] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 265–275, 2008.

[8] Don Carney, Uğur Cetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Very Large Data Bases (VLDB)*, pages 838–849, 2003.

[9] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *International Conference on Management of Data (SIGMOD)*, pages 379–390, 2000.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[11] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[12] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson / Prentice Hall, second edition, 2008.

[13] Bugra Gedik, Kun-Lung Wu, and Philip S. Yu. Efficient construction of compact shedding filters for data stream processing. In *International Conference on Data Engineering (ICDE)*, pages 396–405, 2008.

[14] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[15] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–303, 2002.

[16] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *International Conference on Management of Data (SIGMOD)*, pages 102–111, 1990.

[17] Martin Hirzel, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Giuliano Losa, Mark Mendell, Howard Nasgaard, Robert Soulé, and Kun-Lung Wu. Streams processing language specification. Research Report RC24897, IBM, 2009.

[18] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. Research Report RC25215, IBM, 2011.

[19] Amir Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric M. Rabbah, Trevor N. Mudge, and Scott A. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 214–223, 2009.

[20] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. In *Very Large Data Bases (VLDB)*, pages 1256–1267, 2012.

[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. 1997.

[22] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Lung Kun-Wu, Henrique Andrade, and Bugra Gedik. COLA: Optimizing stream processing applications via graph partitioning. In *International Conference on Middleware*, pages 308–327, 2009.

[23] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The Stream virtual machine. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 267–277, 2004.

[24] E. A. Lee and Messerschmitt D. G. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

[25] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. A performance study of event processing systems. In *TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC)*, pages 221–236, 2009.

[26] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Operating Systems Design and Implementation (OSDI)*, pages 153–167, 1996.

[27] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream processing platform. In *Workshop on Knowledge Discovery Using Cloud and Distributed Computing Platforms (KDCloud)*, 2010.

[28] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[29] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *International Conference on Data Engineering (ICDE)*, pages 49–61, 2006.

[30] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 53–64, 2012.

[31] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 115–126, 2005.

[32] Storm. http://storm-project.net/. Retrieved April, 2013.

[33] Nesime Tatbul, Uğur Cetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Very Large Data Bases (VLDB)*, pages 309–320, 2003.

[34] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376, 2010.

[35] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[36] 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 *International Conference on Middleware*, pages 306–325, 2008.

[37] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Symposium on Operating Systems Principles (SOSP)*, pages 247–260, 2009.