A Catalog of Stream Processing Optimizations

MARTIN HIRZEL, IBM Watson Research Center
ROBERT SOULÉ, University of Lugano
SCOTT SCHNEIDER, IBM Watson Research Center
BUĞRA GEDIK, Bilkent University
ROBERT GRIMM, New York University

Various research communities have independently arrived at stream processing as a programming model for efficient and parallel computing. These communities include digital signal processing, databases, operating systems, and complex event processing. Since each community faces applications with challenging performance requirements, each of them has developed some of the same optimizations, but often with conflicting terminology and unstated assumptions. This article presents a survey of optimizations for stream processing. It is aimed both at users who need to understand and guide the system's optimizer and at implementers who need to make engineering tradeoffs. To consolidate terminology, this article is organized as a catalog, in a style similar to catalogs of design patterns or refactorings. To make assumptions explicit and help understand tradeoffs, each optimization is presented with its safety constraints (when does it preserve correctness?) and a profitability experiment (when does it improve performance?). We hope that this survey will help future streaming system builders to stand on the shoulders of giants from not just their own community.

1. INTRODUCTION

Streaming applications are programs that process continuous data streams. These applications have become ubiquitous due to increased automation in telecommunications, health care, transportation, retail, science, security, emergency response, and finance. As a result, various research communities have independently developed programming models 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, where each *stream* is a conceptually infinite sequence of data items, and each *operator* consumes data items from incoming streams

Table I. The Optimizations Cataloged in This Survey

Column "Graph" indicates whether or not the optimization changes the topology of the stream graph. Column "Semantics" indicates whether or not the optimization changes the semantics (i.e., the input/output behavior). Column "Dynamic" indicates whether the optimization happens statically (before runtime) or dynamically (during runtime). Entries labeled "(depends)" indicate that both alternatives are well represented in the literature.

Section	Optimization	Graph	Semantics	Dynamic
2.	Operator reordering	changed	unchanged	(depends)
3.	Redundancy elimination	changed	unchanged	(depends)
4.	Operator separation	changed	unchanged	static
5.	Fusion	changed	unchanged	(depends)
6.	Fission	changed	(depends)	(depends)
7.	Placement	unchanged	unchanged	(depends)
8.	Load balancing	unchanged	unchanged	(depends)
9.	State sharing	unchanged	unchanged	static
10.	Batching	unchanged	unchanged	(depends)
11.	Algorithm selection	unchanged	(depends)	(depends)
12.	Load shedding	unchanged	changed	dynamic

and produces data items on outgoing streams. Since operators run concurrently, stream graphs inherently expose parallelism, but since many streaming applications require extreme performance, each community has developed optimizations that go beyond this inherent parallelism. The communities that have focused the most on streaming optimizations are digital signal processing, operating systems and networks, databases, and complex event processing. The latter discipline, for those unfamiliar with it, uses temporal patterns over sequences of events (i.e., data items) and reports each match as a complex event.

Unfortunately, while there is plenty of literature on streaming optimizations, the literature uses inconsistent terminology. For instance, what we refer to as an *operator* is called operator in CQL [Arasu et al. 2006], filter in StreamIt [Thies et al. 2002], box in Aurora and Borealis [Abadi et al. 2003, 2005], stage in SEDA [Welsh et al. 2001], actor in Flextream [Hormati et al. 2009], and module in River [Arpaci-Dusseau et al. 1999]. As another example of inconsistent terminology, push-down in databases and hoisting in compilers are essentially the same optimization, and therefore, we advocate the more neutral term *operator reordering*. To establish common vocabulary, we took inspiration from catalogs for design patterns [Gamma et al. 1995] and for refactorings [Fowler et al. 1999]. Those catalogs have done a great service to practitioners and researchers alike by raising awareness and using consistent terminology. This article is a catalog of the stream processing optimizations listed in Table I.

Besides inconsistent terminology, this article is further motivated by unstated assumptions: certain communities take things for granted that other communities do not. For example, while StreamSQL dialects such as CQL assume that stream graphs are an acyclic collection of query trees, StreamIt assumes that stream graphs are possibly cyclic single-entry, single-exit regions. We have encountered stream graphs in practice that fit neither mold, for example, trading applications with multiple input feeds and feedback. Additionally, several papers focus on one aspect of a problem, such as formulating a mathematical model for the profitability tradeoffs of an optimization, while leaving other aspects unstated, such as the conditions under which the optimization is safe. Furthermore, whereas some papers assume shared memory, others assume a distributed system, where state sharing is difficult and communication is more expensive, since it involves the network. This article describes optimizations for many different kinds of streaming systems, including shared memory and distributed, acyclic and cyclic, among other variations. For each optimization, this article explicitly lists both safety and profitability considerations.

The target audience of this article includes end-users, system implementers, and researchers. For end-users, this article helps understand performance phenomena, guide the automatic optimizer, and, in the worst case, hand-optimize their applications. For system implementers, this article suggests ideas for what optimizations the system should support, illustrates the engineering tradeoffs, and provides starting points for delving deeper into the literature. For researchers, this article helps judge the novelty of ideas, use consistent terminology, predict how widely an optimization applies and how profitable it is, and anticipate interactions between optimizations.

Each optimization is presented in a section by itself, and each section is structured as follows:

—*Tag-line and figure* gives a quick intuition for the policy, algorithm, and transformation underlying the optimization.
—*Example* sketches a concrete real-world application, which illustrates what the optimization does and motivates why it is useful. Taken together, the example subsections for all the optimizations paint a picture of the landscape of modern stream processing domains and applications.
—*Profitability* describes the conditions that a policy needs to consider for the optimization to improve performance. To illustrate the main tradeoffs in a concrete and realistic manner, each profitability subsection is based on a microbenchmark. All experiments were done on a real stream processing system (InfoSphere Streams [Hirzel et al. 2013]). The microbenchmarks serve as an existence proof for a case where the optimization improves performance. They can also serve as a blueprint for testing the optimization in a new application or system.
—*Safety* lists the conditions necessary for the optimization to preserve correctness. Formally, the optimization is only safe if the conjunction of the conditions is true. But beyond that hint of formality, we intentionally kept the conditions informal to make them easier to read, and to make it easier to state side conditions without having to introduce too much notation.
—*Variations* surveys the most influential and unique work on this optimization in the literature. The interested reader can use this as a starting point for further study.
—*Dynamism* identifies established approaches for applying the optimization dynamically instead of statically (i.e., during runtime).

Existing surveys on stream processing do not focus on optimizations [Stephens 1997; Babcock et al. 2002; Johnston et al. 2004], and existing catalogs of optimizations do not focus on stream processing. Parts of this article were the basis of a tutorial at the DEBS 2013 conference [Schneider et al. 2013]. In contrast to prior work on unifying semantic models for stream processing [Jain et al. 2008; Soulé et al. 2010; Botan et al. 2010], this article hones in on optimizations, while keeping the formalisms light. We present a catalog of stream processing optimizations to make them approachable to users, implementers, and researchers.

1.1. Background

This section clarifies the terminology used in this article, based on our prior work on unifying semantic models of streaming [Soulé et al. 2010]. A streaming *application* is represented by a stream *graph*, which is a directed graph whose vertices are operators and whose edges are streams. A streaming *system* is a runtime system that can execute stream graphs. In general, stream graphs might be cyclic, though some systems only support acyclic graphs. Streaming systems implement streams as FIFO (first-in, first-out) queues. Whereas a *stream* is a possibly infinite sequence of data items, at any given point in time, a *queue* contains a finite sequence of in-flight data items. The *data item* is the unit of communication in a streaming application. Different communities

(a) Pipeline-parallel A ∥ B.   (b) Task-parallel D ∥ E.   (c) Data-parallel G ∥ G.
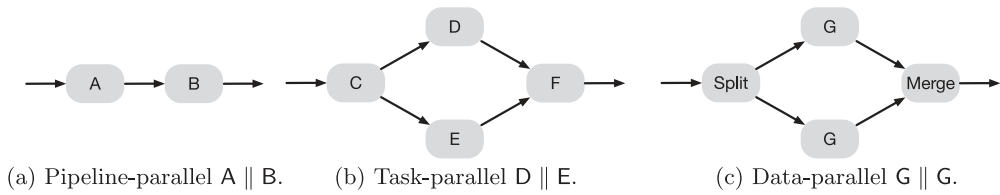
Fig. 1.   Pipeline, task, and data parallelism in stream graphs.

have different notions of data items, including samples in digital signal processing, tuples in databases, or events in complex event processing; this article merely assumes that data items can contain *attributes*, which are smaller units of data. Streaming systems are designed for data in motion and computation at rest, meaning that data items continuously flow through the edges and operators of the graph, whereas the topology of the graph rarely changes. The most common cause for topology changes is *multitenancy*, where a single streaming system runs multiple applications that come and go. Another cause for topology change is fault tolerance, where backup operators and streams take over when their primaries fail.

An *operator* is a continuous stream transformer: each operator transforms its input streams to its output streams, and operators may execute in parallel with each other. It is up to the streaming system to determine when an operator fires; for instance, an operator might have a *firing* each time a data item becomes available in one of its input queues. Operators may or may not have *state*, which is data that the operator remembers between firings. Depending on the streaming system, state might be shared between operators. The *selectivity* of an operator is its data rate measured in output data items per input data item. For example, an operator that produces one output data item for every two input data items has a selectivity of 0.5. An operator with fan-out (i.e., multiple output streams) is called a *split*, and an operator with fan-in (i.e., multiple input streams) is called a *merge*. Many split or merge operators forward data items unmodified, but a relational *join* is an example for a merge operator that includes a nontrivial transformation.

It is often useful to employ specific terminology for the various flavors of parallelism among the operators in a stream graph. Figure 1 illustrates these flavors. *Pipeline parallelism* is the concurrent execution of a producer A with a consumer B. *Task parallelism* is the concurrent execution of different operators D and E that do not constitute a pipeline. And *data parallelism* is the concurrent execution of multiple replicas of the same operator G on different portions of the same data. Data parallelism is also sometimes characterized as SPMD (single program, multiple data).

1.2. Methodology

As mentioned before, each optimization is described in its own section, and each has a subsection on profitability. These subsections contain performance measurements illustrating the tradeoffs of the optimizations. All measurements are based on experiments with microbenchmarks running on InfoSphere Streams, an industry-strength stream processing system, and written in the SPL streaming language [Hirzel et al. 2013].

Most of the measurements we present use normalized throughput as the metric, except for a few optimizations where a different metric is more important (see Section 13.6 for a discussion of metrics). While the charts are based on real runs, they are deliberately kept simple and high level, since this article is concerned with lessons for streaming systems in general, not for InfoSphere Streams in particular. Hence, instead of raw *throughput* in data items per second, we present normalized throughput,

where "1" is chosen to make it easy to compare relative performance. Similarly, several charts use a notion of normalized *operator cost*. Operator cost is the amount of work per operator firing, which we scale using a busy-loop. Again, instead of raw operator cost, we normalize, making "1" an easy-to-read baseline for the experiment at hand.

All charts include error bars indicating the standard deviation over multiple runs (≥ 3). However, in most cases, the performance is so stable across runs that the error bars are too small to see.

2. OPERATOR REORDERING (A.K.A. HOISTING, SINKING, ROTATION, PUSH-DOWN)

*Move more selective operators upstream to filter data early.*

2.1. Example

Consider a health care application that continuously monitors patients, alerting physicians when it detects that a patient requires immediate medical assistance. The input stream contains patient identification and real-time vital signs. A first operator A enriches each data item with the full patient name and the result of the last exam by a nurse. The next operator B is a selection operator, which only forwards data items with alarming vital signs. In this ordering, many data items will be enriched by operator A and will be sent on stream q_1 only to be dropped by operator B. Reordering B in front of A eliminates this unnecessary overhead.

2.2. Profitability

Reordering is profitable if it moves selective operators before costly operators. The selectivity of an operator is the number of output data items per input data item. For example, an operator that forwards only 30% of data items and drops the rest has selectivity of 0.3. The chart shows throughput given two operators A and B of equal cost, where the selectivity of A is fixed at 0.5, and the selectivity of B varies on the x-axis. Assume that the drop probabilities of A and B are independent. If A comes before B, then no matter what the selectivity of B is, A processes all data and B processes 50% of the data, so the performance does not change. If B comes before A, then B processes all data, but the amount of data processed by A is determined by the selectivity of B, and overall throughput is higher when B drops more data. The cross-over point is when both are equally selective.
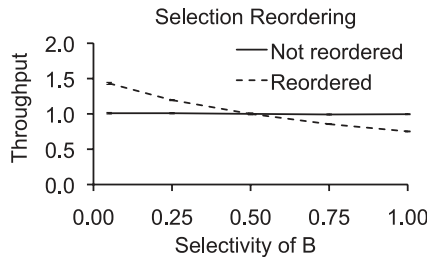
2.3. Safety

Operator reordering is safe if the following conditions hold:

—*Ensure attribute availability.* The second operator B must only rely on attributes of the data item that are already available before the first operator A. In other words,

the set of attributes that B reads from a data item must be disjoint from the set of attributes that A writes to a data item.

—*Ensure commutativity.* The result of executing B before A must be the same as the result of executing A before B. In other words, A and B must commute. Given attribute availability, a sufficient condition for commutativity is if both A and B are stateless. However, there are also cases where reordering is safe past stateful operators; for instance, in some cases, an aggregation can be moved before a split.

2.4. Variations

Algebraic reorderings. Operator reordering is popular in streaming systems built around the relational model, such as the STREAM system [Arasu et al. 2006]. These systems establish the safety of reordering based on the formal semantics of relational operators, using algebraic equivalences between different operator orderings. Such equivalences can be found in standard texts on database systems, such as Garcia-Molina et al. [2008]: besides moving selection operators early to reduce the number of data items, another common optimization moves projection operators (operators that strip away some attributes from data items) early to reduce the size of each data item. And a related optimization picks a relative ordering of relational join operators to minimize intermediate result sizes: by moving the more selective join first, the other join has less work. Some streaming systems reorder operators based on extended algebras that go beyond the relational model. For example, Galax uses nested-relational algebra for XML processing [Ré et al. 2006], and SASE uses a custom algebra for finding temporal patterns across sequences of data items [Wu et al. 2006]. More generally, commutativity analysis on operator implementations could be used to discover reorderings even without an operator-level algebra [Rinard and Diniz 1996]. A practical consideration is whether or not to treat floating point arithmetic as commutative, since floating-point rounding can lead to different results after reordering. Hueske et al. analyze the read-set and write-set of user-defined operators to determine safety of reorderings [2012].

Synergies with other optimizations. While operator reordering yields benefits of its own, it also interacts with several of the streaming optimizations cataloged in the rest of this article. Redundancy elimination (Section 3) can be viewed as a special case of operator reordering, where a Split operator followed by redundant copies of an operator A is reordered into a single copy of A followed by the Split. Operator separation (Section 4) can be used to separate an operator B into two operators B_1 and B_2; this can enable a reordering of one of the operators B_i with a neighboring operator A. After reordering operators, they can end up near other operators where fusion (Section 5) becomes beneficial. For instance, a selection operator can be fused with a Cartesian-product operator into a relational join; except in the degenerate case where the selection drops nothing, this is usually faster because it never needs to create all tuples in the product. Fission (Section 6) introduces parallel regions; when two parallel regions are back to back, reordering the Merge and Split eliminates a serialization bottleneck, as in the Exchange operator in Volcano [Graefe 1990]. The following figure illustrates this Split/Merge rotation:

2.5. Dynamism

The optimal ordering of operators is often dependent on the input data. Therefore, it is useful to be able to change the operator ordering at runtime. The Eddy operator enables a dynamic version of the operator-reordering optimization with a static graph transformation [Avnur and Hellerstein 2000]. As shown in the following figure, an Eddy operator is connected to every other operator in the pipeline and dynamically routes data after measuring which ordering is the most profitable. This has the advantage that selectivity need not be known ahead of time but incurs some extra overhead for tuple routing. The Eddy operator assumes that the probability of a data item getting filtered by one operator is independent of its probability of getting filtered by another operator. Babu et al. provide an alternative solution to dynamic operator reordering with an approximation algorithm that handles dependent probabilities and is guaranteed to be within a small constant factor of optimal [2004].
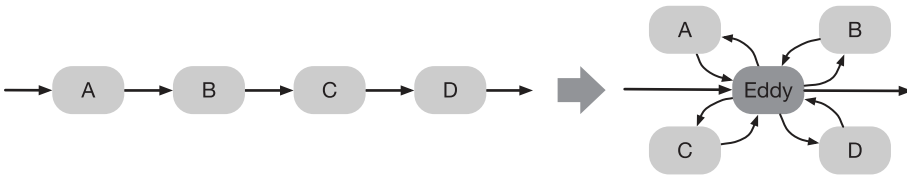
3. REDUNDANCY ELIMINATION (A.K.A. SUBGRAPH SHARING, MULTIQUERY OPTIMIZATION)

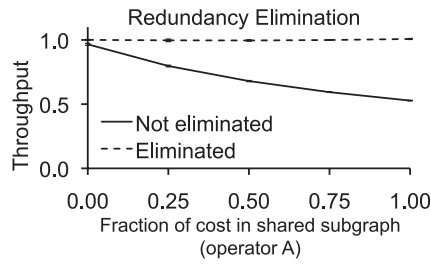*Eliminate redundant computations.*
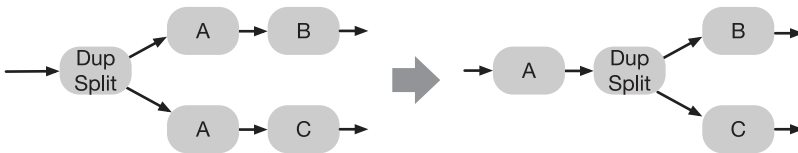
3.1. Example

Consider two telecommunications applications, one of which continuously updates billing information, and the other of which monitors for network problems. Both applications start with an operator A that deduplicates call-data records and enriches them with caller information. The first application consists of operator A followed by an operator B that filters out everything except long-distance calls and calculates their costs. The second application consists of operator A followed by an operator C that performs quality control based on dropped calls. Since operator A is common to both applications, redundancy elimination can share A, thus saving resources.

3.2. Profitability

Redundancy elimination is profitable if resources are limited and the cost of redundant work is significant. The chart shows the performance of running two applications together on a single core, one with operators A and B, the other with operators A and C. The total cost of operators A, B, and C is held constant. However, without redundancy elimination, throughput degrades when a large fraction of the cost belongs to operator A, since this work is duplicated. In fact, when A does all the work, redundancy elimination improves throughput by a factor of two, because it runs A only once instead of twice.

3.3. Safety

Redundancy elimination is safe if the following conditions hold:

—*Ensure same algorithm.* The redundant operators must, indeed, perform an equivalent computation. General program equivalence is a classical undecidable problem. In practice, a sufficient (but not necessary) condition is that the operators have identical code. Alternatively, equivalence can be based on algebra.
—*Ensure combinable state.* Redundant operators are easy to combine if they are stateless. If they are stateful and work on different data, more care is needed. For instance, a simple counter on a combined stream would differ from separate counters on subsets of the stream.

3.4. Variations

Multitenancy. Redundant subgraphs as described earlier often occur in streaming systems that are shared by many different streaming applications. Redundancies are likely when many users launch applications composed from a small set of data sources and built-in operators. While redundancy elimination could be viewed as just a special case of operator reordering (Section 2), in fact, the literature has taken it up as a domain in its own right. This separate treatment has been fruitful, leading to more comprehensive approaches. The RETE algorithm is a seminal technique for sharing computation between a large number of continuous applications [Forgy 1982]. NiagaraCQ implements sharing even when operators differ in certain constants by implementing the operators using relational joins against the table of constants [Chen et al. 2000]. YFilter implements sharing between applications written in a subset of XPath by compiling them all into a combined NFA (nondeterministic finite automaton) [Diao et al. 2002].

Other approaches for eliminating operators. Besides the sophisticated techniques for collapsing similar or identical subgraphs, there are other, more mundane ways to remove an operator from a stream graph. An optimizer can remove a no-op (i.e., an operator that has no effect), such as a projection that keeps all attributes unmodified; for example, no-op operators can arise from simple template-based compilers. An optimizer can remove an idempotent operator (i.e., an operator that repeats the same effect as another operator next to it), such as two selections in a row based on the same predicate; for example, idempotent operators can end up next to each other after operator reordering. Finally, an optimizer can remove a dead subgraph (i.e., a subgraph that never produces any output); for example, a developer may choose to disable a subgraph for debugging purposes, or a library may produce multiple outputs, some of which are ignored by a particular application.

3.5. Dynamism

A static compiler can detect and eliminate redundancies, no-ops, idempotent operators, and dead subgraphs in an application. However, the biggest gains come in the multitenancy case, where the system eliminates redundancies between large numbers of

separate applications. In that case, applications are started and stopped independently. When a new application starts, it should share any subgraphs belonging to applications that are already running on the system. Likewise, when an existing application stops, the system should purge any subgraphs that were only used by this one application. These separate starts and stops necessitate dynamic shared subgraph detection, as done, for instance, by Pietzuch et al. [2006]. Some systems take this approach to its extreme by treating the addition or removal of applications as a first-class operation just like the addition or removal of regular data items (e.g., in RETE [Forgy 1982]).

## 4. OPERATOR SEPARATION (A.K.A. DECOUPLED SOFTWARE PIPELINING)

*Separate operators into smaller computational steps.*
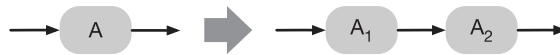
4.1. Example

Consider a retail application that continuously watches public discussion forums to discover when users express negative sentiments about a company's products. Assume that the input stream already contains a sentiment score, obtained by a sentiment extraction operator that analyzes natural-language text to measure how positive or negative it sounds (not shown). Operator A filters data items by sentiment and by product. Since operator A has two filter conditions, it can be separated into two operators A_1 and A_2. This is an enabling optimization: after separation, a reordering optimization (Section 2) can hoist the product selection A_1 before the sentiment analysis, thus reducing the number of data items that the sentiment analysis operator needs to process.

4.2. Profitability
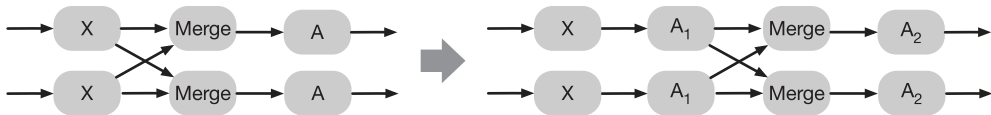
Operator separation is profitable if it enables other optimizations such as operator reordering or fission, or if the resulting pipeline parallelism pays off when running on multiple cores. We report experiments for operator reordering and pipeline parallelism in Sections 2.2 and 5.2, respectively. Therefore, here, we measure an interaction of operator separation not just with reordering but also with fission. Consider an application that consists of a first parallel region with operator X, and a second parallel region with a Merge operator and an aggregation operator A. Assume that the cost of the first region is negligible, and the cost of the second region consists of a cost of 0.5 for Merge plus a cost of 0.5 for A. Therefore, throughput is limited by the second region. With operator separation and reordering, the end of the first parallel region performs a preaggregation A_1 of cost 0.5 before the Merge. This is similar to the idea of combiners in MapReduce [Dean and Ghemawat 2004], except in the context of stream processing instead of batch processing. With selectivity ≤ 0.5, at most half of the data reaches the second region, and thus, the cost of the first region dominates. Since the cost is 0.5, the throughput is double that without optimization. At the other extreme, with selectivity 1, all data reaches the second region, and thus, the throughput is the same as without operator separation.

Separating Aggregation
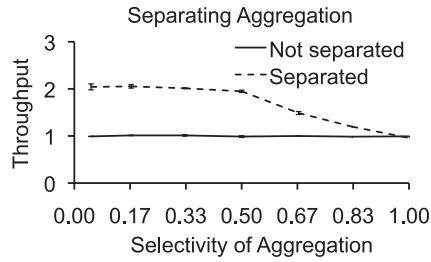
4.3. Safety

Operator separation is safe if the following condition holds:

—*Ensure that the combination of the separated operators is equivalent to the original operator.* Given an input stream s, an operator B can be safely separated into operators B_1 and B_2 only if $B_2(B_1(s)) = B(s)$. As discussed in Section 4.4, establishing this equivalence in the general case is tricky. Fortunately, there are several special cases, particularly in the relational domain, where it is easier. If B is a selection operator, and the selection predicate uses logical conjunction, then B_1 and B_2 can be selections on the conjuncts. If B is a projection that assigns multiple attributes, then B_1 and B_2 can be projections that assign the attributes separately. If B is an idempotent aggregation, then B_1 and B_2 can simply be the same as B itself.

4.4. Variations

Separability by construction. The safety of separation can be established by algebraic equivalences. Database textbooks list such equivalences for relational algebra [Garcia-Molina et al. 2008], and some streaming systems optimize based on these algebraic equivalences [Arasu et al. 2006]. Beyond the algebraic approach, MapReduce can separate the Reduce operator into a preliminary Combine operator and a final Reduce operator if it is associative [Dean and Ghemawat 2004]. This is useful, because subsequently, Combine can be reordered with the shuffle and fused with the Map operator. Similarly, Yu et al. [2009] describe how to automatically separate operators in DryadLINQ [Yu et al. 2008] based on a notion of decomposable functions: the programmer can explicitly provide decomposable aggregation functions (such as Sum or Count), and the compiler can infer decomposability for certain expressions that call them (such as new T(x.Key, x.Sum(), x.Count())), where the constructor T builds a record from the results of aggregations).

Separation by analysis. Separating arbitrary imperative code is a difficult analysis problem. In the compiler community, this has become known as DSWP (decoupled software pipelining [Ottoni et al. 2005]). In contrast to traditional SWP (software pipelining [Lam 1988]), which increases instruction-level parallelism in single-threaded code, DSWP introduces separate threads for the pipeline stages. Ottoni et al. propose a static compiler analysis for fine-grained DSWP [2005]. Thies et al. propose a dynamic analysis for discovering coarse-grained pipelining, which guides users in manually separating operators [2007].

4.5. Dynamism

We are not aware of a dynamic version of this optimization. Separating a single operator into two requires sophisticated analysis and transformation of the code containing the operator. However, the dependent optimizations enabled by operator separation, such

as operator reordering, are often done dynamically, as discussed in the corresponding sections.

5. FUSION (A.K.A. SUPERBOX SCHEDULING)

Avoid the overhead of data serialization and transport.

$$q_0 \rightarrow \boxed{A} \xrightarrow{q_1} \boxed{B} \xrightarrow{q_2} \quad \Rightarrow \quad q_0 \rightarrow \boxed{A\ B} \xrightarrow{q_2}$$

5.1. Example

Consider a security application that continuously scrutinizes system logs to detect security breaches. The application contains an operator A that parses the log messages, followed by a selection operator B that uses a simple heuristic to filter out log messages that are irrelevant for the security breach detection. Assume that the two operators run on separate cores, and that the selection operator B is lightweight compared to the cost of transferring a data item from A to B and firing B. Fusing A and B prevents the unnecessary data transfer and operator firing. The fusion removes the pipeline parallelism between A and B, but since B is lightweight, the savings outweigh the lost benefits from pipeline parallelism.

5.2. Profitability

Fusion trades communication cost against pipeline parallelism. When two operators are fused, the communication between them is cheaper. But without fusion, in a multithreaded system, they can have pipeline parallelism: the upstream operator already works on the next data item while, simultaneously, the downstream operator is still working on the previous data item. The chart shows throughput given two operators of equal cost. The cost of the operators is normalized to a communication cost of 1 for sending a data item between nonfused operators. When the operators are not fused, there are two cases: if operator cost is lower than communication cost, throughput is bounded by communication cost; otherwise, it is determined by operator cost. When the operators are fused, performance is determined by operator cost alone. The break-even point is when the cost per operator equals the communication cost, because the fused operator is $2\times$ as expensive as each individual operator.
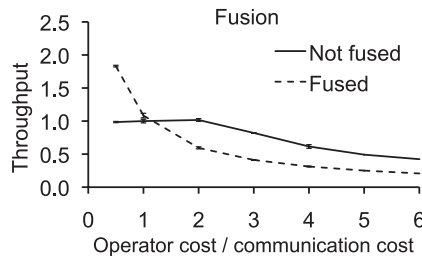
5.3. Safety

Fusion is safe if the following conditions hold:

—*Ensure resource kinds*. The fused operators must only rely on resources, including logical resources such as local files and physical resources such as GPUs, that are all available on a single host.

—*Ensure resource amounts.* The total amount of resources required by the fused operators, such as disk space, must not exceed the resources of a single host.

—*Avoid infinite recursion.* If there is a cycle in the stream graph, for example, for a feedback loop, data may flow around that cycle indefinitely. If the operators are fused and implemented by function calls, this can cause a stack overflow.

5.4. Variations

Single-threaded fusion. A few systems use a single thread for all operators, with or without fusion [Burchett et al. 2007]. But in most systems, fused operators use the same thread, whereas nonfused operators use different threads and can therefore run in parallel. That is the case we refer to as single-threaded fusion. There are different heuristics for deciding its profitability. StreamIt uses fusion to coarsen the granularity of the graph to the target number of cores, based on static cost estimates [Gordon et al. 2002]. Aurora uses fusion to avoid scheduling overhead, picking a fixed schedule that optimizes for throughput, latency, or memory overhead [Carney et al. 2003]. SPADE and COLA fuse operators as much as possible, but only as long as the fused operator performs less work per time unit than the capacity of its host, based on profiling information from a training run [Gedik et al. 2008a; Khandekar et al. 2009]. When fusion is combined with placement on nonuniform hardware such as Cell or GPUs, the optimization problem becomes intricate, giving rise to papers that apply integer linear programming to it [Hormati et al. 2009; Udupa et al. 2009].

Optimizations enabled by fusion. Fusion often opens up opportunities for traditional compiler optimizations to speed up the code. For instance, in StreamIt, fusion is followed by constant propagation, scalar replacement, register allocation, and instruction scheduling across operator boundaries [Gordon et al. 2002]. In relational systems, fusing two projections into a single projection means that the fused operator needs to allocate only one data item, not two, per input item. Fusion can also open up opportunities for algorithm selection (see Section 11). For instance, when SASE fuses a source operator that reads input data with a downstream operator, it combines them such that the downstream operator is piggy-backed incrementally on the source operator, producing fewer intermediate results [Wu et al. 2006]. The benefits of fusion are even recognized beyond traditional streaming; for instance, Coutts et al. avoid allocation of intermediate data structures in Haskell by treating lists as streams [2007].

Multithreaded fusion. Instead of combining the fused operators in the same thread of control, fusion may just combine them in the same address space but separate threads of control. That yields the benefits of reduced communication cost, without giving up pipeline parallelism. The fused operators communicate data items through a shared buffer. This causes some overhead for locking or copying data items, except when the operators do not mutate their data items.

5.5. Dynamism

Fusion is most commonly done statically. However, the Flextream system performs dynamic fusion by halting the application, recompiling the code with the new fusion decisions, and then resuming the application [Hormati et al. 2009]. This enables Flextream to adapt to changes in available resources, for instance, when the same host is shared with a different application. However, pausing the application for recompilation causes a latency glitch. Selo et al. mention an even more dynamic fusion scheme as future work in their paper on transport operators [2010]. The idea is to decide at runtime whether to route a data item to a fused operator in the same process or to a version of that same operator in a different process. Finally, Tang and Gedik [2012] apply fusion but leave the decision of which operators share a thread to runtime.

This enables them to control the tradeoffs between pipelining, thread switching, and communication dynamically.

6. FISSION (A.K.A. PARTITIONING, DATA PARALLELISM, REPLICATION)

Parallelize computations.
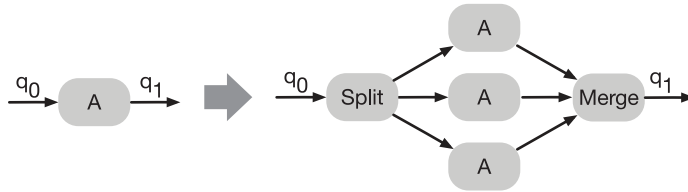

6.1. Example

Consider a scientific application that continuously extracts astronomical information from the raw data produced by radio telescopes. Each input data item contains a matrix, and the central operator in the application is a convolution operator A that performs an expensive, but stateless, computation on each matrix. The fission optimization replicates operator A to parallelize it over multiple cores and brackets the parallel region by Split and Merge operators to scatter and gather the streams.

6.2. Profitability

Fission is profitable if the replicated operator is costly enough to be a bottleneck for the application, and if the benefits of parallelization outweigh the overheads introduced by fission. Split incurs overhead, because it must decide which replica of operator A to send each data item to. Merge may also incur overhead if it must put the streams back in the correct order. These overheads must be lower than the cost of the replicated operator A itself in order for fission to be profitable. The chart shows throughput for fission. Each curve is specified by its p/s/o ratio, which stands for parallel/sequential/overhead. In other words, p is the cost of A itself, s is the cost of any sequential part of the graph that is not replicated, and o is the overhead of Split and Merge. When p/s/o is 1/1/0, the parallel part and the sequential part have the same cost, so no matter how much fission speeds up the parallel part, the overall time remains the same due to pipeline parallelism and Amdahl's law. When p/s/o is 1/0/1, then fission has to overcome an initial overhead equal to the cost of A, and therefore only turns a profit above two cores. Finally, a p/s/o of 1/0/0 enables fission to turn a profit right away.
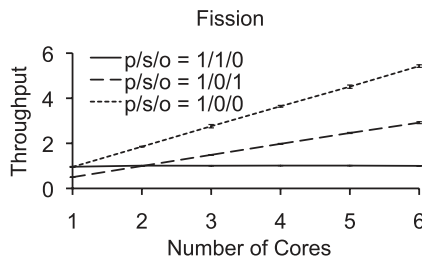
6.3. Safety

Fission is safe if the following conditions hold:

—*If there is state, keep it disjoint, or synchronize it*. Stateless operators are trivially safe;
they can be replicated much in the same way that SIMD instructions can operate
on multiple data items at once. Operators with partitioned state can benefit from
fission, if the operator is replicated strictly on partitioning boundaries. An operator
with *partitioned state* is one that maintains disjoint state based on a particular key
attribute of each data item, for example, a separate average stock price based on the
value of the stock-ticker attribute. Such operators are, in effect, multiple operators
already. Applying fission to such operators makes them separate in actuality as well.
Finally, if operators share the same address space after fission, they can share state
as long as they perform proper synchronization to avoid race conditions.

—*If ordering is required, merge in order*. Ordering is a subtle constraint, because it
is not the parallelized operator itself that determines whether ordering matters.
Rather, it is the downstream operators that consume the operator's data items. If a
downstream operator is commutative across data items, then the order in which the
data items are processed is irrelevant. If downstream operators must see data items
in a particular order, then the transformation must ensure that the output data is
merged in the same order as the input data was split. There are various approaches
for either re-establishing order or tolerating disorder. CQL uses logical timestamps
[Arasu et al. 2006]. StreamIt uses round-robin or duplication [Gordon et al. 2006].
MapReduce, instead of re-establishing the old order, uses a distributed "sort" stage
[Dean and Ghemawat 2004]. And CEDR is a streaming system whose primary design
objective is handling out-of-order streams [Barga et al. 2007].

—*Avoid deadlocks*. Both of the previous two safety constraints involved synchroniza-
tion: shared-state accesses synchronize to avoid race conditions, and mergers syn-
chronize to wait for out-of-order data items. Synchronization poses a deadlock risk
if there can be circular wait conditions. In the shared-state case, circular wait can
happen if an operator waits for a shared-variable lock while another waits for a data
item on a stream. This can be avoided by moving communication out of the criti-
cal section [Soulé et al. 2012]. In the in-order merge case, circular wait can happen
if the split cannot send data because buffers along one channel filled up, and the
merge cannot receive data because another channel is empty. This can be avoided by
periodic dummy messages [Li et al. 2010].

6.4. Variations

Fission for large batch jobs. Large batch jobs can be viewed as a special case of stream
processing where the computation is arranged as a data-flow graph, streams are finite,
and operators process data in a single pass. However, a significant difference between
large batch jobs and streaming is that batch jobs can write intermediate data to disk
and can reorder it in its entirety before proceeding to the next stage. Systems using
fission for large batch jobs include distributed databases [Graefe 1990; DeWitt et al.
1990], MapReduce [Dean and Ghemawat 2004], and Dryad [Isard et al. 2007]. The
approach dates back at least to early distributed databases such as Volcano [Graefe
1990] and Gamma [DeWitt et al. 1990]. Both support fission even for stateful operators,
as long as the state is grouped by keys. By default, even without fission, stream graphs
already have inherent parallelism, with one thread of control per operator. However,
as DeWitt and Gray explain, the number of operators in the graph before fission may
not be sufficient for the number of cores [1992]. In contrast, fission offers much larger
scaling opportunities. More recently, fission by keys for large batch jobs has also been
the centerpiece of NoSQL systems like MapReduce [Dean and Ghemawat 2004] and

Dryad [Isard et al. 2007]. As discussed in Section 2, fission is commonly combined with a reordering of split and merge operators at the boundaries between parallel regions.

Fission for infinite streams. In contrast to batch processing, streaming applications process conceptually infinite amounts of data. A good example for fission of infinite streams is StreamIt [Gordon et al. 2006]. StreamIt addresses the safety question of fission by only replicating operators either that are stateless or whose operator state is a read-only sliding window, which can be replicated along with the operator itself. In terms of profitability, the StreamIt experience shows that fission is preferable to pipeline and task parallelism, because it balances load more evenly. Schneider et al. generalize fission beyond the StreamIt setting to also work on stateful operators with dynamic data rates in a distributed system [Schneider et al. 2012]. Besides these papers, there is other work on fission for infinite streams, discussed later under dynamism. When the streaming language is designed explicitly for fission, the language constructs help programmers express code where fission is safe by construction, so the compiler does not need to do much additional work to establish safety. When the language is not designed for fission, safety must be established either by static or by dynamic dependency analysis. An example for a static analysis that discovers fission opportunities is parallel-stage decoupled software pipelining [Raman et al. 2008]. An example for dynamic analysis that discovers fission opportunities is presented by Thies et al. [2007].

6.5. Dynamism

To make the *profitability* decision for fission dynamic, we need to dynamically adjust the width of the parallel region, in other words, the number of replicated parallel operators. SEDA does that by using a thread-pool controller, which keeps the size of the thread pool below a maximum but may adjust to a smaller number of threads to improve locality [Welsh et al. 2001]. MapReduce dynamically adjusts the number of workers dedicated to the map task [Dean and Ghemawat 2004]. And "elastic operators" adjust the number of parallel threads based on trial and error with observed profitability [Schneider et al. 2009].

To make the *safety* decision for fission dynamic, we need to dynamically resolve conflicts on state and ordering. Brito et al. use software transactional memory, where simultaneous updates to the same state are allowed speculatively, with roll-back if needed [2008]. The ordering is guaranteed by ensuring that transactions are only allowed to commit in the same order in which the input data arrived.

7. PLACEMENT (A.K.A. LAYOUT)

Assign operators to hosts and cores.
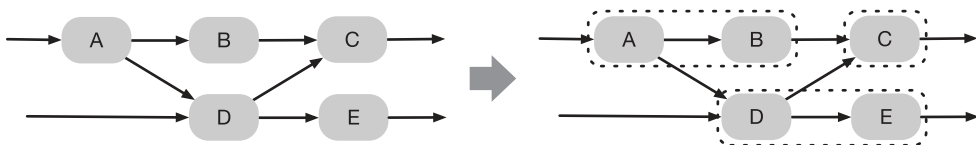
7.1. Example

Consider a telecommunications application that continuously computes usage information for long-distance calls. The input stream consists of call-data records. The example has three operators: operator A preprocesses incoming data items, operator B selects long-distance calls, and operator C computes and records billing information for the selected calls. In general, the stream graph might contain more operators, such as D and E, which perform additional functions, such as classifying customers based on their

calling profile and determining targeted promotions. If we assume that preprocessing (operator A) and billing (operator C) are both expensive, it makes sense to place them on different hosts. On the other hand, selection (operator B) is cheap, but it reduces the data volume substantially. Therefore, it should be placed on the same host as A, because that reduces the communication cost by eliminating data that would otherwise have to be sent between hosts.

7.2. Profitability

Placement trades communication cost against resource utilization. When multiple operators are placed on the same host, they compete for common resources, such as disk, memory, or CPU. The chart is based on a scenario where two operators compete for disk only. In other words, each operator accesses a file each time it fires. The two operators access different files, but since there is only one disk, they compete for the I/O subsystem. The host is a multicore machine, so the operators do not compete for CPU. When communication cost is low, the throughput is roughly twice as high when the operators are on separate hosts because they can each access separate disks and the cost of communicating across hosts is marginal. When communication costs are high, the benefit of accessing separate disks is overcome by the expense of communicating across hosts, and it becomes more profitable to share the same disk even with contention.
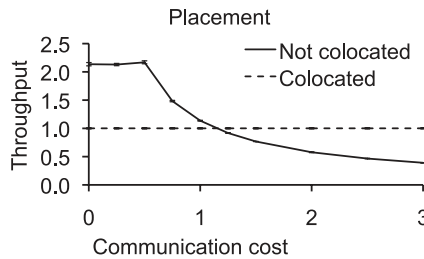
7.3. Safety

Placement is safe if the following conditions hold:

—*Ensure resource kinds*. Placement is safe if each host has the right resources for all the operators placed on it. For example, source operators in financial streaming applications often run on FPGAs, and the Lime streaming language supports operators on both CPUs and FPGAs [Auerbach et al. 2010]. Operators compiled for an FPGA must be placed on hosts with FPGAs.
—*Ensure resource amounts*. The total amount of resources required by the fused operators, such as FPGA capacity, must not exceed the resources of a single host.
—*Obey security and licensing restrictions*. Besides resource constraints, placement can also be restricted by security, where certain operators can only run on trusted hosts. In addition to these technical restrictions, legal issues may also apply. For example, licensing may restrict a software package to be installed on only a certain number of hosts.
—*If placement is dynamic, move only relocatable operators*. Dynamic placement requires operator migration (i.e., moving an operator from one host to another). Doing this safely requires moving the operator's state and ensuring that no in-flight data items are lost in the switch-over. Depending on the system, this may only be possible for certain operators, for instance, operators without state, or without OS resources such as sockets or file descriptors.

7.4. Variations

Placement for load balancing. Section 8 discusses placement algorithms that focus primarily on load balancing [Xing et al. 2005; Amini et al. 2006].

Placement for other constraints. While load balancing is usually at least part of the consideration for placement, often other constraints complicate the problem. Pietzuch et al. present a decentralized placement algorithm for a geographically distributed streaming system, where some operators are geographically pinned [Pietzuch et al. 2006]. SODA performs placement for load balancing while also taking into account constraints arising from resource matching, licensing, and security [Wolf et al. 2008]. SPADE allows the programmer to guide placement by specifying host pools [Gedik et al. 2008a]. When StreamIt is compiled to a multicore with a software-programmable communication substrate, placement considers not just load balancing, but also communication hops in the grid of cores, and the compiler generates custom communication code [Gordon et al. 2002].

7.5. Dynamism

The majority of the placement decisions are usually made statically, either during compilation or at job submission time. However, some placement algorithms continue to be active after the job starts, to adapt to changes in load or resource availability. As discussed in Section 7.3, this poses additional safety requirements. Published algorithms assume that the safety requirements are satisfied by a system mechanism for migrating operators between hosts [Xing et al. 2005; Pietzuch et al. 2006].

8. LOAD BALANCING

Distribute workload evenly across resources.
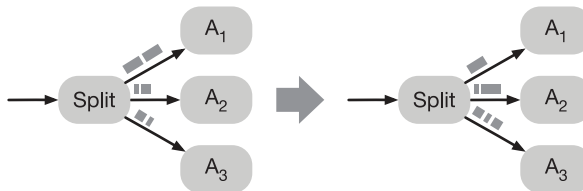
8.1. Example

Consider a security application that continuously checks that outgoing messages from a hospital do not reveal confidential patient information. The application uses a natural-language processing operator A to check whether outgoing messages contain text that could reveal confidential information, such as social security numbers or medical conditions, to unauthorized people. Operator A is expensive, and furthermore, its cost varies based on the size and contents of the data items. Since A is expensive, the fission optimization (see Section 6) has been applied to create parallel replicas A_1, A_2, and A_3. When one of the replicas is busy with a message that takes a long time to process, but another replica is idle, this optimization sends the next message to the idle replica so it gets processed quickly. In other words, when the load is unevenly distributed, the optimization balances it to improve overall performance.

8.2. Profitability

Load balancing is profitable if it compensates for skew. The chart shows the impact of load balancing in an experiment consisting of a Split operator that streams data to three or four replicated operators. With perfect load balancing, throughput is close to

four with four replicas, and close to three with three replicas. Without load balancing, there is skew, and throughput is bounded by whichever replica receives the most load. For example, with keyed partitions, this replica might be responsible for data items corresponding to a popular key. If the bottleneck replica receives 33% of the load, then even with a total of four replicas, the throughput is only three.
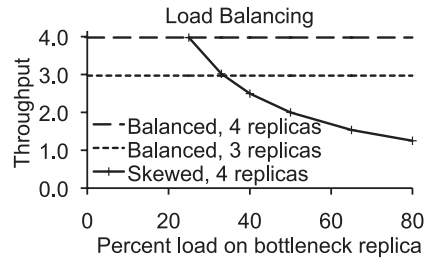
8.3. Safety

Load balancing is safe if the following conditions hold:

—*Avoid starvation.* The work assignment must ensure that every data item eventually gets processed.
—*Ensure each worker is qualified.* If load balancing is done after fission, each replica must be capable of processing each data item. That means replicas must be either stateless or have access to common shared state.
—*Establish placement safety.* If load balancing is done while placing operators, the placement safety conditions from Section 7.3 must be met.

8.4. Variations

Balancing load while placing operators. StreamIt uses fusion (Section 5) and fission (Section 6) to balance load at compile time, by adjusting the granularity of the stream graph to match the target number and capacity of cores [Gordon et al. 2002]. Xing et al. use operator migration to balance load at runtime by placing operators on different hosts if they tend to experience load spikes at the same time, and vice versa [2005]. While Xing et al. focus only on computation cost, Wolf et al. use operator placement at job submission time to balance both computation cost and communication cost [2008]. After placing operators on hosts, their load can be further balanced via priorities [Amini et al. 2006].

Balancing load while assigning work to operators. Instead of balancing load by deciding how to arrange the operators, an alternative approach is to first use fission (Section 6) to replicate operators and then balance load by deciding how much streaming data each replica gets to process. The distributed queue component in River [Arpaci-Dusseau et al. 1999] offers two approaches for this: in the push-based approach, the producer keeps track of consumer queue lengths and uses a randomized credit-based scheme for routing decisions, whereas in the pull-based approach, consumers request data when they are ready. Another example for the push-based approach is the use of back-pressure for load balancing in System S [Amini et al. 2006]. The pull-based approach works best for batch processing and is used in MapReduce [Dean and Ghemawat 2004]; in contrast, Condie et al. argue that the push-based approach works better for streaming [2010]. In MapReduce, as in other systems with fission by keys, the load balance depends on the quality of the hash function and the skew in the data. Work stealing is an approach for rearranging work even after it has been pushed or pulled to operators [Blumofe et al. 1995].

8.5. Dynamism

As the previous discussion of variations shows, there are two main techniques for load balancing: based on placement or based on tuple routing. Roughly speaking, the placement-based variants tend to be static, whereas the routing-based variants are dynamic. Placement has the advantage that it does not necessarily require fission. Placement can be made dynamic too, but that has issues: operator migration causes freeze times; if load spikes are sudden, changing the placement may take too long; and migrating a stateful operator is an engineering challenge [Douglis and Ousterhout 1991]. Routing incurs a frequent small overhead for each data item instead of an occasional large overhead for each reconfiguration.

9. STATE SHARING (A.K.A. SYNOPSIS SHARING, DOUBLE-BUFFERING)

*Optimize for space by avoiding unnecessary copies of data.*
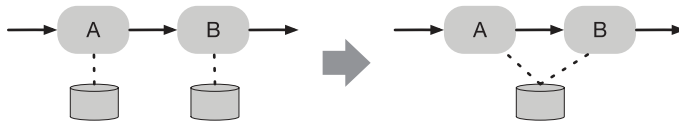
9.1. Example

Consider a financial application that continuously computes the volume-weighted average price and other statistics of stocks for both 1 hour and 1 day. Assume that the application maintains large windows for each aggregation—enough so that their memory requirements may be substantial fractions of a single host. However, if the only difference between the aggregations is their time granularity, then they can share the same aggregation window, thereby reducing the total amount of memory required for both operators.

9.2. Profitability

State sharing is profitable for throughput if it reduces stalls due to cache misses or disk I/O, by decreasing the memory footprint. The chart shows the results of an experiment with two operators, both acting on the same stream of data. To provide measurably bad locality, each operator walks a fixed number of randomly selected locations in the state each time it fires. At low state sizes, all state fits in the 32KB L1 cache, and throughput for both versions is high. As the state size increases, the not-shared version does not fit in the L1 cache anymore, and its throughput degrades. Eventually, the shared version does not fit in the L1 cache anymore either, but both still fit in the L2 cache, so the throughput becomes the same again. This phenomenon is repeated at the L2 cache size: the throughput of the not-shared version degrades first, and the throughput of the shared version follows later when it does not fit in the L2 cache anymore either.
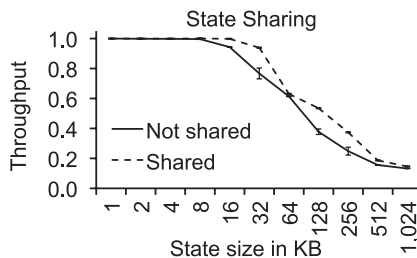
9.3. Safety

State sharing is safe if the following conditions hold:

—*Ensure state is visible to both operators.* The operators that share the state must have common access to it. Typically, this is accomplished by fusion, putting them in the same operating-system process.
—*Avoid race conditions.* State sharing must prevent race conditions, either by ensuring that the data is immutable or by properly synchronizing accesses.
—*Manage memory safely.* The memory for the shared state is managed properly. It is neither reclaimed too early nor allowed to grow without bounds (i.e., leak).

9.4. Variations

State-sharing techniques vary by what kind of state is being shared. We discuss the prominent variations from the literature in order from most general to least general.

Shared operator state. The most general variant deals with operators that have arbitrary nontrivial state. It imposes the most challenging requirements on synchronization and memory management. The straightforward approach is to use shared memory and mutual-exclusion locks. But when conflicts are rare, this may unnecessarily restrict concurrency. Therefore, another approach uses STM (software transactional memory) to manage shared data representing a table or a graph [Brito et al. 2008].

Shared window. In this variant, multiple consumers can peek into the same window. Even though operators with windows are technically stateful, this is a simple case of state that is easier to share [Gordon et al. 2006]. CQL implements windows by nonshared arrays of pointers to shared data items, such that a single data item might be pointed to from multiple windows and event queues [Arasu et al. 2006].

Shared queue. In this variant, the producer can write a new item into a queue at the same time that the consumer reads an old item. To ensure proper synchronization without sacrificing actual concurrency or requiring extra data copies, the queue must have a capacity of at least two data items; therefore, this variant is sometimes called double-buffering. Sermulins et al. show how to further optimize a shared queue by making it local and computing all offsets at compile time, so that it can be implemented by scalar variables instead of an array [Sermulins et al. 2005]. Once this is done, traditional compiler optimizations can improve the code even further by allocating queue entries to registers.

9.5. Dynamism

We are not aware of a dynamic version of this optimization: the decision whether or not state can be shared is made statically. However, once that decision is made, the implementation techniques can be more or less dynamic. StreamIt uses a fully static approach, where a static schedule prescribes exactly what data can be accessed by which operator at what time [Sermulins et al. 2005]. The work of Brito et al. is more dynamic, where access to shared state is reconciled by software transactional memory [2008].

10. BATCHING (A.K.A. TRAIN SCHEDULING, EXECUTION SCALING)

*Process multiple data items in a single batch.*

10.1. Example

Consider a health care application that repeatedly fires an FFT (fast Fourier transform) operator for medical imaging (this example is inspired by Sermulins et al. [2005]). Efficient FFT implementations contain enough code for instruction cache locality to become an issue. If the FFT is used as an operator in a larger application together with other operators, batching can amortize the cost of bringing the FFT code into cache over multiple data items. In other words, each time the FFT operator fires, it processes a batch of data items in a loop. This will increase latency, because data items are held until the batch fills up. But depending on the application, this latency can be tolerated if it leads to higher fidelity otherwise.

10.2. Profitability

Batching trades throughput for latency. Batching can improve throughput by amortizing operator-firing and communication costs over more data items. Such amortizable costs include calls that might be deeply nested; warm-up costs, in particular, for the instruction cache; and scheduling costs, possibly involving a context switch. On the other hand, batching leads to worse latency, because a data item will not be processed as soon as it is available, but only later, when its entire batch is available. The figure shows this tradeoff for batch sizes from 1 to 10 data items. For throughput, higher is better; initially, there is a large improvement in throughput, but the throughput curve levels off when the per-batch cost has been amortized. For latency, lower is better; latency increases linearly with batch size, getting worse the larger the batch is.
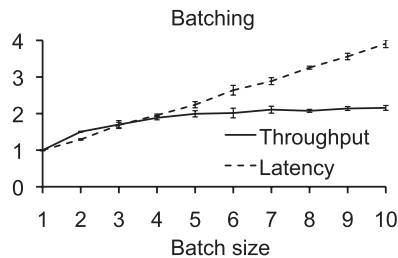
10.3. Safety

Batching is safe if the following conditions hold:

—*Avoid deadlocks.* Batching is only safe if it does not cause deadlocks. Batching can cause deadlock if the operator graph is cyclic. This happens if an operator waits for a number of data items to form a batch, but some of those data items must go around a feedback loop, and the feedback loop is depleted because the operator is waiting. Batching can also cause deadlock if the batched operator shares a lock with an upstream operator. An example is if the batched operator waits for a number of data items to form a batch while holding the lock, thus preventing the upstream operator from sending data items to complete the batch.
—*Satisfy deadlines.* Certain applications have hard real-time constraints; others have quality-of-service (QoS) constraints involving latency. In either case, batching must take care to keep latency within acceptable levels. For instance, video processing must keep up a frame rate to avoid jitter.

10.4. Variations

Batching is a streaming optimization that plays well into the hands of more traditional (not necessarily streaming) compiler optimizations. In particular, batching gives rise

to loops, and the compiler may optimize these loops with unrolling or with software pipelining [Lam 1988]. The compiler for a streaming language may even combine the techniques directly [Sermulins et al. 2005].

10.5. Dynamism

The main control variable in batching is the batch size (i.e., the number of data items per batch). The batch size can be controlled either statically or dynamically. On the static side, *execution scaling* [Sermulins et al. 2005] is a batching algorithm for StreamIt that trades the instruction-cache benefits of batching against the data-cache cost of requiring larger buffers. On the dynamic side, *train scheduling* [Carney et al. 2003] is a batching algorithm for Aurora that amortizes context-switching costs when sharing few cores among many operators, leaving the batch size open. And SEDA [Welsh et al. 2001] uses a *batching controller* that dynamically finds the largest batch size that still exhibits acceptable latency, allowing the system to react to changing load conditions.

11. ALGORITHM SELECTION (A.K.A. TRANSLATION TO PHYSICAL QUERY PLAN)

*Use a faster algorithm for implementing an operator.*

11.1. Example

Consider a transportation application that, for tolling purposes, continuously monitors which vehicles are currently on congested road segments (this example is inspired by the Linear Road benchmark [Arasu et al. 2006]). The application joins two input streams: one stream sends, at regular intervals, a table of all congested road segments, and the other stream sends location updates that map vehicles to road segments. A too-obvious implementation would implement every relational join as a nested-loop join A_α. However, in this case, the join checks the equality of road segment identifiers. Therefore, a better join algorithm, such as a hash join A_β, can be chosen.

11.2. Profitability

Algorithm selection is profitable if it replaces a costly operator with a cheaper operator. In some cases, neither algorithm is better in all circumstances. For example, algorithm A_α may be faster for small inputs and A_β may be faster for large inputs. In other cases, the algorithms optimize for different metrics. For example, algorithm A_α may be faster, but algorithm A_β may use less memory. Finally, there are cases with tradeoffs between performance and generality: algorithm A_α may be faster, but algorithm A_β may work in a wider set of circumstances. The chart compares throughput of a nested loop join versus a hash join. At small window sizes, the performance difference is in the noise, whereas at large window sizes, the hash join clearly performs better. On the other hand, hash joins are less general, since their join condition must be an equality, not an arbitrary predicate.
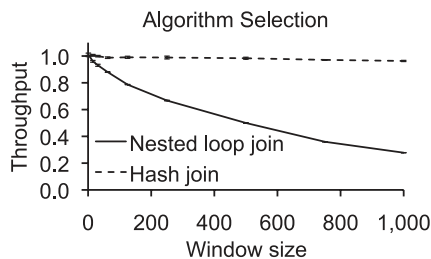
11.3. Safety

Algorithm selection is safe if the following condition holds:

—*Ensure same behavior.* Both operators must behave the same for the given inputs. If algorithm A_α is less general than algorithm A_β, then choosing the operator with A_α instead of A_β is only safe if A_α is general enough for the particular usage. The join example from Section 11.1 illustrates this.

11.4. Variations

Physical query plans. The motivating example for this section, where the choice is between a nested-loop join and a hash join, is common in database systems. Compilers for databases typically first translate an application (or query) into a graph (or plan) of logical operators, and then translate that to a graph (or plan) of physical operators [Garcia-Molina et al. 2008]. The algorithm selection happens during the translation from logical to physical operators. Join operators in particular have many implementation choices; for instance, an index lookup join may speed up join conditions like $a > 5$ with a B-tree. When join conditions get more complex, deciding the best strategy becomes more difficult. A related approach is Sase, which can fuse certain operators with the source operator and then implement these operators by a different algorithm [Wu et al. 2006].

Auto-tuners. Outside of streaming systems, there are several successful software packages that perform "empirical optimization." In order to tune itself to a specific hardware platform, the software package automatically runs a set of performance experiments during installation to select the best-performing algorithms and parameters. Prominent examples include Fftw [Frigo and Johnson 1998], Spiral [Xiong et al. 2001], and Atlas [Whaley et al. 2001]. Yotov et al. compare this empirical optimization approach to more traditional, model-based compiler optimizations [Yotov et al. 2003].

Different semantics. Algorithm selection can be used as a simple form of load shedding. There is much work in the algorithms literature on approximation algorithms for streaming, often termed "sketching"; the interested reader can refer to Babcock et al. for a survey [2002]. While most approaches to load shedding work by dropping data items (as described in Section 12), load shedding by algorithm selection merely switches to a cheaper implementation. Unlike the other variations of algorithm selection, this is, by definition, not safe, because the algorithms are not equivalent. This choice can happen either at job admission time [Wolf et al. 2008] or dynamically, as described next.

11.5. Dynamism

When algorithm selection is used to react to runtime conditions, it must be dynamic. In Seda, each operator can decide its own policy for overload, and one alternative is to provide degraded service (i.e., algorithm selection [Welsh et al. 2001]). In Borealis, operators have control inputs, for instance, to select a different algorithm variant for the operator [Abadi et al. 2005]. To implement dynamic algorithm selection, the compiler statically provisions both variants of the algorithm, and the runtime system dynamically picks one or the other as needed. In other words, this approach does for algorithm selection what the Eddy [Avnur and Hellerstein 2000] does for operator reordering: it statically inserts a dynamic routing component.

12. LOAD SHEDDING (A.K.A. ADMISSION CONTROL, GRACEFUL DEGRADATION)

*Degrade gracefully when overloaded.*

12.1. Example

Consider an emergency management application that provides logistics information to police and fire companies as well as to the general public. Under normal conditions, the system can easily keep up with the load and display information to everyone who asks. However, when disaster strikes, the load can increase by orders of magnitude and exceed the capacity of the system. Without load shedding, the requests would pile up, and nobody would get timely responses. Instead, it is preferable to shed some of the load by only providing complete and accurate replies to requests from police or fire companies and degrading accuracy for everyone else.

12.2. Profitability

Load shedding improves throughput at the cost of reducing accuracy. Consider an aggregate operator A that constructs a histogram over windows of 1,000 tuples each, for instance, to visualize system state in a graphical dashboard. For each window, it counts each data item as belonging to a "bucket." The selectivity of an operator is the number of output data items per input data item. When there is no load shedding (i.e., when selectivity is 1), the histogram has perfect accuracy (i.e., an accuracy of 1). On the other hand, if the load shedder only forwards 10 out of every thousand data items (i.e., when selectivity is 0.01), the histogram has a lower accuracy. The chart measures accuracy as 1 minus error, where the error is the Pythagorean distance between the actual histogram and the expected histogram.
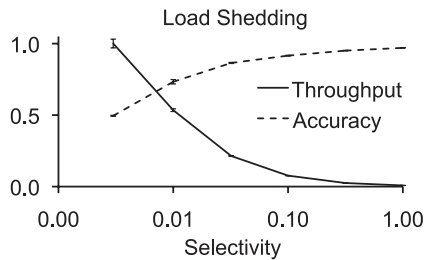
12.3. Safety

Unlike the other optimizations in this article, load shedding is, by definition, *not* safe. While the other optimizations try to compute the same result as in the unoptimized case, load shedding computes a different, approximate, result; the quality of service of the application will degrade. However, depending on the particular application, this drop in quality may be acceptable. Some applications deal with inherently imprecise data to begin with: for example, sensor readings from the physical world have limited precision. Other applications produce outputs where correctness is not a clear-cut issue: for example, advertisement placement and prioritization. Finally, there are applications that are inherently resilient to imprecision: for example, iterative page-rank computation uses a convergence check [Page et al. 1998].

12.4. Variations

Load shedding in network applications. Network stacks and web servers are vulnerable to load spikes. Implementing them as graphs of streams and operators facilitates load

shedding, because graphs make internal communication more explicit. If load shedding happens too late, the system will be busy doing lots of things with no results. This effect was identified by Mogul and Ramakrishnan and dubbed *receive livelock* [1997]. The Scout operating system drops data items early in the graph if it can predict that they will miss their deadline [Mosberger and Peterson 1996]. The Click router puts load shedders into their own separate operators to modularize the application [Morris et al. 1999]. And in the SEDA architecture for event-based servers, each operator can elect between different approaches for dealing with overload, by back-pressure, load shedding, or even algorithm selection (see Section 11) [Welsh et al. 2001].

Load shedding in relational systems. Papers on load shedders for both Aurora and STREAM observe that in general, shedders should be as close to sources as possible, but in the presence of subgraph sharing (see Section 3), shedders may need to be delayed until just after the shared portion [Tatbul et al. 2003; Babcock et al. 2004]. Somewhat related to load shedding, which drops data items from queues, is state spilling, which moves data of stateful operators out of main memory and onto disk [Liu et al. 2006].

Load shedding to preserve network bandwidth. Besides preserving compute power, load shedding can also preserve network bandwidth. In the case of battery-powered sensors, saving network bandwidth helps conserve radio power. Adaptive filters shed traffic at the source and guarantee that even though the answer is approximate, it is within user-specified bounds [Olston et al. 2003]. Constraint chaining combines temporal suppression (only send updates when values change) with spatial suppression (only send updates when neighbors differ) [Silberstein et al. 2006]. And compact shedding filters drop data that does not contribute to any of a large number of queries or data that has little effect on quality of service [Gedik et al. 2008b].

12.5. Dynamism

By definition, load shedding is always applied dynamically.

13. DISCUSSION

The previous sections surveyed the major streaming optimizations one by one. A bigger picture emerges when making observations across multiple optimizations. This section discusses these observations, puts them in context, and proposes avenues for future research on streaming optimizations.

13.1. How to Specify Streaming Applications

Not only is there a large number of streaming languages, but also there are several language families and other approaches for implementing streaming applications. The programming model is relevant for optimizations, since it influences how and where they apply. The following list of programming models is ordered from low level to high level. For conciseness, we only list one representative example for each.

—*Nonstreaming language.* This is the lowest-level approach, where the application is written in a traditional language like C or Fortran, and the compiler must do all the work of extracting streams, as in decoupled software pipelining [Ottoni et al. 2005].
—*Annotated nonstreaming language.* This approach adds pragmas to indicate streams in a traditional language like C or Fortran. An example is Brook [Buck et al. 2004].
—*Extension to nonstreaming language.* This approach adds language features to turn a traditional language like C into a streaming language. An example is Hancock [Cortes et al. 2004].

—*Framework in object-oriented language.* In this approach, an operator is specified as a subclass of a class with abstract event-handling methods. An example is Storm [2013].

—*Graph, specified textually.* Some streaming languages allow the user to specify the stream graph directly in terms of operators and streams. An example is SPL [Hirzel et al. 2013].

—*Graph, specified visually.* Instead of specifying the stream graph textually in a language, some systems, such as Aurora, provide a visual environment for that [Abadi et al. 2003].

—*Graph, composed with combinators.* Some streaming languages support graph construction only with a small set of built-in combinators. For example, StreamIt provides three combinators: pipeline, split-join, and feedback loop [Gordon et al. 2006].

—*Queries written in SQL dialect.* The databases community has developed dialects of SQL for streaming, for example, the language for StreamInsight [Ali et al. 2009].

—*Rules written in Datalog dialect.* There are also dialects of logic languages for streaming, for example, Overlog [Loo et al. 2005].

—*Patterns compiled to automatons.* The complex event processing community has developed pattern languages, which can be compiled into state machines for detecting events on streams. An example is SASE [Wu et al. 2006].

—*Tag-based planner.* This is the highest-level approach, where the user merely selects tags and the system synthesizes an application, as in Mario [Riabov et al. 2008]. The user experience more closely resembles search than programming.

As a rule of thumb, the advantages of low-level approaches are generality (pretty much any application can be expressed) and predictability (the program will perform as the author expects). On the other hand, the advantages of high-level approaches are usability (certain applications can be expressed concisely) and optimizability (the safety conditions are easy to discover). Of course, this rule of thumb is oversimplified, since generality, predictability, usability, and optimizability depend on more factors than whether the programming model is low level or high level.

Avenues for future work. For low-level stream programming models, research is needed to make them easier to use and optimize, for example, by providing more powerful analyses. For high-level stream programming models, research is needed to make them more general and to make it easier for users to understand the performance characteristics of their application after optimization. Given the diversity of streaming languages, another direction for future work is in intermediate languages that would allow the same optimization to apply to multiple languages [Soulé et al. 2012].

13.2. How Streaming Optimizations Enable Each Other

Figure 2 sketches the most important ways in which stream processing optimizations enable each other. We defer the discussion of interactions with traditional compiler analyses and optimizations to the next subsection. Among the streaming optimizations, the primary enablers are operator separation and operator reordering. Both also have benefits on their own, but much of their power comes from facilitating other optimizations.

Figure 2 has a cycle between operator reordering and fission: operator reordering enables more effective fission by bringing operators together that can be part of the same parallel region, whereas fission enables reordering of the split and merge operators inserted by fission. There is another cycle between fission and fusion: fission introduces data parallelism to make up for the pipeline parallelism lost by fusion, whereas fusion adds more work to the parallel region to amortize the split/merge cost of fission. In
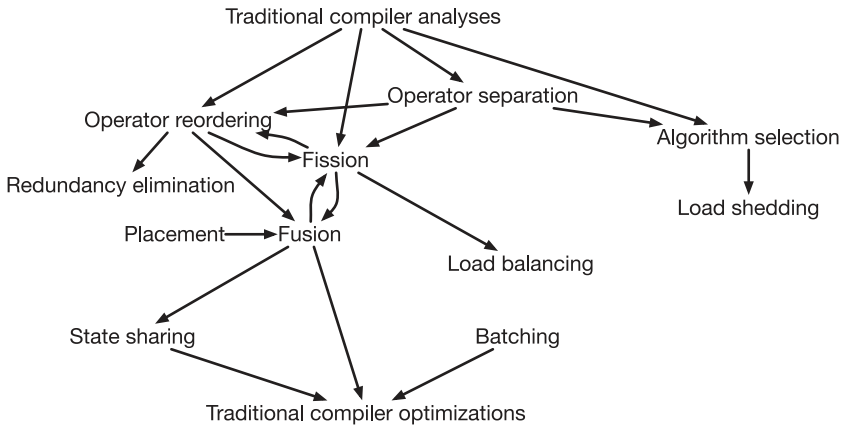
Fig. 2.   Interactions of streaming optimizations with each other and with traditional compilers. An edge from X to Y indicates that X can help to enable Y.

addition, fission makes it easier to balance load, because data parallelism tends to be more homogeneous and malleable than pipeline or task parallelism.

A streaming system that implements multiple optimizations is well advised to apply them in some order consistent with the direction of the edges in Figure 2. That way, optimizations performed earlier create more opportunities by enabling optimizations performed later. It can even make sense to repeatedly attempt optimizations that form an enabling cycle.

Avenues for future work. Finding the right sequence in which to apply optimizations is an interesting problem when there are variants of optimizations with complex interactions. Furthermore, while there is literature with cost models for individual optimizations, extending those to work on multiple optimizations is challenging; in part, that is because the existing cost models are usually sophisticated and custom-tailored for their optimization. Furthermore, models for optimizations must capture characteristics not just of the application, but also of the system and the input data. Capturing these characteristics accurately and with moderate cost is another avenue for future work.

13.3. How Streaming Optimizations Interact With Traditional Compilers

By *traditional compiler*, we refer to compilers for languages such as Fortran, C, C++, or Java. These languages do not have streaming constructs and rely heavily on functions, loops, arrays, objects, and similar shared-memory control constructs and data structures. Traditional compilers excel at optimizing code written in that style.

The top-most part of Figure 2 sketches the most important ways in which traditional compiler analyses can enable streaming optimizations. Specifically:

—*Operator reordering* can be enabled by commutativity analysis [Rinard and Diniz 1996] and read/write set analysis [Hueske et al. 2012].
—*Operator separation* can be supported by compiler analysis for decoupled software pipelining (DSWP) [Ottoni et al. 2005].
—*Fission* can also be supported by compiler analysis for parallel-stage DSWP [Ottoni et al. 2005].
—*Algorithm selection* can be supported by worst-case execution time (WCET) analysis [Lim et al. 1995].

That does not mean that without compiler analysis, these optimizations are impossible. To the contrary, many streaming systems apply the optimizations successfully by using a programming model that is high-level enough to establish certain safety properties by construction instead of by analysis.

At the other end, the bottom-most part of Figure 2 sketches the most important ways in which streaming optimizations have been used to enable traditional compiler optimizations. Specifically:

—*Fusion* enables function inlining, and that in turn is a core enabler for many other compiler optimizations, such as constant folding and register allocation.
—*State sharing* enables scalar replacement. If the compiler can statically determine the size of a communication queue based on fixed data rates and can determine at which offset each iteration accesses the queue, then instead of implementing the queue as an array shared between operators, it can implement the queue by one local variable per element [Sermulins et al. 2005].
—*Batching* enables loop unrolling, software pipelining [Lam 1988], vectorization [Hormati et al. 2010], and other loop transformations. Compilers for high-performance computing perform sophisticated optimizations on loops over arrays. These optimizations aim at taking the maximum advantage of available hardware (registers, cache, functional units, vector instructions, etc.). Batching makes them applicable to streams, since each batch can serve as an array. For the optimizations to reach their full potential, batches must be large, since the optimized loops are usually fastest only after reaching steady state.

In each case, the streaming optimization increases the amount of information available to the traditional compiler. Note, however, that this in itself does not automatically lead to improved optimization. Some engineering is usually needed to ensure that the traditional compiler will indeed take advantage of its optimization opportunities [Mosberger et al. 1996]. For instance, when the generated code uses pointers or deeply nested calls, the traditional compiler cannot always establish the safety or profitability of transformations.

Avenues for future work. One fruitful area for research would be new compiler analyses to help enable streaming optimizations in more general cases. Another area of research is in how to communicate source-level information to a low-level compiler for optimization of generated code.

13.4. Dynamic Optimization for Streaming Systems

Several streaming optimizations have both static and dynamic variants. Table I summarizes these variations, and each optimization section has a subsection on dynamism. In general, the advantages of static optimization are that they can afford to be more expensive; it is easier to make them more comprehensive; and it is easier for them to interact with traditional compilers. On the other hand, the advantages of dynamic optimization are that they are more autonomous; they have access to more information to support profitability decisions; they can react to changes in resources or load; and they can even speculate on safety, as long as they have a safe fall-back mechanism. The literature lists some intermediate approaches, which either optimize at application launch time or periodically rerun a static optimizer at runtime, as in Flextream [Hormati et al. 2009]. This is in contrast to the fully dynamic approach, where the application is transformed for maximum runtime flexibility, as in Eddies [Avnur and Hellerstein 2000].

Avenues for future work. There are several open problems in supporting more dynamic optimizations. One is low-overhead profiling and simple cost models to support

profitability tradeoffs. Another is the runtime support for dynamic optimization, for instance, efficient and safe migration of stateful operators.

13.5. Assumptions, Stated or Otherwise

Stream processing has become popular in several independent research communities, and these communities have different assumptions that influence the shape and feasibility of streaming optimizations.

Even, predictable, and balanced load. Pretty much all static optimizations make this assumption. On the other hand, other communities, such as the systems community, assume to the contrary that load can fluctuate widely. In fact, that is a primary motivation for two of the optimizations: load balancing and load shedding.

Centralized system. Many optimizations assume shared memory and/or a shared clock and are thus not directly applicable to distributed streaming systems. This is true for most cases of state sharing and for dynamic techniques such as changing the degree of parallelism in fission to adapt to load. Authors of distributed systems tend to emphasize distribution, but it does not always occur to authors of centralized systems to state the centralized assumptions.

Fault tolerance. Many optimizations are orthogonal to whether or not the system is fault tolerant. However, for some optimizations, making them fault tolerant requires significant additional effort. An example is the Flux operator, which makes fission fault tolerant by maintaining hot stand-by operators and implementing protocols for fault detection, take-over, and catch-up [Shah et al. 2004].

Avenues for future work. For any optimization that explicitly states or silently makes restrictive assumptions, coming up with a way to overcome the restrictions can be a rewarding research project. Examples include getting a centralized optimization to work (and scale!) in a distributed system and removing the dependence on fault tolerance from an optimization.

13.6. Metrics for Streaming Optimization Profitability

There are many ways to measure whether a streaming optimization was profitable, including throughput, latency, quality of service (QoS), power, and network utilization. The goals are frequently in line with each other: many optimizations that improve throughput will also improve the other metrics. For that reason, most of this survey focuses on throughput. Notable exceptions include the tradeoff between throughput and latency seen in batching, fission, and operator separation; the tradeoff between throughput and QoS or accuracy in load shedding; and the tradeoff between throughput and power in fission. As a concrete example for such tradeoffs, *slack* refers to the permissible wiggle room for degrading latency up to a deadline, which can be exploited by a controller to optimize throughput [Welsh et al. 2001].

Avenues for future work. For performance evaluation, standard benchmarks would be a great service to the streaming optimization community. Existing benchmarking work includes the Stanford stream query repository including Linear Road [Arasu et al. 2006], the BiCEP benchmarks [Mendes et al. 2009], and the StreamIt benchmarks [Thies and Amarasinghe 2010], but more work is needed. Another direction for future research is multimetric optimizers.

14. CONCLUSION

This article presents a catalog of optimizations for stream processing. It consolidates the extensive prior optimizations work and also provides a practical guide for users and

implementors. The challenge in organizing such a catalog is to provide a framework in which to understand the optimizations. To that end, this article is structured in a similar style to catalogs of design patterns or refactoring. This survey establishes a common terminology across the various research communities that have embraced stream processing. This enables members from the different communities to easily understand and apply the optimizations and lays a foundation for continued research in streaming optimizations.

ACKNOWLEDGMENTS

REFERENCES

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research (CIDR)*. 277–289.

Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal* 12, 2 (Aug. 2003), 120–139.

Mohamed H. Ali, Ciprian Gerea, Balan Raman, Beysim Sezgin, Tiho Tarnavski, Tomer Verona, et al. 2009. Microsoft CEP Server and Online Behavioral Targeting. In *Demo at the Conference on Very Large Data Bases (VLDB-Demo)*. 1558–1561.

Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. 2006. Adaptive Control of Extreme-Scale Stream Processing Systems. In *International Conference on Distributed Computing Systems (ICDCS)*.

Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *Journal* 15, 2 (June 2006), 121–142.

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

Joshua S. Auerbach, David F. Bacon, Perry Cheng, and Rodric M. Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 89–108.

Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *International Conference on Management of Data (SIGMOD)*. 261–272.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002. Models and Issues in Data Stream Systems. In *Principles of Database Systems (PODS)*. 1–16.

Brian Babcock, Mayur Datar, and Rajeev Motwani. 2004. Load Shedding for Aggregation Queries over Data Streams. In *International Conference on Data Engineering (ICDE)*. 350–361.

Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive Ordering of Pipelined Stream Filters. In *International Conference on Management of Data (SIGMOD)*. 407–418.

Roger S. Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. 2007. Consistent Streaming through Time: A Vision for Event Stream Processing. In *Conference on Innovative Data Systems Research (CIDR)*. 363–373.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Principles and Practice of Parallel Programming (PPoPP)*. 207–216.

Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *Conference on Very Large Data Bases (VLDB)*. 232–243.

Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2008. Speculative Out-of-Order Event Processing with Software Transaction Memory. In *Conference on Distributed Event-Based Systems (DEBS)*. 265–275.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. In *Computer Graphics and Interactive Techniques (SIGGRAPH)*. 777–786.

Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. 2007. Lowering: A Static Optimization Technique for Transparent Functional Reactivity. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. 1–80.

Don Carney, Uğur Cetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator Scheduling in a Data Stream Manager. In *Conference on Very Large Data Bases (VLDB)*. 309–320.

Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *International Conference on Management of Data (SIGMOD)*. 379–390.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Networked Systems Design and Implementation (NSDI)*. 313–328.

Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. 2004. Hancock: A Language for Analyzing Transactional Data Streams. *ACM Transactions on Programming Languages and Systems* 26, 2 (March 2004), 301–338.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *International Conference on Functional Programming (ICFP)*. 315–326.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Operating Systems Design and Implementation (OSDI)*. 137–150.

David DeWitt and Jim Gray. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM (CACM)* 35, 6 (June 1992), 85–98.

D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. 1990. The Gamma Database Machine Project. *Transations on Knowledge and Data Engineering (TKDE)* 2, 1 (March 1990), 44–62.

Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. 2002. YFilter: Efficient and Scalable Filtering of XML Documents. In *Demo at International Conference on Data Engineering (ICDE-Demo)*. 341–342.

Fred Douglis and John Ousterhout. 1991. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software—Practice and Experience* 21, 8 (Aug. 1991), 757–785.

Charles L. Forgy. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* 19, 1 (1982), 17–37.

Martin Fowler, Kent Beck, John Brant, and William Opdyke. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Matteo Frigo and Steven G. Johnson. 1998. FFTW: An Adaptive Software Architecture for the FFT. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1381–1384.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (second ed.). Prentice Hall.

Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008a. SPADE: The System S Declarative Stream Processing Engine. In *International Conference on Management of Data (SIGMOD)*. 1123–1134.

Buğra Gedik, Kun-Lung Wu, and Philip S. Yu. 2008b. Efficient Construction of Compact Shedding Filters for Data Stream Processing. In *International Conference on Data Engineering (ICDE)*. 396–405.

Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 151–162.

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. 2002. A Stream Compiler for Communication-Exposed Architectures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 291–303.

Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *International Conference on Management of Data (SIGMOD)*. 102–111.

Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. 2013. IBM

Streams Processing Language: Analyzing Big Data in Motion. *IBM Journal of Research and Development (IBMRD)* 57, 3/4 (May/July 2013), 7:1–7:11.

Amir Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric M. Rabbah, Trevor N. Mudge, and Scott A. Mahlke. 2009. Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures. In *Parallel Architectures and Compilation Techniques (PACT)*. 214–223.

Amir H. Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. 2010. MacroSS: Macro-SIMDization of Streaming Applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 285–296.

Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. In *Conference on Very Large Data Bases (VLDB)*. 1256–1267.

Michael Isard, Mihai Budiuand Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-Parallel Program from Sequential Building Blocks. In *European Conference on Computer Systems (EuroSys)*. 59–72.

Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Cetintemel, Mitch Cherniack, Richard Tibbets, and Stan Zdonik. 2008. Towards a Streaming SQL Standard. In *Conference on Very Large Data Bases (VLDB)*. 1379–1390.

Westley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Computing Surveys (CSUR)* 36, 1 (March 2004), 1–34.

Rohit Khandekar, Irsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. 2009. COLA: Optimizing Stream Processing Applications Via Graph Partitioning. In *International Conference on Middleware*. 308–327.

Monica Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Programming Language Design and Implementation (PLDI)*. 318–328.

Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain, and Joseph M. Lancaster. 2010. Deadlock-Avoidance for Streaming Applications with Split-Join Structure: Two Case Studies. In *Application-specific Systems Architectures and Processors (ASAP)*. 333–336.

Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moonm, and Chong Sang Kim. 1995. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering (TSE)* 21, 6 (July 1995), 593–604.

Bin Liu, Yali Zhu, and Elke A. Rundensteiner. 2006. Run-Time Operator State Spilling for Memory Intensive Long-Running Queries. In *International Conference on Management of Data (SIGMOD)*. 347–358.

Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005. Implementing Declarative Overlays. In *Symposium on Operating Systems Principles (SOSP)*. 75–90.

Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. 2009. A Performance Study of Event Processing Systems. In *TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC)*. 221–236.

Jeffrey C. Mogul and K. K. Ramakrishnan. 1997. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems (TOCS)* 15, 3 (Aug. 1997), 217–252.

Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. 1999. The Click Modular Router. In *Symposium on Operating Systems Principles (SOSP)*. 263–297.

David Mosberger and Larry L. Peterson. 1996. Making Paths Explicit in the Scout Operating System. In *Operating Systems Design and Implementation (OSDI)*. 153–167.

David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. 1996. Analysis of Techniques to Improve Protocol Processing Latency. In *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 73–84.

Chris Olston, Jing Jiang, and Jennifer Widom. 2003. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *International Conference on Management of Data (SIGMOD)*. 563–574.

Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *International Symposium on Microarchitecture (MICRO)*. 105–118.

Larry Page, Sergey Brin, R. Motwani, and T. Winograd. 1998. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford Digital Libraries Working Paper* (1998).

Peter Pietzuch, Jonathan Ledlie, Jeffrey Schneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *International Conference on Data Engineering (ICDE)*. 49–61.

Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-Stage Decoupled Software Pipelining. In *Code Generation and Optimization (CGO)*. 114–123.

Christopher Ré, Jérôme Siméon, and Mary F. Fernàndez. 2006. A Complete and Efficient Algebraic Compiler for XQuery. In *International Conference on Data Engineering (ICDE)*. 14–25.

Anton V. Riabov, Eric Bouillet, Mark D. Feblowitz, Zhen Liu, and Anand Ranganathan. 2008. Wishful Search: Interactive Composition of Data Mashups. In *International World Wide Web Conferences (WWW)*. 775–784.

Martin C. Rinard and Pedro C. Diniz. 1996. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. In *Programming Language Design and Implementation (PLDI)*. 54–67.

Scott Schneider, Henrique Andrade, Buğra Gedik, Alain Biem, and Kun-Lung Wu. 2009. Elastic Scaling of Data Parallel Operators in Stream Processing. In *International Parallel & Distributed Processing Symposium (IPDPS)*. 1–12.

Scott Schneider, Buğra Gedik, and Martin Hirzel. 2013. Tutorial: Stream Processing Optimizations. In *Conference on Distributed Event-Based Systems (DEBS)*. 249–258.

Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2012. Auto-Parallelizing Stateful Distributed Streaming Applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 53–64.

Philippe Selo, Yoonho Park, Sujay Parekh, Chitra Venkatramani, Hari K. Pyla, and Fang Zheng. 2010. Adding Stream Processing System Flexibility to Exploit Low-overhead Communication Systems. In *Workshop on High Performance Computational Finance (WHPCF)*. 1–8.

Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. 2005. Cache Aware Optimization of Stream Programs. In *Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*. 115–126.

Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. 2004. Highly Available, Fault-Tolerant, Parallel Dataflows. In *International Conference on Management of Data (SIGMOD)*. 827–838.

Adam Silberstein, Rebecca Braynard, and Jun Yang. 2006. Constraint Chaining: On Energy-Efficient Continuous Monitoring in Sensor Networks. In *International Conference on Management of Data (SIGMOD)*. 157–168.

Robert Soulé, Martin Hirzel, Buğra Gedik, and Robert Grimm. 2012. From a Calculus to an Execution Environment for Stream Processing. In *Conference on Distributed Event-Based Systems (DEBS)*. 20–31.

Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A Universal Calculus for Stream Processing Languages. In *European Symposium on Programming (ESOP)*. 507–528.

Robert Stephens. 1997. A Survey of Stream Processing. *Acta Informatica* 34, 7 (July 1997), 491–541.

Storm. 2013. Distributed and Fault-Tolerant Realtime Computation. Retrieved September 2013 from http://storm-project.net/.

Yuzhe Tang and Buğra Gedik. 2012. Auto-Pipelining for Data Stream Processing. In *Transactions on Parallel and Distributed Systems (TPDS)*. 10.1109/TPDS.2012.333.

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

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

William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *International Symposium on Microarchitecture (MICRO)*. 314–327.

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction (CC)*. 179–196.

Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2009. Software Pipelined Execution of Stream Programs on GPUs. In *Code Generation and Optimization (CGO)*. 200–209.

Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles (SOSP)*. 230–243.

R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated Empirical Optimizations of Software and the ATLAS Project. In *Parallel Computing (PARCO)*. 3–35.

Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. 2008. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. In *International Conference on Middleware*. 306–325.

Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-Performance Complex Event Processing over Streams. In *International Conference on Management of Data (SIGMOD)*. 407–418.

Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic Load Distribution in the Borealis Stream Processor. In *International Conference on Data Engineering (ICDE)*. 791–802.

Jianxin Xiong, Jeremy Johnson, Robert Johson, and David Padua. 2001. SPL: A Language and Compiler for DSP Algorithms. In *Programming Language Design and Implementation (PLDI)*. 298–308.

Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. 2003. A Comparison of Empirical and Model-Driven Optimization. In *Programming Language Design and Implementation (PLDI)*. 63–76.

Yuan Yu, Pradeep K. Gunda, and Michael Isard. 2009. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *Symposium on Operating Systems Principles (SOSP)*. 247–260.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Operating Systems Design and Implementation (OSDI)*. 1–14.