RC25215 (W1109-134) September 28, 2011
Computer Science

IBM Research Report

A Catalog of Stream Processing Optimizations

Martin Hirzel[1], Robert Soulé[2], Scott Schneider[1], Bugra Gedik[1], Robert Grimm[2]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[2]New York University

A Catalog of Stream Processing Optimizations

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

Various research communities have independently arrived at stream processing as a programming model for high-performance and parallel computation, including digital signal processing, databases, operating systems, and complex event processing. Each of these communities has developed some of the same optimizations, but often with conflicting terminology and unstated assumptions. This paper 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 trade-offs. To consolidate terminology, this paper is organized as a catalog, in a style similar to catalogs of design patterns or refactorings. To make assumptions explicit and help with trade-offs, 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 optimization inventors to stand on the shoulders of giants from not just their own community.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; H.2.4 [**Systems**]: QueryProcessing

General Terms: Languages

Additional Key Words and Phrases: Stream Processing, Optimizations

ACM Reference Format:
Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R. 2011. A Catalog of Stream Processing Optimizations. ACM Comput. Surv. V, N, Article A (January YYYY), 29 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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* conceptually has its own thread of control. 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, complex even processing, and databases.

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; Abadi et al. 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 for 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

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.	Load balancing	unchanged	unchanged	(depends)
8.	Placement	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

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 paper is a catalog of the stream processing optimizations listed in Table I.

Besides consistent terminology, another motivation for this paper is unstated assumptions: certain communities take things for granted that other communities do not. For example, while the StreamSQL community assumes that stream graphs are forests (acyclic sets of trees), StreamIt assumes that stream graphs are possibly cyclic single-entry, single-exit regions. We have observed stream graphs in practice that fit neither mold. Additionally, several papers focus on one aspect of a problem, such as formulating a mathematical model for the profitability trade-offs 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, other papers assume a distributed system, where state sharing is difficult and communication is more expensive, since it involves the network. This paper describes optimizations for many different kinds of streaming systems, including shared-memory and distributed, relational and synchronous, among other variations. For each optimization, this paper explicitly lists both safety and profitability considerations.

The target audience of this paper includes end users, system implementers, and researchers. For end users, this paper helps understand performance phenomenons, and helps users to guide the automatic optimizer, or in the worst case, to hand-optimize their applications. For system implementers, this paper suggests ideas for what optimizations the system may want to support, illustrates the engineering trade-offs, and provides starting points for digging deeper into the literature. For researchers, this paper helps judge the novelty of ideas, use consistent terminology, and anticipate interactions of optimizations with each other and with other concerns.

Each section is structured as follows:

— *Tag-line and figure,* giving a quick intuition for what the optimization does.
— *Example,* describing 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 optimization paint a picture of the landscape of modern stream processing domains and applications.
— *Profitability,* describing the conditions under which the optimization improves performance. To illustrate the main trade-offs in a concrete and realistic manner, each profitability subsection is based on a micro-benchmark. All experiments were done on a real stream processing system, and each chart shows error bars indicating the

standard deviation. The micro-benchmarks serve as an existence proof for a case where the optimization improves performance. It can also serve as a blue-print for testing the optimization in a new application or system.
— *Safety,* listing 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, 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,* surveying 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,* identifying established approaches for applying the optimization statically or dynamically, i.e., before the application starts or during runtime.

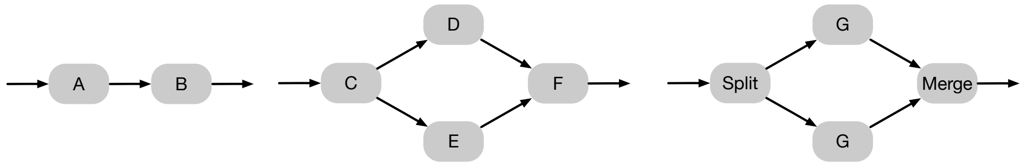
Existing surveys on stream processing do not focus on optimizations [Stephens 1997; Babcock et al. 2002; Johnston et al. 2004], and existing catalogs of optimizations do not focus on stream processing. This paper provides both: it presents a catalog of stream processing optimizations, and makes them approachable to users, implementers, and researchers.

1.1. Background

This section clarifies terminology used in this paper. 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 have different notions of data items, including samples in digital signal processing, tuples in databases, or events in complex event processing; this paper 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 *multi-tenancy*, where a single streaming system runs multiple applications that come and go.

An *operator* is a continuous stream transformer: each operator conceptually has its own thread of control, and it transforms its input streams to its output streams. 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*. Most split or merge operators forward data items unmodified, but a relational *join* is an example for a merge operator that includes a non-trivial transformation.

It is often useful to use specific terminology for the various flavors of parallelism among the operators in a stream graph. Fig. 1 illustrates these flavors. *Pipeline parallelism* is the concurrent execution of a producer A with a consumer B. *Task parallelism*



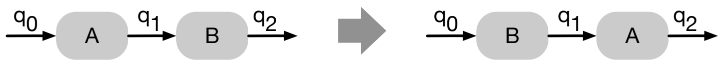
(a) Pipeline-parallel A || B. (b) Task-parallel D || E. (c) Data-parallel G || G.

Fig. 1. Pipeline, task, and data parallelism in stream graphs.

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. The architecture community refers to data parallelism as SIMD (single instruction, multiple data).

2. OPERATOR REORDERING (A.K.A. HOISTING, SINKING, ROTATION, PUSHDOWN)

Move more selective operators upstream to filter data early.

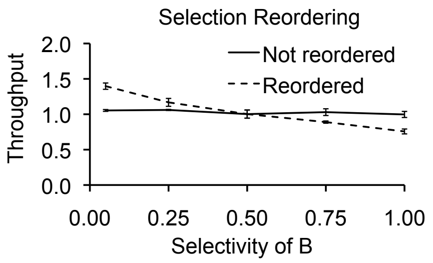


2.1. Example

Consider a healthcare 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. Hoisting 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 drops 70% of all data items outputs only 30% and thus has selectivity 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. If A comes before B, then independently of the selectivity of B, 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 depends on 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

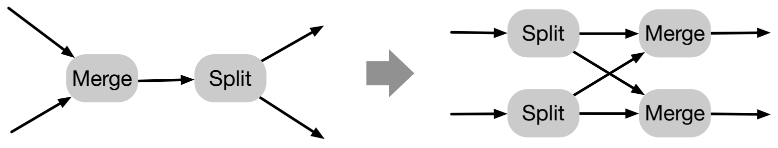
Operator reordering is safe if the following conditions hold:

— *Ensure 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. 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.
— *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.

2.4. Variations

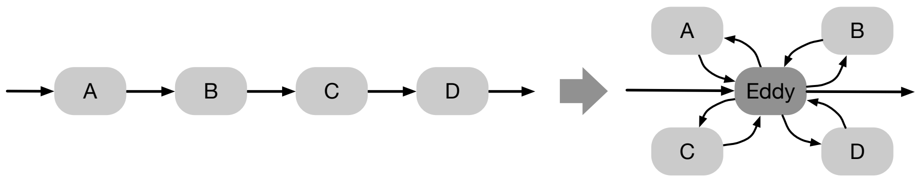
Algebraic reorderings. Operator reordering is popular in streaming systems built around the relational model, such as the STREAM system [Arasu et al. 2006]. These systems establish the safety of reordering based on the formal semantics of relational operators, using algebraic equivalences between different 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 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. Some streaming systems reorder operators based on their own 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]. Finally, 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.

Synergies with other optimizations. While operator reordering yields benefits on its own, it also interacts with several of the streaming optimizations cataloged in the rest of this paper. 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_1 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. Fission (Section 6) introduces parallel segments; when two parallel segments are back-to-back, reordering the Merge and Split eliminates a serialization bottle-neck, as in the Exchange operator in Volcano [Graefe 1990]. The following figure illustrates this Split/Merge rotation:



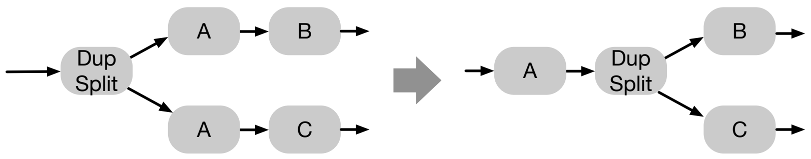
2.5. Dynamism

The optimal ordering of operators is often dependent on the input data. Therefore, it is useful to be able to change the 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 figure below, an Eddy operator is connected to every other operator in the pipeline, and dynamically routes data after measuring which ordering would be the most profitable.



3. REDUNDANCY ELIMINATION (A.K.A. SUBGRAPH SHARING, MULTI-QUERY OPTIMIZATION)

Eliminate redundant computations.

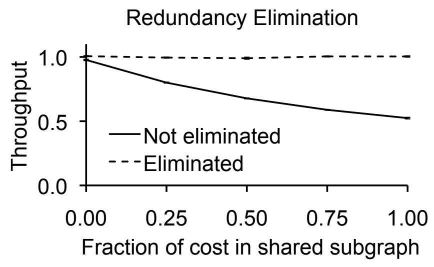


3.1. Example

Consider two telecommunications applications, one of which continuously updates billing information, and the other continuously monitors for network problems. Both applications start with an operator A that deduplicates CDRs (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 it to save 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. For example, if both of them compute an average, but one uses the arithmetic mean and the other the geometric mean, they cannot be shared.
—*Ensure available data.* If the redundant operators A are stateful, then they must operate on the exact same stream of input data. If they are stateless, then they must operate on subsets of a common superset of input data. For example, if both input streams are CDRs (call-data records), but from different geographies, then the operators can only be shared if they are stateless.

3.4. Variations

Multi-tenancy. Redundant subgraphs as described above 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 fruitful domain in its own right, 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 (non-deterministic finite automaton) [Diao et al. 2002].

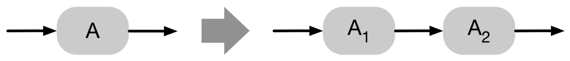
Other approaches for eliminating operators. Besides the sophisticated techniques for collapsing similar or identical subgraphs, there are other, more mundane ways to remove an operator from a stream graph. An optimizer can remove a no-op, i.e., an operator that has no effect, such as a projection that 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 chose to disable a subgraph for debugging purposes, or a library may produce multiple outputs, some of which are unused 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 multi-tenancy 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 sub-graph detection, as done e.g. in [Pietzuch et al. 2006]. Some systems take this 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)

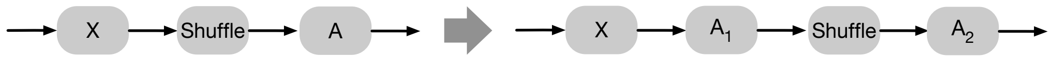
Separate operators into smaller computational steps.



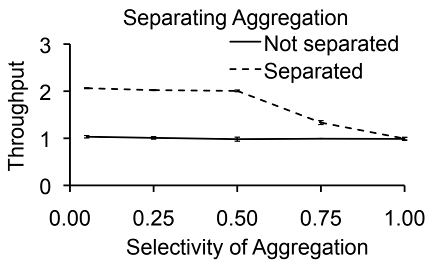
4.1. Example

Consider a retail application that continuously watches public discussion forums to discover when users express negative sentiments about a company's products. The application would contain a sentiment-extraction operator that analyzes natural-language text to score its sentiment, and an operator A that 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) could 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 leads to beneficial pipeline parallelism, or if it enables other optimizations, particularly operator reordering. Consider an application that consists of a first parallel segment X, a Shuffle operator, and a second parallel segment with an aggregation operator A. The cost of the first segment is negligible, and the cost of the second segment consists of a cost of 0.5 for Shuffle plus a cost of 0.5 for A. Therefore, throughput is limited by the second segment. With operator separation and reordering, the end of the first parallel segment performs a pre-aggregation A_1 of cost 0.5 before the Shuffle. That increases the normalized cost of the first segment, but it reduces the cost of the second segment based on the selectivity of the pre-aggregation. When the selectivity approaches 1, that means that no data items are dropped, and the second segment performs as much work as in the non-separated case. Due to pipeline parallelism, the overall throughput of the application is bounded by the slower of the two segments, which is why the throughput curves touch at selectivity 1.



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 $B_2(B_1(s)) = B(s)$. Establishing this equivalence in the general case is tricky, but 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 "and", 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 that are based on the relational model optimize based on them [Arasu et al. 2006]. Beyond this algebraic approach, MapReduce can separate the Reduce stage to get a separate Combine operator [Dean and Ghemawat 2004]. 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 average or sum), and the compiler can infer decomposability for expressions that call them (such as a constructor that uses different aggregators for different attributes).

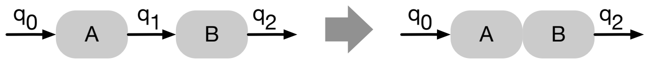
Separation by analysis. Separating arbitrary imperative code is a difficult analysis problem. In the compiler community, this has become known as DSWP (decoupled software pipelining [Ottoni et al. 2005]). In contrast to traditional SWP (software pipelining [Lam 1988]), which increases instruction-level 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.

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

Avoid the overhead of data serialization and transport.

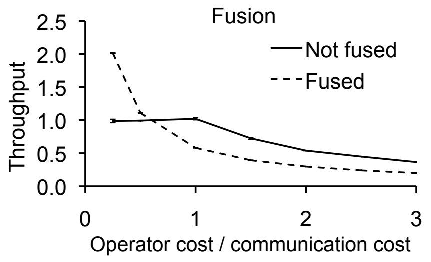


5.1. Example

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. The selection operator B is light-weight 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 light-weight, 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, they could have pipeline parallelism: the upstream operator can already work 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 non-fused operators. When the operators are not fused, there are two cases: if operator cost is lower than communcation cost, then throughput is bounded by communication cost, else 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 is half of the communication cost, because the fused operator is 2× 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, such as local files or 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 non-fused 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. 2008; Khandekar 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. Fusion can also open up opportunities for algorithm selection (see Section 11). For instance, when SASE fuses a sequence-scan operator with a window operator, it combines them such that the sequence scan produces fewer intermediate results to begin with [Wu et al. 2006].

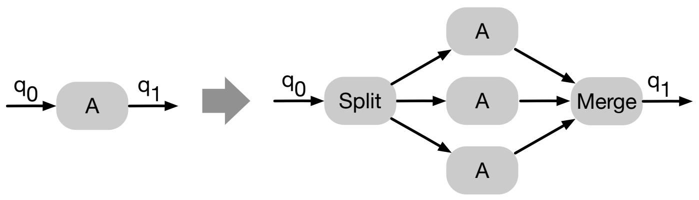
Multi-threaded fusion. Instead of combining the fused operators in the same thread of control, fusion may just combine them in the same address space, but separate threads of control. That yields the benefits of reduced communication cost, without giving up pipeline parallelism. The fused operators communicate data items through a shared buffer. In the general case, use of a shared buffer implies copying data items. However, in the cases of operators that do not mutate their data items or operators that can synchronize their access to the buffer, this copying overhead can be eliminated.

5.5. Dynamism

Fusion is most commonly done statically. However, the Flextream system performs dynamic fusion by halting the application, re-compiling 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. 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.

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

Parallelize computations.

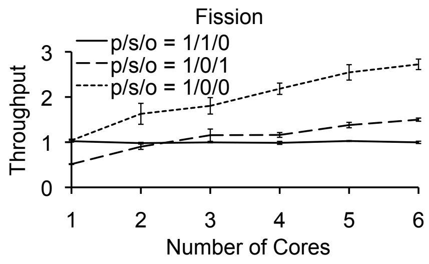


6.1. Example

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 segment 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 bottle-neck 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. 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 duplicated 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 duplicated strictly on partitioning boundaries. *Partitioned state* is when an operator maintains disjoint state based on a particular key attribute of each data item. 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 operator itself that determines whether ordering matters. Rather, it is the downstream operators that consume the operator's data items. If an operation 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 but the operator itself is commutative, then the transformation must ensure that the output data is combined in the same order that the input data was partitioned. There are various approaches for re-establishing the right order, if required. CQL uses logical timestamps [Arasu et al. 2006]. StreamIt uses round-robin or duplication [Gordon

et al. 2006]. And MapReduce, instead of re-establishing the old order, uses a distributed "sort" stage [Dean and Ghemawat 2004].
— *Obey licensing constraints.* If the software in an operator is only licensed for a single host, fission must not place replicas on multiple hosts.

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, and operators process data in a single pass, but streams are finite. In other words, streams are an implementation techniques for relational databases, and fission for this style of database dates back at least to Volcano [Graefe 1990] and Gamma [Dewitt et al. 1990]. This form of fission applies to stateful operators, as long as the state is grouped by keys. It may at first appear strange to simply refer to fission as "parallelization", since by default, 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. That is why, more recently, this form of fission by keys for large batch jobs has been the center-piece of 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 segments.

Fission for infinite streams. Continuous streaming applications have conceptually infinite streams. 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 that are either 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. Besides StreamIt, there is other work on fission for infinite streams, which is discussed below under dynamism. In most systems, the streaming language is designed explicitly for fission, making it easy for the compiler to establish safety. When the language is not designed for fission, safety must be established either by static or by dynamic dependence analysis. An example for a static analysis that discovers fission opportunities is parallel-stage decoupled software pipelining [Raman et al. 2008]. An example for a dynamic analysis that discovers fission opportunities is [Thies et al. 2007].

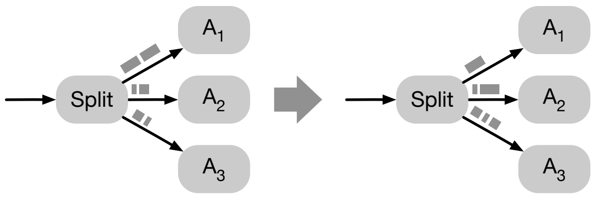
6.5. Dynamism

To make the *profitability* decision for fission dynamic, we need to dynamically adjust the width of the parallel segment, 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 STM (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 order.

7. LOAD BALANCING

Distribute workload evenly across resources.

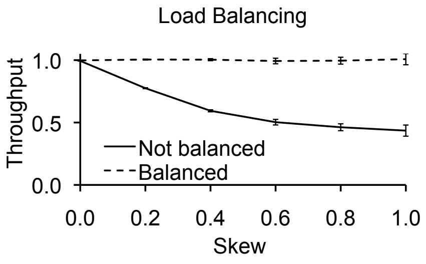


7.1. Example

Consider a security application that continuously checks that outgoing messages from a hospital do not reveal confidential patient information. The application uses natural-language processing algorithms to check whether outgoing messages contain text that could reveal confidential information, such as social security numbers or medical conditions, to unauthorized people. Assume that the operator A that does this is expensive, and therefore, 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.

7.2. Profitability

Load balancing is profitable if it compensates for skew in resource utilization. The chart shows the results of an experiment consisting of a Split operator that sends data to three replicated operators. There are two variants of the experiment: in one variant, the Split operator sends its data items to downstream operators using round-robin scheduling, i.e., not balanced. In the second variant, the Split operator sends its data items to whichever downstream operator is free, i.e., balanced. The chart compares the throughput as a function of the skew, i.e., the variation in the amount of time it takes to process each data item (either due to changes in the data item size or load of the downstream operator's CPU).



7.3. 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 a common shared state.

—*Establish placement safety.* If load balancing is done while placing operators, the safety conditions from Section 8 must be met.

7.4. 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].

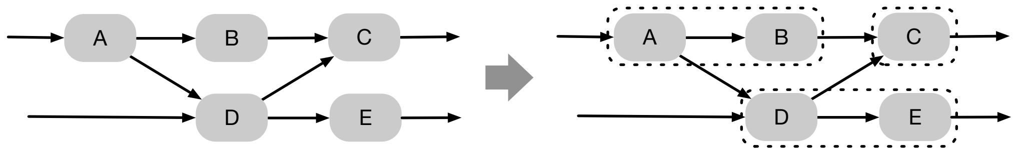
Balancing load while assigning work to operators. Instead of balancing load by deciding how to arrange the operators, an alternative approach is to first use fission (Section 6) to replicate operators, and then balance load by deciding how much streaming data each replica needs to process. The distributed 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, the MapReduce Online paper argues that the push-based approach works better for streaming [Condie et al. 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 re-arranging work even after it has been pushed or pulled to operators [Blumofe et al. 1995].

7.5. Dynamism

Roughly speaking, the placement-based variants of load balancing tend to be static, whereas the work-based variants of load balancing 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 performance hick-ups; if load spikes are sudden, changing the placement may take too long; and migrating a stateful operator is an engineering challenge.

8. PLACEMENT (A.K.A. LAYOUT)

Assign operators to hosts and cores.

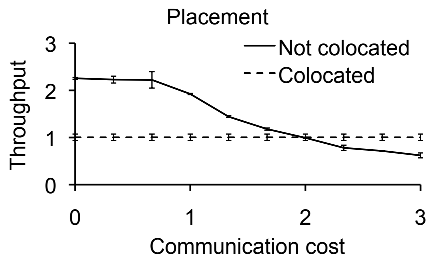


8.1. Example

Consider a telecommunications application that continuously computes billing information for long-distance calls. The input stream consists of call-data records (CDRs). The example has three operators: operator A cleans incoming CDRs, 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. We assume that operators A and C are both expensive, and therefore, it makes sense to place them on different hosts. On the other hand, 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.

8.2. 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 multi-core machine, so the operators do not compete for CPU. When communication cost is low, the throughput is higher 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.



8.3. 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, certain operators might involve code compiled for an FPGA, and must therefore only 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 licensing and security restrictions.* Besides resource constraints, placement can also be restricted by licensing (where a software package can only be installed on a certain number of hosts) and security (where certain operators can only run on trusted 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 the switch-over is clean without data loss. Depending on the system, this may only be possible for certain operators, for instance, operators without state.

8.4. Variations

Placement for load balancing. Section 7 discussed placement algorithms that focus primarily on load balancing [Xing et al. 2005; Amini et al. 2006].

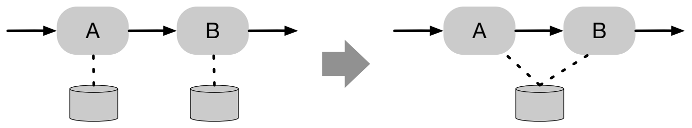
Placement for other constraints. While load balancing is usually at least part of the consideration for placement, often other constraints complicate the problem. Pietzuch et al. present a decentralized placement algorithm for a geographically distributed streaming system, where some operators are geographically 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. 2008]. When StreamIt is compiled to a communication-exposed architecture, 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].

8.5. 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. Those algorithms assume that the system provides a mechanism for migrating operators between hosts [Xing et al. 2005; Pietzuch et al. 2006].

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

Optimize for space by avoiding unnecessary copies of data.

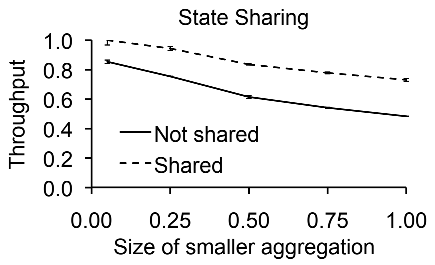


9.1. Example

Consider a financial application that continuously computes the volume-weighted average price (VWAP) of stocks for both a full day and a half day. This application will have to maintain 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 if it reduces stalls due to cache misses, by decreasing the memory footprint and/or avoiding redundant accesses to the same data. The chart shows the results of an experiment with two aggregation operators, both acting on the same stream of data. The difference between the aggregations is their granularity: the window in one of the aggregations is a fraction of the size of the window in the other aggregation. Since the state required for the smaller window is a subset of that required for the larger one, both operators can use the same memory. Even as the size of the smaller aggregation window increases, state sharing remains profitable because the aggregation window is traversed only once.



9.3. Safety

State sharing is safe if the following conditions hold:

— *Provide common access to state.* The operators that share the state 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 is it allowed to grow without bounds, i.e., leak.

9.4. Variations

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 avoid interference and extra data copies, the queue must have a capacity of at least two data items; therefore, this variant is sometimes called double-buffering. Note that the operators themselves can be stateless. 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.

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 easy to share [Gordon et al. 2006]. CQL implements windows by non-shared array of pointers to shared data items, such that a single data item might be pointed to from multiple windows and even queues [Arasu et al. 2006].

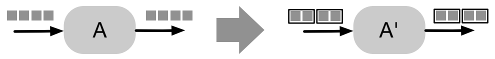
Shared operator state:. This variant deals with operators that have non-trivial state, not just simple windows. It imposes the most challenging requirements on synchronization and memory management. For example, the operators may use STM (software transactional memory) to manage shared data representing a table or a graph [Brito et al. 2008].

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, there is some variation in how much of the synchronization and memory management is decided at runtime. StreamIt uses a fully-static approach, where a static schedule prescribes exactly what data can be accessed by which operator at a particular time [Sermulins et al. 2005]. Brito et al.'s work is more dynamic, where access to shared state is reconciled by STM (software transactional memory) [2008].

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

Process multiple data items in a single batch.

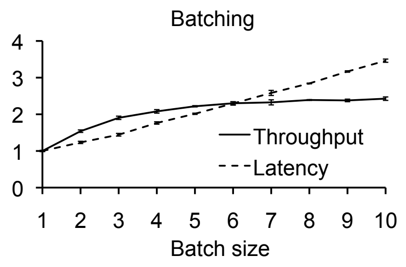


10.1. Example

Consider a healthcare application that repeatedly fires an FFT (Fast Fourier Transform) operator for medical imaging. Efficient FFT implementations contain enough code such that instruction cache locality becomes 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 in 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 against latency. Batching can improve throughput by amortizing operator-firing 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 trade-off 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 out 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, which might happen when the stream-graph is cyclic, or when the batched operator shares a lock with downstream operators.
— *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.

10.4. Variations

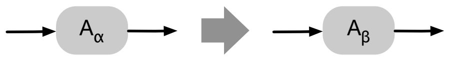
Batching is a streaming optimization that plays well into the hands of more traditional (not necessarily streaming) compiler optimizations. In particular, batching gives rise to loops, and the compiler may optimize these loops with unrolling or with software pipelining [Lam 1988]. The compiler for a 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 i-cache benefits of batching against the d-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, making the system react to changing load conditions.

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

Use a faster algorithm for implementing an operator.

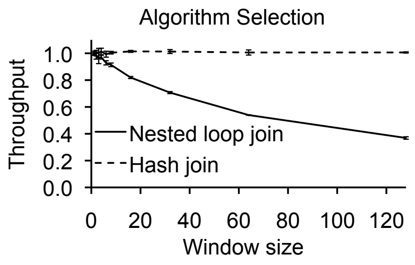


11.1. Example

Consider a transportation application that, for tolling purposes, continuously monitors which vehicles are currently on congested road segments. The application has 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 trade-offs between performance and generality: algorithm A_α may be faster, but algorithm A_β may be more general. The chart compares throughput of a nested loop join vs. 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. For example, hash join is less general than nested-loop join, and can only be chosen when the join condition is an equality.

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, where the algorithmic choice is between weaving certain operators into sequence construction vs. keeping them as separate pipeline stages [Wu et al. 2006].

Auto-tuners. Outside of streaming systems, there are several successful software packages that perform “empirical optimization”. The idea is that 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].

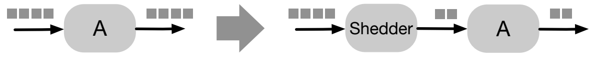
Different semantics. Algorithm selection can be used as a simple form of load shedding. While most approaches to load shedding work by dropping data items (as described in Section 12), load shedding by algorithm selection merely switches to a cheaper implementation. Unlike the other variations of algorithm 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 below.

11.5. Dynamism

When algorithm selection is used to react to runtime conditions, it must be dynamic. For instance, as mentioned above, algorithm selection can be used to implement dynamic load shedding. 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.

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

Degrade gracefully when overloaded.

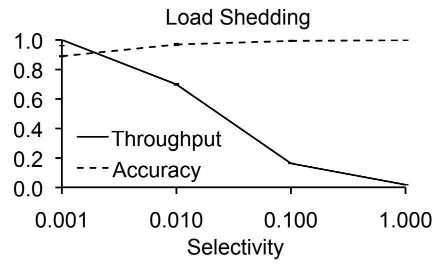


12.1. Example

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 a large 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 aggregator that constructs a histogram by counting 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 one out of every thousand data items, i.e., when selectivity is 0.001, 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 paper, 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.

12.4. Variations

Load shedding in network applications. Network stacks and web servers are vulnerable to load spikes, and load shedding has been a prime motivator for implementing them as graphs of streams and operators. The Scout operating system uses paths in drop data items early if it can be predicted that they will miss their deadline by the time they make their way through a path of streaming operators [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]. Gedik et al. propose going one step further, by moving the load shedders out of the streaming system entirely and onto the sensors that produce the data in the first place [2008].

12.5. Dynamism

This optimization 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 individual 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, 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.

— *Non-streaming 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, like in decoupled software pipelining [Ottoni et al. 2005].
— *Annotated non-streaming language.* This approach adds pragmas to indicate streams in a traditional language like C or Fortran. An example is Brook [Buck 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. Examples are common in the systems community, e.g., SEDA [Welsh et al. 2001].
— *Graph, specified textually.* Some streaming languages allow the user to specify the stream graph directly in terms of operators and streams. An example is SPADE [Gedik et al. 2008].
— *Graph, specified visually.* Instead of specifying the stream graph textually in a language, some systems, such as Aurora, provide a visual environment for that instead [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, CQL [Arasu et al. 2006].
— *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 (CEP) 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].

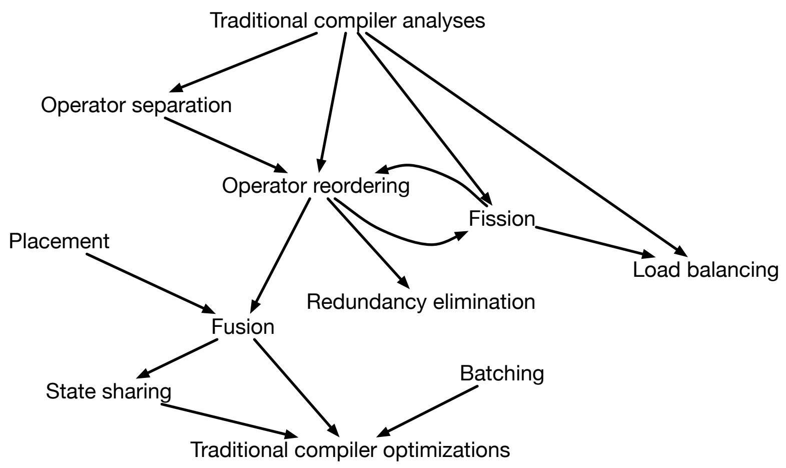


Fig. 2. Interactions of streaming optimizations with each other and with traditional compilers. An edge from X to Y indicates that X can help to enable Y.

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 over-simplified, 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 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 characteristic 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.

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. There is a circular enablement between operator reordering and fission: operator reordering enables more effective fission by bringing operators together that can be part of the same parallel segment, whereas fission enables reordering of the split and merge operators that fission inserts. In addition, fission makes it easier to balance load, because it introduces data parallelism, which tends to be more homogeneous and malleable than pipeline or task parallelism. A streaming system that implements multiple of these optimizations is well advised to apply them in some order consistent with the direction of the edges in Figure 2.

Avenues for future work. Phase ordering 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.

13.3. How streaming optimizations interact with traditional compilers

By *traditional compiler*, we refer to compilers for languages like 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].
— *Operator separation* can be supported by compiler analysis for decoupled software pipelining (DSWP) [Ottoni et al. 2005].
— *Fission* can be supported by compiler analysis for parallel-stage DSWP [Ottoni et al. 2005].
— *Load balancing* 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.

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.
— *State sharing* enables scalar replacement [Sermulins et al. 2005].
— *Batching* enables loop unrolling and/or software pipelining [Lam 1988].

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].

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 making sure the compiler actually follows up on information given to it by the streaming language and its optimizations.

13.4. Dynamic optimization for streaming systems

Several streaming optimizations have both static and dynamic variants. Table I summarizes this, 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 optimize either at submission time, or periodically re-run 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 flexibilities, 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 trade-offs. 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 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. 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 [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, or 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 trade-off between throughput and latency seen in batching, fission, and operator separation; the trade-off between throughput and QoS or accuracy in load shedding; and the trade-off between throughput and power in fission. As a concrete example for such trade-offs, *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 [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 multi-metric optimizers.

14. CONCLUSION

This paper 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 paper 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.

REFERENCES

ABADI, D. J., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. 2005. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*. 277–289.

ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2003. Aurora: A new model and architecture for data stream management. *The VLDB Journal 12,* 2 (Aug.), 120–139.

AMINI, L., JAIN, N., SEHGAL, A., SILBER, J., AND VERSCHEURE, O. 2006. Adaptive control of extreme-scale stream processing systems. In *International Conference on Distributed Computing Systems (ICDCS)*.

ARASU, A., BABU, S., AND WIDOM, J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal 15,* 2 (June), 121–142.

ARPACI-DUSSEAU, R. H., ANDERSON, E., TREUHAFT, N., CULLER, D. E., HELLERSTEIN, J. M., PATTERSON, D., AND YELICK, K. 1999. Cluster I/O with River: Making the fast case common. In *Workshop on I/O in Parallel and Distributed Systems (IOPADS)*. 10–22.

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously adaptive query processing. In *International Conference on Management of Data (SIGMOD)*. 261–272.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Principles of Database Systems (PODS)*. 1–16.

BABCOCK, B., DATAR, M., AND MOTWANI, R. 2004. Load shedding for aggregation queries over data streams. In *International Conference on Data Engineering (ICDE)*. 350–361.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: an efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming (PPoPP)*. 207–216.

BRITO, A., FETZER, C., STURZREHM, H., AND FELBER, P. 2008. Speculative out-of-order event processing with software transaction memory. In *Conference on Distributed Event-Based Systems (DEBS)*. 265–275.

BUCK, I. ET AL. 2004. Brook for GPUs: Stream computing on graphics hardware. In *Computer Graphics and Interactive Techniques (SIGGRAPH)*. 777–786.

BURCHETT, K., COOPER, G. H., AND KRISHNAMURTHI, S. 2007. Lowering: a static optimization technique for transparent functional reactivity. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. 1–80.

CARNEY, D., CETINTEMEL, U., RASIN, A., ZDONIK, S., CHERNIACK, M., AND STONEBRAKER, M. 2003. Operator scheduling in a data stream manager. In *Very Large Data Bases (VLDB)*. 309–320.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *International Conference on Management of Data (SIGMOD)*. 379–390.

CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. 2010. MapReduce Online. In *Networked Systems Design and Implementation (NSDI)*. 313–328.

DEAN, J. AND GHEMAWAT, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Operating Systems Design and Implementation (OSDI)*. 137–150.

DEWITT, D. AND GRAY, J. 1992. Parallel database systems: the future of high performance database systems. *Communications of the ACM (CACM) 35,* 6, 85–98.

DEWITT, D. J., GHANDEHARIZADEH, S., SCHNEIDER, D. A., BRICKER, A., HSIAO, H. I., AND RASMUSSEN, R. 1990. The Gamma database machine project. *Transations on Knowledge and Data Engineering (TKDE) 2*, 44–62.

DIAO, Y., FISCHER, P. M., FRANKLIN, M. J., AND TO, R. 2002. YFilter: Efficient and scalable filtering of XML documents. In *Demo at International Conference on Data Engineering (ICDE-Demo)*.

FORGY, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence 19*, 17–37.

FOWLER, M., BECK, K., BRANT, J., AND OPDYKE, W. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

FRIGO, M. AND JOHNSON, S. G. 1998. FFTW: an adaptive software architecture for the FFT. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1381–1384.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GARCIA-MOLINA, H., ULLMAN, J. D., AND WIDOM, J. 2008. *Database Systems: The Complete Book*, Second ed. Prentice Hall.

GEDIK, B., ANDRADE, H., WU, K.-L., YU, P. S., AND DOO, M. 2008. SPADE: The System S declarative stream processing engine. In *International Conference on Management of Data (SIGMOD)*. 1123–1134.

GEDIK, B., WU, K.-L., AND YU, P. S. 2008. Efficient construction of compact shedding filters for data stream processing. In *International Conference on Data Engineering (ICDE)*. 396–405.

GORDON, M. I., THIES, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 151–162.

GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LAMB, A. A., LEGER, C., WONG, J., HOFFMANN, H., MAZE, D., AND AMARASINGHE, S. 2002. A stream compiler for communication-exposed architectures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 291–303.

GRAEFE, G. 1990. Encapsulation of parallelism in the Volcano query processing system. In *International Conference on Management of Data (SIGMOD)*. 102–111.

HORMATI, A., CHOI, Y., KUDLUR, M., RABBAH, R. M., MUDGE, T. N., AND MAHLKE, S. A. 2009. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Parallel Architectures and Compilation Techniques (PACT)*. 214–223.

ISARD, M., YU, M. B. Y., BIRRELL, A., AND FETTERLY, D. 2007. Dryad: Distributed data-parallel program from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*. 59–72.

JOHNSTON, W. M., HANNA, J. R. P., AND MILLAR, R. J. 2004. Advances in dataflow programming languages. *ACM Computing Surveys 36*, 1–34.

KHANDEKAR, R., HILDRUM, I., PAREKH, S., RAJAN, D., WOLF, J., WU, K.-L., ANDRADE, H., AND GEDIK, B. 2009. COLA: Optimizing stream processing applications via graph partitioning. In *International Conference on Middleware*. 308–327.

LAM, M. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Programming Language Design and Implementation (PLDI)*. 318–328.

LIM, S.-S., BAE, Y. H., JANG, G. T., RHEE, B.-D., MIN, S. L., PARK, C. Y., SHIN, H., PARK, K., MOONM, S.-M., AND KIM, C. S. 1995. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering (TSE) 21,* 6.

LOO, B. T. ET AL. 2005. Implementing declarative overlays. In *Symposium on Operating Systems Principles (SOSP)*. 75–90.

MENDES, M. R. N., BIZARRO, P., AND MARQUES, P. 2009. A performance study of event processing systems. In *TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC)*. 221–236.

MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. 1999. The Click modular router. In *Symposium on Operating Systems Principles (SOSP)*. 263–297.

MOSBERGER, D. AND PETERSON, L. L. 1996. Making paths explicit in the Scout operating system. In *Operating Systems Design and Implementation (OSDI)*. 153–167.

MOSBERGER, D., PETERSON, L. L., BRIDGES, P. G., AND O'MALLEY, S. 1996. Analysis of techniques to improve protocol processing latency. In *Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. 73–84.

OTTONI, G., RANGAN, R., STOLER, A., AND AUGUST, D. I. 2005. Automatic thread extraction with decoupled software pipelining. In *International Symposium on Microarchitecture (MICRO)*. 105–118.

PIETZUCH, P., LEDLIE, J., SCHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. 2006. Network-aware operator placement for stream-processing systems. In *International Conference on Data Engineering (ICDE)*. 49–61.

RAMAN, E., OTTONI, G., RAMAN, A., BRIDGES, M. J., AND AUGUST, D. I. 2008. Parallel-stage decoupled software pipelining. In *Code Generation and Optimization (CGO)*. 114–123.

RÉ, C., SIMÉON, J., AND FERNÀNDEZ, M. F. 2006. A complete and efficient algebraic compiler for XQuery. In *International Conference on Data Engineering (ICDE)*. 14–25.

RIABOV, A. V., BOUILLET, E., FEBLOWITZ, M. D., LIU, Z., AND RANGANATHAN, A. 2008. Wishful search: Interactive composition of data mashups. In *International World Wide Web Conferences (WWW)*. 775–784.

RINARD, M. C. AND DINIZ, P. C. 1996. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Programming Language Design and Implementation (PLDI)*. 54–67.

SCHNEIDER, S., ANDRADE, H., GEDIK, B., BIEM, A., AND WU, K.-L. 2009. Elastic scaling of data parallel operators in stream processing. In *International Parallel & Distributed Processing Symposium (IPDPS)*. 1–12.

SELO, P., PARK, Y., PAREKH, S., VENKATRAMANI, C., PYLA, H. K., AND ZHENG, F. 2010. Adding stream processing system flexibility to exploit low-overhead communication systems. In *Workshop on High Performance Computational Finance (WHPCF)*.

SERMULINS, J., THIES, W., RABBAH, R., AND AMARASINGHE, S. 2005. Cache aware optimization of stream programs. In *Languages, Compiler, and Tool Support for Embedded Systems (LCTES)*. 115–126.

SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. 2004. Highly Available, Fault-Tolerant, Parallel Dataflows. In *International Conference on Management of Data (SIGMOD)*. 827–838.

STEPHENS, R. 1997. A survey of stream processing. *Acta Informatica 34,* 7 (July), 491–541.

TATBUL, N., CETINTEMEL, U., ZDONIK, S., CHERNIACK, M., AND STONEBRAKER, M. 2003. Load shedding in a data stream manager. In *Very Large Data Bases (VLDB)*. 309–320.

THIES, W. AND AMARASINGHE, S. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In *Parallel Architectures and Compilation Techniques (PACT)*. 365–376.

THIES, W., CHANDRASEKHAR, V., AND AMARASINGHE, S. 2007. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *International Symposium on Microarchitecture (MICRO)*. 314–327.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction (CC)*. 179–196.

WELSH, M., CULLER, D., AND BREWER, E. 2001. SEDA: An architecture for well-conditioned, scalable Internet services. In *Symposium on Operating Systems Principles (SOSP)*. 230–243.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimizations of software and the ATLAS project. In *Parallel Computing (PARCO)*.

WOLF, J., BANSAL, N., HILDRUM, K., PAREKH, S., RAJAN, D., WAGLE, R., WU, K.-L., AND FLEISCHER, L. 2008. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *International Conference on Middleware*. 306–325.

WU, E., DIAO, Y., AND RIZVI, S. 2006. High-performance complex event processing over streams. In *International Conference on Management of Data (SIGMOD)*. 407–418.

XING, Y., ZDONIK, S., AND HWANG, J.-H. 2005. Dynamic load distribution in the Borealis stream processor. In *International Conference on Data Engineering (ICDE)*. 791–802.

XIONG, J., JOHNSON, J., JOHSON, R., AND PADUA, D. 2001. SPL: A language and compiler for DSP algorithms. In *Programming Language Design and Implementation (PLDI)*. 298–308.

YOTOV, K., LI, X., REN, G., CIBULSKIS, M., DEJONG, G., GARZARAN, M., PADUA, D., PINGALI, K., STODGHILL, P., AND WU, P. 2003. A comparison of empirical and model-driven optimization. In *Programming Language Design and Implementation (PLDI)*. 63–76.

YU, Y., GUNDA, P. K., AND ISARD, M. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Symposium on Operating Systems Principles (SOSP)*. 247–260.

YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. 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.