From a Calculus to an Execution Environment for Stream Processing

Robert Soulé
New York University
soule@cs.nyu.edu

Martin Hirzel
IBM Research
hirzel@us.ibm.com

Buğra Gedik
Bilkent University
bgedik@cs.bilkent.edu.tr

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Abstract

At one level, this paper is about River, a virtual execution environment for stream processing. Stream processing is a paradigm well-suited for many modern data processing systems that ingest high-volume data streams from the real world, such as audio/video streaming, high-frequency trading, and security monitoring. One attractive property of stream processing is that it lends itself to parallelization on multicores, and even to distribution on clusters when extreme scale is required. Stream processing has been co-evolved by several communities, leading to diverse languages with similar core concepts. Providing a common execution environment reduces language development effort and increases portability. We designed River as a practical realization of Brooklet, a calculus for stream processing. So at another level, this paper is about a journey from theory (the calculus) to practice (the execution environment). The challenge is that, by definition, a calculus abstracts away all but the most central concepts. Hence, there are several research questions in concretizing the missing parts, not to mention a significant engineering effort in implementing them. But the effort is well worth it, because using a calculus as a foundation yields clear semantics and proven correctness results.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—*parallel languages*; D.3.4 [*Programming Languages*]: Processors—*compilers*

Keywords Stream Processing, Domain Specific Language, Intermediate Language, CQL, Sawzall, StreamIt

1. Introduction

It is widely accepted that virtual execution environments help programming languages by decoupling them from the target platform and vice versa. At its core, a virtual execution environment provides a small interface with well-defined behavior, facilitating robust, portable, and economic language implementations. Similarly, a calculus is a formal system that mathematically defines the behavior of the essential features of a domain. This paper demonstrates how to use a calculus as the foundation for an execution environment for stream processing. Stream processing makes it convenient to exploit parallelism on multicores or even clusters. Streaming languages are diverse [2, 3, 10, 31, 34], because they address many real-world domains, including transportation, audio and video processing, network monitoring, telecommunications, healthcare, and finance.

The starting point for this paper is the Brooklet calculus [32]. A Brooklet application is a stream graph, where each edge is a conceptually infinite stream of data items, and each vertex is an operator. Each time a data item arrives on an input stream of an operator, the operator fires, executing a pure function to compute data items for its output streams. Optionally, an operator may also maintain state, consisting of variables to remember across operator firings. Prior work demonstrated that Brooklet is a natural abstraction for different streaming languages.

The finishing point for this paper is the River execution environment. Extending a calculus into an execution environment is challenging. A calculus deliberately abstracts away features that are not relevant in theory, whereas an execution environment must add them back in to be practical. The question is how to do that while (1) maintaining the desirable properties of the calculus, (2) making the source language development effort economic, and (3) safely supporting common optimizations and reaching reasonable target-platform performance. The answers to these questions are the research contributions of this paper.

On the implementation side, we wrote front-ends for dialects of three very different streaming languages (CQL [2], Sawzall [31], and StreamIt [34]) on River. We wrote a back-end for River on System S [1], a high-performance distributed streaming runtime. And we wrote three high-level optimizations (placement, fusion, and fission) that work at the River level, decoupled from and thus reusable across front-ends. This is a significant advance over prior work, where source languages, optimizations, and target platforms are tightly coupled. For instance, since River's target platform, System S, runs on a shared-nothing cluster, this paper reports the first distributed CQL implementation, making CQL more scalable.

Overall, this paper shows how to get the best of both theory and practice for stream processing. Starting from a calculus supports formal proofs showing that front-ends realize the semantics of their source languages, and that optimizations are safe. And finishing in an execution environment lowers the barrier to entry for new streaming language implementations, and thus grows the ecosystem of this crucial style of programming.

2. Maintaining Properties of the Calculus

Being a calculus, Brooklet makes abstractions. In other words, it removes irrelevant details to reduce stream processing to the features that are essential for formal reasoning. On the other hand, River, being a practical execution environment, has to take a stand on each of the abstracted-away details. This section describes these decisions, and explains how the execution environment retains the benefits of the calculus.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS '12, July 16–20, 2012, Berlin, Germany.
Copyright © 2012 ACM 978-1-4503-1315-5. . . $10.00

In order for this paper to be self-contained and suitable for a systems audience, it first restates the formal semantics from the Brooklet paper [32] in more informal terms. Here is an example Brooklet program:

```
(appleTrades) <- SelectApple(trades);
($volume)     <- Count(appleTrades, $volume);
```

The `SelectApple` operator filters data items from stream `trades` into stream `appleTrades`, where the `Count` operator counts their volume in variable `$volume`. Note that `$volume` appears both on the left (written) and right (read) side of the `Count` operator. Brooklet models streams as FIFO queues of data items. Queues are one-to-one: they have at most one producer and one consumer. Queues without producers, such as `trades`, are input queues, and queues without consumers are output queues. At the beginning of the execution, only input queues are non-empty. Convergent or divergent graphs can be constructed by using the same operator as the consumer or producer of multiple queues, respectively. A queue is eligible to fire if and only if it is non-empty and has a consumer. Execution proceeds as follows:

while there is at least one queue eligible to fire:
  atomically perform the firing as a single step:
  - Non-deterministically pick a queue that is eligible to fire. By definition, the queue has a unique consumer operator.
  - Pop one data item from the front of the firing queue.
  - Read the values of the input variables of the operator.
  - Call the function that implements the operator, passing the data item from the firing queue and the values of the input variables as parameters.
  - The result of the function call contains, for each output queue of the operator, zero or more data items, and for each output variable of the operator, its new value.
  - Push the data items to the operator's output queues.
  - Write the values to the operator's output variables.

At the end, only the program's output queues are non-empty. The result of the execution consists of the contents of variables and output queues.

2.1 Brooklet Abstractions and their Rationale

The following is a list of simplifications in the Brooklet semantics, along with the insights behind them.

Atomic steps. Brooklet defines execution as a sequence of atomic steps. Being a small-step operational semantics makes it amenable to proofs. Each atomic step contains an entire operator firing. By not sub-dividing firings further, it avoids interleavings that unduly complicate the behavior. In particular, Brooklet does not require complex memory models.

Pure functions. Functions in Brooklet are pure, without side effects and with repeatable results. This is possible because state is explicit and separate from operators. Keeping state separate also makes it possible to see right away which operators in an application are stateless or stateful, use local or shared state, and read or write state.

Opaque functions. Brooklet elides the definition of the functions for operator firings, because semantics for local sequential computation are well-understood.

Non-determinism. Each step in a Brooklet execution non-deterministically picks a queue to fire. This non-deterministic choice abstracts away from concrete schedules. In fact, it even avoids the need for any centralized scheduler, thus enabling a distributed system without costly coordination. Note that determinism can be implemented on top of Brooklet with the appropriate protocols, as shown by translations from deterministic languages [32].

No physical platform. Brooklet programs are completely independent from any actual machines they would run on.

Finite execution. Stream processing applications run conceptually forever, but a Brooklet execution is a finite sequence of steps. One can reason about an infinite execution by induction over each finite prefix [16].

2.2 River Concretizations and their Rationale

This section shows how the River execution environment fills in the holes left by the Brooklet calculus. For each of the abstractions from the previous section, it briefly explains how to concretize it and why. The details and correctness arguments for these points come in later sections.

Atomic steps. Whereas Brooklet executes firings one at a time, albeit in non-deterministic order, River executes them concurrently whenever it can guarantee that the end result is the same. This concurrency is crucial for performance. To guarantee the same end result, River uses a minimum of synchronization that keeps firings conceptually atomic. River shields the user from concerns of locking or memory models.

Pure functions. Both the calculus and the execution environment separate state from operators. However, whereas the calculus passes variables in and out of functions by copies, the execution environment uses pointers instead to avoid the copying cost. Using the fact that state is explicit, River automates the appropriate locking discipline where necessary, thus relieving users from this burden. Furthermore, instead of returning data items to be pushed on output queues, functions in River directly invoke call-backs for the run-time library, thus avoiding copies and simplifying the function implementations.

Opaque functions. Functions for River are implemented in a traditional non-streaming language. They are separated from the River runtime library by a well-defined API. Since atomicity is preserved at the granularity of operator firings, River does not interfere with any local instruction-reordering optimizations the low-level compiler or the hardware may want to perform.

Non-determinism. Being a virtual execution environment, River ultimately leaves the concrete schedule to the underlying platform. However, it reduces the flexibility for the scheduler by bounding queue sizes, and by using back-pressure to prevent deadlocks when queues fill up.

No physical platform. River programs are target-platform independent. However, at deployment time, the optimizer takes target-platform characteristics into consideration for placement.

Finite execution. River applications run indefinitely and produce timely outputs along the way, fitting the purpose and intent of practical stream processing applications.

2.3 Maximizing Concurrency while Upholding Atomicity

This section gives the details for how River upholds the sequential semantics of Brooklet. In particular, River differs from Brooklet in how it handles state variables and data items pushed on output queues. These differences are motivated by performance goals: they avoid unnecessary copies and increase concurrency.

River requires that each operator instance is single-threaded and that all queue operations are atomic. Additionally, if variables are shared between operator instances, each operator instance uses locks to enforce mutual exclusion. River's locking discipline follows established practice for deadlock prevention, i.e., an operator instance first acquires all necessary locks in a standard order, then performs the actual work, and finally releases the locks in

reverse order. Otherwise, River does not impose further ordering constraints. In particular, unless prevented by locks, operator instances may execute in parallel. They may also enqueue data items as they execute and update variables in place without differing in any observable way from Brooklet's call-by-value-result semantics. To explain how River's execution model achieves this, we first consider execution without shared state.

Operator instance firings without shared state behave as if atomic. In River, a downstream operator instance o_2 can fire on a data item while the firing of the upstream operator instance o_1 that enqueued it is still in progress. The behavior is the same as if o_2 had waited for o_1 to complete before firing, because queues are one-to-one and queue operations are atomic. Furthermore, since each operator is single-threaded, there cannot be two firings of the same operator instance active simultaneously, so there are no race conditions on operator-instance-local state variables. □

1. $AllClasses.add(AllSharedVars)$
2. **for all** $o \in OpInstances$ **do**
3. $\quad UsedByO = sharedVariablesUsedBy(o)$
4. $\quad$ **for all** $v \in UsedByO$ **do**
5. $\quad\quad EquivV = v.equivalenceClass$
6. $\quad\quad$ **if** $EquivV \not\subseteq UsedByO$ **then**
7. $\quad\quad\quad AllClasses.remove(EquivV)$
8. $\quad\quad\quad AllClasses.add(EquivV \cap UsedByO)$
9. $\quad\quad\quad AllClasses.add(EquivV \setminus UsedByO)$

Figure 1. Algorithm for assigning shared variables to equivalence classes, that is, locks.

In the presence of shared state, River uses the lock assignment algorithm shown in Figure 1. The algorithm finds the minimal set of locks that covers the shared variables appropriately. The idea is that locks form equivalence classes over shared variables: every shared variable is protected by exactly one lock, and shared variables in the same equivalence class are protected by the same lock.

Two variables only have separate locks if there is an operator instance that uses one but not the other. The algorithm starts with a single equivalence class (lock) containing all variables in line 1. The only way for variables to end up under different locks is by the split in lines 7–9. Without loss of generality, let v be in $EquivV \cap UsedByO$ and w be in $EquivV \setminus UsedByO$. That means there is an operator instance o that uses $UsedByO$, which includes v but excludes w. □

An operator instance only acquires locks for variables it actually uses. Let's say operator instance o uses variable v but not w. We need to show that v and w are under separate locks. If they are under the same lock, then the algorithm will arrive at a point where $UsedByO$ contains v but not w and $EquivV$ contains both v and w. That means that $EquivV$ is not a subset of $UsedByO$, and lines 7–9 split it, with v and w in two separate parts of the split. □

Shared state accesses behave as if atomic. An operator instance locks the equivalence classes of all the shared variables it accesses. □

2.4 Bounding Queue Sizes

In Brooklet, communication queues are infinite, but real-world systems have limited buffer space, raising the question of how River should manage bounded queues. One option is to drop data items when queues are full. But this results in an unreliable communication model, which significantly complicates application development [15], wastes effort on data items that are dropped later on [26],

1. $process(dataItem, submitCallback, variables)$
2. $\quad lockSet = \{v.equivalenceClass()$ **for** $v \in variables\}$
3. $\quad$ **for all** $lock \in lockSet.iterateInStandardOrder()$ **do**
4. $\quad\quad lock.acquire()$
5. $\quad tmpCallback = \lambda d \Rightarrow tmpBuffer.push(d)$
6. $\quad opFire(dataItem, tmpCallback, variables)$
7. $\quad$ **for all** $lock \in lockSet.iterateInReverseOrder()$ **do**
8. $\quad\quad lock.release()$
9. $\quad$ **while** $!tmpBuffer.isEmpty()$ **do**
10. $\quad\quad submitCallback(tmpBuffer.pop())$

Figure 2. Algorithm for implementing back-pressure.

and is inconsistent with Brooklet's semantics. A more attractive option is to automatically apply back-pressure through the operator graph.

A straightforward way to implement back-pressure is to let the enqueue operation performed by an operator instance block when the output queue is full. While easy to implement, this approach could deadlock in the presence of shared variables. To see this, consider an operator instance o_1 feeding another operator instance o_2, and assume that both operator instances access a common shared variable. Further assume that o_1 is blocked on an enqueue operation due to back-pressure. Since o_1 holds the shared variable lock during its firing, o_2 cannot proceed while o_1 is blocked on the enqueue operation. On the other hand, o_1 will not be able to unblock until o_2 makes progress to open up space in o_1's output queue. They are deadlocked.

The pseudocode in Figure 2 presents River's solution to implementing back-pressure. It describes the *process* function, which is called by the underlying streaming runtime when data arrives at a River operator. The algorithm starts in line 2 with an operator's lock set. The lock set is the minimal set of locks needed to protect an operator's shared variables, as described in Section 2.3. Before an operator fires, it first must acquire all locks in its lock set, as shown in lines 3-4. Once all locks are held, the process function invokes the operator's *opFire* method, which contains the actual operator logic. The *opFire* does not directly enqueue its resultant data for transport by the runtime. Instead, it writes its results to a dynamically-sized intermediate buffer, which is passed to *opFire* as a callback. Lines 5-6 show the callback and invocation of the operator logic. Next, lines 7-8 release all locks. Finally, lines 9-10 drain the temporary buffer, enqueuing each data item for transport by calling the streaming runtime's *submit* callback.

The key insight is that lines 9-10 might block if the downstream queue is full, but there is no deadlock because at this point the algorithm has already released its shared-variable locks. Furthermore, *process* will only return after it has drained the temporary buffer, so it only requires enough space for a single firing. If *process* is blocked on a downstream queue, it may in turn block its own upstream queue. That is what is meant by back-pressure. The algorithm in Figure 2 restricts the scheduling of operator firings. In Brooklet, an operator instance can fire as long as there is at least one data item in one of its input queues. In River, an additional condition is that all intermediate buffers must be empty. This does not impact the semantics of the applications or the programming interface of the operators. It simply impacts the scheduling decisions of the runtime.

3. Making Language Development Economic

River is intended as the target of a translation from a source language, which may be an existing streaming language or a newly invented one. In either case, the language implementer wants to make use of the execution environment without spending too much effort on the translation. In other words, language development should

```
select avg(speed), segNo, dir, hwy
from segSpeed[range 300];
```

(a) One of the Linear-Road queries in CQL.

```
congested: { speed: int; seg_no: int;
             dir: int; hwy: int } relation
= select avg(speed), seg_no, dir, hwy
  from seg_speed[range 300];
```

(b) One of the Linear-Road queries in River-CQL.

```
proto "querylog.proto"
queryOrigins: table sum[url: string] of count: int;
queryTargets: table sum[url: string] of count: int;
logRecord: QueryLogProto = input;
emit queryOrigins[logRecord.origin] <- 1;
emit queryTargets[logRecord.target] <- 1;
```

(c) Batch log query analyzer in Sawzall.

```
queryOrigins: table sum[url:string] of count: int;
queryTargets: table sum[url:string] of count: int;
logRecord: { origin: string; target: string } = input;
emit queryOrigins[logRecord.origin] <- 1;
emit queryTargets[logRecord.target] <- 1;
```

(d) Batch log query analyzer in River-Sawzall.

```
pipeline {
  pipeline {
    splitjoin {
      split duplicate;
      filter { /* ... */ }
      filter { /* ... */ }
      join roundrobin; }
    filter { /* subtractor */ } }
  filter { /* ... */ } }
```

(e) FM Radio in both StreamIt versions (filter bodies elided).

```
float->float filter {
  work pop 2 push 1 {
    push(peek(1) - peek(0)); pop(); pop(); } }
```

(f) One of the FM Radio filters in StreamIt.

```
filter {
  work {
    s,tc <- subtractor(s,peek(1)); push(tc); pop();} }
```

(g) One of the FM Radio filters in River-StreamIt.

Figure 3. Example source code in original languages and their River dialects.

be economic. We address this requirement by an intermediate language that is easy to target, and in addition, by providing an eco-system of tools and reusable artifacts for developing River compilers. We demonstrate the economy by implementing three existing languages. The foundation for the translation support is, again, the Brooklet calculus. Hence, we review the calculus first, before exploring what it takes to go from theory to practice in River.

3.1 Brooklet Treatment of Source Languages

The Brooklet paper contained formalizations, but no implementations, for translating the cores of CQL [2], Sawzall [31], and StreamIt [34] to the Brooklet calculus [32]. This exercise helped prove that the calculus can faithfully model the semantics of two languages that had already been formalized elsewhere, and helped provide the first formal semantics of a third language that had not previously been formalized.

Brooklet translation source. *CQL*, the continuous query language, is a dialect of the widely used SQL database query language for stream processing [2]. CQL comes with rigorously defined semantics, grounded in relational algebra. The additions over SQL are windows for turning streams into tables, as well as operators for observing changes in a table to obtain a stream. *Sawzall* [31] is a language for programming MapReduce [7], a scalable distributed batch processing system. Finally, *StreamIt* [34] is a synchronous data flow (SDF) [23] language with a denotational semantics [35]. StreamIt relies on fixed data rates to compute a static schedule, thus reducing runtime overheads for synchronization and communication. These three languages have fundamentally different constructs, optimized for their respective application domains. They were developed independently by different communities: CQL originated from databases, Sawzall from distributed computing, and StreamIt from digital signal processing. The Brooklet paper abstracted away many details of the source languages that are not relevant for the calculus, such as the type system and concrete operator implementations.

Brooklet translation target. A Brooklet program consists of two parts: the stream graph and the operator functions. For specifying stream graphs, Brooklet provides a topology language, as seen in the `appleTrades` example at the beginning of Section 2. For operator functions, on the other hand, Brooklet does not provide any notation; instead, it just assumes they are pure opaque functions. Where necessary, the Brooklet paper uses standard mathematical notation, including functions defined via currying. Remember that currying is a technique that transforms a multi-argument function into a function that takes some arguments, and returns another residual function for the remaining arguments. This is useful for language translation in that the translator can supply certain arguments statically, while leaving others open to firing time.

Brooklet translation specification. The Brooklet paper specifies translations in sequent calculus notation. The translation is syntax-directed, in the sense that each translation rule matches certain syntactic constructs, and translations of larger program fragments are composed from translations of their components. These translations are restricted to the core source language subsets, and are only specified on paper, not implemented. The mathematical notation and restriction to the essentials make the translations amenable to proofs. However, they omit many of the details necessary for a practical implementation, which is what this paper is about.

3.2 River Implementation of Source Languages

As the previous section shows, Brooklet abstracts three aspects of source language support: it simplifies the source languages, it provides a core target language, and it specifies but does not implement translations. The following sections describe how River and its eco-system concretize these three areas, with the goal of economy in language development. In other words, not only does River make it *possible* to implement various streaming languages, it makes it *easier*.

3.3 River Translation Source

We implemented the same three streaming languages (CQL, Sawzall, and StreamIt) on River. The River dialects are more complete than the Brooklet language subsets, but they are not identical to the original published versions of the languages. They add some features

that were missing in the original versions in order to make them more type-safe. They omit some infrequently used features from the original versions to reduce implementation effort. And they replace some features, notably expressions, with equivalent features to enable code reuse. While we took the liberty to modify the source languages, we retained their essential aspects. In practice, it is not uncommon for source languages to change slightly when they are ported to a virtual execution environment. For example, the JVM supports Jython rather than C Python, and the CLR supports F# rather than OCaml. Those cases, like ours, are motivated by economic language development.

River-CQL. River's dialect of CQL is more complete than the original language. Figures 3 (a) and (b) show an example. The original version lacks types. River-CQL includes types to make the language easier to use, for example, by reporting type errors at compile time. Moreover, because the types are preserved during translation, River-CQL avoids some overheads at runtime. The type syntax of any functional or imperative language would do for this purpose; we used the syntax for types in River's implementation language, described in Section 3.4. Since we already had compiler components for the type sublanguage, we could just reuse those, simplifying source language development.

River-Sawzall. River's dialect of Sawzall replaces protocol buffers by a different notation. Protocol buffers are a data definition language that is part of Google's eco-system. The River eco-system, on the other hand, has its own type notation, which we reuse across source languages. Figures 3 (c) and (d) illustrate this change. As this feature was not central to Sawzall to begin with, changing it was justified to ease language development.

River-StreamIt. River's dialect of StreamIt elides the feature called *teleport messaging*. Teleport messages are an escape hatch to send out-of-band messages that side-step the core streaming paradigm. Only very few StreamIt programs use teleport messaging [33]. They require centralized support, and are thus only implemented in the single-node back-end of StreamIt. Since River runs on multi-node clusters, we skipped this feature altogether. Furthermore, another change in River's dialect of StreamIt is that it uses out-of-line work function implementations for filter operators, as seen in Figures 3 (f) and (g). Since work functions in StreamIt contain traditional imperative code, it is a matter of taste where and in what syntax to write them. We chose not to spend time literally emulating a notation that is inessential to the main language.

3.4 River Translation Target

A River program consists of two parts: the stream graph and the operator implementations. For the stream graph, River simply reuses the topology language of Brooklet. For operators, on the other hand, River must go beyond Brooklet by supplying an implementation language. The primary requirements for this implementation language are that (1) the creation and decomposition of data items be convenient, to aid in operator implementation, and (2) mutable state be easily identifiable, in keeping with the semantics. An explicit non-goal is support for traditional compiler optimizations, which we leave to an off-the-shelf traditional compiler.

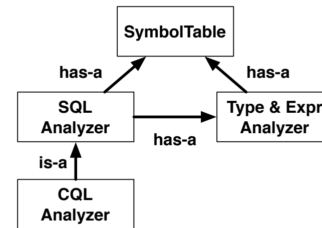
Typed functional languages clearly meet both requirements and a lower-level traditional intermediate language such as LLVM [22] can also meet them, given library support for higher-level language features such as pattern matching. In our current implementation, we rely on OCaml as River's implementation sublanguage. It features a high-quality native code compiler and a simple foreign function interface, which facilitates integration with existing streaming runtimes written in C/C++.

3.5 River Translation Specification

A language implementer who wants to create a new language translator needs to implement a parser, a type-checker, and a code generator. We facilitate this task by decomposing each language into sublanguages, and then reusing common sublanguage translator modules across languages. Principally, we follow the same approach as the Jeannie language [18], which composes C and Java. However, our work both increases the granularity of the components (by combining parts of languages) and the number of languages involved.

Modular parsers. The parsers use component grammars written as modules for the *Rats!* parser generator [14]. Each component grammar can either *modify* or *import* other grammar modules. For example, the CQL grammar consists of several modules: SQL's select-from-where clauses, streaming constructs modifying SQL to CQL, an imported expression sublanguage for operators like projection or selection, and an imported type sublanguage for schemas. The grammar modules for expressions and types are the same as in other River languages. The result of parsing is an abstract syntax tree (AST), which contains tree nodes drawn from each of the sublanguages.

Modular type checkers. Type checkers are also implemented in a compositional style. Type checkers for composite languages are written as groups of visitors. Each visitor is responsible for all AST nodes corresponding to a sublanguage. Each visitor can either dispatch to or inherit from other visitors, and all visitors share a common type representation and symbol table. For example, the CQL analyzer inherits from an SQL analyzer, which in turn dispatches to an analyzer for expressions and types. All three analyzers share a symbol table.



If there are type errors, the type analyzer reports those and exits. Otherwise, it populates the symbol table, and decorates the AST with type annotations.

Modular code generators. The implementation language of the River IL allows language developers to write language-specific libraries of standard operators, such as select, project, split, join, and aggregate. However, the operator implementations need to be specialized for their concrete application. Consider, for example, an implementation for a selection operator:

```
Bag.filter (fun x -> #expr) inputs
```

where `#expr` stands for a predicate indicating the filter condition.

How best to support this specialization was an important design decision. One approach would be to rely on language support, i.e., OCaml's support for generic functions and modules (i.e. *functors*) as reflected in the River IL. This approach is well-understood and statically safe. But it also requires abstracting away any application-specific operations in callbacks, which can lead to unwieldy interfaces and performance overhead. Instead, we chose to implement common operators as IL templates, which are instantiated inline with appropriate types and expressions. Pattern variables (of form `#expr`) are replaced with concrete syntax at compile time. This eliminates the overhead of abstraction at the cost of code size.

The templates are actually parsed by grammars derived from the original language grammars. As a result, templates benefit from both the convenience of using concrete syntax, and the robustness of static syntax checking. Code generation templates in River play the same role as currying in Brooklet, i.e., they bind the function to its arguments.

Thus, code generation is also simplified by the use of language composition. The input to the code generator is the AST annotated with type information, and the output is a stream graph and a set of operator implementations. Our approach to producing this output is to first create the AST for the stream graph and each operator implementation, and then pretty-print those ASTs. In the first step, we splice together subtrees obtained from the templates with subtrees obtained from the original source code. In the second step, we reuse pretty-printers that are shared across source language implementations. Overall, we found that the use of language composition led to a smaller, more consistent implementation with more reuse, making the changes to the source languages well worth it.

4. Safe and Portable Optimizations

One of the benefits of a virtual execution environment is that it can provide a single implementation of an optimization, which benefits multiple source languages. In prior work on stream processing, each source language had to re-implement similar optimizations. The River execution environment, on the other hand, supports optimization reuse across languages. Here, we are primarily interested in optimizations from the streaming domain, which operate at the level of a stream graph, as opposed to traditional optimizations at the level of functional or imperative languages. By working at the level of a stream graph, River can optimize an entire distributed application. As with the other contributions of River, the Brooklet calculus provides a solid foundation, but new ideas are needed to build an execution environment upon it.

4.1 Brooklet Treatment of Optimizations

The Brooklet paper decouples optimizations from their source languages [32]. It specifies each optimization by a *safety guard* and a *rewrite rule*. The safety guard checks whether a subgraph satisfies the preconditions for applying the optimization. It exploits the one-to-one restriction on queues and the fact that state is explicit to establish these conditions. If a subgraph passes the safety guard, the rewrite rule replaces it by a transformed subgraph. The Brooklet paper then proceeds to prove that the optimizations leave the observable input/output behavior of the program unchanged.

The Brooklet paper discusses three specific optimizations: (1) *Fusion* replaces two operators by a single operator, thus reducing communication costs at the expense of pipeline parallelism. (2) *Fission* replaces a single operator by a splitter, a number of data-parallel replicas of the operator, and a merger. The Brooklet paper only permits fission for stateless operators. (3) *Selection hoisting* (i.e., selection pushdown) rewrites a subgraph $A \rightarrow \sigma$ into a subgraph $\sigma \rightarrow A$, assuming that A is a stateless operator and σ is a selection operator that only relies on data fields unchanged by A. Selection hoisting can improve performance by reducing the number of data items that A has to process.

4.2 River Optimization Support

We made the observation that River's source languages are designed to make certain optimizations safe by construction, without requiring sophisticated analysis. For example, Sawzall provides a set of built-in aggregations that are known to be commutative, and partitioned by a user-supplied key, thus enabling fission. Rather than loosing safety information in translation, only to have to discover it again before optimization, we wanted to add it to River's intermediate language (IL). However, at the same time, we did not want to make the IL source-language specific, which would jeopardize the reusability of optimizations and the generality of River.

We resolved this tension by adding extensible annotations to River's graph language. An annotation next to an operator specifies *policy* information, which encompasses safety and profitability. Safety policies are usually passed down by the translator from source language to IL, such as, which operators to parallelize. Profitability policies usually require some knowledge of the execution platform, such as the number of machines to parallelize on. In this paper, we use simple heuristics for profitability; prior work has also explored more sophisticated analyses for this, which are beyond the scope of this paper [13]. Policy is separated from *mechanism*, which implements the actual code transformation that performs the optimization. River's annotation mechanism allows it to do more powerful optimizations than Brooklet. For example, fission in River works not just on stateless operators, but also on stateful operators, as long as the state is keyed and the key fields are listed in annotations. Both CQL and Sawzall are designed explicitly to make the key evident from the source code, so all we needed to do is preserve that information through their translators.

To keep annotations extensible, they share a common, simple syntax, inspired by Java. Each use of an operator is preceded by zero or more annotations. Each annotation is written as an at-sign (`@`), an identifier naming the annotation, and a comma-separated list of expressions serving as parameters. River currently makes use of the following annotations:

Annotation	Description
`@Fuse(`*ID*`)`	Directive to fuse operators with the same *ID* in the same process.
`@Parallel()`	Directive to perform fission on an operator.
`@Commutative()`	Declares that an operator's function is commutative.
`@Keys(`$k_1,\ldots,k_n$`)`	Declares that an operator's state is partitionable by the key fields $k_1,\ldots,k_n$ in each data item.
`@Group(`*ID*`)`	Directive to place operators with the same *ID* on the same machine.

We anticipate adding more annotations as we implement more source languages and/or more optimizations. The translator from River IL to native code invokes the optimizers one by one, transforming the IL at each step. A specification passed to the translator determines the order in which the optimizations are applied.

4.3 Fusion Optimizer

Intuition. Fusion combines multiple stream operators into a single stream operator, to avoid the overhead of data serialization and transport [12, 20].

Policy. The policy annotation is `@Fuse(`*ID*`)`. Operators with the same *ID* are fused. Applying fusion is a tradeoff. It eliminates a queue, reducing communication cost, but it prohibits operators from executing in parallel. Hence, fusion is profitable if the savings in communication cost exceed the lost benefit of parallelism. As shown in the Brooklet calculus, a sufficient safety precondition for fusion is if the fused operators form a straight-line pipeline without side entries or exits.

Mechanism. The current implementation replaces internal queues by direct function calls. A possible future enhancement would be to allow fused operators to share the same process but run on different threads. This would reduce the cost for communication, but still maintain the benefits of pipeline parallelism on multicores.

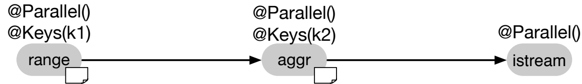
4.4 Fission Optimizer

Intuition. Fission replicates an operator or a stream subgraph to introduce parallel computations on subsets of the data [7, 8, 12].

Policy. Fission uses three annotations. The `@Parallel()` annotation is a directive to parallelize an operator. The `@Commutative()` annotation declares that a given operator's function commutes. Finally, the `@Keys(`k_1`,...,`k_n`)` annotation declares that an operator is stateful, but that its state is keyed (i.e. partitioned) by the key in fields $k_1, \ldots, k_n$. Fission is profitable if the computation in the parallel segment is expensive enough to make up for the the overhead of the inserted split and merge operators.

The safety conditions for fission depend on state and order. In terms of state, there must be no memory dependencies between replicas of the operator. This is trivially the case when the operator is stateless. The other way to accomplish this is if the state of the operator can be partitioned by key, such that each operator replica is responsible for a separate portion of the key space. In that case, the splitter routes data items by using a hash on their key fields. When a parallel segment consists of multiple operators, they must be either stateless or have the same key. To understand how order affects correctness, consider the following example. Assume that in the unoptimized program, the operator pushes data item d_1 before d_2 on its output queue. In the optimized program, d_1 and d_2 may be computed by different replicas of the operator, and depending on which replica is faster, d_2 may be pushed first. That would be unsafe if any down-stream operator depends on order. That means that fission is safe with respect to order either if all down-stream operators commute, or if the merger brings data items from different replicas back in the right order. Depending on the source language, the merger can use different ordering strategies: CQL embeds a logical timestamp in every data item that induces an ordering; Sawzall has commutative aggregations and can hence ignore order; and StreamIt only parallelizes stateless operators and can hence use round-robin order.

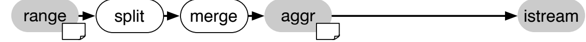
Mechanism. River's fission optimization consists of multiple steps. Consider the following example in which three operators appear in a pipeline. The first two operators, `range` and `aggr`, are stateful, and keyed by k_1 and k_2 respectively. The third, `istream`, is stateless. The figures indicate stateful operators by a rectangle with a folded corner. All three operators have the `@Parallel()` annotation, indicating that fission should replicate them.



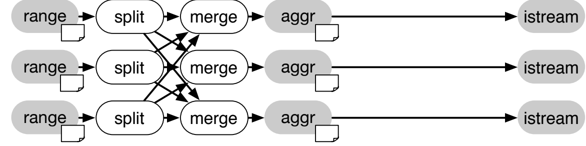
Step 1 adds split and merge operators around parallelizable operators. This trivially parallelizes each individual operator. At the same time, it introduces bottlenecks, as data streamed through adjacent mergers and splitters must pass through a single machine. Note that for source or sink operators, only merge or split, respectively, are needed. This is because the partitioning of input data and the combining of output data is assumed to occur outside of the system.

Step 2 removes the bottlenecks. There are two in the example; each calls for a different action. First, the merge and split between `aggr` and `istream` can be safely removed, because `istream` is stateless.

Next, the merge and split between `range` and `aggr` cannot be removed, because both operators partition state by different keys, k_1 and k_2. Instead, we apply a *rotation*. A rotation switches the order of the merge and split to remove the bottleneck. This is the same approach as the *shuffle* step in MapReduce [7].



Finally, Step 3 replicates the operators to the desired degree of parallelism, and inserts `@Group(`*ID*`)` annotations for the placement optimizer, to ensure that replicas actually reside on different machines.



In this example, all operators are parallelized. In other applications, only parts may be parallelized, so performance improvements will be subject to Amdahl's law: for example, if the unoptimized program spends 1/5th of its time in non-parallelizable operators, the optimized program can be no more than 5× faster.

4.5 Placement Optimizer

While Brooklet explored selection hoisting, River explores placement instead, because it is of larger practical importance, and illustrates the need for the optimizer to be aware of the platform.

Intuition. Placement assigns operators to machines and cores to better utilize physical resources [36].

Policy. The policy annotation is `@Group(`*ID*`)`. Operators with the same *ID* are assigned to the same machine. Several operators can be assigned the same *ID* to take advantage of machines with multiple cores. Placement is profitable if it reduces network traffic (by placing operators that communicate a lot on the same machine) and/or improves load balance (by placing computationally intensive operators on different machines). In our current implementation, annotations are applied by hand. The annotations could be added automatically by leveraging prior work on determining optimal placement [30, 36].

Mechanism. The placement mechanism does not transform the IL, but rather, directs the runtime to assign the same machine to all operators that use the same identifier. Information such as the number of machines is only available at the level of the virtual execution environment, a trait that River shares with other language-based virtual machines. Placement is complicated if operators share state. In general, River could support sharing variables across machines, but relies on the underlying runtime to support that functionality. Because our current backend does not provide distributed shared state, the placement optimizer has an additional constraint. It ensures that all operators that access the same shared variable are placed on the same machine. Fortunately, none of the streaming languages we encountered so far need cross-machine shared state.

4.6 When to Optimize

The Brooklet calculus abstracts away the timing of optimizations. The River execution environment performs optimizations once the available resources are known, just before running the program. In other words, the translations from source languages to IL happen ahead of time, but the translation from IL to native code for a specific target platform is delayed until the program is launched. That enables River to make more informed profitability decisions.

As just described, River implements *static* optimizations. In future work, we plan to address *dynamic* optimizations that make decisions at runtime. One promising approach to implementing dynamic optimizations is to statically create a more general graph, and then adapt how data flows through it at runtime. A seminal example for this is the Eddy operator, which performs dynamic operator

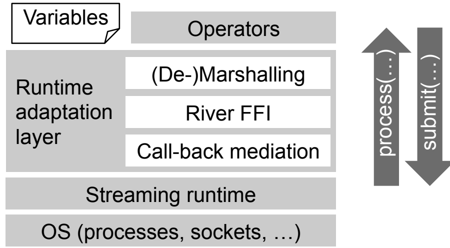


Figure 4. Stack of layers for executing River.

reordering without physically changing the graph at runtime [4]. River could use annotations to decide where an optimization applies, and statically rewrite the graph to add in control operators that dynamically route data items for optimization.

5. Runtime Support

The main goal for River's runtime support is to insulate the IL and the runtime from each other. Figure 4 shows the architecture. It consists of a stack of layers, where each layer only knows about the layers immediately below and above it. Above the operating system is the streaming runtime, which provides the software infrastructure for process management, data transport, and distribution. Above that is the runtime adaptation layer, which provides the interface between River operators and the distributed runtime. At the highest level are the operator instances and their associated variables. River's clean separation of concerns ensures that it can be ported to additional streaming runtimes. The rest of this section describes each layer in detail.

5.1 Streaming Runtime

A distributed streaming runtime for River must satisfy the following requirements. It must launch the right processes on the right machines in the distributed system to host operators and variables. It must provide reliable transport with ordered delivery to implement queues. It must arbitrate resources and monitor system health for reliability and performance. Finally, our placement optimizer relies on placement support in the runtime. There is no strict latency requirement: the semantics tolerate an indefinite transit time.

The streaming runtime sits on top of an operating system layer, which provides basic centralized services to the runtime layer, such as processes, sockets, etc. However, building a high-performance streaming runtime satisfying the above-listed requirements is a significant engineering effort beyond the OS. Therefore, we reuse an existing runtime instead of building our own. We chose System S [10], a distributed streaming runtime developed at IBM that satisfies the requirements, and that is available to universities under an academic license. While System S has its own streaming language, we bypassed that in our implementation, instead interfacing with the runtime's C++ API.

5.2 Runtime Adaptation

The runtime adaptation layer provides the interface between River operators and the distributed runtime. As shown in Figure 4, it consists of three sub-layers.

Call-Back Mediation Layer. This layer mediates between the runtime-specific APIs and call-backs, and the River-specific functions. The code in this layer peels off runtime-specific headers from a data item, and then passes the data item to the layer above. Similarly, it adds the headers to data on its way down. If there are shared variables, this layer performs locking, and implements output buffers for avoiding back-pressure induced dead-lock as described in Section 2.4. The call-back mediation layer is linked into the streaming runtime.

River FFI Layer. A foreign function interface, or FFI for short, enables calls across programming languages. In this case, it enables C++ code from a lower layer to call a River function in an upper layer, and it enables River code in an upper layer to call a C++ function in a lower layer. The River FFI is the same as the OCaml FFI. Each OS process for River contains an instance of the OCaml runtime, which it launches during start-up.

(De-)Marshalling Layer. This layer converts between byte arrays and data structures in River's implementation language. It uses an off-the-shelf serialization module. The code in this layer is auto-generated by the River compiler. It consists of `process(...)` functions, which are called by the next layer down, demarshal the data, and call the next layer up; and of `submit(...)` functions, which are called by the next layer up, marshal the data, and call the next layer down. Since this layer is crucial for performance, we plan to optimize it further by specialized code generation.

Overall, implementing River on System S required 1,636 lines of Java code for the River-to-C++ translator, and 780 lines of boilerplate C++ code used by the translator. Since that is only a moderate amount of code, we believe porting to a different platform would not be too much work.

5.3 Variables and Operators

As described in Section 3.5, operators are generated from templates written by the language developer. Their implementation strikes a balance between the functional purity of operators in Brooklet, and performance demands of a practical IL that needs to update data in place, and make callbacks instead of returning values. Variables and operators are implemented in the implementation sublanguage of River. An operator firing takes the form of a function call from the next lower layer. If the operator accesses variables, then the call passes those as references, so that the operator can perform in-place updates if needed. Instead of returning data as part of the function's return value, the operator invokes a call-back for each data item it produces on an output queue. Note that this simple API effectively hides any lower-layer details from the variables and operators.

6. Evaluation

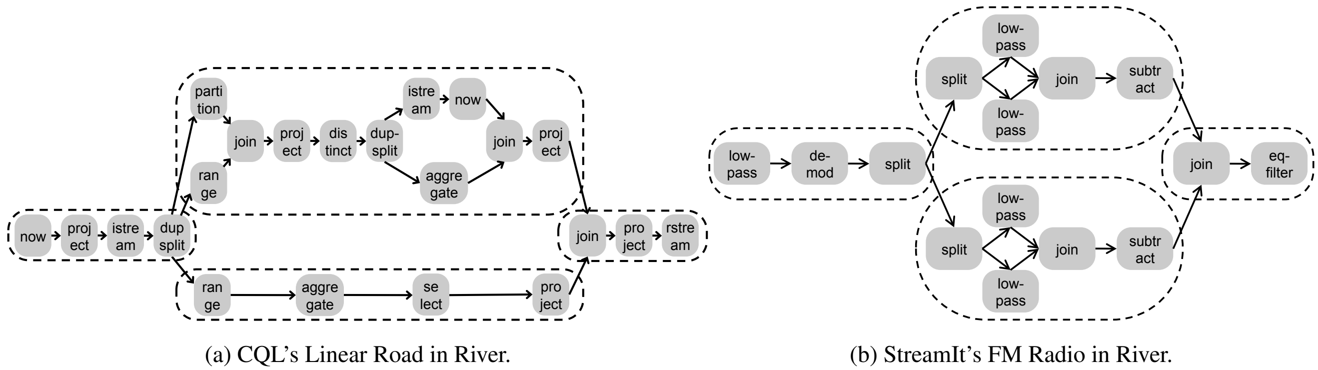
We have built a proof-of-concept prototype of River, including front-ends for the three source languages, implementations of the three optimizations, and a back-end on the System S distributed streaming system. We have not yet tuned the absolute performance of our prototype; the goal of this paper was to show its feasibility and generality. Therefore, while this section presents some experimental results demonstrating that the system works and performs reasonably well, we leave further efforts on absolute performance to future work.

All performance experiments were run on a cluster of 16 machines. Each machine has two 4-core 64-bit Intel Xeon (X5365) processors running at 3GHz, where each core has 32K L1i and 32K L1d caches of its own, and a 4MB unified L2 cache that is shared with another core. The processors have a FSB speed of 1,333 MHz and are connected to 16GB of uniform memory. Machines in the cluster are connected via 1Gbit ethernet.

6.1 Support for Existing Languages

To verify that River is able to support a diversity of streaming languages, we implemented the language translators described in Section 3, as well as illustrative benchmark applications. The benchmarks exercise a significant portion of each language, demonstrating the expressivity of River. They are described below:

CQL Linear Road. Linear Road [2] is the running example in the CQL paper. It is a hypothetical application that computes tolls for



(a) CQL's Linear Road in River.

(b) StreamIt's FM Radio in River.

Figure 5. Structural view for the CQL and StreamIt benchmarks. The dashed ovals group operators that are placed onto the same machine.

vehicles traveling on the Linear Road highway. Figure 5 (a) shows the operator graph of the application. Each vehicle is assigned a unique id, and its location is specified by three attributes: speed, direction, and position. Each highway is also assigned an id. Vehicles pay a toll when they drive on a congested highway. A highway is congested if the average speed of all vehicles on the highway over a 5 minute span is less than 40 mph.

StreamIt FM Radio. This benchmark implements a multi-band equalizer [12]. As shown in Figure 5 (b), the input passes through a demodulator to produce an audio signal, and then an equalizer. The equalizer is implemented as a splitjoin with two band-pass filters; each band-pass filter is the difference of a pair of low-pass filters.

Sawzall Batch Log Analyzer. Figure 3 (d) shows this query, which is based on an exemplar Sawzall query in Pike et al. [31]. It is a batch job that analyzes a set of search query logs to count queries per origin based on IP address. The resulting aggregation could then be used to plot query origins on a world map.

CQL Continuous Log Analyzer. This is similar to the Sawzall log query analyzer, but it is a continuous query rather than a batch job. Its input comes from a server farm. Each server reports the origin of the requests it has received, and the analyzer performs an aggregation keyed by the origin over the most recent 5 minute window. Note that the data is originally partitioned by the target (server) address, so the application must shuffle the data.

The three source languages, CQL, StreamIt, and Sawzall, occupy three diverse points in the design space for streaming languages and the benchmark applications exercise significant portions of each language. This demonstrates that River is expressive enough to support a wide variety of streaming languages.

6.2 Suitability for Optimizations

To verify that River is extensible enough to support a diverse set of streaming optimizations, we implemented each of the optimizations described in Section 4. We then applied the different optimizations to the benchmark applications from Section 6.1.

Placement. Our placement optimizer distributes an application across machines. Operators from each application were assigned to groups, and each group was executed on a different machine. As a first step, we used the simple heuristic of assigning operators to groups according to the branches of the top-level split-merge operators, although there has been extensive prior work on determining the optimal assignments [36]. In the non-fused version, each operator had its own process, and it was up to the OS to schedule processes to cores. Figure 6 (a) and (b) show the results of running both Linear Road and the FM Radio applications on 1, 2, and 4 machines. Figure 5 shows the partitioning scheme for the 4-machine case using dashed ovals. These results are particularly exciting because the original implementation of CQL was not distributed. Despite the fact that the Linear Road application shows only limited amounts of task and pipeline parallelism, the first distributed CQL implementation achieves a 3.70× speedup by distributing execution on 4 machines. The FM Radio application exhibits a 1.84× speedup on 4 machines.

Fission. Our fission optimizer replicates operators, and then distributes those replicas evenly across available machines. We tested two applications, the Sawzall batch log analyzer and the CQL continuous log analyzer, with increasing amounts of parallelism. In these experiments, the degree of parallelism corresponded to the number of available machines, from 1 to 16. We additionally ran the Sawzall query on 16 machines with 16, 32, 48, and 64 degrees of parallelism, distributing the computation across cores. The results are shown in Figures 6 (c) and (d).

Fission adds split and merge operators to the stream graph. Therefore, in the non-fissioned case, there are fewer processing steps. Despite of this, the Sawzall program's throughput increased when the degree of parallelism went from 1 to 2. As the degree of parallelism and the number of machines increased from 2 to 16, the increase in throughput was linear, with an 8.92× speedup on 16 machines. When further distributed across cores, the Sawzall program also experiences a large performance increase. However, the 32 replicas case showed better performance than 64 replicas. This makes sense, since the unfused Sawzall query has 5 operators, each of which was replicated 64 times ($64 * 5 = 320$), while the total number of available cores across all machines was $16 * 8 = 128$.

The CQL continuous log analyzer saw a performance improvement with fission, but only achieved at best a 2.19× speedup, with no benefit past 6 machines. Unlike Sawzall, all data items in CQL are timestamped. To maintain CQL's deterministic semantics, mergers must wait for messages from all input ports for a given timestamp. The lesson we learned was that when implementing a distributed, data-parallel CQL, we need to give more consideration to how to enforce deterministic execution efficiently. Unlike our River-CQL implementation, the original CQL did not do fission and thus did not encounter this issue.

Fusion. The Linear Road results in Figure 6 (a) illustrate the tradeoffs during fusion, which often comes at the expense of pipeline parallelism. The fusion optimization only improves performance in the 4-machine case, where it achieves a 4.02× speedup, which is overall better than the 4-machine results without fusion. Figure 6 (c) shows that fusion is particularly beneficial to the Sawzall log query analyzer. In this case, fusion eliminates much

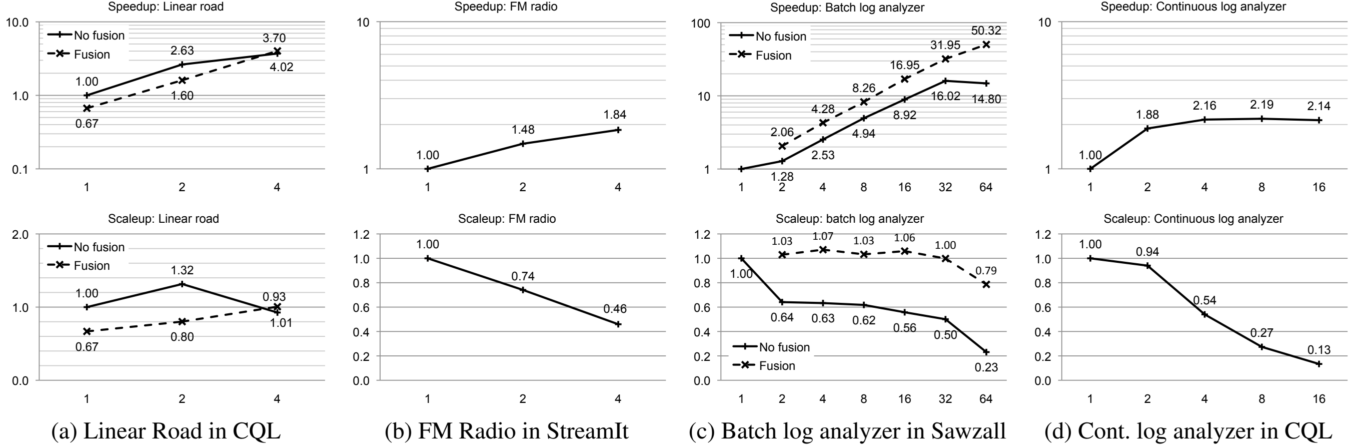


Figure 6. Speedup is the throughput relative to single machine. Scaleup is the speedup divided by number of machines in (a) and (b), and the speedup divided by the degree of parallelism in (c) and (d).

of the per-data item processing overhead, and therefore allows the fission optimization to yield much better results. Fusion combines each mapper with its splitter and each reducer with its merger. With both fusion and fission, the Sawzall log query analyzer speeds up 50.32× on 16 machines with 64 degrees of parallelism.

Calibration to Native Implementation. The placement, fission, and fusion optimizations demonstrate that River can support a diverse set of streaming optimizations, thus validating our design. While we did not focus our implementation efforts on absolute performance metrics, we were interested to see how River's performance compared to a native language implementation. The Sawzall compiler translates Sawzall programs into MapReduce jobs. We therefore implemented the Sawzall web log analyzer query as a Hadoop job. We ran both Hadoop and River on a cluster of 16 machines, with an input set of 10^9 data items, around 24GB. Hadoop was able to process this data in 96 seconds. We observed that the Hadoop job started 92 mappers during its execution, so we ran River with 92 mappers, and computed the results in 137 seconds. However, we should point out that the execution model for both systems differs. While Hadoop is a batch processing system and stores its intermediate results to disk, River is a streaming system, in which the mapper and the reducer are running in parallel, and intermediate results are kept in memory. Given that there are 16 machines with 8 cores each, we ran River again with 64 mappers and 64 reducers ($16 * 8 = 64 + 64 = 128$). Under this scenario, River completed the computation in 115 seconds.

Despite the fact that Hadoop is a well-established system that has been heavily used and optimized, the River prototype ran about 83% as fast. There are two reasons for this performance difference. First, our current fission optimizer always replicates the same number of mappers as reducers, which is not an optimal configuration. Additional optimization work is needed to adjust the map-to-reduce ratio. Second, our implementation has some unnecessary serialization overheads. A preliminary investigation suggests that eliminating those would give us performance on par with Hadoop.

6.3 Concurrency

This section explores the effectiveness of the locking algorithm from Figure 1.

Coarse-Grained Locking. For the coarse-grained locking experiment, we used the following setup, described in River's topology language:

```
(q2, $v1)      <- f1(q1, $v1     );
(q3, $v1, $v2) <- f2(q2, $v1, $v2);
(q4,      $v2) <- f3(q3,      $v2);
```

In this example, `f1` and `f3` are expensive (implemented by sleeping for a set time), whereas `f2` is cheap (implemented as the identity function). Operator instances `f1` and `f2` share the variable `$v1`, and `f2` and `f3` share the variable `$v2`. With naive locking, we would put all three variables under a single lock. That means that all three operator instances are mutually exclusive, and we would expect the total execution time to be approximately:

$$cost(\texttt{f1}) + cost(\texttt{f2}) + cost(\texttt{f3})$$

On the other hand, with our smarter locking scheme, `f1` and `f3` have no locks in common and can therefore execute in parallel. Hence, you would expect an approximate total execution time of:

$$\max\{cost(\texttt{f1}), cost(\texttt{f3})\} + cost(\texttt{f2})$$

We tested this by varying the cost (i.e., delay) of the operators `f1` and `f3` over a range of 0.001 to 1 seconds. The results in Figure 7 (a) behave as we expected, with our locking scheme performing approximately twice as fast as with the naive version.

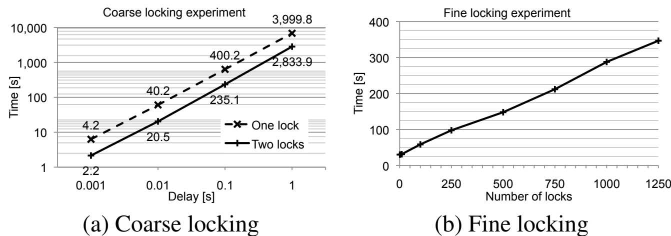
Fine-Grained Locking. As a second micro-benchmark for our locking algorithm, we wanted to quantify the overhead of locking on a fine-grained level. We used the following setup, described in River's topology language:

```
(q2, $v1, ..., $vn) <- f1(q1, $v1, ..., $vn);
(q3, $v1, ..., $vn) <- f2(q2, $v1, ..., $vn);
```

Operators `f1` and `f2` share variables `$v1` ... `$vn`, where we incremented `n` from 1 to 1,250. With naive locking, each variable would be protected by its own lock, as opposed to our locking scheme, which protects all `n` variables under a single lock. Figure 7 (b) shows that the overhead of the naive scheme grows linearly with the number of locks, just as expected.

7. Related Work

River is an **execution environment for streaming**, which runs on a distributed system and supports multiple source languages. SVM, the stream virtual machine, is a C framework for streaming on both CPU and GPU back-ends, but the paper does not describe any source-language support [21]. MSL, the multicore streaming layer, executes only the StreamIt language on the Cell architecture [38]. Erbium is a set of intrinsic functions for a C compiler for streaming



(a) Coarse locking (b) Fine locking

Figure 7. Locking experiments

on an x86 SMP, but the paper describes no source-language support [25]. And Lime is a new streaming language with three backends: Java, C, and FPGA bit-files [3]. None of SVM, MSL, Erbium, or Lime are distributed on a cluster, none of them are founded on a calculus, and they all have at most a single source language each. While we are not aware of prior work on execution environments for distributed streaming, there are various execution environments for distributed batch dataflow. MapReduce [7] has emerged as a de-facto execution environment for various batch-processing languages, including Sawzall [31], Pig Latin [29], and FlumeJava [6]. Dryad [19] is a batch execution environment that comes with its own language Nebula, but is also targeted by DryadLINQ [37]. CIEL is a batch execution environment with its own language Sky-Writing [27]. Like MapReduce, Dryad, and CIEL, River runs on a shared-nothing cluster, but in contrast, River is designed for continuous streaming, and derived from a calculus.

Stream processing is closely related to **complex event processing** (CEP), with the latter placing greater emphasis on the timing intervals between the arrival of data items processed by an operator. Because CEP functionality can be encapsulated in a streaming operator [17], River should also be able to serve as a substrate for those languages.

River comes with an eco-system for **economic source-language development**. The LINQ framework (language integrated query) also facilitates front-end implementation, but using a different approach: LINQ embeds SQL-style query syntax in a host language such as C#, and targets different back-ends such as databases or Dryad [24]. Our approach follows more traditional compiler frameworks such as Polyglot [28] or MetaBorg [5]. We use the *Rats!* parser generator to modularize grammars [14]. Our approach to modularizing type-checkers and code-generators has some resemblance to Jeannie [18], though Jeannie is not related to stream processing, and River composes more language components at a finer granularity.

Several communities have come up with similar **streaming optimizations**, but unlike River, they do not decouple the optimizations from the source-language translator and reuse them across different source languages. In *parallel databases* [8], the IL is relational algebra. Similarly to the fission optimization in this paper, parallel databases use hash-splits for parallel execution. But to do so, they rely on algebraic properties of a small set of built-in operators. In contrast, River supports an unbounded set of user-defined operators. There has been surprisingly little work on generalizing database optimizations to the more general case [9, 11], and that work is still limited to the database domain. The *StreamIt* compiler implements its own variant of fission [12]. It relies on fixed data rates and stateless operators for safety, and indeed, the StreamIt language is designed around making those easy to establish. Our fission is more general, since it can parallelize even in the presence of state. MapReduce has data-parallelism hard-wired into its design. Safety relies on keys and commutativity, but those are up to the user or a higher-level language to establish. River supports language-independent optimization by making such language-specific safety properties explicit in the IL.

8. Conclusion

This paper presents River, an execution environment for distributed stream processing. River is based on Brooklet, a stream-processing calculus [32]. Stream processing is widely used, easy to distribute, and has language diversity. By providing an execution environment for streaming, we are making a lot of common infrastructure portable and reusable, and thus facilitating further innovation. And by building our execution environment on a calculus, we are giving a clear definition of how programs should behave, thus enabling reasoning about correctness.

One contribution of this paper is to show how to maintain the properties of the calculus in the execution environment. The Brooklet calculus has a small-step operational semantics, which models execution as a sequence of atomic operator firings using pure functions. In River, operator firings still behave as if atomic, but are concurrent and do not need to be pure.

A second contribution of this paper is to make source language development economic. We show how to reuse common language components across diverse languages, thus limiting the effort of writing parsers, type checkers, and code generators to the unique features of each language. The River execution environment includes an intermediate language for describing stream graphs and operator implementations. We use code generation for operator implementations based on a simple templating mechanism.

A third contribution of this paper is to provide safe streaming optimizations that work across different source languages. Each source language is designed to make the safety of certain optimizations evident. We provide an annotation syntax, so that on the one hand, translators from the source languages can retain the safety information, while on the other hand, optimizers can work at the IL level without being source-language specific. Each optimizer uses the annotations as policy and implements the program manipulation as mechanism.

To conclude, River is an execution environment for running multiple streaming languages on a distributed system, and comes with an eco-system for making it easy to implement and optimize multiple source-languages. River is based on a calculus, giving it a clear semantics and strong correctness guarantees.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We also thank Nagui Halim for his encouragement. This work is supported by NSF CCF-1162444.

References

[1] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *Proc. 4th International Workshop on Data Mining Standards, Services, and Platforms*, pp. 27–37, Aug. 2006.

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

[3] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 89–108, Oct. 2010.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD International Conference on Management of Data*, pp. 261–272, June 2000.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 365–383, Oct. 2004.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 363–375, June 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX Symposium on Operating Systems Design and Implementation*, pp. 137–150, Dec. 2004.

[8] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[9] L. Fegaras. Optimizing queries with object updates. *Journal of Intelligent Information Systems*, 12(2–3):219–242, Mar. 1999.

[10] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1123–1134, June 2008.

[11] G. Ghelli, N. Onose, K. Rose, and J. Siméon. XML query optimization in the presence of side effects. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 339–352, June 2008.

[12] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 151–162, Oct. 2006.

[13] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proc. 10th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 291–303, Dec. 2002.

[14] R. Grimm. Better extensibility through modular syntax. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 38–51, June 2006.

[15] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421–486, Nov. 2004.

[16] Y. Gurevich, D. Leinders, and J. Van Den Bussche. A theory of stream queries. In *Proc. 11th International Conference on Database Programming Languages*, vol. 4797 of *LNCS*, pp. 153–168, Sept. 2007.

[17] M. Hirzel. Partition and compose: Parallel complex event processing. In *Proc. 6th International Conference on Distributed Event-Based Systems*, July 2012.

[18] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 19–38, Oct. 2007.

[19] M. Isard, M. B. Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel program from sequential building blocks. In *Proc. 2nd European Conference on Computer Systems*, pp. 59–72, Mar. 2007.

[20] R. Khandekar, I. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. COLA: Optimizing stream processing applications via graph partitioning. In *Proc. 10th ACM/IFIP/USENIX International Conference on Middleware*, vol. 5896 of *LNCS*, pp. 308–327, Nov. 2009.

[21] F. Labonte, P. Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. In *Proc. 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 267–277, Sept./Oct. 2004.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. 2nd IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 75–88, Mar. 2004.

[23] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[24] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 706, June 2006.

[25] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton. Erbium: A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 11–20, Oct. 2010.

[26] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, Aug. 1997.

[27] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, pp. 113–126, Mar. 2011.

[28] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Conference on Compiler Construction*, vol. 2622 of *LNCS*, pp. 138–152, Apr. 2003.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1099–1110, June 2008.

[30] P. Pietzuch, J. Ledlie, J. Schneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. 22nd International Conference on Data Engineering*, pp. 49–61, Apr. 2006.

[31] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[32] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *Proc. 19th European Symposium on Programming*, vol. 6012 of *LNCS*, pp. 507–528, Mar. 2010.

[33] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. 19th International Conference on Parallel Architectures and Compilation Techniques*, pp. 365–376, Sept. 2010.

[34] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction*, vol. 2304 of *LNCS*, pp. 179–196, Apr. 2002.

[35] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications. Technical Report MIT-LCS-TM-622, Massachusetts Institute of Technology, Dec. 2001.

[36] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proc. 9th ACM/IFIP/USENIX International Conference on Middleware*, vol. 5346 of *LNCS*, pp. 306–325, Dec. 2008.

[37] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–14, Dec. 2008.

[38] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. *ACM SIGARCH Computer Architecture News*, 36(2):18–27, May 2008.