StreamJIT: A Commensal Compiler for High-Performance Stream Programming

Jeffrey Bosboom
MIT CSAIL
jbosboom@csail.mit.edu

Sumanaruban Rajadurai
Weng-Fai Wong
National University of Singapore
{sumanaruban,wongwf}@nus.edu.sg

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

Abstract

There are many domain libraries, but despite the performance benefits of compilation, domain-specific languages are comparatively rare due to the high cost of implementing an optimizing compiler. We propose *commensal compilation*, a new strategy for compiling embedded domain-specific languages by reusing the massive investment in modern language virtual machine platforms. Commensal compilers use the host language's front-end, use host platform APIs that enable back-end optimizations by the host platform JIT, and use an autotuner for optimization selection. The cost of implementing a commensal compiler is only the cost of implementing the domain-specific optimizations. We demonstrate the concept by implementing a commensal compiler for the stream programming language StreamJIT atop the Java platform. Our compiler achieves performance 2.8 times better than the StreamIt native code (via GCC) compiler with considerably less implementation effort.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages, Data-flow languages; D.3.4 [*Programming Languages*]: Processors—Compilers, code generation, optimization

Keywords Domain-specific languages, embedded domain-specific languages

1. Introduction

Today's software is built on multiple layers of abstraction which make it possible to rapidly and cost-effectively build complex applications. These abstractions are generally provided as domain-specific libraries, such as LAPACK [3] in the linear algebra domain and ImageMagick [18] in the image processing domain. However, applications using these performance critical domain libraries lack the compiler optimization support needed for higher performance, such as judicious parallelization or fusing multiple functions for locality. On the other hand, domain-specific languages and compilers are able to provide orders of magnitude more performance by incorporating such optimizations, giving the user both simple abstractions and high performance. In the image processing domain, for example, Halide [28] programs are both faster than the hand-optimized code and smaller and simpler than the naive code.

Despite these benefits, compiled domain-specific languages are extremely rare compared to library-based implementations. This is mainly due to the enormous cost of developing a robust and capable compiler along with tools necessary for a practical language, such as debuggers and support libraries. There are attempts to remedy this by building reusable DSL frameworks such as Delite [12], but it will require a huge investment to make these systems as robust, portable, usable and familiar as C++, Java, Python, etc. Systems like Stream$^{\text{Hs}}$ [21] generate high-performance binary code with an external compiler, but incur the awkwardness of calling it through a foreign function interface and cannot reuse existing code from the host language. Performance-critical domains require the ability to add domain-specific optimizations to existing libraries without the need to build a full-fledged optimizing compiler.

In this paper we introduce a new strategy, *commensal compilation*, for implementing embedded domain-specific languages by reusing the existing infrastructure of modern language VMs. Commensal[1] compilers use the host language's front-end as their front-end, an autotuner to replace their middle-end optimization heuristics and machine model, and the dynamic language and standard JIT optimization support provided by the virtual machine as their back-end

[1] In ecology, a commensal relationship between species benefits one species without affecting the other (e.g., barnacles on a whale).

OOPSLA '14, October 20–24, 2014, Portland, Oregon, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2585-1/14/10.
http://dx.doi.org/10.1145/2660193.2660236

code generator. Only the essential complexity of the domain-specific optimizations remains. The result is good performance with dramatically less implementation effort compared to a compiler targeting virtual machine bytecode or native code. Commensal compilers inherit the portability of their host platform, while the autotuner ensures performance portability. We prove the concept by implementing a commensal compiler for the stream programming language StreamJIT atop the Java platform. Our compiler achieves performance on average 2.8 times better than StreamIt's native code (via GCC) compiler with considerably less implementation effort.

As StreamJIT makes no language extensions, user code written against the StreamJIT API is compiled with standard `javac`, and for debugging, StreamJIT can be used as "just a library", in which StreamJIT programs run as any other Java application. For performance, the StreamJIT commensal compiler applies domain-specific optimizations to the stream program and computes a static schedule for automatic parallelization. Writing portable heuristics is very difficult because the best combination of optimizations depends on both the JVM and the underlying hardware; instead, the StreamJIT compiler uses the OpenTuner extensible autotuner [6] to make its optimization decisions. The compiler then performs a simple pattern-match bytecode rewrite and builds a chain of MethodHandle (the JVM's function pointers) combinators to be compiled by the JVM JIT. The JVM has complete visibility through the method handle chain, enabling the full suite of JIT optimizations. By replacing the traditional compiler structure with this existing infrastructure, we can implement complex optimizations such as fusion of multiple filters, automatic data parallelization and distribution to a cluster with only a small amount of compilation, control and communication code.

Our specific contributions are

- commensal compilation, a novel compilation strategy for embedded domain-specific languages, that dramatically reduces the effort required to implement a compiled domain-specific language,
- a commensal compiler for the stream programming language StreamJIT, which demonstrates the feasibility of commensal compilation by achieving on average 2.8 times better performance than StreamIt's native code compiler with an order of magnitude less code.

Section 2 explains commensal compilers abstractly, separate from any particular language. Section 3 discusses related work in compilers and stream programming. Section 4 gives an overview of the Java-embedded stream programming language StreamJIT. Section 5 describes specifics of implementing commensal compilers in Java, using StreamJIT as an example. Section 6 describes the StreamJIT API and workflow, Section 7 describes interpreted mode, Section 8 describes the commensal compiler in detail, Section 9 describes how the StreamJIT compiler uses an autotuner, and Section 10 describes how StreamJIT programs are distributed to multiple machines. Section 11 evaluates StreamJIT's performance against the StreamIt native code compiler and Section 12 concludes.

2. Commensal Compiler Design

Commensal compilers are defined by their reuse of existing infrastructure to reduce the cost of implementing an optimizing compiler. In this section we explain the design of commensal compiler front-, middle- and back-ends.

Front-end Commensal compilers compile embedded domain-specific languages expressed as libraries (not language extensions) in a host language that compiles to bytecode for a host platform virtual machine. To write a DSL program, users write host language code extending library classes, passing lambda functions, or otherwise using standard host language abstraction features. This includes code to compose these new classes into a program, allowing the DSL program to be generated at runtime in response to user input. This host language code is compiled with the usual host language compiler along with the rest of the application embedding the DSL program. Implementing a DSL using a library instead of with a traditional front-end reduces implementation costs, allows users to write code in a familiar language, reuse existing code from within the DSL program, easily integrate into the host application build process, and use existing IDEs and analysis tools without adding special language support.

At run time, the compiled code can be executed directly, as "just a library", analogous to a traditional language interpreter. In this interpreted mode, users can set breakpoints and inspect variables with standard graphical debuggers. By operating as a normal host language application, the DSL does not require debugging support code.

Middle-end For increased performance, a commensal compiler can reflect over the compiled code, apply domain-specific optimizations, and generate optimized code. A commensal compiler leaves standard compiler optimizations such as constant subexpression elimination or loop unrolling to the host platform, so its intermediate representation can be at the domain level, tailored to the domain-specific optimizations.

In a few domains, using simple algorithms or heuristics to guide the domain-specific optimizations results in good (or good enough) performance, but most domains require a nuanced understanding of the interaction between the program and the underlying hardware. In place of complex heuristics and machine performance models, commensal compilers can delegate optimization decisions to an autotuner, simultaneously reducing implementation costs and ensuring performance portability.

Back-end Many platforms provide APIs for introspectable expressions or dynamic language support that can be reused for code generation in place of compiling back to bytecode. Code generators using these APIs can compose normal host language code instead of working at the bytecode level, keeping the code generator modular enough to easily swap implementation details at the autotuner's direction. Finally, code generated through these APIs can include constant references to objects and other runtime entities, allowing the host platform's JIT compiler to generate better code than if the DSL compiled to bytecode.

The compiled DSL code runs on the host platform as any other code, running in the same address space with the same data types, threading primitives and other platform features, so interaction with the host application is simple; a foreign function interface is not required. Existing profilers and monitoring tools continue to report an accurate picture of the entire application, including the optimized DSL program.

Result The result of these design principles is an efficient compiler without the traditional compiler costs: the front-end is the host front-end, the middle-end eliminates all but domain-specific optimizations using an autotuner, and the back-end leaves optimizations to the host platform. Only the essential costs of the domain-specific optimizations remains.

In this paper we present a commensal compiler for the stream programming language StreamJIT. The host language is Java, the host platform is the Java Virtual Machine, and code generation is via the MethodHandle APIs originally for dynamic language support, with a small amount of bytecode rewriting. But commensal compilers are not specific to Java. The .NET platform's `System.Reflection.Emit` [2] allows similar bytecode rewriting and the Expression Trees [1] feature can replace method handles.

3. Related Work

The Delite DSL compiler framework [12] uses Lightweight Modular Staging [30] to build a Scala-level intermediate representation (IR), which can be raised to Delite IR to express common parallel patterns like foreach and reduce. DSLs can provide additional domain-specific IR nodes to implement, e.g., linear algebra transformations. The Delite runtime then compiles parts of the IR to Scala, C++ or CUDA and heuristically selects from these compiled implementations at runtime. Delite represents the next step of investment beyond commensal compilers; where commensal compilers reuse existing platform infrastructure, Delite is a platform unto itself. The Forge meta-DSL [33] adds additional abstractions to shield DSL authors and users from having to understand Delite.

Truffle [43], built on top of the Graal extensible JIT compiler [44], aims to efficiently compile abstract syntax tree interpreters by exploiting their steady-state type information. Truffle optimizes languages that have already invested in a separate front-end and interpreter by adding specialized node implementations, while our method handle strategy enables the JVM's existing type profiling and devirtualization. Our compiler is portable to all Java 7 JVMs, while Truffle is dependent on Graal.

Java 7 introduced MethodHandles along with the `invokedynamic` instruction [31] to support efficient implementation of dynamic languages. Major JVM JIT compilers implement optimizations for code using method handles ([34] describes the OpenJDK JIT compiler changes; [17] describes J9's). JRuby [23] and Jython [11], among other languages, are using them for this purpose. Garcia [15] proposed using method handles encapsulating individual statements to implement primitive specializations for Scala. JooFlux [27] uses method handles for live code modification during development and aspect-oriented programming. To our knowledge, we are the first to use method handles for DSL code generation.

Domain-specific languages may want to check semantic properties not enforced by a general-purpose compiler such as `javac`. Java 8 introduced type annotations [36], which can be used to check custom properties at compile time via an annotation processor. The Checker Framework [13] uses type annotations to extend Java's type system, notably to reason about uses of `null`. The commensal compiler can also perform checks as part of its compilation; for example, StreamJIT enforces that pipelines and splitjoins do not contain themselves during compilation.

StreamJIT is strongly inspired by StreamIt [37], a synchronous dataflow language programmed with single-input, single-output filters composed using pipelines and splitjoins (called structured streams by analogy with structured programming). Together, the combination of static data rates, lack of aliasing, and structured streams allows the StreamIt compiler [16] great insight into the program structure, allowing filter fusion or fission to arrive at an optimal amount of parallelism for a particular machine. The StreamIt compiler is heuristically-driven and its heuristics require modification when it is ported to a new machine. As a statically-compiled language, StreamIt programs must commit to their structure at compile time; applications needing parameterized streams must compile many variants and select one at runtime.

The Feldspar DSL for signal processing [10] embeds a low-level C-like language in Haskell, generating parallelized C code for a C compiler; Stream$^{\text{Hs}}$ [21] embeds StreamIt in Haskell to add metaprogramming capabilities StreamIt lacks, then invokes the usual StreamIt compiler. Commensal compilers use their host language directly, so users write programs in a familiar language and the compiled programs (not just the language) can be embedded without using a foreign function interface.

Other streaming languages have dropped static data rates in favor of dynamic fork-join parallelism, becoming more expressive at the cost of performance. XJava [24–26] is a Java extension also closely inspired by StreamIt. Its runtime exposes a small number of tuning parameters; [26] gives heuristics performing close to exhaustive search. Java 8 introduces `java.util.stream`[35], a fluent API for bulk operations on collections similar to .NET's LINQ [22]. Without data rates, fusion and static scheduling are impossible, so these libraries cannot be efficiently compiled.

StreamFlex [32] is a Java runtime framework for real-time event processing using a stream programming paradigm; while it uses Java syntax, it requires a specially-modified JVM to provide latency bounds.

Lime [8] is a major Java extension aiming to exploit both conventional homogeneous multicores and heterogeneous architectures including reconfigurable elements such as FPGAs. Similar to Delite, the Lime compiler generates Java code for the entire program, plus OpenCL code for GPUs and Verilog for FPGAs for subsets of the language, then the Liquid Metal runtime [9] selects which compiled code to use.

Dryad [19] is a low-level distributed execution engine supporting arbitrary rateless stream graphs. DryadLINQ [46] heuristically maps LINQ expressions to a stream graph for a Dryad cluster. Spark [47] exposes an interface similar to DryadLINQ's, but focuses on resiliently maintaining a working set. Besides lacking data rates, these systems are more concerned with data movement than StreamJIT's distributed runtime, which distributes computation.

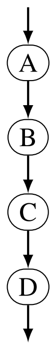
ATLAS [42] uses autotuning to generate optimized linear algebra kernels. SPIRAL [45] autotunes over matrix expressions to generate FFTs; [14] also tunes FFTs. Hall et al [39] use autotuning to find the best order to apply loop optimizations. PetaBricks [4] uses autotuning to find the best-performing combination of multiple algorithm implementations for a particular machine (e.g, trading off between mergesort, quicksort and insertion sort). These systems use autotuning to generate optimized implementations of small, constrained kernels, whereas StreamJIT tunes larger programs.

SiblingRivalry [5] is a system for robust online autotuning that divides the machine in half and runs the current best program variant in parallel with tuning trials, ensuring at least the current best performance. StreamJIT's online autotuning could adopt this system.

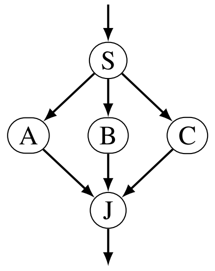
Wang et al [41] apply machine learning to train a performance predictor for filter fusion and fission in StreamIt graphs, then search the space of partitions for the best predicted value, effectively learning a heuristic. StreamJIT uses an autotuner to search the space directly to ensure performance portability.

4. StreamJIT Overview

In this section we present an overview of StreamJIT, a Java-embedded stream programming language derived from StreamIt, so we can use StreamJIT as an example when explaining how to implement a commensal compiler in Java in Section 5.



(a) Pipelines compose one-to-one elements (filters, pipelines or splitjoins) by connecting each element's output to the input of the next element.



(b) Splitjoins compose a splitter, a joiner, and one or more one-to-one elements (filters, pipelines or splitjoins) by connecting the splitter's outputs to the inputs of the branches and the outputs of the branches to the joiner's input.

Figure 1: StreamJIT composition structures.

StreamJIT programs share the same structure as StreamIt programs, being stream graphs composed of filters, splitters and joiners (collectively called *workers* as they all have work methods specifying their behavior). *Filters* are single-input, single-output workers; despite their name, they need not remove items from the stream. *Splitters* and *joiners* have multiple outputs and inputs respectively. All workers declare static *peek rates* stating how many items they examine on each input, *pop rates* stating how many of those items they consume, and *push rates* stating how many items they produce on each input for each execution of their work method.

StreamJIT programs are *stream graphs* built using pipelines and splitjoins, which compose workers vertically or horizontally, respectively (see Figure 1). Pipelines connect elements in sequence, with the output of each element connected to the input of the next (see Figure 1a). Splitjoins connect the outputs of a splitter to the input of each of the splitjoin's branch and the output of each branch to the inputs of a joiner (see Figure 1b). Splitters and joiners have multiple outputs and inputs respectively, so they can only appear at the beginning or end of a splitjoin. Filters, pipelines and splitjoins are all single-input single-output, and thus can be easily composed with one another.

Data items flow through the stream graph as it executes. Each worker can be executed any time its input requirements (peek and pop rates) are met, and multiple executions of stateless workers can proceed in parallel. As all communication between workers occurs via the stream graph edges, a compiler is free to select an execution schedule that exploits data, task and pipeline parallelism.

5. Commensal Compilers in Java

This section presents techniques for implementing commensal compilers targeting the Java platform. While this section uses examples from StreamJIT, the focus is on the compiler's platform-level operations.

5.1 Front-end

Commensal compilers use their host language's front-end for lexing and parsing, implementing their domain-specific abstractions using the host language's abstraction features. In the case of Java, the embedded domain-specific language (EDSL) is expressed using classes, which the user can extend and instantiate to compose an EDSL program.

The basic elements of a StreamJIT program are instances of the abstract classes `Filter`, `Splitter` or `Joiner`. User subclasses pass rate information to the superclass constructor and implement the `work` method using `peek`, `pop` and `push` to read and write data items flowing through the stream. (`Filter`'s interface for subclasses is shown in Figure 2; see Figure 4 for an example implementation.)

User subclasses are normal Java classes and the `work` method body is normal Java code, which is compiled to bytecode by `javac` as usual. By reusing Java's front-end, StreamJIT does not need a lexer or parser, and can use complex semantic analysis features like generics with no effort.

`Filter` contains implementations of `peek`, `pop` and `push`, along with some private helper methods, allowing `work` methods to be executed directly by the JVM without using the commensal compiler. By using the `javac`-compiled bytecodes directly, users can debug their filters with standard graphical debuggers such as those in the Eclipse and NetBeans IDEs. In this "interpreted mode", the EDSL is simply a library. (Do not confuse the EDSL interpreted mode with the JVM interpreter. The JVM JIT compiler can compile EDSL interpreted mode bytecode like any other code running in the JVM.)

StreamJIT provides common filter, splitter and joiner subclasses as part of the library. These built-in subclasses are implemented the same way as user subclasses, but because they are known to the commensal compiler, they can be intrinsified. For example, StreamJIT's built-in splitters and joiners can be replaced using index transformations (see Section 8.2).

```
public abstract class Filter<I, O>
 extends Worker<I, O>
 implements OneToOneElement<I, O> {
  public Filter(int popRate, int pushRate);
  public Filter(int popRate, int pushRate,
      int peekRate);
  public Filter(Rate popRate, Rate pushRate,
      Rate peekRate);

  public abstract void work();

  protected final I peek(int position);
  protected final I pop();
  protected final void push(O item);
}
```

Figure 2: `Filter`'s interface for subclasses. Subclasses pass rate information to one of `Filter`'s constructors and implement `work` using `peek`, `pop` and `push` to read and write data items flowing through the stream. See Figure 4 for an example implementation.

5.2 Middle-end

A commensal compiler's middle-end performs domain-specific optimizations, either using simple heuristics or delegating decisions to an autotuner. Commensal compilers use high-level intermediate representations (IR) tailored to their domain. In Java, input programs are object graphs, so the IR is typically a tree or graph in which each node decorates or mirrors a node of the input program. Basic expression-level optimizations such as common subexpression elimination are left to the host platform's JIT compiler, so the IR need not model Java expressions or statements. (The back-end may need to understand bytecode; see Section 5.3).

The input program can be used directly as the IR, using package-private fields and methods in the library superclasses to support compiler optimizations. Otherwise, the IR is built by traversing the input program, calling methods implemented by the user subclasses or invoking reflective APIs to obtain type information or read annotations. In particular, reflective APIs provide information about type variables in generic classes, which serve as a basis for type inference through the DSL program to enable unboxing.

The StreamJIT compiler uses the unstructured stream graph as its intermediate representation, built from the input stream graph using the visitor pattern. In addition to the filter, splitter or joiner instance, the IR contains input and output type information recovered from Java's reflection APIs. Type inference is performed when exact types have been lost to Java's type erasure (see Section 8.3). Other IR attributes support StreamJIT's domain-specific optimizations; these include the schedule (Section 8.1) and index functions used for built-in splitter and joiner removal (Section 8.2). The high-level IR does not model lower-level details such as the expressions or control flow inside `work` methods, as StreamJIT leaves optimizations at that level to the JVM JIT compiler.

StreamJIT provides high performance by using the right combination of data, task and pipeline parallelism for a particular machine and program. Finding the right combination heuristically requires a detailed model of each machine the program will run on; Java applications additionally need to understand the underlying JVM JIT and garbage collector. As StreamJIT programs run wherever Java runs, the heuristic approach would require immense effort to develop and maintain many machine models. Instead, the StreamJIT compiler delegates its optimization decisions to the OpenTuner extensible autotuner as described in Section 9.

Autotuner use is not essential to commensal compilers. Commensal compilers for languages with simpler optimization spaces or that do not require maximum performance might prefer the determinism of heuristics over an autotuner's stochastic search.

5.3 Back-end

The back-end of a commensal compiler generates code for further compilation and optimization by the host platform. A commensal compiler targeting the Java platform can either emit Java bytecode (possibly an edited version of the user subclasses' bytecode) or generate a chain of `java.lang.invoke.MethodHandle` objects which can be invoked like function pointers in other languages.

The Java Virtual Machine has a stack machine architecture in which bytecodes push operands onto the operand stack, then other bytecodes perform some operation on the operands on top of the stack, pushing the result back on the stack. Each stack frame also contains local variable slots with associated bytecodes to load or store slots from the top of the stack. Taking most operands from the stack keeps Java bytecode compact. In addition to method bodies, the Java bytecode format also includes symbolic information about classes, such as the class they extend, interfaces they implement, and fields they contain. Commensal compilers can emit bytecode from scratch through libraries such as ASM [7] or read the bytecode of existing classes and emit a modified version. Either way, the new bytecode is passed to a `ClassLoader` to be loaded into the JVM.

Method handles, introduced in Java 7, act like typed function pointers for the JVM. Reflective APIs can be used to look up a method handle pointing to an existing method, then later invoked through the method handle. Method handles can be partially applied as in functional languages to produce a bound method handle that takes fewer arguments. Method handles are objects and can be used as arguments like any

```
private static void loop(MethodHandle loopBody,
    int begin, int end, int increment) throws
    Throwable {
  for (int i = begin; i < end; i += increment)
    loopBody.invokeExact(i);
}
```

Figure 3: This loop combinator invokes the `loopBody` argument, a method handle taking one `int` argument, with every `increment`-th number from `begin` to `end`. StreamJIT uses similar loop combinators with multiple indices to implement a schedule of filter executions (see Section 8.1).

other object, allowing for method handle combinators (see the loop combinator in Figure 3). These combinators can be applied repeatedly to build a chain of method handles encapsulating arbitrary behavior. The bound arguments are constants to the JVM JIT compiler, so if a method handle chain is rooted at a constant (such as a `static final` variable), the JVM JIT can fully inline through all the method handles, effectively turning method handles into a code generation mechanism.

StreamJIT uses bytecode rewriting to enable use of method handles for code generation. The StreamJIT compiler copies the bytecode of each filter, splitter or joiner class that appears in the stream graph, generating a new work method with calls to `peek`, `push` and `pop` replaced by invocations of new method handle arguments. To support data-parallelization, initial read and write indices are passed as additional arguments and used in the method handle invocations. (See Section 8.4 for details and Figure 11 for an example result.)

For each instance in the stream graph, a method handle is created pointing to the new work method. The method handle arguments introduced by the bytecode rewriting are bound to method handles that read and write storage objects (see Section 8.5). The resulting handle (taking initial read and write indices) is bound into loop combinators similar to the one in Figure 3 to implement the schedule, producing one method handle chain per thread. To make the root method handles of these chains compile-time constants for the JVM JIT, the StreamJIT compiler emits bytecode that stores them in `static final` fields and immediately calls them.

Because the root method handles are JIT-compile-time constants, the methods they point to will not change, so the JVM JIT can inline the target methods. The bound arguments are also constants, so all the method handles will inline all the way through the chain, loops with constant bounds can be unrolled, and the addresses of the storage arrays can be baked directly into the generated native code. Using method handles for code generation is easier than emitting bytecode, yet produces faster machine code.

```
class LowPassFilter
    extends Filter<Float, Float> {
  private final float rate, cutoff;
  private final int taps, decimation;
  private final float[] coeff;
  LowPassFilter(float rate, float cutoff,
      int taps, int decimation) {
    //pop, push and peek rates
    super(1 + decimation, 1, taps);
    /* ...initialize fields... */
  }
  public void work() {
    float sum = 0;
    for (int i = 0; i < taps; i++)
      sum += peek(i) * coeff[i];
    push(sum);
    for (int i = 0; i < decimation; i++)
      pop();
    pop();
  }
}
```

Figure 4: LowPassFilter from the FMRadio benchmark. Peeks (nonconsuming read) at its input, pushes its output, and pops (consuming read) the input it is finished with. Pushes and pops can be freely intermixed, but the peek-push-pop style is common in our benchmarks.

6. StreamJIT API and Workflow

We now break down the StreamJIT workflow to describe what occurs at development time (when the user is writing their code), at compile time, and at run time.

Development time Users write workers by subclassing `Filter`, `StatefulFilter`, `Splitter`, or `Joiner`, passing their data rates to the superclass constructor. The worker's computation is specified by the `work` method, operating on data items read from input with the `pop` and `peek` methods and writing items to the output with `push`. (See Figure 4.) Work methods may contain nearly arbitrary Java code (including library calls), with some restrictions to permit automatic parallelization:

- Filters maintaining state must extend `StatefulFilter` to avoid being data-parallelized.
- To avoid data races, once a data item is pushed to the output, workers should not modify it, and to avoid deadlocking with the StreamJIT runtime, workers should not perform their own synchronization.

StreamJIT does not attempt to verify these properties. While simple workers can be easily verified, workers that call into libraries would require sophisticated analysis.

```
private static final class BandPassFilter
    extends Splitjoin<Float, Float> {
  private BandPassFilter(float rate,
    float low, float high, int taps) {
    super(new DuplicateSplitter<Float>(),
      new SubtractJoiner<Float>(),
      new LowPassFilter(rate, low, taps, 0),
      new LowPassFilter(rate, high, taps, 0))
    );
  }
}
```

Figure 5: BandPassFilter from the FMRadio benchmark. This splitjoin is instantiated once for each band being processed.

Users also write code to assemble stream graphs using `Pipelines` and `Splitjoin` objects, which contain filters or other pipelines or splitjoins. In addition to instantiating `Pipeline` and `Splitjoin`, users can subclass them to facilitate reuse in multiple places in the stream graph or in other stream programs (see Figure 5).

Finally, the constructed stream graph along with input and output sources (e.g., input from an `Iterable` or file, output to a `Collection` or file) is passed to a `StreamCompiler` for execution. The graph will execute until the input is exhausted (when all workers' input rates cannot be satisfied), which the application can poll or wait for using a `CompiledStream` object returned by the `StreamCompiler`.

Importantly, stream graph construction and compilation occurs at run time, and so can depend on user input; for example, a video decoder can be instantiated for the video's size and chroma format. In StreamIt, a separate stream graph must be statically compiled for each set of parameters, then the correct graph loaded at run time, leading to code bloat and long compile times when code changes are made (as each graph is recompiled separately).

Compile time StreamJIT does not make any language extensions, so user code (both workers and graph construction code) is compiled with a standard Java compiler such as `javac`, producing standard Java class files that run on unmodified JVMs. Thus integrating StreamJIT into an application's build process merely requires referencing the StreamJIT JAR file, without requiring the use of a separate preprocessor or compiler.

Run time At run time, user code constructs the stream graph (possibly parameterized by user input) and passes it to a `StreamCompiler`. During execution, there are two levels of interpretation or compilation. The StreamJIT level operates with the workers in the stream graph, while the JVM level operates on all code running in the JVM (StreamJIT

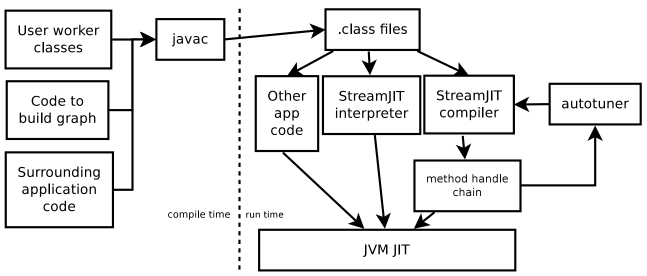


Figure 6: StreamJIT workflow. At compile time, user worker classes and graph construction code are compiled by `javac` just like the surrounding application code. At runtime, the StreamJIT interpreter runs `javac`'s output as is, while the compiler optimizes and builds a method handle chain. Either way, the executed code can be compiled by the JVM JIT like any other code. The compiler can optionally report performance to the autotuner and recompile in a new configuration (see Section 9).

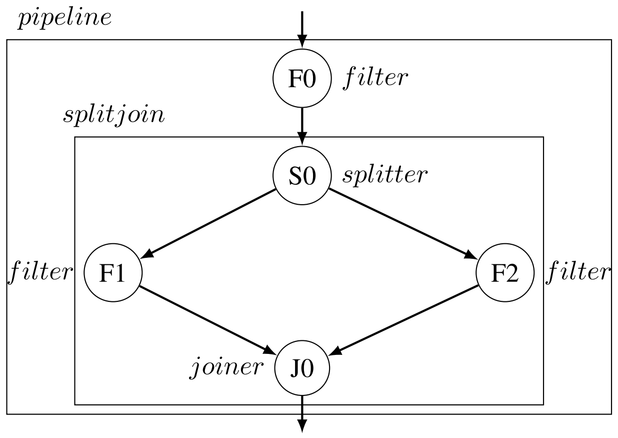


Figure 7: A simple stream graph in which a filter and a splitjoin are enclosed in a pipeline. This graph's data rates are shown in Figure 9.

or otherwise). (See Figure 6.) The two levels are independent: even when StreamJIT is interpreting a graph, the JVM is switching between interpreting bytecode and running just-in-time-compiled machine code as usual for any Java application. The user's choice of `StreamCompiler` determines whether StreamJIT interprets or compiles the graph.

The StreamJIT runtime can partition StreamJIT programs to run in multiple JVMs (usually on different physical machines). The partitions are then interpreted or compiled in the same way as the single-JVM case, with data being sent via sockets between partitions. The partitioning is decided by the autotuner, as described in Section 10.

7. StreamJIT Interpreted Mode

In interpreted mode, the StreamJIT runtime runs a fine-grained pull schedule (defined in [37] as executing upstream workers the minimal number of times required to execute the last worker) on a single thread using the code as compiled by `javac`, hooking up `peek`, `pop` and `push` behind the scenes to operate on queues. Because interpreted mode uses the original bytecode, users can set breakpoints and inspect variables in StreamJIT programs with their IDE's graphical debugger.

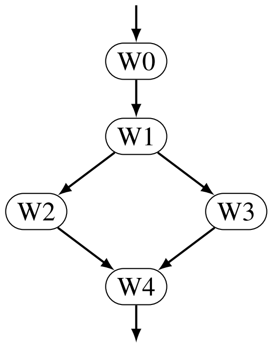
8. The StreamJIT Commensal Compiler

In compiled mode, the StreamJIT commensal compiler takes the input stream graph, applies domain-specific optimizations, then (using worker rate declarations) computes a parallelized schedule of worker executions. To implement this schedule, the compiler emits new bytecode from the `javac` bytecode by replacing `peek`, `pop` and `push` with invocations of method handles that read and write backing storage, reflects these new work methods as method handles, and composes them with storage implementations and loop combinators. The StreamJIT runtime then repeatedly calls these method handles until the schedule's input requirements cannot be met, after which execution is transferred to the interpreter to process any remaining input and drain buffered data. See Section 5.3 for background information on how bytecode and method handles are uesd in Java-based commensal compilers.

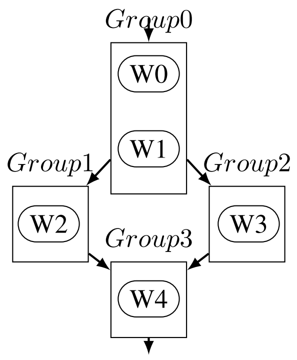
This section describes the compiler flow in temporal order. Sections 8.4 and 8.7 describe code generation; the other sections explain StreamJIT's domain-specific optimizations. In this section we note which decisions are made by the autotuner, but defer discussion of the search space parameterization until Section 9. All examples are based on the stream graph in Figure 7.

8.1 Fusion and scheduling

Fusion The compiler first converts the input stream graph made of pipelines and splitjoins into an unstructured stream graph containing only the workers (from Figure 7 to Figure 8a). The compiler then fuses workers into groups (from Figure 8a to Figure 8b). Groups have no internal buffering, while enough data items are buffered on inter-group edges to break the data dependencies between groups (software pipelining [29]). This enables independent data-parallelization of each group without synchronization. Each worker initially begins in its own group. As directed by the autotuner, each group may be fused upward, provided it does not peek and all of its predecessor workers are in the same group. Peeking workers are never fused upward as they would introduce buffering within the group. Stateful filters (whose state is held in fields, not as buffered items) may be fused, but their statefulness infects the entire group, preventing it from being data-parallelized.



(a) The unstructured form of the example graph in Figure 7. The pipeline and splitjoin have been discarded.

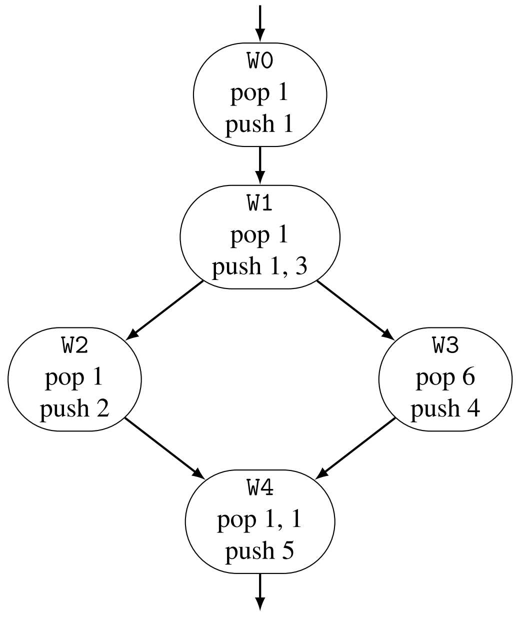


(b) The example graph from Figure 7 after fusion. Workers `W0` and `W1` are fused together.

Figure 8: Grouping and fusion

Scheduling To execute with minimal synchronization, the compiler computes a *steady-state schedule* of filter executions which leaves the buffer sizes unchanged [20]. Combined with software pipelining between groups, synchronization is only required at the end of each steady-state schedule. The buffers are filled by executing an *initialization schedule*. When the input is exhausted, the stream graph is *drained* by migrating buffered data and worker state to the interpreter, which runs for as long as it has input. Draining does not use a schedule, but migration to the interpreter uses *liveness information* tracking the data items that are live in each buffer.

The compiler finds schedules by formulating integer linear programs. The variables are execution counts (of workers or groups); the constraints control the buffer delta using push and pop rates as coefficients. For example, to leave the buffer size unchanged between worker x_1 with push rate 3 and worker x_2 with pop rate 4, the compiler would add the constraint $3x_1 - 4x_2 = 0$; to grow the buffer by (at least) 10 items, the constraint would be $3x_1 - 4x_2 \geq 10$. (See Figure 9 for a steady-state example.) The resulting programs are solved using lp_solve[2]. While integer linear programming is NP-hard, and thus slow in the general case, in our experience StreamJIT schedules are easily solved.



$$\begin{gathered}
\texttt{W0} - \texttt{W1} = 0\\
\texttt{W1} - \texttt{W2} = 0\\
3\texttt{W1} - 6\texttt{W3} = 0\\
2\texttt{W2} - \texttt{W4} = 0\\
4\texttt{W3} - \texttt{W4} = 0\\
\texttt{W0} \geq 0\\
\texttt{W1} \geq 0\\
\texttt{W2} \geq 0\\
\texttt{W3} \geq 0\\
\texttt{W4} \geq 0\\
\texttt{W0} + \texttt{W1} + \texttt{W2} + \texttt{W3} + \texttt{W4} \geq 1
\end{gathered}$$

Figure 9: An intra-group steady-state schedule example, showing the integer linear program generated if all workers in Figure 7 are fused into one group. Each variable represents the number of executions of the corresponding worker in the steady-state schedule. The first five constraints (one per edge) enforce that the buffer sizes do not change after items are popped and pushed. The second five (one per worker) enforce that workers do not execute negative times, and the last enforces that at least one worker is executed (ensuring progress is made). Inter-group scheduling is the same, except with groups instead of workers.

Steady-state scheduling is hierarchical. First intra-group schedules are computed by minimizing the total number of executions of the group's workers, provided each worker ex-

[2] http://lpsolve.sourceforge.net/

ecutes at least once and buffer sizes are unchanged. Then the inter-group schedule minimizes the number of executions of each group's intra-group schedule, provided each group executes at least once and buffer sizes are unchanged (see Figure 9). The inter-group schedule is multiplied by an autotuner-directed factor to amortize the cost of synchronizing at the end-of-steady-state barrier.

The compiler then computes an initialization inter-group schedule to introduce buffering between groups as stated above. Each group's inputs receive at least enough buffering to run a full steady-state schedule (iterations of the intra-group schedule given by the inter-group schedule), plus additional items if required to satisfy a worker's peek rate. The initialization and steady-state schedules share the same intra-group schedules.

Based on the initialization schedule, the compiler computes the data items buffered on each edge. This *liveness information* is updated during splitter and joiner removal and used when migrating data from initialization to steady-state storage and during draining (all described below).

8.2 Built-in splitter and joiner removal

The built-in `RoundrobinSplitter`, `DuplicateSplitter`, and `RoundrobinJoiner` (and their variants taking weights) classes split or join data in predictable patterns without modifying them. As directed by the autotuner, these workers can be replaced by modifying their neighboring workers to read (for splitters) or write (for joiners) in the appropriate pattern. For example, splitter `S0` in Figure 7 has two downstream workers and it distributes one item to its first child (`F1`) and 3 items to its second child (`F2`) in turn. It can be removed by modifying its downstream workers to read from indices $4i$ and $4*\lfloor i/3 \rfloor + i\%3+1$ in its upstream storage (see Figure 10). Joiner removal results in similar transformations of write indices. Nested splitjoins may result in multiple removals, in which the index transformations are composed.

Instances of the built-in `Identity` filter, which copies its input to its output, could also be removed at this stage (with no index transformation), but the current implementation does not.

When a worker is removed, liveness information from its input edge(s) is propagated to the input edges of its downstream workers via simulation. Because the interpreter does not perform removal, the liveness information remembers on which edge the data item was originally buffered and its index within that buffer so it can be returned to its original location for draining. When removing a `DuplicateSplitter`, the compiler duplicates the liveness information, but marks all but one instance as a duplicate so only one item is restored when draining.

Splitter and joiner removal is performed after scheduling to ensure each removed worker executes an integer number of times (enforced by the ILP solver). Otherwise, splitters and joiners may misdistribute their items during initialization or draining.

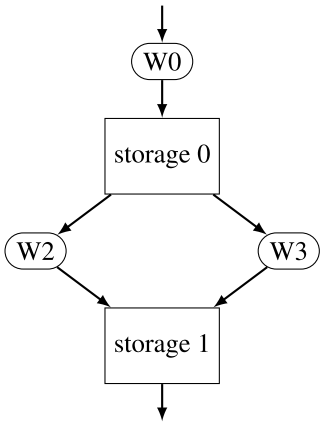


Figure 10: The example stream graph from Figure 7 after splitter and joiner removal. `W2` and `W3` read directly from `W0`'s output storage and write directly into the overall graph output storage, avoiding the copies performed by the splitter and joiner.

8.3 Type inference and unboxing

StreamJIT workers define their input and output types using Java generics. The types of a `Filter<Integer, Integer>` can be recovered via reflection, but reflecting a `Filter<I,O>` only provides the type variables, not the actual type arguments of a particular instance. Using the Guava library's `TypeToken` [40], the compiler follows concrete types through the graph to infer the actual arguments. For example, if a `Filter<Float, Integer>` is upstream of a `Filter<T, List<T>>`, the compiler can infer `T` to be `Integer`. Storage types are then inferred to be the most specific common type among the output types of workers writing to that storage. This analysis depends on finding at least some concrete types via reflection; if the entire graph's types are type variables, the analysis cannot recover the actual types. In practice, most graphs have enough "inference roots" to find uses of wrapper types for unboxing.

After type inference, worker input and output types and edge storage types may be unboxed, as directed by the autotuner. Input, output and storage decisions are independent; the JVM will introduce boxing or unboxing later if, e.g., a storage location was unboxed while a downstream worker's input was not. (Primitives are stored more compactly than wrapper objects, which may make up for the boxing cost with better cache usage.) Separate unboxing decisions are made for each instance of a worker class (indeed, each instance may have different input and output types). However, the multiple outputs or inputs of a splitter or joiner instance will either all be unboxed or not unboxed.

8.4 Bytecode generation

The original worker bytecode as compiled by `javac` (and used by the interpreter) assumes a serial execution strategy, popping and pushing items in queues. To run multiple iterations of a worker in parallel, the compiler must transform the bytecode. For each worker class, the compiler creates an archetype class containing one or more `static` *archetypal work methods*. One archetypal work method is created for each pair of actual input and output types for workers of that class; for example, a `HashCodeFilter<T, Integer>` could result in generation of `HashCodeFilterArchetype` containing `workObjectInteger`, `workIntegerInteger` and `workObjectint` methods. Because workers may use private fields, a seperate *field holder* class is also created to work around access control, containing copies of the worker class fields accessible by the archetype class.

All filter work methods share the signature `void (FieldHolderSubclass state, MethodHandle readHandle, MethodHandle writeHandle, int readIndex, int writeIndex)`. The read and write handles provide indexed access to the worker's input and output channels, while the indices define which iteration of the worker is being executed. If a worker has pop rate o and push rate u, the tth iteration has read index to and write index tu. Splitters and joiners have multiple outputs or inputs respectively, so their work methods take arrays of indices and their read or write handles take an additional parameter selecting the channel.

The original worker's work method bytecode is cloned into each archetypal work method's body. References to worker class fields are remapped to state holder fields. `peek(i)` and `pop()` calls are replaced with read handle invocations at the read index (plus `i` for peeks); `pop()` additionally increments the read index. Similarly, `push(x)` is replaced by writing `x` at the current write index via the write handle, then incrementing the write index. If the input or output types have been unboxed, existing boxing or unboxing calls are removed. Figure 11 shows the result of rewriting the example filter from Figure 4.

A bytecode optimization is performed for the common case of filters that peek, push and pop in that order (as in the example in Figure 4). The pops are translated into unused read handle invocations. If splitter or joiner removal introduced index transformations, the JVM JIT cannot always prove the read handle calls to be side-effect-free due to potential index-out-of-bounds exceptions. The StreamJIT compiler knows the indices will always be valid based on the schedule, so the unused invocations can be safely removed. The JVM can then remove surrounding loops and read index increments as dead code. This is the only bytecode-level optimization the StreamJIT compiler performs; teaching the JVM JIT about more complex index expressions may obviate it.

```java
class LowPassFilterArchetype {
  public static void workfloatfloat(
      LowPassFilterStateHolder state,
      MethodHandle readHandle,
      MethodHandle writeHandle,
      int readIndex, int writeIndex) {
    float sum = 0;
    for (int i = 0; i < state.taps; i++)
      sum +=
       readHandle.invokeExact(readIndex + i)
       * state.coeff[i];
    writeHandle.invokeExact(writeIndex, sum);
    writeIndex++;
    for (int i = 0; i < state.decimation; i++)
    {
      readHandle.invokeExact(readIndex);
      readIndex++;
    }
    readHandle.invokeExact(readIndex);
    readIndex++;
  }
}
```

Figure 11: LowPassFilterArchetype: the result of bytecode rewriting on LowPassFilter from Figure 4, before performing StreamJIT's only bytecode-level optimization (see the text). StreamJIT emits bytecode directly; for purposes of exposition, this figure shows the result of decompiling the generated bytecode back to Java.

8.5 Storage allocation

The compiler allocates separate storage for the initialization and steady-state schedules. To compute initialization buffer sizes, the compiler multiplies the initialization inter-group schedule by the intra-group schedules to find each worker's total number of executions, then multiplies by the push rate to find the total items written to each storage. Steady-state buffer sizes are computed similarly using the steady-state inter-group schedule, but additional space is left for the buffering established by the initialization schedule. The storage implementation is a composition of backing storage with an addressing strategy. It provides read and write handles used by archetypal work methods, plus an adjust handle that performs end-of-steady-state actions.

The actual backing storage may be a plain Java array, or for primitive types, a direct NIO `Buffer` or native memory allocated with `sun.misc.Unsafe`. Each implementation provides read and write method handles taking an index. Steady-state backing storage implementations are chosen by the autotuner on a per-edge basis; initialization always uses Java arrays.

Addressing strategies translate worker indices into physical indices in backing storage. Direct addressing simply passes indices through to the backing storage and is used during initialization and for storage fully internal to a group (read and written only by the group). Storage on other edges needs to maintain state across steady-state executions (to maintain software pipelining). Circular addressing treats the backing storage as a circular buffer by maintaining a head index; at the end of each steady-state execution, elements are popped by advancing the index. Double-buffering alternates reading and writing between two separate backing storage implementations at the end of each steady-state execution, but can only be used when the storage is fully external to all groups that use it (read or written but not both), as otherwise items written would need to be read before the buffers are flipped. Steady-state addressing strategies are chosen by the autotuner on a per-edge basis.

Data parallelization assumes random access to storage, including the overall input and output of the stream graph. If the overall input edge is part of the compilation and the source provides random access (e.g., a `List` or a memory-mapped file), the storage normally allocated for that edge may be replaced by direct access to the source; otherwise data is copied from the source into the storage. This copy-avoiding optimization is important because the copy is performed serially at the end-of-steady-state barrier. A similar optimization could be performed for random-access output sinks, but the current implementation does not.

8.6 Work allocation

The compiler then divides the steady-state inter-group schedule among the cores, as directed by the autotuner. The buffering established by the initialization schedule ensures there are no data dependencies between the groups, so the compiler is free to choose any allocation, except for groups containing a stateful filter, which must be allocated to a single core to respect data-dependence on the worker state. The initialization schedule itself does not have this guarantee, so all work is allocated to one core in topological order.

8.7 Code generation

To generate code to run the initialization and steady-state schedules, the compiler builds a method handle chain for each core.

- For each worker, a new instance of its corresponding state holder class is initalized with the values of the worker's fields. The archetypal work method for the worker is reflected as a method handle and the state holder instance, plus the read and write handles for the storage used by the worker, are bound (the handle is partially applied), yielding a handle taking a read and write index.
- Each worker handle is bound into a worker loop combinator that executes a range of iterations of the worker, computing the read and write indices to pass to the worker handle using the worker's pop and push rates.
- The worker loop handles for all workers in a group are then bound into a group loop combinator that executes a range of interations of the intra-group schedule, multiplying the group iteration index by the intra-group schedule's worker execution counts to compute the worker iteration indices to pass to the worker loops.
- Finally, the group's iteration range allocated for that core is bound into the group loop and the group loops are bound by a sequence combinator to form a core handle.

8.8 Preparing for run time

At the end of compilation, the compiler creates a *host* instance, which is responsible for managing the compiled stream graph through the lifecycle phases described in Section 8.1: initialization, steady-state and draining. From the compiler, the host receives the initialization and steady-state core code handles, liveness information, and counts of the items read and written from and to each graph input and output. The host creates the threads that execute the method handles and the barrier at which the threads will synchronize at the end of each steady-state iteration. The barrier also has an associated *barrier action*, which is executed by one thread after all threads have arrived at the barrier but before they are released.

The code actually run by each thread is in the form of newly-created Runnable proxies with `static final` fields referencing the core code method handles. Because the handles are compile-time constant, the JVM JIT can inline through them and their constituent handles, including object references bound into the handle chain. For example, a circular buffer storage computes a modulus to contain the index in the bounds of the backing storage, but the JVM JIT sees the modulus field as a constant and can avoid generating a division instruction. The JVM's visibility into the generated code is essential to good performance.

8.9 Run time

At run time, threads run core code and synchronize at the barrier. The behavior of each thread and of the barrier action is managed using `SwitchPoint` objects, which expose the lifecycle changes to the JVM. The JVM will speculatively compile assuming the switch point points to its first target, then recompile when the switch point is invalidated, avoiding branching on each call. The first switch point is invalidated when transitioning to the steady-state after initialization and the second is invalidated after draining is complete. Figures 12 and 13 show thread behavior and barrier action behavior, respectively.

Initialization Initially, the core code is a no-op, so all threads immediately reach the barrier. The barrier action calls the initialization core handle after reading data from graph inputs. Any output produced is written to graph out-

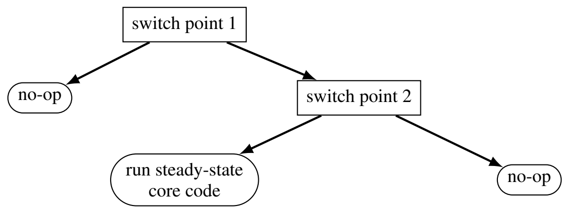


Figure 12: Thread behavior during the host lifecycle. Initialization and draining are single-threaded, so they occur at the barrier action. *no-op* (no operation) indicates a method handle that does nothing.

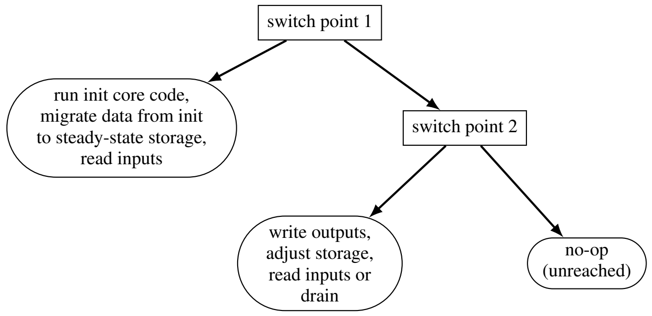

switch point 1
run init core code, migrate data from init to steady-state storage, read inputs
switch point 2
write outputs, adjust storage, read inputs or drain
no-op (unreached)

Figure 13: Barrier action behavior during the host lifecycle. In the steady-state, if insufficient inputs can be read for the next steady-state iteration, draining begins immediately. After draining is complete, the threads stop before returning to the barrier, so the third method handle (rightmost node in the figure) is never called.

puts, data buffered in initialization storage is migrated to steady-state storage using the liveness information, and input is read for the steady-state. The first switch point is then invalidated to transition to the steady-state phase and the cores are released from the barrier.

Steady-state In the steady-state phase, the core code is the core method handle built during compilation. When all cores come to the barrier after running their core handle, output is written to the graph outputs, storage is adjusted (e.g., circular buffer head indices are incremented), and input is read from graph inputs to be used in the next cycle. This phase continues until not enough input remains to execute the steady-state code, at which point draining begins (still in the barrier action).

Draining During draining, data buffered in steady-state storage is migrated into queues for the interpreter based on the liveness information. The interpreter runs until the input is exhausted or progress stops due to unsatisfiable input (peek and pop) rates. When online autotuning, any remaining buffered data is passed to the StreamJIT runtime for use in a future compilation. The StreamJIT runtime requests the threads to stop, the second switch point is invalidated, and the cores are released from the barrier. Invalidating the second switch point makes the core code a no-op, to make race conditions between the cores and the StreamJIT runtime harmless.

9. Autotuning

In place of heuristics, the StreamJIT compiler uses the OpenTuner [6] extensible autotuner to make its optimization decisions. The compiler defines the tuner's search space with a number of parameters based on the stream graph being compiled. The tuner requests throughput measurements for a particular set of values (called a *configuration*), allocating trials between different search techniques according to their payoff. Tuning can be performed offline, in which the program state is reset for each trial, or online, in which the stream graph is recompiled preserving the state of the previous compilation.

Search space For each non-peeking worker in the graph, a boolean parameter determines whether that worker is fused upward. For each built-in splitter or joiner in the graph, a boolean parameter determines whether that splitter or joiner is removed. For each worker, two boolean parameters control unboxing that worker's input and output types. For each edge, one boolean parameter controls whether that edge's storage is unboxed, one enumerated parameter selects the backing storage (Java array, NIO `Buffer` or native memory via `sun.misc.Unsafe`), and one enumerated parameter selects between double-buffering and circular addressing strategies if that storage is not internal to a group.

Core allocation is controlled by four parameters per worker: a count n and permutation parameter that define a subset of the available cores to allocate to (the first n cores in the permutation), plus a bias count b and a bias fraction f between 0 and 1. The parameters corresponding to the first worker in each group are selected. Stateful groups cannot be data-parallelized, so if the group contains a stateful worker, all group executions in the inter-group schedule are assigned to the first core in the permutation. Otherwise, the group executions are divided equally among the n cores in the subset. Then the first b cores (or $n-1$ if $b \geq n$) have f times their allocation removed and redistributed equally among the other $n-b$ cores. Biasing allows the autotuner to load-balance around stateful groups while preserving equal allocation for optimal data-parallelization.

Interactions between parameters may leave some parameters unused or ignored in some configurations; the search space has varying dimensionality. For example, if after (not) fusing preceding groups, a group has more than one predecessor, it cannot be fused upward, so the corresponding fusion parameter is ignored. Throughput is very sensitive to the core allocation, and both fusion and removal parameters affect which sets of core allocation parameters are used. Unused sets accumulate random mutations, which can prevent the tuner from recognizing profitable changes in fusion or removal because the resulting core allocation is poor. To address this, all unused core allocation parameters in each group are overwritten with the used set's values, which empirically gives better tuning performance. More elegant ways to search spaces of varying dimensionality are a topic for future work in optimization.

Custom techniques In addition to its standard optimization algorithms, OpenTuner can be extended with custom techniques, which StreamJIT uses in two ways. One technique suggests three fixed configurations with full fusion, removal and unboxing, equal allocation to all cores (maximum data parallelism), and multipliers of 128, 1024 and 4096. These configurations help the tuner find a "pretty good" portion of the search space more quickly than by testing random configurations.

For most stream graphs, fusion, splitter and joiner removal, and unboxing are profitable at all but a few places in the graph. Fusion, for example, is usually good except for stateful workers whose state infects their entire group, preventing data parallelism. To convey this knowledge to the tuner, three custom techniques modify the current best known configuration by applying full fusion, removal or unboxing. If the result has already been tested, the techniques proceed to the second-best configuration and so on. These techniques keep the tuner in a portion of the search space more likely to contain the best configuration, from which the tuner can deviate in a small number of parameters where the optimizations are not profitable. However, these techniques are merely suggestions. Because the tuner allocates trials to techniques based on their expected payoff, if for example a graph contains many stateful workers, the tuner will learn to ignore the full-fusion technique.

Online autotuning Online tuning works similarly to offline tuning, except that graph edges may contain buffered items left behind by the previous execution. The initialization schedule cannot remove buffering from within fused groups because it shares intra-group code with the steady-state schedule, so the downstream groups on those edges cannot be fused upward. Removal of splitters and joiners on these edges is also disabled due to a bug in the implementation. While this limits online tuning performance compared to offline tuning (which never has previous buffered items), the graph is drained as completely as possible before recompiling, so usually only a few edges are affected.

10. Automatic Distribution

To scale to multiple nodes, the StreamJIT runtime partitions the stream graph into connected subgraphs, which are compiled or interpreted separately. Because StreamJIT uses an autotuner to find the partitioning instead of heuristics, implementing distribution only requires defining some autotuning parameters and writing the communication code.

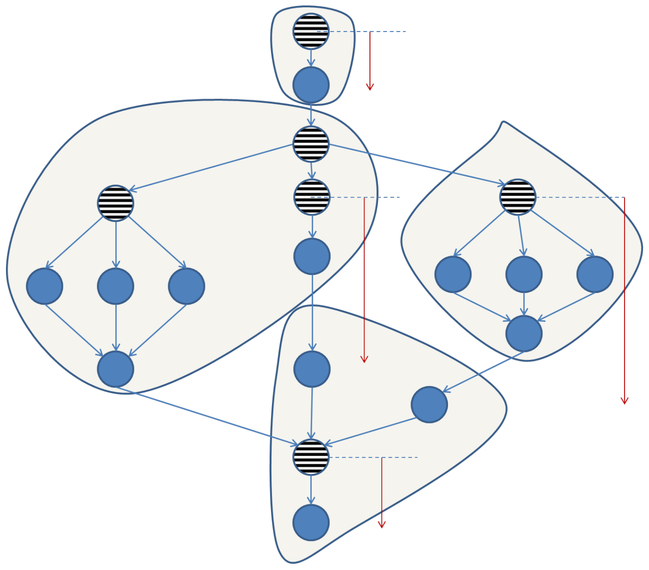


Figure 14: Partitioning a stream graph for distribution. Hatched workers are keys; arrows represent the cut-distance parameters. Both inner splitjoins are treated as single workers; the left does not have a cut-distance parameter as it's immediately followed by another key.

Each partition is a connected subgraph and the graph of partitions is acyclic. (See Figure 14.) Compiled partitions execute dynamically once enough input to execute their schedule is available. Because each worker is placed in only one partition, it is not possible to data-parallelize a worker across multiple nodes; adding nodes exploits task and pipeline parallelism only. As a workaround, the user can introduce a roundrobin splitjoin (simulating data parallelism as task parallelism), allowing the autotuner to place each branch in a different partition, allowing task parallelism across nodes.

Autotuning partitioning The partitioning is decided by the autotuner. Usually one partition per node provides the best throughput by minimizing communication, but sometimes the better load-balancing allowed by using more partitions then nodes overcomes the extra communication cost of creating extra data flows.

A naive parameterization of the search space would use one parameter per worker specifying which node to place it on, then assemble the workers on each node into the largest partitions possible (keeping the workers in a partition connected). Empirically this results in poor autotuning performance, as most configurations result in many small partitions spread arbitrarily among the nodes. Small partitions inhibit fusion, resulting in inefficient data-parallelization, and poor assignment causes a large amount of inter-node communication.

Instead, some workers in the graph are selected as *keys* (highlighted in Figure 14) which are assigned to nodes. Every kth worker in a pipeline is a key, starting with the first.

Splitjoins containing s or fewer workers total are treated as though they are single workers; otherwise, the splitter, joiner, and first worker of each branch are keys and key selection recurses on each branch. For each key, one parameter specifies which node to place it on and another parameter selects how many workers after this key to cut the graph (0 up to the distance to the next key). If there are too many partitions, most configurations have unnecessary communication back and forth between nodes, but if there are too few partitions, load balancing becomes difficult. We chose $k = 4$ and $s = 8$ empirically as a balance that performs well on our benchmarks. These constants could be meta-autotuned for other programs.

Given a configuration, the graph is then cut at the selected cut points, resulting in partitions containing one key each, which are then assigned to nodes based on the key's assignment parameter. Adjacent partitions assigned to the same node are then combined unless doing so would create a cycle among the partition graph (usually when a partition contains a splitter and joiner but not all the intervening branches). The resulting partitions are then compiled and run.

Sockets Partitions send and receive data items over TCP sockets using Java's blocking stream I/O. Each partition exposes its initialization and steady-state rates on each edge, allowing the required buffer sizes to be computed using the ILP solver, which avoids having to synchronize to resize buffers. However, execution of the partitions is dynamically (not statically) scheduled to avoid a global synchronization across all nodes. The runtime on the node initiating the compilation additionally communicates with each partition to coordinate draining, which proceeds by draining each partition in topological order. During draining, buffers dynamically expand to prevent deadlocks in which one partition is blocked waiting for output space while a downstream partition is blocked waiting for input on a different edge (which reading would free up output space). When using online autotuning, the initiating node then initiates the next compilation.

11. Evaluation

11.1 Implementation effort

Excluding comments and blank lines, StreamJIT's source code consists of 33,912 lines of Java code (26,132 excluding benchmarks and tests) and 1,087 lines of Python code (for integration with OpenTuner, which is written in Python); see Figure 15 for a breakdown. In comparison, the StreamIt source code (excluding benchmarks and tests) consists of 266,029 lines of Java, most of which is based on the Kopi Java compiler with custom IR and passes, plus a small amount of C in StreamIt runtime libraries. StreamIt's Eclipse IDE plugin alone is 30,812 lines, larger than the non-test code in StreamJIT.

User API (plus private interpreter plumbing)	1,213
Interpreter	1,032
Compiler	5,437
Distributed runtime	5,713
Tuner integration	713
Compiler/interp/distributed common	4,222
Bytecode-to-SSA library	5,166
Utilities (JSON, ILP solver bindings etc.)	2,536
Total (non-test)	26,132
Benchmarks and tests	7,880
Total	33,912

Figure 15: StreamJIT Java code breakdown, in non-blank, non-comment lines of code. An additional 1,087 lines of Python are for tuner integration.

11.2 Comparison vs. StreamIt

Single-node offline-tuned StreamJIT throughput was compared with StreamIt on ten benchmarks from the StreamIt benchmark suite [38]. When porting, the algorithms used in the benchmarks were not modified, but some implementation details were modified to take advantage of StreamJIT features. For example, where the StreamIt implementations of DES and Serpent use a roundrobin joiner immediately followed by a filter computing the exclusive or of the joined items, the StreamJIT implementations use a programmable joiner computing the exclusive or. The ported benchmarks produce the same output as the originals modulo minor differences in floating-point values attributable to Java's differing floating-point semantics.

Benchmarking was performed on a cluster of 2.4GHz Intel Xeon E5-2695v2 machines with two sockets and 12 cores per socket and 128GB RAM. HyperThreading was enabled but not used (only one thread was bound to each core).

StreamIt programs were compiled with `strc --smp 24 -O3 -N 1000 benchmark.str`. The emitted C code was then compiled by `gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3` with `gcc -std=gnu99 -O3 -march=corei7-avx -mtune=corei7-avx`. `strc` was built with the SMP backend's `FAKE_IO` option enabled, which replaces file output with a write to a `volatile` variable; this ensures I/O does not affect performance while preventing GCC from optimizing code away. `strc`'s `-N` option adds instrumentation to count the CPU cycles per output generated by the stream graph when executing the steady-state schedule, which is converted to nanoseconds per output by multiplying by the CPU cycle time. StreamIt is no longer maintained and `strc` fails to compile MPEG2 and Vocoder, so while we present StreamJIT performance, we cannot compare on those benchmarks.

StreamJIT programs ran on 64-bit OpenJDK build 1.8.0-ea-b124. StreamJIT programs were autotuned for 12 hours in three independent sessions. In each trial, the program is

benchmark	StreamJIT	StreamIt	relative perf
Beamformer	2,320,186	1,204,215	1.9
BitonicSort	9,771,987	6,451,613	1.5
ChannelVocoder	551,065	796,548	0.7
DCT	23,622,047	6,434,316	3.7
DES	17,441,860	6,469,003	2.7
FFT	25,210,084	2,459,016	10.3
Filterbank	924,499	1,785,714	0.5
FMRadio	2,272,727	2,085,143	1.1
MPEG2	32,258,065	-	-
Serpent	2,548,853	6,332,454	0.4
TDE-PP	12,605,042	2,357,564	5.3
Vocoder	406,394	-	-

Figure 16: Single-node 24-core throughput comparison, in outputs per second. Relative performance is StreamJIT throughput divided by StreamIt throughput. StreamIt fails to compile MPEG2 and Vocoder.

compiled with the autotuner's requested configuration, the initialization schedule is run, and the steady-state schedule is run for at least 10 seconds (rounded up to a whole schedule) to ensure the JVM JIT compiles the steady-state code. Then for at least 5 seconds rounded up to a whole schedule, the number of output items is counted and this count is divided by the actual elapsed time (measured with `System.nanoTime()`) to compute the throughput. The throughput of the best-performing configurations from each tuning run is averaged for comparison purposes.

Across all ten benchmarks, StreamJIT's average throughput is 2.8 times higher than StreamIt, despite being implemented with considerably less effort. The results are shown in Figure 16. StreamJIT's autotuner chooses better parallelizations than StreamIt's heuristics, but GCC vectorizes much more than HotSpot (the OpenJDK JIT compiler). On our benchmarks, parallelization generally outweighs vectorization, but on ChannelVocoder and Filterbank StreamJIT cannot overcome the vectorization disadvantage.

11.3 Online Autotuning and Distribution

The StreamIt benchmarks have a high communication-to-computation ratio, for which our task-parallel distribution strategy performs poorly. Our naive communication code compounds this flaw by making multiple unnecessary copies. In the current implementation, online autotuning is implemented using the distributed runtime with one node, incurring the cost of loopback sockets and the extra copies. In this section we use modified versions of the benchmarks, so the numbers presented are not directly comparable to single-node offline performance.

Online single-node throughput Single-node online tuning was evaluated on the same machines used above for single-node offline tuning (with 24 cores). Figure 17 shows online tuning performance on the modified ChannelVocoder benchmark and Figure 18 shows the same metric for the modified FMRadio benchmark. The graph shows the time taken to output 30000 data items after reaching the steady-state in each configuration (inverse throughput, thus lower numbers are better), as well as the time of the best configuration found so far. Both graphs show the variance increasing when the autotuner has not discovered a new best configuration, then decreasing again when it does. The autotuner is making a tradeoff between exploring the search space and exploiting the parts of the space it already knows give high performance. If this oscillation is unacceptable for a particular program, the current best configuration could be run in parallel with the search to provide a fallback, as in SiblingRivalry [5].

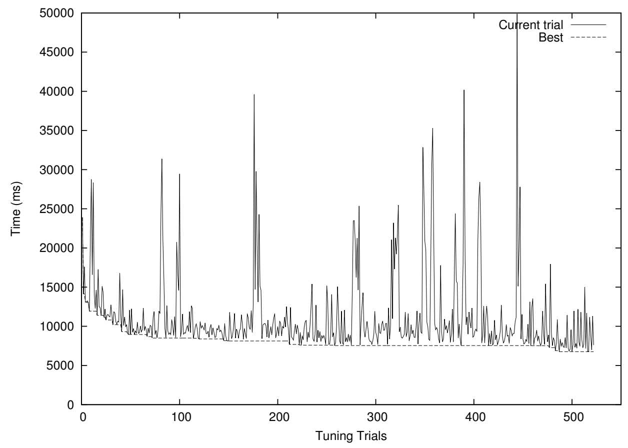


Figure 17: Online autotuning progress on the (modified) ChannelVocoder benchmark, showing the time in milliseconds to produce 30000 outputs (inverse throughput, lower is better).

Offline distributed throughput Distributed offline tuning was evaluated on the same machines used for single-node offline tuning, but using only 16 cores per node for the StreamJIT compiler to leave cores free for the distributed runtime's communication thread. Figure 19 shows speedup on the modified FMRadio and ChannelVocoder benchmarks across 4, 8 and 16 nodes relative to 2-node performance after a 24-hour tuning run. ChannelVocoder, the more compute-intensive of the two benchmarks, scales well up to 16 nodes. FMRadio has less computation and only scales well up to 4 nodes.

Figure 20 compares tuning progress for the modified FMRadio benchmark using 2, 4, 8, and 16 nodes over the 24-hour tuning run (Figure 19 is based on the best configuration found during these runs). Each trial tests the time to produce 30000 stream graph outputs. The number of trials performed in each tuning run varies based on autotuner randomness.

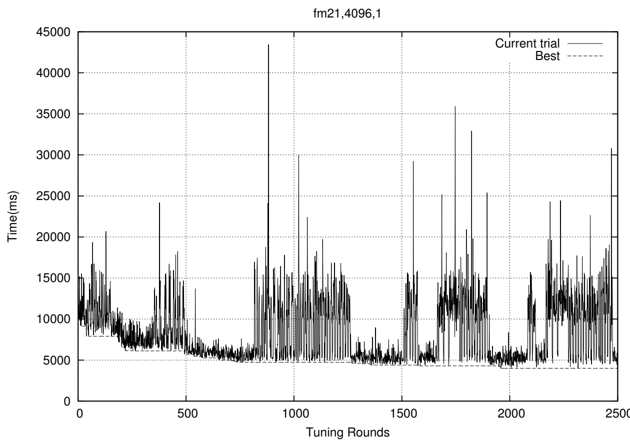


Figure 18: Online autotuning progress on the (modified) FMRadio benchmark, showing the time in milliseconds to produce 30000 outputs (inverse throughput, lower is better).

benchmark	2 nodes	4	8	16
ChannelVocoder	1.00	1.38	3.58	10.13
FMRadio	1.00	3.18	3.75	4.60

Figure 19: Speedup after distributed offline autotuning on 4, 8 and 16 nodes, relative to 2 nodes.

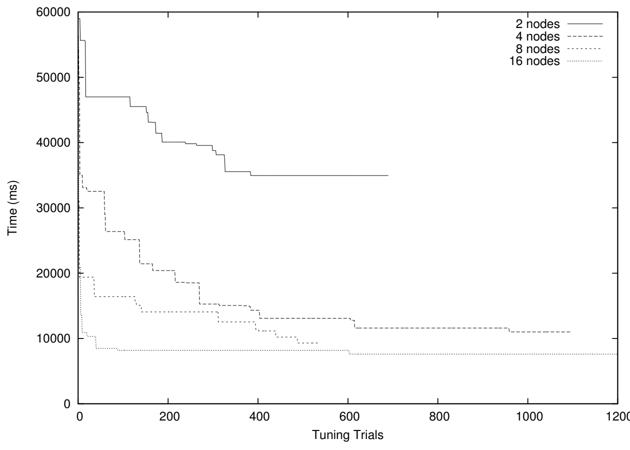


Figure 20: Distributed offline autotuning progress on the (modified) FMRadio benchmark, showing the time in milliseconds to produce 30000 outputs (inverse throughput, lower is better) of the best configuration up to that point. The benchmark was tuned for each node count for 24 hours.

12. Conclusion

Modern applications are built on the abstractions provided by domain libraries. While domain-specific languages can offer better performance through domain-specific optimizations, the cost of implementing an optimizing compiler has caused few domain-specific languages to be built. Commensal compilers substantially reduce compiler implementation effort by leveraging existing infrastructure. By reusing their host language's front-end, delegating middle-end decisions to an autotuner, and using existing APIs to enable optimized code generation by the virtual machine back-end, commensal compilers need only implement domain-specific optimizations. We demonstrated the power of our approach by implementing a commensal compiler for StreamJIT, a Java-embedded stream programming language, that gives performance 2.8 times better than StreamIt's native code compiler with a fraction of the engineering effort. We additionally implemented automatic distribution over multiple nodes at the low cost of writing the socket communication code.

Acknowledgements

Mike Gordon helped us get started with StreamIt and explained the lessons learned during its development. Jason Ansel provided valuable insight into designing effective search spaces for OpenTuner.

This material is based upon work supported by the U.S. Department of Energy, Office of Science under the X-Stack award DE-SC0008923, a grant from the Intel Science and Technology Center for Big Data, and a SMART[3] graduate fellowship.

References

[1] Expression Trees, . URL `http://msdn.microsoft.com/en-us/library/bb397951%28v=vs.110%29.aspx`.

[2] Emitting Dynamic Methods and Assemblies, . URL `http://msdn.microsoft.com/en-us/library/8ffc3x75%28v=vs.110%29.aspx`.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542481.

[5] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. SiblingRivalry: Online Autotuning Through Local Competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures*

[3] http://smart.mit.edu/

and Synthesis for Embedded Systems, CASES '12, pages 91–100, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1424-4. doi: 10.1145/2380403.2380425.

[6] J. Ansel, S. Kamil, K. Veeramachaneni, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. 2013. URL `http://hdl.handle.net/1721.1/81958`.

[7] ASM. ASM bytecode transformation library. `http://asm.ow2.org/`.

[8] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869469.

[9] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. A Compiler and Runtime for Heterogeneous Computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 271–276, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228411.

[10] E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegrd, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178, July 2010. doi: 10.1109/MEMCOD.2010.5558637.

[11] S. Bharadwaj. invokedynamic and Jython. In *JVM Language Summit 2011*, JVMLS 2011, 2011.

[12] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100, Oct 2011. doi: 10.1109/PACT.2011.15.

[13] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[14] M. Frigo and S. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, May 1998. doi: 10.1109/ICASSP.1998.681704.

[15] M. Garcia. Runtime metaprogramming via java.lang.invoke.MethodHandle. May 2012. URL `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2012Q2/RuntimeMP.pdf`.

[16] M. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology, 2010.

[17] D. Heidinga. MethodHandle Implementation Tips and Tricks. In *JVM Language Summit 2011*, JVMLS 2011, 2011.

[18] ImageMagick. ImageMagick. URL `http://www.imagemagick.org/`.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273005.

[20] M. Karczmarek. *Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language*. PhD thesis, Massachusetts Institute of Technology, 2002.

[21] A. Kulkarni and R. R. Newton. Embrace, Defend, Extend: A Methodology for Embedding Preexisting DSLs. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-specific Languages*, FPCDSL '13, pages 27–34, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2380-2. doi: 10.1145/2505351.2505355.

[22] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142552.

[23] C. Nutter. Adding invokedynamic Support to JRuby. In *JVM Language Summit 2011*, JVMLS 2011, 2011.

[24] F. Otto, V. Pankratius, and W. Tichy. High-level Multicore Programming with XJava. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 319–322, May 2009. doi: 10.1109/ICSE-COMPANION.2009.5071011.

[25] F. Otto, V. Pankratius, and W. Tichy. XJava: Exploiting Parallelism with Object-Oriented Stream Programming. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 875–886. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03868-6. doi: 10.1007/978-3-642-03869-3_81.

[26] F. Otto, C. Schaefer, M. Dempe, and W. Tichy. A Language-Based Tuning Mechanism for Task and Pipeline Parallelism. In P. DAmbra, M. Guarracino, and D. Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 328–340. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15290-0. doi: 10.1007/978-3-642-15291-7_30.

[27] J. Ponge and F. L. Mouël. JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications. *CoRR*, abs/1210.1039, 2012.

[28] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, June 2013.

[29] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, MICRO 14, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.

[30] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1. doi: 10.1145/1868294.1868314.

[31] J. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, page 2. ACM, 2009.

[32] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput Stream Programming in Java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 211–228, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297043.

[33] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts &#38; Experiences*, GPCE '13, pages 145–154, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2373-4. doi: 10.1145/2517208.2517220.

[34] C. Thalinger and J. Rose. Optimizing Invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 1–9, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852763.

[35] The Java Tutorials. Aggregate Operations, . URL `http://docs.oracle.com/javase/tutorial/collections/streams/index.html`.

[36] The Java Tutorials. Type Annotations and Pluggable Type Systems, . URL `http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html`.

[37] W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[38] W. Thies and S. Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854319.

[39] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. A Scalable Auto-tuning Framework for Compiler Optimization. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009. doi: 10.1109/IPDPS.2009.5161054.

[40] TypeToken. TypeToken. URL `https://code.google.com/p/guava-libraries/wiki/ReflectionExplained#TypeToken`.

[41] Z. Wang and M. F. P. O'boyle. Using Machine Learning to Partition Streaming Programs. *ACM Trans. Archit. Code Optim.*, 10(3):20:1–20:25, Sept. 2008. ISSN 1544-3566. doi: 10.1145/2512436.

[42] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.

[43] C. Wimmer and T. Würthinger. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1563-0. doi: 10.1145/2384716.2384723.

[44] T. Würthinger. Extending the Graal Compiler to Optimize Libraries. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, SPLASH '11, pages 41–42, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048168.

[45] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 298–308, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378860.

[46] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.