StreamJIT: A Commensal Compiler for High-Performance Stream Programming

by

Jeffrey Bosboom

Bachelor of Science, Information and Computer Science
University of California, Irvine, 2011

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
May 21, 2014

Signature redacted

Certified by ....
Saman Amarasinghe
Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

StreamJIT: A Commensal Compiler for High-Performance Stream Programming

by
Jeffrey Bosboom

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

There are domain-specific libraries for many domains, enabling rapid and cost-effective development of complex applications. On the other hand, domain-specific languages are rare despite the performance advantages of compilation. We believe the reason is the multiple orders-of-magnitude higher cost of building a compiler compared to building a library. 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 an autotuner instead of optimization heuristics, and use host platform APIs that enable back-end optimizations by the host platform JIT. 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. The StreamJIT commensal compiler takes advantage of the structure of stream programs to find the right amount of parallelism for a given machine and program. Our compiler achieves 2.4 times better performance than StreamIt's native code (via GCC) compiler with considerably less implementation effort.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

Sumanaruban Rajadurai, a Ph.D. student at the National University of Singapore, joined the StreamJIT project in March 2013 through the Singapore-MIT Alliance for Research and Technology (SMART) program[1]. He implemented StreamJIT's distributed backend, described in section 5.7, and our discussions at the whiteboard were very productive, especially when working out the index transformations described in section 5.5.3. He performed the experiments on online and distributed autotuning in section 5.8.3, though the presentation is my own. He drew figure 5-14.

My advisor, Saman Amarasinghe, kept me focused on what was important to the project. He also provided the StreamJIT name, concisely expressing the project's original scope – StreamIt, in Java, with a JIT.

Jason Ansel, Kalyan Veeramachaneni, Shoaib Kamil, and Una-May O'Reilly from the OpenTuner project provided valuable insight into search space design (see section 6.3).

Fredrik Kjolstad and Vladimir Kiriansky, fellow Ph.D. students in Saman's research group, provided valuable comments on the conference paper draft that turned into this thesis.

Martin Rinard and Daniel Jackson gave me helpful outside advice.

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

Contents

Chapter 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 [ABB$^+$99] in the linear algebra domain and ImageMagick [Ima] in the image processing domain. However, applications using these performance critical domain libraries lack the compiler optimization support needed for higher performance, such as judicious parallelization or fusing multiple functions for locality. On the other hand, domain-specific languages and compilers are able to provide orders of magnitude more performance by incorporating such optimizations, giving the user both simple abstractions and high performance. In the image processing domain, for example, Halide [RKBA$^+$13] programs are both faster than hand-optimized code and smaller and simpler than 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 [BSL$^+$11], but it will require a huge investment to make these systems as robust, portable, usable and familiar as C++, Java, Python, etc. Systems like Stream$^{\text{Hs}}$ [KN13] 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 thesis 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 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

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

for the stream programming language StreamJIT atop the Java platform. Our compiler achieves performance on average 2.4 times better than StreamIt's native code (via GCC) compiler with considerably less implementation effort.

StreamJIT programs are dataflow graphs of workers connected by channels. Workers perform atomic, data-independent operations on the data items flowing through the stream graph. Workers specify the number of data items they produce and consume each time they are invoked (their data rates). The StreamJIT compiler uses the data rates to compute a schedule of worker executions that leaves buffer sizes unchanged. All communication between workers occurs via the channels and most workers are stateless, so the StreamJIT compiler is free to exploit data, task and pipeline parallelism when executing the schedule.

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 [AKV+13] 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 2.4 times better performance than StreamIt's native code compiler with an order of magnitude less code.

Chapter 2 discusses related work. Chapter 3 explains commensal compilers abstractly, separate from any particular language; chapter 4 provides implementation details specific to the Java platform. Chapter 5 describes and evaluates the StreamJIT system, and chapter 6 presents alternative approaches and lessons learned during StreamJIT development.

Chapter 2

Related Work

In this chapter we review related work on compilers for embedded languages, stream programming (and particularly the StreamIt language), use of autotuners in compilers, and relevant aspects of the Java platform.

2.1 Embedded Language Compilers

The Delite DSL compiler framework [BSL$^+$11] uses Lightweight Modular Staging [RO10] to build a Scala-level intermediate representation (IR), which can be raised to Delite IR to express common parallel patterns like foreach and reduce. DSLs can provide additional domain-specific IR nodes to implement, e.g., linear algebra transformations. The Delite runtime then compiles parts of the IR to Scala, C++ or CUDA and heuristically selects from these compiled implementations at runtime. Delite represents the next step of investment beyond commensal compilers; where commensal compilers reuse existing platform infrastructure, Delite is a platform unto itself. The Forge meta-DSL [SGB$^+$13] adds abstractions to shield DSL authors and users from having to understand Delite.

Truffle [WW12], built on top of the Graal extensible JIT compiler [W1̈1], 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.

2.2 Stream Programming

Stream programs apply a fixed, regular series of operations to a large set of input data items, called the *stream.* Operations are grouped into atomic units called *actors* or *filters*, each of which independently consumes one or more data items from its input and produces one or more data items from its output. By connecting the input of one filter to the output of another, complex programs called *stream graphs* can be created out of small, reusable components. While a filter might occasionally send an out-of-band control message to another filter in response to a data item, the majority of filter communication is by passing items through the data stream in a regular pattern. Some filters require state, but most operations can be performed by stateless filters, or by filters whose computation refers to (*peeks* at) future inputs. Application domains from suitable for stream programming include signal processing, media compression, and XML processing.

Stream programming is primarily influenced by Kahn process networks [Kah74], Hoare's communicating sequential processes [Hoa78], and synchronous dataflow [LM87]. All three of these models can be interpreted as graphs whose nodes compute on data items passed through the channels, differing mostly in what synchronization behavior is implied by the channels. Nodes in Kahn process networks block when reading from an empty channel, but can always output. A CSP process blocks on both input and output and no buffering is provided (processes must rendezvous to transfer data), but processes can wait on multiple channels simultaneously. Synchronous dataflow nodes statically declare their input and output rates (items consumed/produced per firing), but can be scheduled to execute without blocking. Other models related to stream programming include petri nets, computation graphs, and actors; for a review of streaming in languages, see [Ste97].

2.2.1 StreamIt

StreamJIT is strongly inspired by StreamIt [Thi09], a domain-specific language for stream programming following the synchronous dataflow (SDF) paradigm. In SDF, filters in the stream graph declare how many input items they consume per execution (their *pop rate*) and how many output items they produce per execution (their *push rate*). Filters that peek at future inputs without consuming them also declare a *peek rate*. Data items are notionally copied when placed into a channel; thus StreamIt has no pointers. StreamIt stream graphs are based around filters, vertical containers called pipelines, and horizontal containers called splitjoins (with special filters called splitters and joiners that have multiple outputs or inputs respectively). These stream graphs are called *structured streams* by analogy with structured programming. Together, the combination of defined rates, lack of aliasing, and structured streams allows the StreamIt compiler [Gor10] great insight into the program structure, allowing for analyses that fuse filters together or fiss them apart to arrive at an optimal amount of parallelism for a particular machine. By buffering a large number of data items, the StreamIt compiler generates a steady-state schedule (consisting of an ordering of filter executions) that allows each filter to execute the number of times specified in the schedule on data not used by other filters, such that synchronization is not required to ensure mutual exclusion. 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.

Some applications require limited out-of-band communication between filters, often in a way synchronized with the data stream. For example, a video decoder processing a variable bit rate video might need to transmit compression parameters upstream from the filter that recognizes them to the filters doing the decompression. This communication must occur at a defined time relative to the data stream; if the new parameters arrive too early or too late, subsequent frames of the video will be garbled and the decoder may desynchronize from the compressed data stream. StreamIt supports this type of communication with *teleport messaging* [TKS$^+$05]. In teleport messaging, filters that wish to receive messages declare a message handler that will be invoked when the message arrives to modify the filter's state. Filters can then send messages through a *portal* to broadcast to filters registered with that portal. The semantics are as if outgoing data items produced by the sender are tagged with the message; receivers execute their message handler when the tagged data item arrives.

Messages can also be sent with a specified latency, which logically tags items that many executions later; negative latencies can be used for communicating with upstream filters. Unlike shared-memory communication, teleport messaging is fully visible to the compiler and can be integrated into the schedule.

Some stream applications inherently have dynamic data rates, such as decompressors. StreamIt supports dynamic rate declarations, which allow a filter to specify a minimum, maximum and average rate, or a fully unknown rate. The StreamIt compiler compiles static subgraphs of the stream graph as normal, then dynamically fires them at runtime [SGA+13].

2.2.2 Other Streaming Languages

Many other streaming languages exist, which we will broadly categorize here. Prototyping environments, typified by Ptolemy [BHLM91], are used to simulate and validate complex systems; some environments can also generate code from the prototype design, though its quality varies. Dataflow languages such as Lucid [AW77], Id [KP78], and VAL [AD78] only allow each variable to be assigned once and without side-effects, effectively reducing control flow to data dependencies, yielding a very fine-grained stream graph. Functional languages using lazy evaluation over lists [HMJ76] can be thought of as stream programming languages in the same way as dataflow languages. Synchronous languages such as Esterel [BG92] target real-time control and communication applications in which control flow is driven by real-time events; in these programs, filters transform input events into output actions of the controlled system. Finally, general-purpose languages with stream support include Occam [Inm88], based on CSP, and Erlang [Arm07], based on actors. Other general-purpose languages have been the basis of domain-specific derived or embedded languages for streaming, a few of which we will examine further.

KernelC and StreamC are C-derived languages designed for the Imagine media processor architecture [Mat02]. Stream programs for Imagine are written using StreamC to connect kernels written in KernelC. KernelC is a restricted subset of C, without global variables, pointers, function calls, or conditional control flow, but with added operations for stream programming, such as a loop-over-stream construct, conditional assignments, and conditional reads and writes. StreamC is a C++ extension, with built-in classes representing streams and built-in functions for creating, draining and copying between streams. Kernels are represented as functions taking input and output streams as arguments. In the Imagine system, the KernelC compiler is responsible for allocating functional units and instruction-level scheduling, while the StreamC compiler is responsible for scheduling data movement through the on-chip memory to keep the kernels fed.

Brook is a C-derived stream programming language intended to maximize the ratio of computation to memory bandwidth. Originally designed for the Merrimac stream supercomputer [DLD+03], Brook was retargeted to provide abstractions for portable GPU programming without compromising performance [BFH+04]. Allowing most operations of C, including pointers, Brook is highly programmable, but this flexibility complicates compiler analysis, and performance gains beyond those driven by new GPUs are limited.

PeakStream [Pap07] and Accelerator [TPO06] are C++ and C# libraries, respectively, for just-in-time compilation of data-parallel array operations to GPU code. Like Brook, these libraries focus on GPU abstraction while retaining most of the semantics of traditional languages. Neither library provides a stream abstraction; the programmer manually marshals data into and out of data-parallel array objects. Accelerator performs limited optimizations on expression trees during the just-in-time compilation, obtaining performance competitive with hand-coded pixel shaders performing the same operations.

The Feldspar DSL for signal processing [ACD$^+$10] embeds a low-level C-like language in Haskell, generating parallelized C code for a C compiler; Stream$^{\mathrm{Hs}}$ [KN13] 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.

Lime [ABCR10] 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 [ABB$^+$12] selects which compiled code to use.

StreamFlex [SPGV07] 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.

Flextream [HCK$^+$09] implements a hybrid static-dynamic parallelizing compiler for StreamIt targeting the Cell architecture, based on the Multicore Streaming Layer runtime [ZLRA08]. Flextream statically compiles optimistically for a large resource budget, then uses heuristics to dynamically rebalance the schedule to the resources available. Flextream cannot scale up a schedule if more resources are available at runtime than assumed during static compilation, and it uses heuristics tuned for the Cell processor.

Other streaming languages have dropped static data rates in favor of dynamic fork-join parallelism, becoming more expressive at the cost of performance. XJava [OPT09a, OPT09b, OSDT10] is a Java extension also closely inspired by StreamIt. Its runtime exposes a small number of tuning parameters; [OSDT10] gives heuristics performing close to exhaustive search. Java 8 introduces `java.util.stream` [Thea], a fluent API for bulk operations on collections similar to .NET's LINQ [MBB06]. Without data rates, fusion and static scheduling are impossible, so these libraries cannot be efficiently compiled.

Dryad [IBY$^+$07] is a low-level distributed execution engine supporting arbitrary rateless stream graphs. DryadLINQ [YIF$^+$08] heuristically maps LINQ expressions to a stream graph for a Dryad cluster. Spark [ZCF$^+$10] 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.

2.3 Autotuning

ATLAS [WD98] uses autotuning to generate optimized linear algebra kernels. SPIRAL [XJJP01] autotunes over matrix expressions to generate FFTs; FFTW [FJ98] also tunes FFTs. Hall et al [TCC$^+$09] use autotuning to find the best order to apply loop optimizations. PetaBricks [ACW$^+$09] 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 [APW$^+$12] is an online autotuning system that maintains quality-of-service while autotuning by running a known safe configuration in parallel with an experimental configuration, taking the result of whichever completes first. While this requires dividing the available resources in half, it permits learning while preventing the application from getting stuck in pathological configurations. Despite the resource division, the authors report performance better than a statically computed known-safe configuration using the whole machine, especially when adapting to changing loads caused by other programs. StreamJIT's online autotuning could adopt this system.

Wang et al [WO08] 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.

2.4 The Java Platform

The Java platform consists of the Java Virtual Machine (JVM) [LYBB14] and the Java standard libraries. The JVM executes programs specified by platform-independent Java bytecode, whether compiled from the Java programming language or other source languages, or generated at runtime. Java bytecode uses symbolic representations that closely mirror the Java programming language; for example, object fields are accessed by their name and type, not by an offset into the storage allocated for the object. This allows the Java standard libraries to provide rich reflective information, which commensal compilers can use.

Java 7 introduced MethodHandles along with the `invokedynamic` instruction [Ros09] to support efficient implementation of dynamic languages. Direct method handles act as function pointers that can point to any method, constructor, or field accessible to the code creating the handle. Adaptor method handles transform the arguments or return values of an existing method handle; for example, an adapter method handle could bind an argument to implement partial function application. Major JVM JIT compilers implement optimizations for code using method handles ([TR10] describes the OpenJDK JIT compiler changes; [Hei11] describes J9's).

JRuby [Nut11] and Jython [Bha11], among other languages, use method handles for their original purpose. Garcia [Gar12] proposed using method handles encapsulating individual statements to implement primitive specializations for Scala. JooFlux [PM12] 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.

JVMs employ sophisticated adaptive optimizing compilers to improve performance compared to bytecode interpretation. Many JVMs begin by interpreting methods, gathering both high-level (method frequency) and low-level (type profiles of receiver objects at individual virtual call instructions) profiling information. Methods executed frequently ("hot" methods) are compiled to native code, using the low-level profiling information to aid code generation. For example, type profiles of receiver objects can be used to implement inline caches for faster dynamic dispatch. Because code generation is deferred to run time, the JVM can also perform speculative optimizations such as devirtualization, deoptimizing if the assumptions required to make the optimization safe are invalidated (in the case of devirtualization, when dynamic class loading loads another class in the inheritance hierarchy). For an overview of adaptive optimizations performed by virtual machines, see [AFG$^+$05]. The optimizations implemented by individual JVMs varies widely; see their documentation or developer resources (e.g., OpenJDK's HotSpot compiler [HS]).

Domain-specific languages may want to check semantic properties not enforced by a general-purpose compiler such as javac. Java 8 introduced type annotations [Theb], which can be used to check custom properties at compile time via an annotation processor. The Checker Framework [DDE$^+$11] uses type annotations to extend Java's type system, notably to reason about uses of null. A 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.

Chapter 3

Commensal Compiler Design

Commensal compilers are defined by their reuse of existing infrastructure to reduce the cost of implementing an optimizing compiler. In this chapter 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 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 thesis 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` [NETa] allows similar bytecode rewriting and the Expression Trees [NETb] feature can replace method handles.

Chapter 4

Commensal Compilers in Java

This chapter presents techniques for implementing commensal compilers targeting the Java platform. While this chapter uses examples from StreamJIT, the focus is on the compiler's platform-level operations; the StreamJIT language is described in section 5.1 and the StreamJIT compiler's domain-specific optimizations are explained in section 5.5. As StreamJIT is implemented in Java, this chapter assumes the host language is the Java programming language.

4.1 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 4-1; see figure 5-3 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 5.5.3).

Users then compose filters, splitters and joiners using the `Pipeline` and `Splitjoin` con-

```
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 4-1: `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 5-3 for an example implementation.

tainer classes to produce a stream graph. This stream graph is the input to the interpreter or commensal compiler.

4.2 Middle-end

A commensal compiler's middle-end performs domain-specific optimizations, either using 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 4.3).

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 5.5.4). Other IR attributes support StreamJIT's domain-specific optimizations; these include the schedule (section 5.5.2) and index functions used for built-in splitter and joiner removal (section 5.5.3). 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 5.6.

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.

4.3 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 [ASM] 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 other object, allowing for method handle combinators (see the loop combinator in figure 4-2). 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 5.5.5 for details and figure 5-10 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

```
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 4-2: 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 5.5.2).

bound to method handles that read and write storage objects (see section 5.5.6). The resulting handle (taking initial read and write indices) is bound into loop combinators similar to the one in figure 4-2 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.

Chapter 5

StreamJIT

In this chapter we present StreamJIT, a Java-embedded stream programming language with a commensal compiler. We first describe the StreamJIT API and user workflow. We then present the StreamJIT commensal compiler in detail, followed by how the compiler presents problems to the autotuner. Next we describe the distributed back-end. Finally, we evaluate the implementation effort and performance of StreamJIT by comparison with StreamIt.

5.1 StreamJIT API

5.1.1 Overview

Stream programs apply a fixed series of operations to a stream of data items. Consider figure 5-1, which presents a block diagram of an FM radio receiver with an equalizer. Starting with samples from an antenna, this receiver filters out high-frequency noise, then demodulates the signal to recover audio samples. These samples are duplicated into each band of the equalizer, which uses a band-pass filter (built out of two low-pass filters and a subtractor) to select just that band. Each band's amplification factors are applied and the results are summed to produce the final output to a speaker.

The corresponding StreamJIT program directly mirrors the block diagram's structure. Each operation is specified by a *worker* class whose `work` method reads data items from its input and writes them to its output. *Filters* are single-input, single-output workers, while *splitters* and *joiners* have multiple outputs and inputs respectively. StreamJIT programs are *stream graphs* built using pipelines and splitjoins, which compose workers vertically or horizontally, respectively (see figure 5-2). The FM radio has four nesting levels of composition: the top-level pipeline contains a splitjoin for each equalizer band, which in turn is a pipeline containing a splitjoin forming a band-pass filter. This stream graph is the input to the StreamJIT commensal compiler.

Data rates To enable automatic parallelization, workers are required to declare data rates. Workers declare *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. A worker's work method can be invoked any time enough input is available to satisfy its input requirements (peek and pop rates), and multiple executions of stateless workers can proceed in parallel. All communication between workers occurs by the flow of data items through the channels in

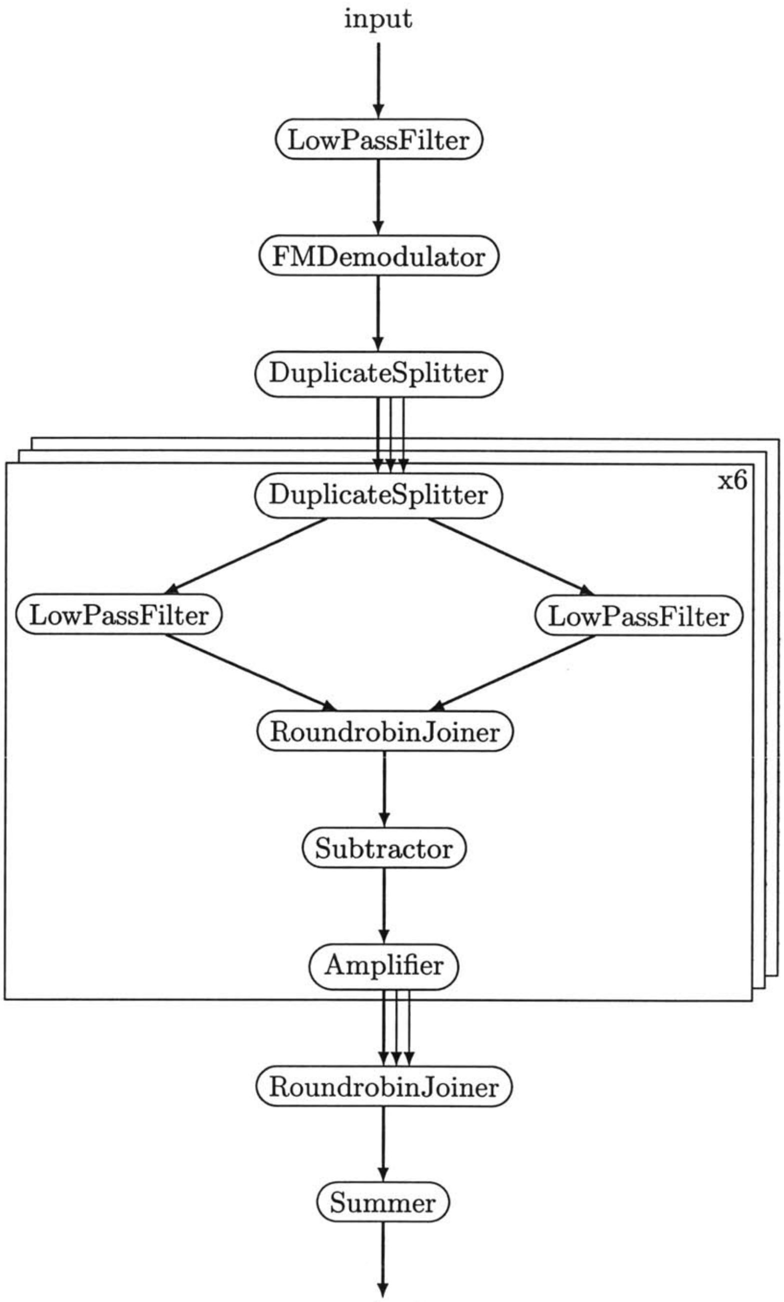
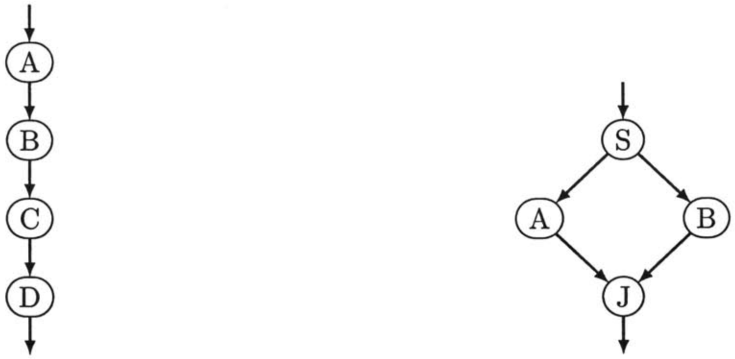


Figure 5-1: Block diagram of an FM radio receiver with an equalizer. The boxed components are replicated six times, one for each equalizer band. DuplicateSplitters replicate their input to each of their outputs; RoundrobinJoiner interleaves its inputs to its output. The StreamJIT program implementing this FM radio receiver mirrors this structure.



(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 5-2: StreamJIT composition structures.

the graph, exposed to the StreamJIT commensal compiler, so the compiler can use the data rates to compute a parallel schedule of filter executions tailored for a particular program and machine.

Structured streams To promote composition and reuse, StreamJIT enforces the use of pipelines and splitjoins, preventing the creation of arbitrary graph structures, an approach we term *structured streams* by analogy to structured programming. Filters, pipelines and splitjoins are collectively called *one-to-one elements* or simply *elements* because they all have one input and one output. Pipeline bodies and splitjoin branches can contain arbitrary one-to-one elements, including other pipelines and splitjoins, allowing for convenient recursive composition. Structured streams also promote reuse by enforcing a standard number of inputs and outputs, allowing existing components to be easily plugged in to a new program. Structured streams are occasionally limiting, but their benefits outweigh their costs. (See section 6.1 for further discussion.)

The next subsections describe writing worker classes and composing stream graphs, using the FM radio as an example.

5.1.2 Implementing Workers

Users write workers by subclassing `Filter`, `StatefulFilter`, `Splitter`, or `Joiner`. The worker's computation is specified by the `work` method, operating on data items read from input with the `peek` and `pop` methods and writing items to the output with `push`. Filters pass their data rates to the superclass constructor; splitters and joiners have more output or input rates respectively, and implement rate-getters instead.

Filters are workers with a single input and a single output. Despite their name, filters are not restricted to filtering their input; the name is derived from signal processing. Figure 5-3 shows the code for `LowPassFilter` from the FM radio example. This filter uses the peek-push-pop pattern: it computes its output by peeking at its input, pushes that output, then

```
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 5-3: LowPassFilter from the FMRadio benchmark. Peeks (nonconsuming read) at its input, pushes its output, and pops (consuming read) the input it is finished with. Peeks, pops and pushes can be freely intermixed, but the peek-push-pop style is common in our benchmarks.

pops some of the input it read. Peeks, pops and pushes can be freely intermixed (as long as the total number of each conforms to the declared data rates), but this style is common in our benchmark suite.

Splitters and joiners are written similarly, except their `push` or `pop` method (respectively) takes an additional parameter to select the output or input. Figure 5-4 shows the code for `RoundrobinJoiner` from the FM radio example. `RoundrobinJoiner` copies `itemsPerInput` from each input to its output in order, effectively interleaving the inputs. The `supportedInputs` method specifies how many inputs the joiner supports (in this case, an unlimited number), and the rate getters return the joiner's data rates. Because it is commonly used and presents special optimization opportunities (see section 5.5.3), `RoundrobinJoiner` is part of the StreamJIT API, but its code is the same as if it had been written by a user.

Restrictions on work methods Work methods may contain nearly arbitrary Java code (including library calls), with the following restrictions to permit automatic parallelization:

- Workers with state must extend `StatefulFilter` to avoid being data-parallelized. (StreamJIT currently does not support stateful splitters or joiners, though they could be trivially added by creating `StatefulSplitter` and `StatefulJoiner` classes.)

- Stateless workers must be reentrant to permit data parallelization. If a physically stateful but logically stateless (parallelizable) worker, such as a pseudorandom number

```
public final class RoundrobinJoiner<T> extends Joiner<T, T> {
  private final int itemsPerInput;
  public RoundrobinJoiner() {
    this(1);
  }
  public RoundrobinJoiner(int itemsPerInput) {
    this.itemsPerInput = itemsPerInput;
  }
  @Override
  public void work() {
    for (int i = 0; i < inputs(); ++i)
      for (int j = 0; j < itemsPerInput; ++j)
        push(pop(i));
  }
  @Override
  public int supportedInputs() {
    return Joiner.UNLIMITED;
  }
  @Override
  public List<Rate> getPeekRates() {
    return Collections.nCopies(inputs(), Rate.create(0));
  }
  @Override
  public List<Rate> getPopRates() {
    return Collections.nCopies(inputs(), Rate.create(itemsPerInput));
  }
  @Override
  public List<Rate> getPushRates() {
    return Arrays.asList(Rate.create(itemsPerInput * inputs()));
  }
}
```

Figure 5-4: RoundrobinJoiner, a built-in joiner provided by the StreamJIT API and used in many programs, including the FMRadio benchmark. In joiners, pop takes an argument to specify which input to pop from.

```
new Pipeline<>(
  new Splitjoin<>(
    new DuplicateSplitter<Float>(),
    new RoundrobinJoiner<Float>(),
    new LowPassFilter(rate, low, taps, 0),
    new LowPassFilter(rate, high, taps, 0)
  ),
  new Subtractor()
);
```

Figure 5-5: BandPassFilter from the FMRadio benchmark, instantiating `Pipeline` and `Splitjoin` directly.

generator, needs to synchronize to protect state, it must do so reentrantly to avoid deadlocking if the StreamJIT compiler schedules multiple executions of the worker in parallel.

- To avoid data races with other workers, workers may modify data items only after popping them, and all workers must not modify data items after pushing them to an output. Data items that will be modified by a downstream worker may not even be read after being pushed.

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

5.1.3 Composing Graphs Using Pipelines and Splitjoins

The StreamJIT API classes `Pipeline` and `Splitjoin` are used to compose stream graphs, either instantiated directly or subclassed. When used directly, workers or one-to-one elements are provided to the container's constructor. Direct instantiation is convenient for elements only used once in the stream graph, while subclassing allows reuse at multiple points in the stream graph. Figure 5-5 gives the code to create the band-pass filter used in the FM radio using `Pipeline` and `Splitjoin` directly; compare the `BandPassFilter` class given in figure 5-6.

A splitjoin's branches must have the same effective data rate (outputs pushed by the last worker in the branch per input popped by the first worker). If the branch rates are unbalanced, either one or more branches will halt for lack of input or input buffers on other branches will grow without bound. In either case, the graph cannot be compiled. For debugging, StreamJIT provides a `CheckVisitor` class that can identify the offending splitjoin.

5.1.4 Compiling Stream Graphs

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 the `CompiledStream` object

```
class BandPassFilter extends Pipeline<Float, Float> {
  BandPassFilter(float rate, float low, float high, int taps) {
    super(new Splitjoin<>(
        new DuplicateSplitter<Float>(),
        new RoundrobinJoiner<Float>(),
        new LowPassFilter(rate, low, taps, 0),
        new LowPassFilter(rate, high, taps, 0)),
      new Subtractor());
  }
}
```

Figure 5-6: BandPassFilter from the FMRadio benchmark, extending Pipeline to create a reusable component. This pipeline is instantiated once for each equalizer band.

```
Input<Float> input = Input.fromBinaryFile(Paths.get("fmradio.in"),
  Float.class, ByteOrder.LITTLE_ENDIAN);
Output<Float> output = Output.toBinaryFile(Paths.get("fmradio.out"),
  Float.class, ByteOrder.LITTLE_ENDIAN);
StreamCompiler compiler = new Compiler2StreamCompiler(24);
OneToOneElement<Float, Float> graph = new FMRadioCore();
CompiledStream stream = compiler.compile(graph, input, output);
stream.awaitDrained();
```

Figure 5-7: Compiling the FMRadio benchmark for 24 cores.

returned by the StreamCompiler. The user can switch between the StreamJIT interpreter, commensal compiler and distributed back-end just by using a differentStreamCompiler implementation; no changes to the stream graph are required. Figure 5-7 shows how to compile the FM radio example, whose top-level element is a FMRadioCore instance.

5.2 StreamJIT Compared to StreamIt

The StreamJIT language is strongly inspired by StreamIt (see section 2.2.1), making similar use of data rates to enable parallelism and structured streams for composition and reuse. The most obvious difference is that StreamJIT is embedded in Java whereas StreamIt is a separate language. The embedding allows the non-streaming parts of an application to be written in Java rather than shoehorned into StreamJIT. In StreamIt, MPEG-2 bitstream parsing is parallelizable but can only implemented with a filter that only executes once for the entire program (hence why the MPEG2 benchmark starts from a parsed bitstream); a StreamJIT application could perform the parallel parse outside the stream. StreamJIT programs can also transparently call existing Java libraries, where StreamIt did not provide even a foreign function interface.

The embedding also allows stream graph construction at run time in response to 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).

At the language level, StreamJIT supports user-defined splitters and joiners where StreamIt provides only built-in roundrobin and duplicate splitters and roundrobin joiners. StreamJIT currently does not implement teleport messaging; it would be easy to implement in the interpreter but challenging in the compiler and distributed back-end.

5.3 StreamJIT 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 At development time, users implement worker subclasses and write code to assemble and compile stream graphs.

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, here 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 or otherwise). (See figure 5-8.) 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 5.7.

5.4 StreamJIT Interpreted Mode

In interpreted mode, the StreamJIT runtime runs a fine-grained pull schedule (defined in [Thi09] 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.

5.5 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

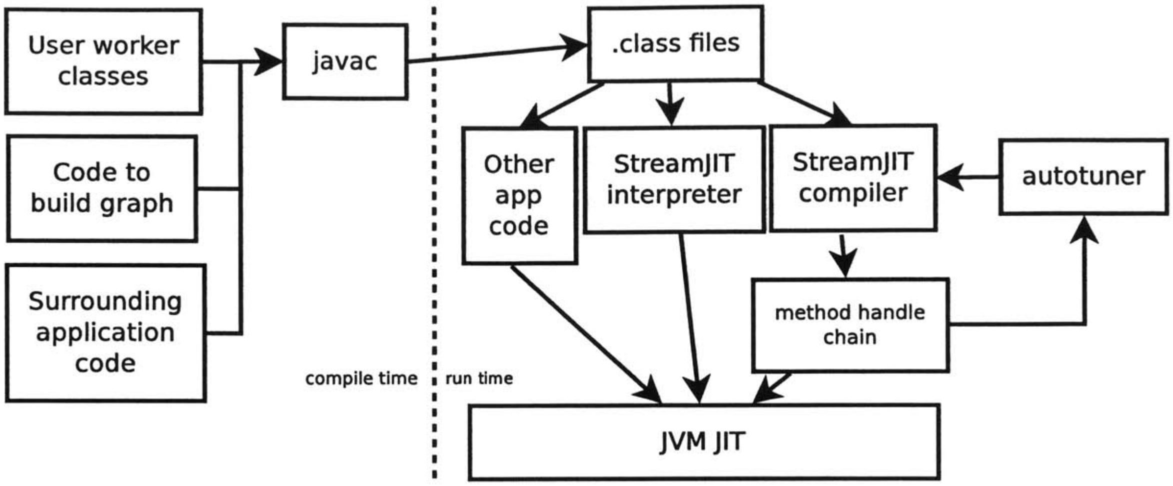


Figure 5-8: StreamJIT workflow. At compile time, user worker classes and graph construction code are compiled by `javac` just like the surrounding application code. At runtime, the StreamJIT interpreter runs `javac`'s output as is, while the compiler optimizes and builds a method handle chain. Either 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 5.6).

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.

The JVM JIT sees all values bound into the method handle chains as compile time constants, allowing the JVM to emit specialized code for each core. Constant loop variables and array lengths enable unrolling, constant object references enable devirtualization without type guards, and the JVM is able to inline as far down as it deems profitable, enabling better JVM optimizations than if the loops were implemented with bytecode. At the same time, building the method handle chain is much easier than emitting bytecode. The loop combinators are simply static methods containing loops, storage implementations are classes implementing the storage interface, and the chain-building code is normal Java code calling the method handle APIs. (See section 4.3 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 5.5.5 and 5.5.8 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 5.6.

5.5.1 Fusion

The compiler first converts the input stream graph made of pipelines and splitjoins into an unstructured stream graph containing only the workers. The compiler then fuses workers into groups. Groups have no internal buffering, while enough data items are buffered on inter-group edges to break the data dependencies between groups (software pipelining [RG81]). 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.

5.5.2 Scheduling

To execute with minimal synchronization, the compiler computes a *steady-state schedule* of filter executions which leaves the buffer sizes unchanged [Kar02]. 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.

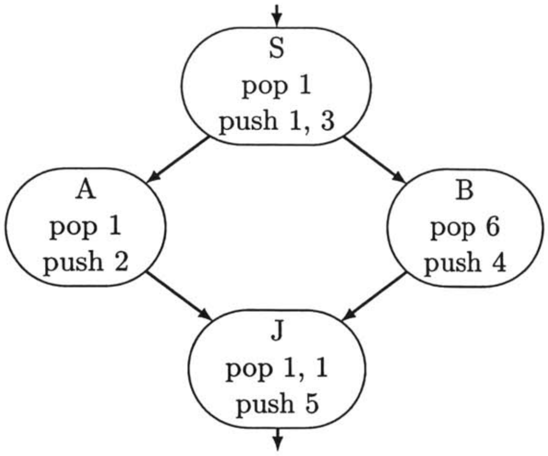
The compiler finds schedules by formulating integer linear programs. The variables are execution counts (of workers or groups); the constraints control the buffer delta using push and pop rates as coefficients. For example, to leave the buffer size unchanged between worker x_1 with push rate 3 and worker 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 5-9 for a steady-state example.) The resulting programs are solved using lp_solve[1]. While integer linear programming is hard in the general case, in our experience with StreamJIT, the instances arising from scheduling are easy.

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 executes 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. 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).

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



$$\begin{gathered} S - A = 0 \\ 3S - 6B = 0 \\ 2A - J = 0 \\ 4B - J = 0 \\ S \geq 0 \\ A \geq 0 \\ B \geq 0 \\ J \geq 0 \\ S + A + B + J \geq 1 \end{gathered}$$

Figure 5-9: An example stream graph group and the constraints generated for it during intra-group steady-state scheduling. Each variable represents the number of executions of the corresponding worker in the steady-state schedule. The first four constraints (one per edge) enforce that the buffer sizes do not change after items are popped and pushed. The second four (one per worker) enforce that workers do not execute negative times, and the last ensures at least one execution happens (progress is made). Inter-group scheduling is the same, except with groups instead of workers.

5.5.3 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, a `RoundrobinSplitter(2)` instance with three downstream workers distributes two items to its first, second and third outputs in turn. It can be removed by modifying its downstream workers to read from indices $6i$, $6i+2$ and $6i+4$ in the splitter's input. 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.

5.5.4 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` [Typ], 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.

5.5.5 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.

```
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 5-10: LowPassFilterArchetype: the result of bytecode rewriting on LowPassFilter from figure 5-3, 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.

Figure 5-10 shows the result of rewriting the example filter from figure 5-3.

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 5-3). 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.

5.5.6 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.

5.5.7 Work Allocation

The compiler then divides the steady-state inter-group schedule among the cores, as directed by the autotuner. Each core receives some or all executions of some or all groups. 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.

5.5.8 Code Generation

To generate code to run the initialization and steady-state schedules, the compiler builds a method handle chain for each core. (For background information on method handle chains, see section 4.3.) For each worker, its corresponding state holder class is initalized with the values of worker 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.

This 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

```
core10 handle
  group0 loop(iter 0 to 256)
    worker0 loop(iter _*4 to _*4+4)
      AdderArchetype::workObjectint(stateholder,
        stor0.readHandle, stor1.writeHandle,
        _*worker0.pop, _*worker0.push)
    worker1 loop(iter _*2 to _*2+2)
      AdderArchetype::workintint(stateholder,
        stor1.readHandle, stor2.writeHandle,
        _*worker1.pop, _*worker1.push)
  group4 loop(iter 512 to 2048)
    worker4 loop(iter _*2 to _*2+2)
      MultArchetype::workintint(stateholder,
        stor4.readHandle, stor8.writeHandle,
        _*worker4.pop, _*worker4.push)
```

Figure 5-11: Structure of a core code handle. Underscores represent the indices being computed in the loops; named values are bound constants. No executions of the group containing workers 2 and 3 were allocated to this core.

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. (See figure 5-11.)

5.5.9 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 5.5.2: 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.

5.5.10 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

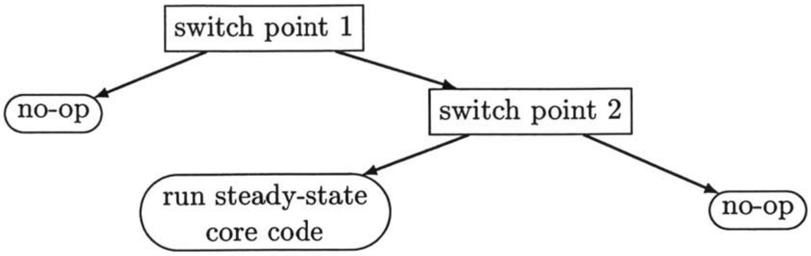


Figure 5-12: Thread behavior during the host lifecycle. Initialization and draining are single-threaded, so they occur at the barrier action. *no-op* (no operation) indicates a method handle that does nothing.

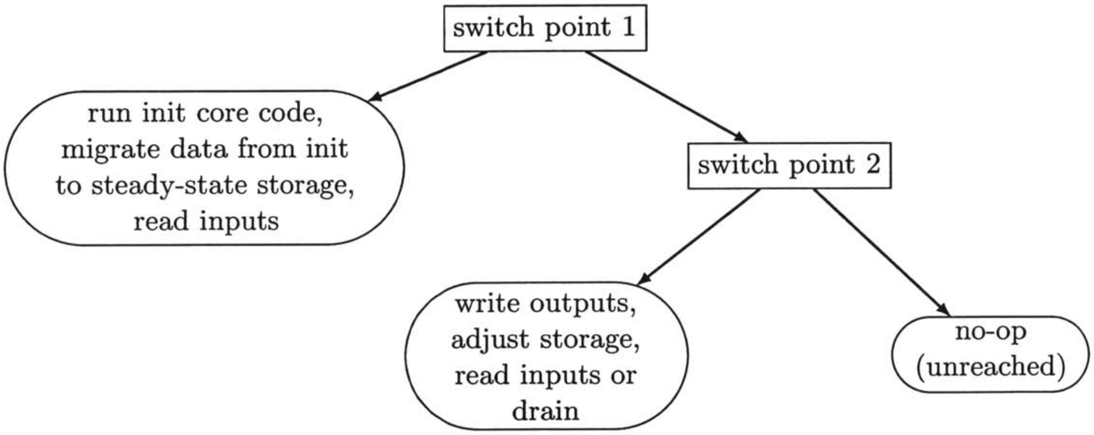

switch point 1
run init core code, migrate data from init to steady-state storage, read inputs
switch point 2
write outputs, adjust storage, read inputs or drain
no-op (unreached)

Figure 5-13: 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.

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 5-12 and 5-13 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 outputs, 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.

5.6 Autotuning

In place of heuristics, the StreamJIT compiler uses the OpenTuner [AKV$^+$13] 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.

5.7 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.

Each partition is a connected subgraph and the graph of partitions is acyclic. (See figure 5-14.) 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 5-14) 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,

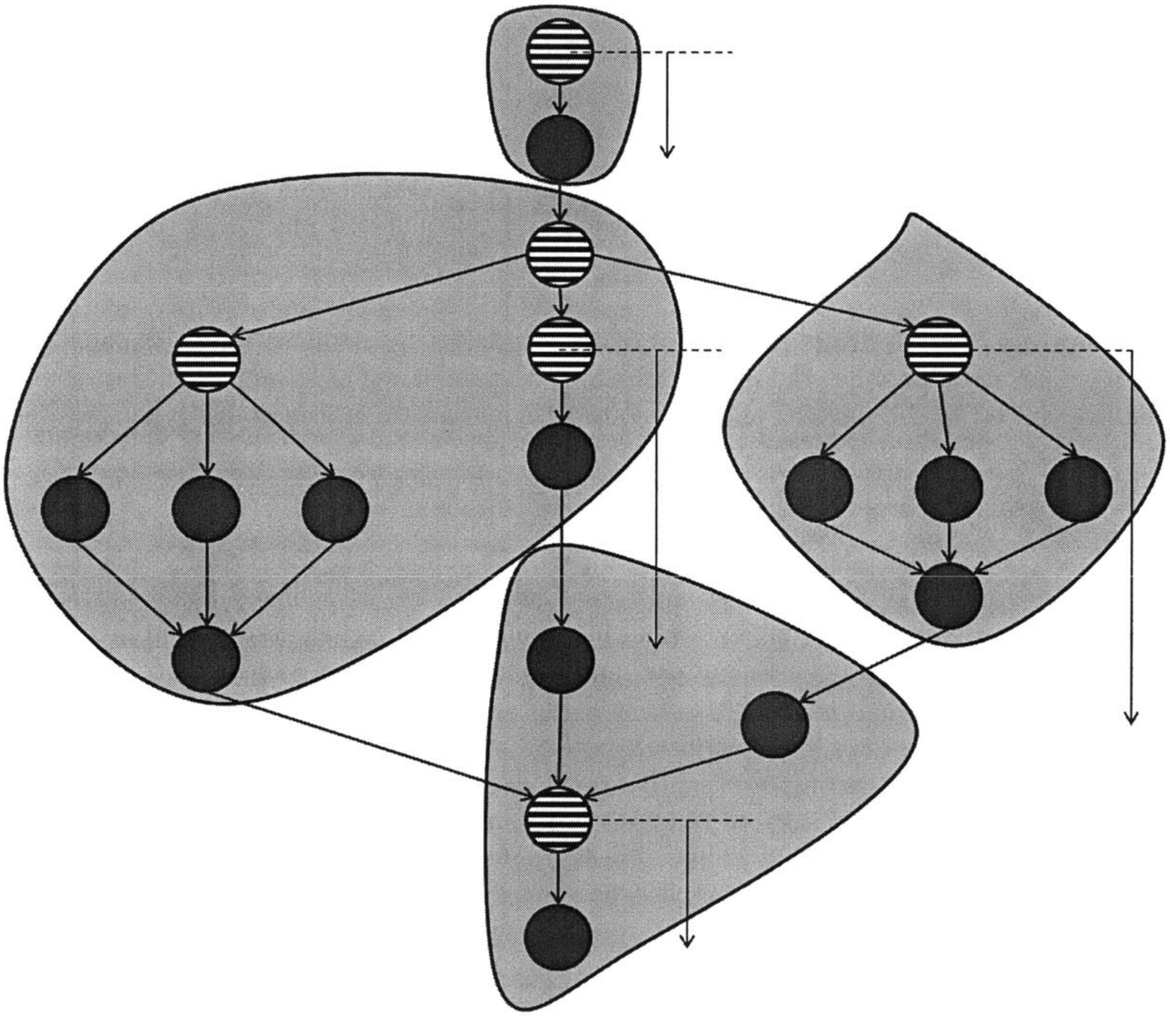


Figure 5-14: Partitioning a stream graph for distribution. Hatched workers are keys; arrows represent the cut-distance parameters. Both inner splitjoins are treated as single workers; the left does not have a cut-distance parameter as it's immediately followed by another key.

User API (plus private interpreter plumbing)	1213
Interpreter	1032
Compiler	5437
Distributed runtime	5713
Tuner integration	713
Compiler/interp/distributed common	4222
Bytecode-to-SSA library	5166
Utilities (JSON, ILP solver bindings etc.)	2536
Total (non-test)	26132
Benchmarks and tests	7880
Total	33912

Figure 5-15: StreamJIT Java code breakdown, in non-blank, non-comment lines of code. An additional 1087 lines of Python are for tuner integration.

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.

5.8 Evaluation

5.8.1 Implementation Effort

Excluding comments and blank lines, StreamJIT's source code consists of 33912 lines of Java code (26132 excluding benchmarks and tests) and 1087 lines of Python code (for integration with OpenTuner, which is written in Python); see figure 5-15 for a breakdown. In comparison, the StreamIt source code (excluding benchmarks and tests) consists of 266029 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 30812 lines, larger than the non-test code in StreamJIT.

5.8.2 Comparison versus StreamIt

Single-node offline-tuned StreamJIT throughput was compared with StreamIt on ten benchmarks from the StreamIt benchmark suite [TA10]. 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.

benchmark	StreamJIT	StreamIt	relative perf
Beamformer	2320186	1204215	1.9
BitonicSort	9771987	6451613	1.5
ChannelVocoder	551065	796548	0.7
DCT	23622047	6434316	3.7
DES	17441860	6469003	2.7
FFT	25210084	2459016	10.3
Filterbank	924499	1785714	0.5
FMRadio	2272727	2085143	1.1
MPEG2	32258065	-	-
Serpent	2548853	6332454	0.4
TDE-PP	12605042	2357564	5.3
Vocoder	406394	-	-

Figure 5-16: 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.

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 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 5-16. 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.

5.8.3 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 5-17 shows online tuning performance on the modified ChannelVocoder benchmark and figure 5-18 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 [APW$^+$12].

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 5-19 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 5-20 compares tuning progress for the modified FMRadio benchmark using 2, 4, 8, and 16 nodes over the 24-hour tuning run (figure 5-19 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.

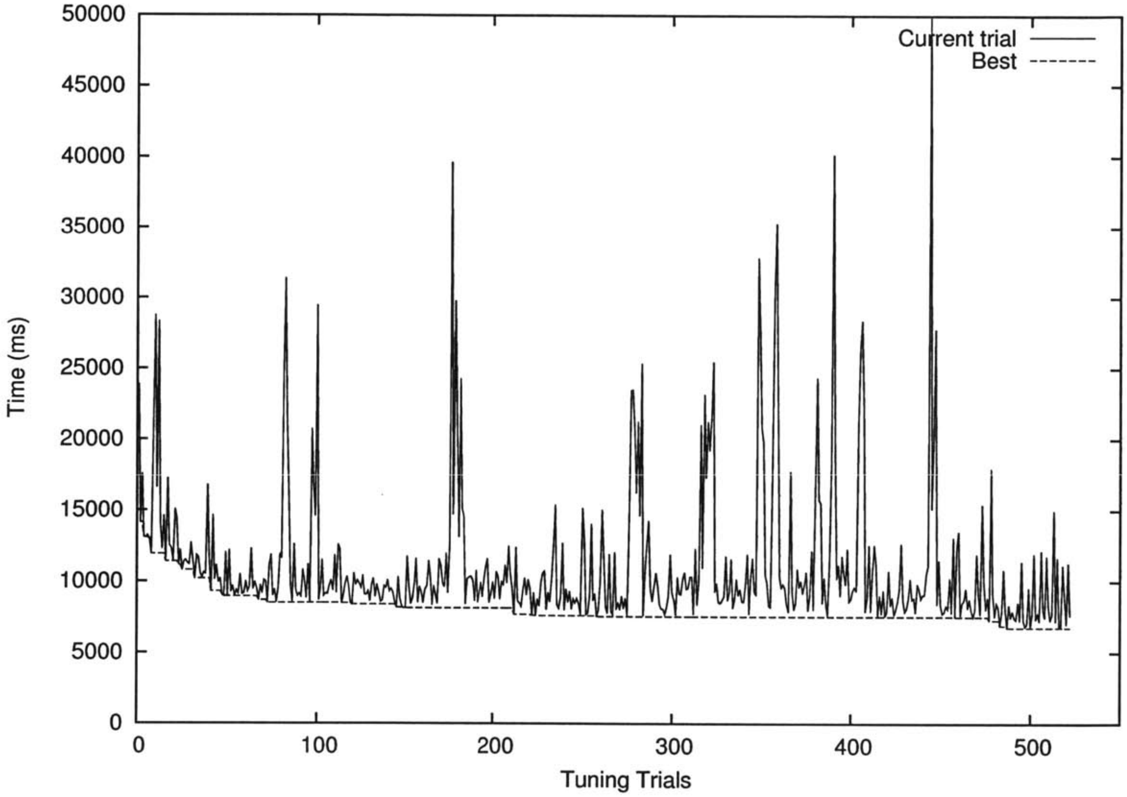


Figure 5-17: Online autotuning progress on the (modified) ChannelVocoder benchmark, showing the time in milliseconds to produce 30000 outputs (inverse throughput, lower is better).

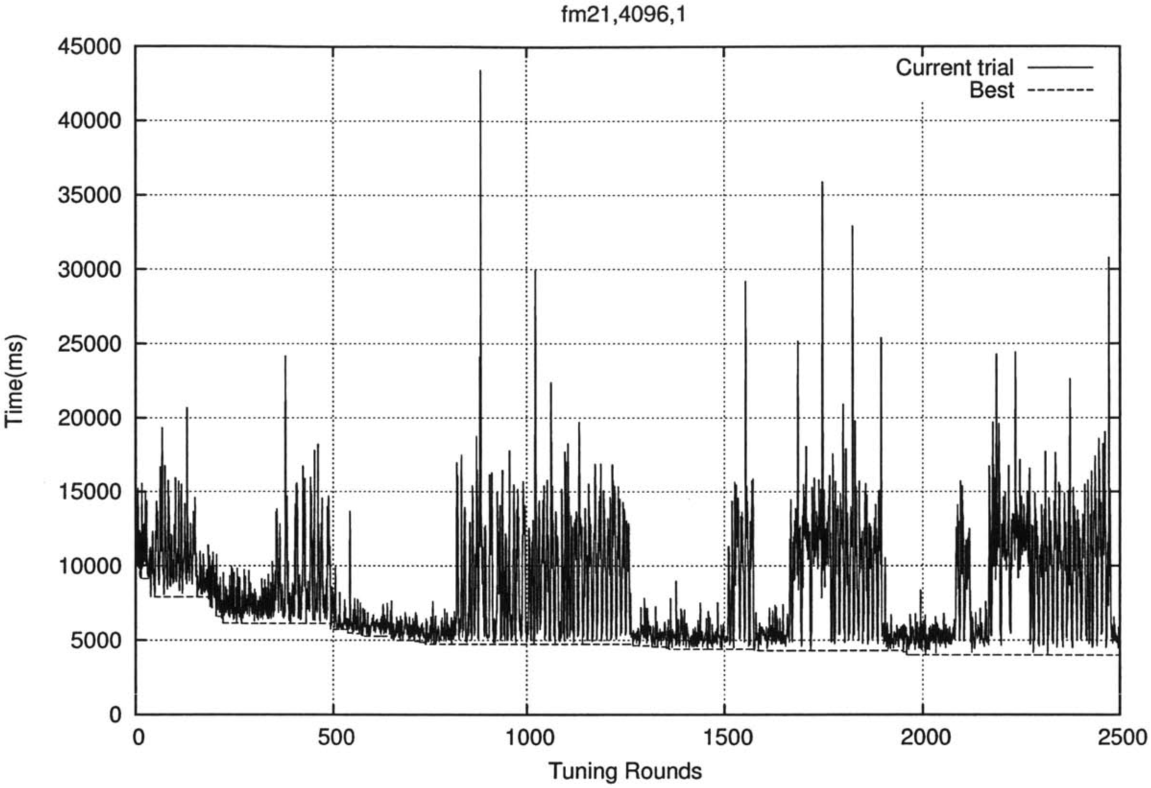


Figure 5-18: Online autotuning progress on the (modified) FMRadio benchmark, showing the time in milliseconds to produce 30000 outputs (inverse throughput, lower is better).

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

Figure 5-19: Speedup after distributed offline autotuning on 4, 8 and 16 nodes, relative to 2 nodes.

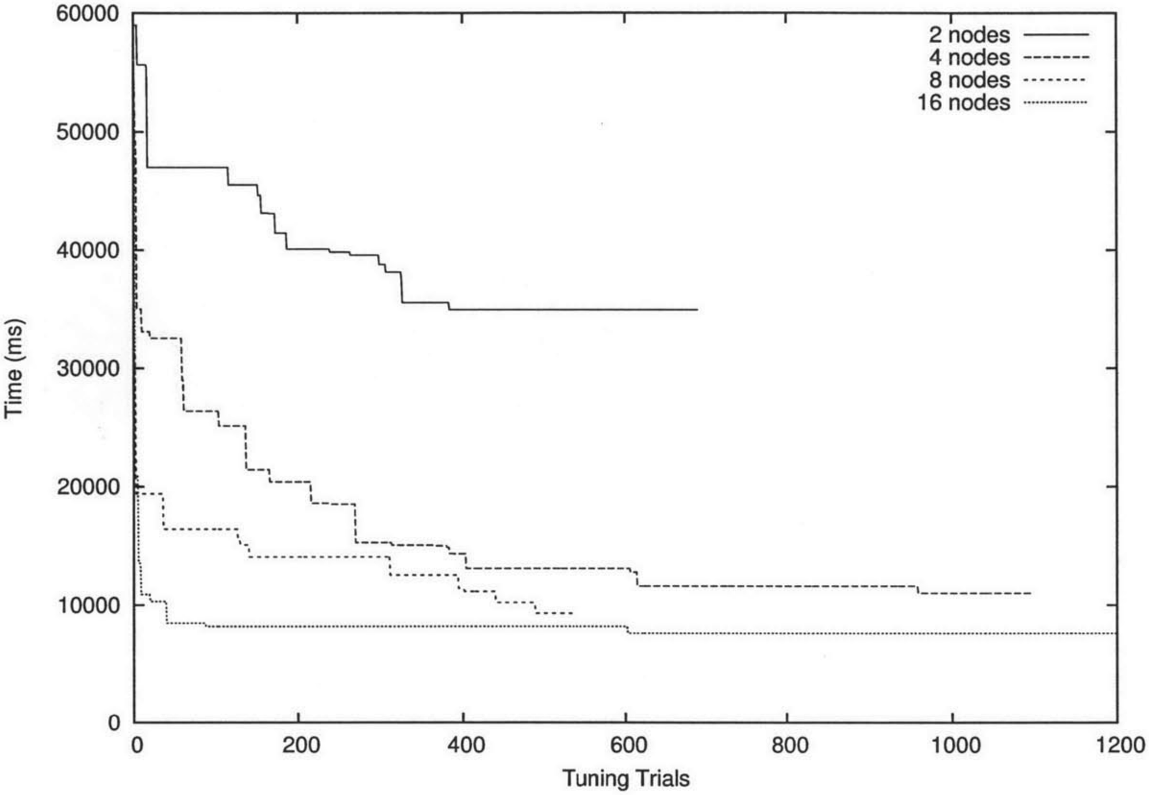


Figure 5-20: Distributed offline autotuning progress on the (modified) FMRadio benchmark, showing the time in milliseconds to produce 30000 outputs (inverse throughput, lower is better) of the best configuration up to that point. The benchmark was tuned for each node count for 24 hours.

Chapter 6

Lessons Learned

This chapter describes lessons learned during StreamJIT development – things usually left unpublished, but very useful to know when building on prior work.

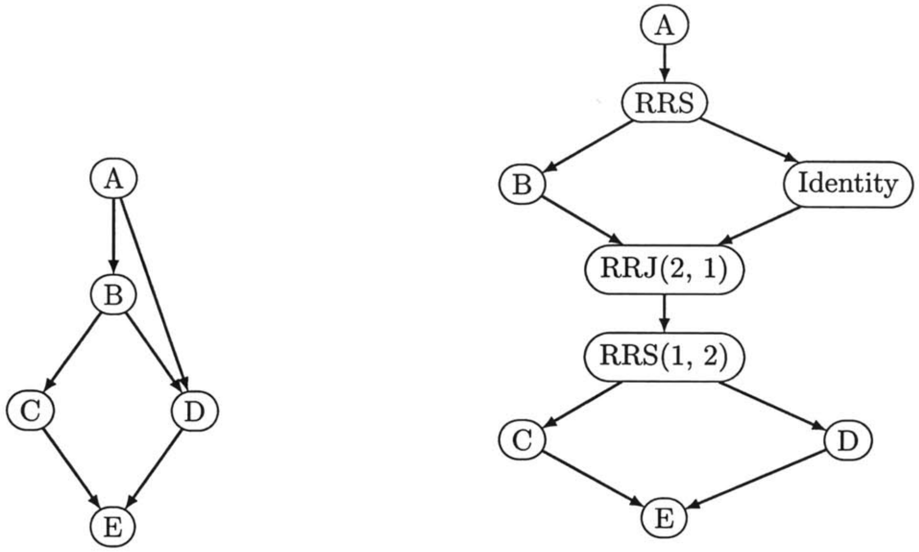
6.1 Structured versus Unstructured Streams

Early in StreamJIT's development we considered supporting unstructured stream graphs. Unstructured streams turned out to provide little benefit and their own difficulties, so StreamJIT returned to structured streams. If, in exploring new applications, unstructured stream patterns turn out to be common, we plan to support them with additional composition structures.

Structured stream limitations StreamIt introduced structured streams, stream graphs containing single-input, single-output filters composed with pipelines and splitjoins. While most of the signal-processing applications targeted by StreamIt can be conveniently expressed in structured streams, implementing some patterns between filters requires multiplexing data items through otherwise-unnecessary splitjoins (see figure 6-1). The compiler can recover the unstructured graph after performing splitter and joiner removal (what StreamIt calls *synchronization removal*), so the only effect of the transformation is to complicate the program, requiring the programmer to remember how data items are being multiplexed. [Thi09] figure 2-19 gives an example from the 3GPP StreamIt benchmark.

[Thi09] section 2.5 offers two possible solutions, tagged structs and teleport messaging, while noting that neither is satisfactory. By encapsulating the data items flowing through the structured graph in structs, a tag field can be added to define which edge in the unstructured graph the data is logically flowing on. Special splitters and joiners could then perform the multiplexing without programmer attention. However, filters must be modified to use tagged structs, limiting where in the stream graph the filter can be reused. Teleport messages, a general feature for communication separate from stream graph edges (not implemented in StreamJIT), are not limited by stream structure, but their out-of-band nature and lack of declared rates prevent them from being efficiently compiled.

Structured streams are also inconvenient for programs with multiple inputs or outputs, which must be multiplexed into or demultiplexed from a stream graph's single input and output. For example, the Beamformer benchmark processes inputs from multiple radar antennas.



(a) An unstructured stream graph.

(b) The same graph, transformed into a structured graph expressible in StreamJIT by multiplexing data items through (weighted) roundrobin splitters and joiners.

Figure 6-1: Unstructured to structured conversion.

StreamJIT wires API With these concerns in mind, the first StreamJIT API proposal did not require structured streams. Pipelines and splitjoins could still be used, but filters could also be composed by manually connecting output channels to input channels. We referred to this design as the *wires API* as the channel connection feature is reminiscent of hardware description languages.

We quickly decided against the wires API proposal as manual channel connection is error-prone and the resulting programs are difficult to read. When writing against the wires API, it is easy to forget or mix up connections, especially with splitters and joiners. When reading a wires API program, some effort is required to understand the graph being constructed. Even for the case of unstructured graphs, programs using manual connection are not much clearer than structured graphs using multiplexing. Finally, the wires API makes it easy to accidentally introduce loops, which we chose not to implement in StreamJIT because they cannot be efficiently compiled.

Bypass and other structures We proceeded to implement a structured streams API. Up to this point, we have focused on matching StreamIt performance on StreamIt benchmark applications, so being restricted to structured streams is not an issue. If future applications require unstructured constructs, additional composition structures besides the built-in `Pipeline` and `Splitjoin` classes can be introduced to support them within the framework of structured streams. For the pattern in figure 6-1, a `Bypass` class would compose two splitters, two joiners (including one at the end of the bypass structure) and an arbitrary one-to-one stream element into a one-to-one element that can be intermixed with pipelines and splitjoins.

```
Splitter a = new ASplitter(), b = new BSplitter();
Filter c = new CFilter();
Joiner d = new DJoiner(), e = new EJoiner();
a.connect(0, b);
a.connect(1, d, 0);
b.connect(0, c);
b.connect(1, d, 1);
c.connect(e, 0);
d.connect(e, 1);
```

Figure 6-2: Connecting filters using the wires API. `connect` connects the receiver's output to another worker's input, with indices provided when connecting from or to splitters and joiners. This code creates the unstructured graph shown in figure 6-1.

6.2 The Bytecode Compiler

Before building the StreamJIT compiler described in chapter 5, we built a one-pass compiler that did not use method handles, but instead used bytecode generation only. In this section, we will refer to the compilers as the *bytecode compiler* and *method handle compiler*. Like the method handle compiler, the bytecode compiler is a just-in-time compiler that runs during the surrounding application's run time. The bytecode compiler performs fusion, scheduling, and bytecode generation similarly to the method handle compiler, but as a one-pass compiler, it does not perform splitter and joiner removal or unboxing.

At the beginning of the StreamJIT project, we were concerned that StreamJIT programs would be hopelessly inefficient due to overheads inherent in the Java virtual machine. The bytecode compiler dispelled these fears by demonstrating performance within an order of magnitude of the StreamIt native code compiler. We abandoned development of the bytecode compiler soon after, partially because its one-pass nature made splitter and joiner removal tricky to implement, but mostly because Java code that generates bytecode is difficult to maintain, especially when multiple forms (chosen via autotuning) must be generated.

Consider backing storage selection. The bytecode compiler uses Java arrays, which are passed directly to the archetypal work methods and indexed with the `arraylength` bytecode instruction. The obvious way to support multiple storage implementations is to introduce a `Storage` interface with `get` and `set` methods. The archetypal work methods then take `Storage` references as arguments. The interface invocation inhibits optimization. If only one storage implementation is used with a particular work method, the JVM JIT can inline the storage methods, but must emit a type guard; while the type check is cheap, it will be performed millions of times per second, and even if it were free, its presence limits what reorderings the JVM can perform. If multiple implementations are used, the JVM will either inline multiple methods with multiple type guards (a polymorphic inline cache) or fall back to a slow virtual call. Using an interface is clean and idiomatic, but incurs overheads not appropriate for a high-performance system.

If a run-time abstraction introduces too much overhead, how about a compile-time one? The code generation differences between storage implementations could be encapsulated behind a `StorageGenerator` interface with methods that return the parameter type and generate the bytecode that reads or writes the storage. (Splitters and joiners would require coordination between generators to handle their multiple outputs or inputs.) Where the run-time abstraction used a problematic interface call, the compile-time abstraction uses separate archetypal work methods for each combination of input and output storage, eliminating the type guards at the expense of some code bloat. This approach resolves the performance issues, but complicates the compiler.

The method handle compiler takes the best of these two worlds: the method handles provide the `Storage` interface abstraction, while allowing full inlining as the method handles bound into the method handle chain are compile-time constants to the JVM JIT. Using method handles has the side benefit that Java code that generates bytecode for an operation (e.g., writing storage) can be replaced by Java code directly performing that operation, making the compiler more maintainable. Only the archetypal work method generation requires working at the bytecode level, using essentially the same code as the bytecode compiler.

At the time we built the method handle compiler, our underlying JVM (OpenJDK with the HotSpot JIT compiler) was reluctant to inline method handles. In particular, as the StreamJIT schedule multiplier increased, the method handle chain was invoked less often to do the same amount of work, fooling the JVM into treating it as cold code. By dynamically creating new classes that directly call a constant method handle, combined with judicious use of the `CompileThreshold` and `ClipInlining` JVM flags, we convinced HotSpot to inline all the way through and generate specialized machine code for each core, resulting in performance comparable to the bytecode compiler. More recent versions of HotSpot use different compilation heuristics for method handles (including incremental inlining) that render precise flag adjustments unnecessary.

6.3 Autotuner Search Space Design

OpenTuner seems to promise users that if they can describe their search space in terms of standard parameters, OpenTuner can find good configurations without requiring the user to understand the search algorithms. Unfortunately, we found this to be a leaky abstraction; autotuning performance is strongly dependent on good search space design.

The search space can be viewed as an undirected graph where the vertices are configurations and the edges connect configurations that are "close" (can be reached by small mutations). The autotuner will trade off between exploring the search space by testing random configurations and exploiting good configurations by testing other nearby configurations to find improvements. If most configurations are bad, exploring will be unprofitable. If most configurations "nearby" a good configuration are worse instead of better, exploitation will be unprofitable. When designing the initial parameterizations of the StreamJIT search space, we focused on covering the entire space without regard to how easy the graph was to search, resulting in poor tuning performance.

In this section, we present how the search space for work allocation evolved over time.

The first parameterization for work allocation used one integer parameter per core per group specifying the number of iterations of the group to put on that core, allocated beginning with the first core. For example, if the schedule specified 30 iterations and one group's eight core parameters were 25, 18, 35, 6, 1, 10, 23 and 40, the first core would be allocated 25 iterations, the second core the remaining 5, and the remaining six cores 0. Random configurations tend to allocate all the work to the first few cores, and once all iterations have been allocated, modifications to the remaining parameters have no effect. In theory, this parameterization fully covers all possible allocations, but in practice the autotuner cannot find configurations that allocate equal work to all cores.

Our next attempt abandoned covering the search space to make equal division more discoverable, in the hope that searching a small space well would produce better average results than searching a large space poorly. Each group used a bitset parameter that controlled which cores the group would be allocated to, then the iterations were divided equally. This configuration still resulted in poor performance because using all the cores (the typical best configuration) requires setting all the bits. When using n cores, this configuration is only $\frac{1}{2^n}$ of the search space, making it hard to discover; in expectation over random configurations, only half the cores will be used. Improving a random configuration is difficult for mutators that toggle each bit with equal probability; as more bits are set, additional mutations are more likely to clear bits than set additional ones.

We next turned to OpenTuner's support for domain-specific parameters with custom mutators. Our composition parameter was composed of an array of floats (one per core) normalized to sum to 1, representing the fraction of work to assign to that core. As equal division over all cores is frequently the best configuration, one mutator set all the floats to $\frac{1}{n}$. Other mutators added a core by moving a fraction of each non-zero float to a zero float, removed a core by zeroing a float and distributing its value over the other floats, or shuffled the floats (for load balancing when some cores have more work than others). OpenTuner's search algorithms can also manipulate the floats directly, to allow hill-climbing. Providing an equal-division mutator allowed OpenTuner to quickly find the equal-division configuration, but tuning progress quickly stopped on graphs containing stateful filters (which require unequal allocations for load balancing). Hill-climbing algorithms were confused by the normalization, while algorithms using our mutators could not make enough progress by just adding or removing cores.

At this point we concluded that OpenTuner could not magically discover the best configurations. Instead we thought about what good configurations would typically look like: equal work divided among some number of cores, with a few cores receiving less work to load-balance around unparallelizable stateful filters. The important variables were the number of cores to use, the amount of work to move for load balancing, and which particular cores to use (core identity becomes important when load balancing). This analysis lead to the subset-bias parameterization described in section 5.6. While random configurations use only half the cores on average, hillclimbers can quickly increase the total core count and decrease the bias count.

Of course, better parameterizations may exist. The subset-bias parameterization performs well on our particular benchmarks, but the empirical nature of autotuning prevents generalization to all StreamJIT programs.

Chapter 7

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 on average 2.4 times higher than StreamIt's native code compiler with an order of magnitude less code. We additionally implemented automatic distribution over multiple nodes at the low cost of writing the socket communication code.

Bibliography

[ABB+99] 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.

[ABB+12] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil 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.

[ABCR10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric 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.

[ACD+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.

[ACW+09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman 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.

[AD78] WB Ackerman and JB Dennis. VAL: A Value Oriented Algorithmic Language. Preliminary Reference Manual, Laboratory for Computer Science. Technical report, MIT, Technical Report TR-218 (June 1979), 1978.

[AFG+05] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[AKV+13] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. 2013.

[APW$^+$12] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. SiblingRivalry: Online Autotuning Through Local Competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 91–100, New York, NY, USA, 2012. ACM.

[Arm07] J. Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

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

[AW77] E.A. Ashcroft and W.W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[BFH$^+$04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[Bha11] Shashank Bharadwaj. invokedynamic and Jython. In *JVM Language Summit 2011*, JVMLS 2011, 2011.

[BHLM91] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Multirate signal processing in Ptolemy. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 1245–1248. IEEE, 1991.

[BSL$^+$11] K.J. Brown, A.K. Sujeeth, Hyouk Joong 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.

[DDE$^+$11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd 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.

[DLD$^+$03] W.J. Dally, F. Labonte, A. Das, P. Hanrahan, J.H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T.J. Knight, et al. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. ACM, 2003.

[FJ98] M. Frigo and S.G. 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.

[Gar12] Miguel Garcia. Runtime metaprogramming via java.lang.invoke.MethodHandle. May 2012.

[Gor10] M.I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures.* PhD thesis, Massachusetts Institute of Technology, 2010.

[HCK+09] A.H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 214–223. IEEE, 2009.

[Hei11] Dan Heidinga. MethodHandle Implementation Tips and Tricks. In *JVM Language Summit 2011*, JVMLS 2011, 2011.

[HMJ76] P. Henderson and J.H. Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103. ACM, 1976.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HS] Hotspot internals for OpenJDK. `https://wikis.oracle.com/display/HotSpotInternals/Home`.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis 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.

[Ima] ImageMagick. `http://www.imagemagick.org/`.

[Inm88] Inmos Corporation. *Occam 2 reference manual.* Prentice Hall, 1988.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[Kar02] Michal Karczmarek. *Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language.* PhD thesis, Massachusetts Institute of Technology, 2002.

[KN13] Abhishek Kulkarni and Ryan 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.

[KP78] Arvind KP. Gostelow, and WE Plouffe. the Id report: An asynchronous programming language and computing machine. Technical report, Technical Report TR114A, University of California, 1978.

[LM87] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, 100(1):24–35, 1987.

[LYBB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. Java Virtual Machine Specification, Java SE 8 Edition, 2014.

[Mat02] Peter Raymond Mattson. *A programming system for the Imagine media processor.* PhD thesis, Stanford, CA, USA, 2002. AAI3040045.

[MBB06] Erik Meijer, Brian Beckman, and Gavin 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.

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

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

[Nut11] Charles Nutter. Adding invokedynamic Support to JRuby. In *JVM Language Summit 2011*, JVMLS 2011, 2011.

[OPT09a] F. Otto, Victor Pankratius, and W.F. 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.

[OPT09b] Frank Otto, Victor Pankratius, and WalterF. Tichy. XJava: Exploiting Parallelism with Object-Oriented Stream Programming. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 875–886. Springer Berlin Heidelberg, 2009.

[OSDT10] Frank Otto, ChristophA. Schaefer, Matthias Dempe, and WalterF. Tichy. A Language-Based Tuning Mechanism for Task and Pipeline Parallelism. In Pasqua DAmbra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 328–340. Springer Berlin Heidelberg, 2010.

[Pap07] M. Papakipos. The PeakStream platform: High-productivity software development for multi-core processors. *PeakStream Inc., Redwood City, CA, USA, April*, 2007.

[PM12] Julien Ponge and Frédéric Le Mouël. JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications. *CoRR*, abs/1210.1039, 2012.

[RG81] 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.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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.

[RO10] Tiark Rompf and Martin 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.

[Ros09] J.R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, page 2. ACM, 2009.

[SGA$^+$13] Robert Soulé, Michael I. Gordon, Saman Amarasinghe, Robert Grimm, and Martin Hirzel. Dynamic Expressivity with Static Optimization for Streaming Languages. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 159–170, New York, NY, USA, 2013. ACM.

[SGB$^+$13] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle 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.

[SPGV07] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan 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.

[Ste97] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[TA10] William Thies and Saman 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.

[TCC$^+$09] A. Tiwari, Chun Chen, J. Chame, M. Hall, and J.K. 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.

[Thea] Aggregate Operations. `http://docs.oracle.com/javase/tutorial/collections/streams/index.html`.

[Theb] Type Annotations and Pluggable Type Systems. `http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html`.

[Thi09] W.F. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[TKS+05] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 224–235, New York, NY, USA, 2005. ACM.

[TPO06] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 325–335. ACM, 2006.

[TR10] Christian Thalinger and John 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.

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

[Wï1] Thomas 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.

[WD98] R. Clint Whaley and Jack 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.

[WO08] Zheng Wang and Michael F. P. O'boyle. Using Machine Learning to Partition Streaming Programs. *ACM Trans. Archit. Code Optim.*, 10(3):20:1–20:25, September 2008.

[WW12] Christian Wimmer and Thomas 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.

[XJJP01] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David 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.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon 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.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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.

[ZLRA08] David Zhang, Qiuyuan J. Li, Rodric Rabbah, and Saman Amarasinghe. A lightweight streaming layer for multicore execution. *SIGARCH Comput. Archit. News*, 36(2):18–27, May 2008.