Compiler Optimizations for an Asynchronous Stream-oriented Programming Language

by

Michael B. Craig

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

Author .....
Department of Electrical Engineering and Computer Science
August 19, 2008

Certified by .....
Samuel Madden
ITT Career Development Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Compiler Optimizations for an Asynchronous Stream-oriented Programming Language

by

Michael B. Craig

Submitted to the Department of Electrical Engineering and Computer Science
on August 19, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Stream-oriented programs allow different opportunities for optimization than procedural programs. Moreover, as compared to purely synchronous stream-oriented programs, optimizing for asynchronous stream-based programs is difficult, owing to the latters' inherent unpredictability. In this thesis, we present several compiler optimizations for WaveScript, a high-level, functional, stream-oriented programming language. We also present a framework for using profiling of stream-graph execution to drive optimizations; two of the optimizations use this profiled information to generate noticeable performance benefits for real-world applications written in WaveScript. Thus, it is shown that profiling presents an important avenue by which to optimize asynchronous stream-based programs.

Thesis Supervisor: Samuel Madden
Title: ITT Career Development Professor

Acknowledgments

Much appreciation is due to my family and friends, who continue to support me in whatever decisions I make; the unconditional love shown by a precious few is absolutely priceless.

I am deeply indebted to Professor Samuel Madden for his continued support, over a (protracted) research period during which I encountered serious personal hurdles. I could scarcely imagine a more helpful thesis advisor.

I am similarly indebted to Ryan Newton for his unwavering commitment over the same period. I am especially grateful for his having brought me in to work on his software, for his masterful knowledge, and regular inspiration.

Finally, I am thankful to a number of others within NMS, and CSAIL more generally, for the support they provided, directly or indirectly relating to my work. From the WaveScope group, Lewis Girod and Stanislav Rost in particular have provided very helpful assistance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents a number of optimization extensions to the compiler for the functional, stream-oriented programming language WaveScript. WaveScript is designed for writing high-data rate stream-processing applications, such as those that perform real-time filtering, audio/video processing, and signal processing more generally [NGC+08].

WaveScript programs describe *asynchronous* stream graphs: the data rates between actors in the graph are not, in general, predictable, and may vary freely. For this reason, it is difficult to apply optimizations from the fairly large synchronous data-flow (SDF) domain directly. The approach presented in this thesis is to use profiling information, gathered during execution of the stream graph, to help infer the rates and execution costs of each module of the program, and to use that profile to determine appropriate optimizations to apply during re-compilation.

This thesis describes how support for profiling was added into the existing WaveScript compiler; it also describes the optimizations that have been added.

1.1 Stream Programming

Stream-oriented programming represents input as an unbounded, time-ordered sequence of data – sometimes called *tuples* to match database nomenclature. A stream program filters and transforms this sequence to produce a similarly unbounded out-

put stream. Such a program is typically structured as a composite of simpler stream "programs," sometimes called operators, or, within the WaveScript compiler, simply *boxes*; the output an upstream box is fed into the input of a downstream box.

Though most would argue that procedural or object-oriented programming represents the dominant paradigm within the software industry, stream programming is much more relevant for a wide variety of applications, especially those that must process unbounded sequences of data in real-time. Audio/video processing, medical- and environmental-sensor monitoring, and financial-data stream mining are examples of such application domains.

As procedural programs are structured as a set of functions which *call* one another, stream programs have the analogous structure of a set of actors connected together. There is a fundamental difference, though: whereas functions in a procedural program pass *control* between themselves, actors in a stream program pass *data*. The difference has very important implications. Code for a (single-threaded) procedural program specifies the exact order of instructions to run (modulo branch instructions that depend on input). Code for a stream program only specifies such ordering within each actor; it does not specify when each actor should be run (other than that it must have some part of its input stream available). The structure of a stream program is such that there is no global state accessible by more than one actor; there is only local, per-actor state. Thus, a stream program requires some further runtime engine to decide when to run each actor, and how to queue intermediate sections of input streams at each actor (the way in which WaveScript decides this is discussed later on).

1.2 WaveScript

Stream-oriented programming is not a new paradigm: it has long been the focus of DSP programming; however, there are few high-level stream-oriented programming environments, and much chip-specific DSP programming must be done at the assembly-code level [KL99].

WaveScript specifically targets asynchronous stream programs, in which neither the data rates between actors, nor the execution times of actors, are specified in advance. This is in contrast to synchronous stream-processing programs, in which the data rates between actors are explicitly specified by the programmer.

1.2.1 The Language

WaveScript is a strongly, statically typed, (mostly) functional language, similar in much of its syntax to ML. In WaveScript, every actor produces a single output stream; however, a WaveScript program is not an explicit declaration of its actors. Instead, it is a higher-level functional program which treats streams as first-class objects. This program is statically elaborated to produce an explicit graph of actors, which completely describes the stream program as defined in the previous section. Figure 2-5 in the next chapter shows two very short example programs.

1.2.2 The Compiler

The WaveScript compiler, written in Scheme, transforms programs into S-expressions – the list-based data structures at the core of Scheme, and its syntax – just after parsing. This intermediate S-expression representation itself looks very much like simple Scheme code, and is manipulated by a long sequence of mini-passes, each of which applies transformations using pattern matching on subtrees of the abstract syntax tree. A "core traverser" walks through the AST and passes subtrees to the mini-pass. Mini-passes typically use the Scheme function `match` to implement pattern matching, and focus only on the syntax elements of interest.

This existing structure has made it very easy to extend the compiler. Though some changes have been introduced to the intermediate representation's syntax, the core of most optimizations can be implemented as single mini-passes.

1.2.3 The Back-Ends

The WaveScript compiler targets several different *back-ends* for execution of the stream graph. A back-end consists of the target programming language, and some form of run-time engine also written in that language. The compiler targets several back-ends, among which are the following:

- XStream, a fairly large engine that includes a memory manager and several different schedulers, written in C++, described in [GMN+08]
- Scheme code, which is nearly identical to the compiler's intermediate S-expression representation, and which can be executed when coupled with a small library of functions – this "simulator" back-end can be run from within the compiler
- Standard ML code for compilation by MLton, a whole-program compiler [Wee06]

In this thesis, though some profiling is performed using the Scheme simulator back-end, we are mostly concerned with the XStream target, and it is what we will use to measure performance improvements.

The rest of the thesis is organized as follows. Chapter 2 provides background information on graph-based, "actor-oriented programming," stream-oriented programming in WaveScript, and some relevant details of the WaveScript compiler. Chapter 3 discusses extensions to the compiler, including two small, illustrative optimizations, followed by the profiling framework, and concluded by two more substantial optimizations based on profiling. Chapter 4 discusses possibilities for future work, and Chapter 5 concludes.

Chapter 2

Background

Here we discuss stream programming as an instance of a class of programming models that are sometimes called "graphical programming" or "actor-oriented programming" [LN04]. We briefly describe Petri nets, a mathematical formalism that describes actor-oriented programming models, and then describe WaveScript stream graphs in similar formal terms. Finally, some further details of the WaveScript language, the compiler, and its internal representation of programs are presented.

2.1 Actor-Oriented Programming

In procedural programming, the chief abstraction is the *function*, by which the flow of program control is managed. In actor-oriented programming, functions may be used, but the important abstraction of *actors* is used to manage the flow of data, rather than program control.

An actor, like a function, is a piece of code which produces output from input. An actor, though, has an explicit notion of input and output *ports*, from which it reads data, and to which it writes data, respectively. An actor is *fired*, meaning its associated code is executed, whenever data is available on (some subset of) its input ports. In an actor-oriented programming, output ports of actors are connected to input ports of other actors to form a data-flow graph.

The actor is typically provided with instructions to read from input ports and write

to output ports at any point in its code; so, unlike a typical procedural function, an actor does not need to end execution to produce its output, and it is not inherently limited in the amount of output it generates during any one firing.

Some actors may be designated as *sinks*, meaning they have no output ports, or *sources*, meaning they have no input ports. Together these essentially form an actor-oriented program's I/O: sinks are where final results are gathered for presentation to the user, and sources are where data are initially presented to the program.

Because an actor can be fired whenever data are available on its input ports, control flow for an actor-oriented program is, in general, non-deterministic. Thus, there is a great amount of freedom in deciding the execution schedule of such a program, and there is opportunity for optimization that is very different from that of a procedural program. For example, a scheduler may run upstream actors many times, buffering their outputs for later delivery to downstream actors; or it may follow the data-flow path of every datum produced, traversing the actor graph in a depth-first manner. In fact, different paradigms within the broad scope of actor-oriented programming may assign their own semantics to the same actor graphs [LN04]. The high-level idea, though, is the same: actors are independent entities, with strictly local code and state, interacting by passing data between connected ports.

WaveScript, a stream-oriented programming language, presents one such paradigm: an actor is free to fire whenever input is available; there is only one output port per actor; and no data-flow cycles are allowed.

2.1.1 Other Platforms

A number of platforms are available to support different types of actor- or stream-oriented programming. Despite some similarities, though, they all differ from WaveScript in important ways.

Aurora – since commercialized as StreamBase – is a stream-based query engine for processing high-volume data [BBC+04]. With a focus on real-time processing, in which approximate results can be tolerated, it supports graceful degradation through intelligent load shedding. Its academic successor, Borealis [AAB+05], includes a num-

ber of advanced features such as dynamic revision of streams, dynamic revision of queries, and multi-host distribution inherited from Medusa [ZSC+03]. The model for application development provided by Aurora and Borealis is that of query processing, similar to that of non-streaming relational databases. Every *box*, or actor[1], in an Aurora stream graph, is an instance of one of a few different types, such as *Map*, *Filter*, *Union*, or *Aggregate*. This is the key difference from WaveScript, which employs a much more general-purpose programming model to support signal-processing applications – especially those of various physical-science domains – for which such query processing is insufficient. Though Aurora and Borealis support user-defined operators, they must be written and compiled separately and are completely opaque to optimizations provided by the systems.

TelegraphCQ has similar goals to Aurora and Borealis, as it is also a distributed, stream-based "continuous query" processing system [CCD+]. TelegraphCQ, though it does expose a language for describing actor graphs, chiefly supports queries written in a stripped-down version of SQL. Thus, a large part of its focus is on automatic distribution and adaptation of query plans. Again, despite similarities, it is not intended for all the same types of signal-processing applications as WaveScript.

StreamIt is a fairly advanced platform that does target signal-processing applications in which most operators are written by the programmer in a general-purpose language [TKA]. StreamIt does provide an actor-oriented model, and its programming language is based on Java. Unlike WaveScript, though, its focus is SDF applications: every operator must be explicit about how many tuples it reads, and how many it writes, per firing, and these two numbers must not change during execution.

Finally, Ptolemy II is a software system for studying actor-oriented programming in as general a setting as possible [LHJ+01]. Graphical *models* in Ptolemy II – programs assembled from operators, any of which may likely be user-defined – are executed by *directors* which control the concrete semantics. For example, a single model (under appropriate constraints), may be excuted under the SDF director, or the

[1]Though it is not especially unique, WaveScript has inherited some of the nomenclature from Aurora, such as that of *boxes*, *tuples*, and *windows*.

process-networks (PN) director [Kah74], or the discrete-event (DE) director [IGH+]. Models in Ptolemy II, then, are not strictly classified as stream-processing, and may be cyclic and highly dependent on feedback. Owing to its great generality – and its use as a tool for studying actor-oriented programming broadly – its performance is not suitable for the high-rate applications targeted by WaveScript.

2.1.2 Petri Nets

Petri nets are a very general mathematical formulation for describing actor-oriented programs. They are described briefly, here, in the language of [Pet77]; WaveScript stream graphs are then described as a specialization of Petri nets.

A Petri net (V, E) is a bipartite directed graph: V is partitioned into *places* P and *transitions* T such that every edge in E connects one place $p \in P$ and one transition $t \in T$ (direction being unimportant). E can be described as a pair of functions $I : T \to \mathcal{P}(P)$ and $O : T \to \mathcal{P}(P)$. $I(t)$ gives the set of places with inputs into t, i.e. $\{p \in P \mid (p,t) \in E\}$. Similarly, $O(t) = \{p \in P \mid (t,p) \in E\}$. Using this language to describe an actor-oriented program, a *transition* defines an actor, and a *place* defines a queue holding data passed between actors.

A Petri net thus models the static structure of a graphical program: each transition represents an actor, and each place represents a queue holding data passed between actors. To describe how the flow of data proceeds during program execution, the concept of *tokens* and *markings* is used. A token represents an individual datum created as output when an actor fires, and consumed as input when another actor fires. A marking $\mu : P \to \mathbb{N}_0^+$ indicates the number of tokens at every place in a Petri net; it represents a snapshot in time of the program, in between actor firings.

Figure 2-1 shows a simple marked Petri net with four places and four transitions. The thick vertical bars represent transitions, the large unfilled circles represent places, and the small filled circles represent tokens.

Evolution though time of a Petri net's markings is described using a *next-state function* $\delta : M \times T \to M$, where M is the set of all possible markings on the Petri net of interest. Applying δ to a marking and a transition, (μ, t), gives the marking

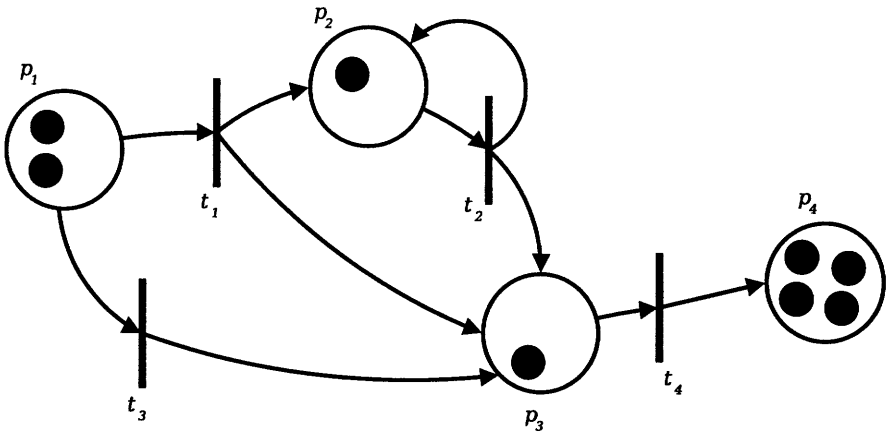

Figure 2-1: sample marked Petri net: the marking μ is $\{(p_1, 2), (p_2, 1), (p_3, 1), (p_4, 4)\}$

that results from firing t. In the definition of Petri nets, the firing of a transition t has rigid requirements and behavior: it can only occur when each of its inputs has at least one token; and after firing, one token is removed from each of its inputs, and one token is placed into each of its outputs. In other words, the requirement for firing t is that $\forall p\ [(p \in I(t)) \Rightarrow (\mu(p) > 0)]$, and the next marking $\mu' = \delta(\mu, t)$ is equal to μ except that $\forall p\ [(p \in I(t) \wedge p \notin O(t)) \Rightarrow (\mu'(p) = \mu(p) - 1)]$ and $\forall p\ [(p \in O(t) \wedge p \notin I(t)) \Rightarrow (\mu'(p) = \mu(p) + 1)]$. Thus, δ is actually a partial function on $M \times T$.

Execution of a marked Petri net can be thought of as a sequence of markings $\{\mu_n\}$ where each $\mu_{i+1} = \delta(\mu, t_i)$ where $t_i \in T$ (in case no transition can be fired under a marking μ_i, we may allow that the execution is "halted" and $\mu_{i+1} = \mu_i$). Given an initial marking μ_0, there are in general many possible executions: each one is essentially defined by the sequence of transitions $\{t_n\}$. The definition of Petri nets specifies δ, but not this sequence. When we describe WaveScript stream graphs in terms of Petri nets, in the next section, we will specify the rules they follow for execution.

Though Petri nets are a very useful tool for analysis of suitably modeled systems – such as graphical programs – we use them here only as a formal description language. In fact, because WaveScript stream graphs are acyclic, the problems of deadlock and

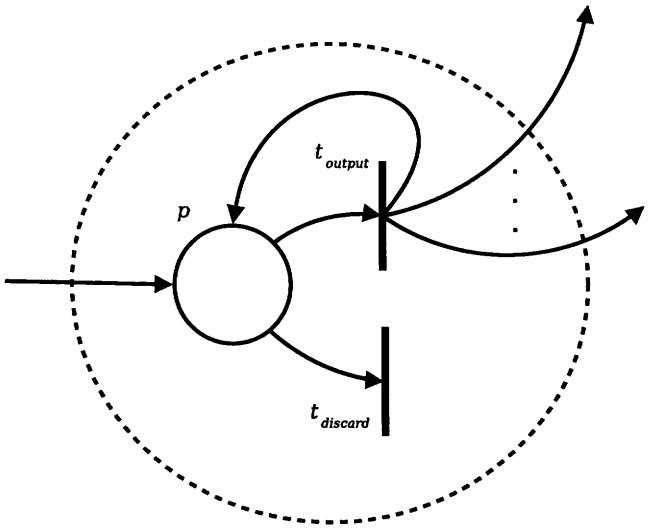

Figure 2-2: WaveScript-graph box as a Petri-net fragment

livelock are stripped away entirely.

2.1.3 WaveScript Stream Graphs

It will take a specialization of Petri nets to describe WaveScript stream graphs. In particular, when a box in a WaveScript graph fires, it reads exactly one tuple from only one of its input streams, and it may write zero, one, or many tuples to its output stream (a box with buggy code may produce infinitely many tuples per firing). Thus, there will not be a one-to-one mapping between Petri-net transitions and WaveScript-graph boxes.

Figure 2-2 shows how a WaveScript-graph box can be modeled by a small Petri-net fragment. The box consists of one place, representing its input queue, into which the input stream flows, and two transitions t_{output} and $t_{discard}$. When a box fires, it consumes one tuple and outputs $n \geq 0$ tuples. This box firing corresponds to n firings of t_{output} followed by one firing of $t_{discard}$. It should be clear that this restriction on execution forces the fragment to behave like a WaveScript-graph box.

It is important to note, though, that there is no correspondence between tokens and tuples, other than on their counts: WaveScript tuples carry real information used

by a box's code; tokens have no such importance assigned to them within the context of Petri nets.

Finally, a WaveScript-graph box may be a *source* that takes in no input streams, but whose place p in the initial marking has an infinite number of tokens (this is a minor violation of the definition of Petri nets – we may choose instead to say that the number is as large as that for which the program should be run). It may also be a *sink* that takes has no output stream, in which case t_{output} is essentially unnecessary. Every WaveScript graph has at least one source, and exactly one sink.

Now, given the restriction discussed, and also given that WaveScript stream graphs are strictly acyclic, it makes sense to use a somewhat simpler language to describe them. As before, the object is a directed graph (V, E), but here V is the set of boxes and every edge in E may connect any two boxes, with the following exceptions: there are no cycles in the graph; the source boxes $S \subset V$ have only outgoing edges; and there is one sink box, $s \notin S$, with no outgoing edges and only one incoming edge. There is one other restriction, that is fairly incidental at this point, but will be discussed further on: every non-source non-sink box has either one or two incoming edges, but no more. If boxes with more than two input streams are desired, they must be constructed with multiple two-input boxes.

2.1.4 The WaveScript Stream-Graph Language

In WaveScript, every non-sink box has a single output stream, so a stream can be defined simply by the box that produces it. That stream may feed into more than one other box, of course: but every tuple sent by a box is given as input to each of its downstream neighbors.

In WaveScript, single-input boxes are defined with the `iterate` construct, with a C-like syntax. A tiny example is shown in Figure 2-3. This code defines a box that expects takes in a stream, `ins`, whose type is expected to be numerical, and produces a stream `outs` which simply has 1 added to each element of `ins`. That is, if `ins` is a stream given by the sequence $(x_0, x_1, x_2, ...)$, then `outs` will be given by $(x_0 + 1, x_1 + 1, x_2 + 1, ...)$.

```
outs = iterate (x in ins)
{
  emit x+1;
}
```

Figure 2-3: Code for a very simple WaveScript-graph box

The fragment `(x in ins)` indicates that, when the box fires, the tuple produced on `ins` that caused the firing will be referred to by `x`.

The `emit` statement is a primitive that writes its argument to the output stream. There may be several `emit` statements, or none, per `iterate` box.

WaveScript supports the usual basic data types: integers, floating-point numbers, booleans, and strings; it also supports strongly typed unions and tuples in the style of ML. There are three dynamically sized types (aside from strings): arrays, lists, and the most unique, *Sigsegs*, or "signal segments." All three are types that are parameterized by the types of their sub-elements.

Arrays and lists require little further discussion, but Sigsegs are unique to WaveScript. A Sigseg is similar to an array in that it is abstractly a sequence of elements of like type. After creation, though, Sigsegs are normally used as read-only objects. They can be written to, but the underlying implementation can be assumed to be copy-on-write. Thus, while Sigsegs are semantically passed by value between boxes, they have nearly the performance of pass-by-reference objects.[2] In reality, to support operations such as subsegmenting and appending, a Sigseg's concrete implementation may be as a linked list of *intervals*, where an interval is typically a pairing of a pointer into a buffer, along with a number giving its length. For long sequences of data, such an implementation is typically very efficient; a similar idea, for example, has been employed by IO-Lite for system-wide buffering and movement of data between I/O devices, also to great benefit [PDZ00]. Further details of how Sigsegs are implemented in XStream are given in [GMN+08].

[2]It should be noted that in WaveScript, it is typically dangerous for a box to emit arrays, as they are always passed by reference. This issue has not been addressed in any detail, because Sigsegs are the preferred structures to be passed.

```
iterate (x in ins)
{
  state {
    arr = Array:null;
    ind = 0;
    startsamp = 0;
  }
  if ind == 0
  then arr := Array:make(128, x);
  arr[ind] := x;
  ind := ind + 1;
  if ind == 128
  then {
    emit toSigseg(arr, startsamp, nulltimebase);
    ind := 0;
    arr := Array:make(128, x);
    startsamp := startsamp + len;
  }
}
```

Figure 2-4: Code for a fixed-size window box

A WaveScript box may have state, which is declared and initialized explicitly with a `state` construct inside of the `iterate` statement. Figure 2-4 shows the code for a *window* box; a generalization of this box is used frequently throughout WaveScript applications, and, along with its inverse, will be discussed in more detail further on. In this box, a new array is constructed every time `ind` is zero, which occurs at the first of every 128 sequential firings; over the course of those firings, the input tuples are stored, in the order received, in that array. At the end of the 128 firings, a Sigseg is created – with the array as its underlying buffer – and emitted. The variables `startsamp` and `nulltimebase` – the latter built into the language – are essentially used to give a *timestamp* to the Sigseg: because WaveScript's target domain is that of signal-processing applications, Sigsegs are generally intended for storing time-stamped data, which are furthermore assumed to represent isochronous samples, and thus require only a single timestamp each. The timestamps (and especially the variable `nulltimebase` here) can essentially be ignored for the purposes of this thesis.

There is only one type of multi-input box in WaveScript, and it is built in as a

primitive: `merge` takes exactly two input streams (which must have the same type), and outputs each tuple it receives, unchanged, as soon as it arrives. `merge` introduces non-determinism into WaveScript stream-graph execution: the programmer has no direct control over over how quickly the two input streams will produce their tuples, and so cannot control the order of their combination by `merge`. If determinism is desired, it must be introduced explicitly, e.g. by disrciminating the input streams and buffering appropriately. Because WaveScript stream graphs are asynchronous, guarantees on the boundedness of such buffering must be insured by the programmer by some other means.

Finally, there is one type of source, and it is also a primitive: `timer`. There is also syntax for specifying which stream should feed into the sink: it is essentially a dummy box called `BASE` which receives only one input stream.

2.1.5 The WaveScript Language

Thus far, only code used to describe the functionality of boxes in a stream graph has been discussed, but WaveScript itself is actually a language for writing *scripts* which themselves evaluate to stream graphs. In other words, a WaveScript script is statically elaborated by the compiler, and the output itself is the stream-graph program.

WaveScript, then, is a functional language in which *streams* are first-class objects. An `iterate` construct that emits tuples of type `t`, for example, is typed generically as a `Stream t`, and we may construct functions with type `Stream t -> Stream t`.

Consider the two pieces of code in Figure 2-5. They look very different, but when evaluated, they produce the same stream graph: a chain of two boxes, the upstream of which adds 1 to every tuple as it passes it along, and the downstream of which similarly adds 2. Note that the type of the function `f` is `(a, Stream a) -> Stream a`.

The optimizations discussed in the next chapter are mostly applied after the "meta-program evalution" stage of the compiler, i.e., after the script has been evaluated and a stream graph has been produced. Still, it is important to realize the distinction between the program before it is statically elaborated, and the stream

```
s1 = iterate (x in input) {
  emit x+1;
};
s2 = iterate (x in s1) {
  emit x+2;
};
```

(a)

```
fun f(c, s) {
  iterate (x in s) {
    emit x+c;
  }
}
s2 = f(2, f(1, input));
```

(b)

Figure 2-5: Two WaveScript code fragments that produce equivalent stream graphs: in (a) the two boxes are described explicitly; in (b) they are described generally, with one `iterate` construct, and are instantiated upon application of the function `f`.

graph produced as a result. Further details on the "intrepret&reify" implementation used to perform the conversion are described in [NGC+08].

2.2 The WaveScript Compiler

The WaveScript compiler is a multi-pass, source-to-source compiler written in Scheme. As described, it compiles a WaveScript code into a stream graph. It then transforms the stream graph into a form suitable for one of its several target *back-ends*, so called to distinguish them from simply target languages: a back-end typically consists of a programming language along with a run-time engine in the same language.

The core part of the compiler's execution is a chain of mini-passes, each of which transforms the abstract syntax tree. A mini-pass, as its name indicates, may be very small, affecting only certain parts of the intermediate syntax, but several are quite large as well.

The first few passes are responsible for parsing, desugaring and typechecking the code. Then, it is evaluated into an explicit, monomorphic stream graph – that is, a stream graph in which all of the types have been resolved, so that none are generic – as described previously. We will be mostly concerned with the stream-graph representation, as it is upon this form that the optimizations to be discussed apply (but there will be one brief occasion to consider the earlier form).

2.2.1 Intermediate Representation

After parsing, WaveScript source code becomes a Scheme S-expression representing the abstract syntax tree. This intermediate representation allows mini-passes to be written very easily, with pattern matching: for each pass, the compiler's core traverser walks down the AST, passing every subtree to the mini-pass, which typically uses the Scheme `match` macro to do pattern matching [WD].[3] Thus, a powerful mini-pass can potentially be written with little code: it need only consider the pieces of syntax upon which it must operate, and the rest can "fall through" to the core traverser.

It is instructive to look at the WaveScript code describing a small stream graph, along with the S-expression describing it at a late stage in the compiler. Figure 2-6 shows WaveScript code that generates a simple three-box stream graph (the first box being the `timer`) in (a), and the S-expression describing the graph within the compiler in (b). The expression in (b) is nearly valid Scheme code; the only difference – aside from being wrapped in an artificial `program` form – is that binding forms (`lambda` and `let`) have additional subforms giving the types of the bound variables. In `let` forms, the type comes immediately following the variable name; in `lambda` forms, the list of types comes right after the list of argument variables.[4] Finally, were this to be truly interpretable Scheme code, suitable functions named `timer` and `iterate` would need to be defined (in fact, the Scheme back-end does just this to simulate stream-graph execution).

An `iterate` box, as we know, is built from two arguments: the first is a piece of code, given by the `let`-wrapped `lambda` form in this representation; the second is the input stream. Here, `tmpsmp_19` corresponds to `s1` in the WaveScript code, and `tmpsmp_21` to `s2`. We see that the second argument to `tmpsmp_19`'s `iterate` is `tmpsmp_17`, bound to the `timer`'s stream, i.e. WaveScript stream `t` feeds into the box defining `s1`. The input to `tmpsmp_21`'s box is `tmpsmp_19`, i.e. `s1` is the input to the

[3]There are a few different popular versions of `match`, each with a slightly different syntax; the version used in the WaveScript compiler is IU-Match, available at the time of this writing at `http://www.bloomington.in.us/ jswitte/scheme.html`.

[4]The `VQ_`-named arguments, with `VQueue`-parameterized types, can be ignored; they are used in Scheme simulation of the stream graph, and are included here only for completeness.

```
t = timer(1.0);
s1 = iterate (_ in t) {
  state { i = 0; }
  emit i;
  i := i + 1;
};
s2 = iterate (x in s1) {
  emit x*x;
  emit x+x;
};
BASE <- s2;
```

(a)

```
(program
  (let ([tmpsmp_17 (Stream #()) (timer '1.0)])
    (let ([tmpsmp_19 (Stream Int)
           (iterate
                    (let ([i_1 (Ref Int) (Mutable:ref '0)])
                      (lambda (_2 VQ_3) (#() (VQueue Int))
                        (begin
                          (emit VQ_3 (deref i_1))
                          (let ([tmpsmp_13 Int (+ (deref i_1) '1)])
                            (set! i_1 tmpsmp_13))
                          (let ([tmpsmp_15 (VQueue Int) VQ_3])
                            tmpsmp_15))))
                    tmpsmp_17)])

      (let ([tmpsmp_21 (Stream Int)
             (iterate
                      (let ()
                        (lambda (x_4 VQ_5) (Int (VQueue Int))
                          (begin
                            (let ([tmpsmp_9 Int (* x_4 x_4)])
                              (emit VQ_5 tmpsmp_9))
                            (let ([tmpsmp_7 Int (+ x_4 x_4)])
                              (emit VQ_5 tmpsmp_7))
                            (let ([tmpsmp_11 (VQueue Int) VQ_5])
                              tmpsmp_11))))
                      tmpsmp_19)])
        tmpsmp_21))))
```

(b)

Figure 2-6: A very small WaveScript program, (a), and its internal representation at a late stage in the compiler, (b)

box defining s2. As interpretable Scheme code, then, the value of the expression is tmpsmp_21, corresponding to s2, which is the stream that feeds into BASE (the sink).

Note that s1's box has one state variable. This is encoded in the intermediate representation by the let form wrapping the lambda given to the iterate box bound to tmpsmp_19; in essence, the box is being rightly described as a closure. Notice also that the box bound to tmpsmp_21 has a let form without any bindings wrapping its lambda: s2's box has no state.

Though this S-expression representation is dense when written on the page, with a little effort, the natural correspondence to its stream graph is apparent. Furthermore, with the compiler's "core traverser" and an appropriate function using the popular Scheme match function, we can write mini-passes with considerable ease. Suppose, for example, we wanted to count the total number of emit statements in the entire graph. We supply the core traverser with a function of two arguments: the first is an expression to match, the second is a "fallthrough" function, by which we return control to the core traverser. It might look as follows:

```
(let ([count 0])
  (lambda (expr fallthru)
    (match expr
      [(emit ,vq ,x) (set! count (+ 1 count))]
      [(program ,[p]) count]
      [,oth (fallthru oth)])))
```

This code may be difficult to understand without prior knowledge of the match macro, but we will walk through it. First, notice that the function is a closure; count keeps track of the number of emit statements encountered. The match call takes the expression expr given by the core traverser, along with three pattern-matching clauses. The first element in each of these three clauses is an S-expression pattern that expr might take. The symbols emit and program must match exactly; comma-unquoted symbols may match any expression. If expr's value is equal to this s-expression – with comma-unquoted symbols replaced by some subform – then this

clause matches, and the `match` form returns the value of the expression in the second part of the clause.

Thus, `emit` statements, as they appear in Figure 2-6, will match the first clause, causing the counter to be incremented.

Notice the expression `,[p]`: this indicates that, if this clause matches, `match` should recursively apply itself to the subform in the corresponding location of `expr`'s value. In this example, our intermediate program will match the second clause exactly once, and the bulk of it – the part wrapped by the `program` form – will be sent through `match` alone. When this recursion is complete, the value of the expression `count` will be returned, which will then be the number of `emit` statements.

Finally, any remaining forms are trivially matched by the third clause, and given to the `fallthru` function, which returns control to the core traverser.

The core traverser, when given this function and the intermediate program, will return the number of `emit` statements contained; however, one would be hard-pressed to call this a compiler "pass," since it is not transforming the program. It is important to note, though, that the program *is* transformed by each of the `match` clauses: the second part of each clause is evaluated to give the value that should *replace* `expr`. It just so happens that, in the code above, the value that replaces the `program` form is a number! Suppose instead that we wanted to change the syntax of `emit`s slightly. We might give `match` the following clause (along with the fall-through clause):

```
[(emit ,vq ,x) `(emit_tuple ,x)]
```

With no other clauses, running such a pass on our program would simply change all of its `emit` statements, e.g. `(emit VQ_3 (deref i_1))` would become `(emit_tuple (deref i_1))`.

This essential mechanism is the basis for writing mini-passes. It will be referred to again later on, during discussion of the added optimizations.

2.2.2 Compilation to Back-ends

The final pass transforms the intermediate program into source code for one of the several "back-end" languages. This transformation is not an algorithmically difficult one, but that fact belies its complexity, as "the devil is in the details" and it is here that the compiler must interact with foreign specifications.

The profiling system to be discussed requires some changes to the XStream back-end transformation, but they are not severe. The code-reuse optimization, also to be discussed, requires further changes to the transformation code, but its explanation will require little further elaboration of the back-end details.

Chapter 3

Extending the Compiler

Here we describe our extensions to the WaveScript compiler. Two small optimizations are presented, which give a feel for how the compiler is modified. Next, profiling of WaveScript stream graphs is discussed, with a description of the modifications necessary to make it work. Finally, two optimizations that make use of profiling are presented, along with their measured performance improvements. First, though, we briefly present four applications written in WaveScript that we use as benchmarks.

3.1 Benchmark WaveScript Applications

The first benchmark is an acoustic marmot-detection program that has been a motivating application since the inception of WaveScript. We test it offline here, with several sets of sample data, but it has been deployed for real-time use by biologists studying marmot alarm calls. It is described in more detail in [NGC+08].

The second benchmark is an EEG, or electroencephalography, filtering application. It processes brain activity monitored by electrodes placed on a patient's scalp.

The third benchmark is a pipeline-leak detection application. Its input data come from vibrational and acoustic sensors attached to water pipelines, and its aim is to detect cracks as early as possible.

The last benchmark, `filterbank`, is borrowed from StreamIt's benchmarks [TKA]. It contains a set of finite impulse response (FIR) filters which process an input signal

at several different rates.

3.2 Small Optimizations

To convey the nature of a mini-pass in the compiler, two small optimizations are presented. Their performance impacts are not dramatic, but they illustrate techniques used by the larger optimizations.

3.2.1 Copy Propagation

Copy propagation is the truncation of a redundant chain of variable bindings down to a single binding, with variable references rewritten as necessary [Muc97]. Though such redundancy is unlikely in well written code, earlier versions of the WaveScript compiler included mini-passes that, for convenience, often inserted many such extra bindings in the intermediate representation. Thus, copy propagation needed to be addressed.

We show how the S-expression form of the intermediate program makes it particularly straightforward to implement copy propagation. Recall that every variable binding occurs in the first subform of a `let` expression, and the scope of that binding is the code in the second subform. Our task becomes one of recursing down the second subforms of `let` expressions, while keeping track of the environments induced by their bindings, as well as of a list of bindings to variables (rather than to compound expressions).

As an example, Figure 3-1 shows two equivalent intermediate programs: on the left is an original, and on the right is the transformation desired by propagating copies. On the left, the variables `y_4` and `z_5` are redundant: in terms of program correctness, there is no reason for them to exist; references to them can be replaced by references to `x_3`.

Note that, in keeping with the functional nature of WaveScript, mutable variables must be explicitly declared as such by the programmer. In the intermediate program, these variables are given parametric types with base type `Ref`, e.g. as `Ref Int` in-

```
(let ([tmpsmp_9
       (Stream #())
       (timer '1.0)])
  (let ([tmpsmp_11
         (Stream Int)
         (iterate
           (let ()
             (lambda
               (__1 VQ_2)
               (#() (VQueue Int))
               (let ([x_3 Int '0])
                 (let ([y_4 Int x_3])
                   (emit VQ_2 y_4)
                   (let ([z_5 Int y_4])
                     (emit VQ_2 z_5))))
               VQ_2))
           tmpsmp_9)])
    tmpsmp_11))
```

(a)

```
(let ([tmpsmp_9
       (Stream #())
       (timer '1.0)])
  (let ([tmpsmp_11
         (Stream Int)
         (iterate
           (let ()
             (lambda
               (__1 VQ_2)
               (#() (VQueue Int))
               (let ([x_3 Int '0])
                 (emit VQ_2 x_3)
                 (emit VQ_2 x_3))
               VQ_2))
           tmpsmp_9)])
    tmpsmp_11))
```

(b)

Figure 3-1: The intermediate representation for a short program, (a), and the equivalent representation after applying copy propagation, (b)

stead of `Int`. Currently our copy-propagation mini-pass ignores mutable variables, as new ones are never inserted by the compiler (as immutable bindings may be). The programmer may introduce redundant mutable bindings, but it would be rare (and indeed strange) for none of them to ever then change value, so that case is ignored for now.

The approach we take is as follows: our mini-pass matches on `iterate` forms, and initializes a *substitution mapping* of variable names to substitutable variable names. The mini-pass also matches on `let` forms, for which it checks each binding to see whether its right-hand side is a symbol (indicating that it's a variable reference); if so, it adds an entry to the substitution mapping, (v_l, v_r), where v_l is the left-hand side (the variable name to be bound) and v_r the right-hand side (the substitutable variable name). This updated mapping is used to traverse the body of the `let` form; the mapping is *not* valid, and is not used, outside of that body. In the updated `let` form, we leave out that binding. Finally, we match on variable references, and we

replace it with its substitution in the current mapping, if there is one.

Below we show the code for the `match`-clause for matching `let` forms.

```
[(let ([,lhs* ,ty* ,[rhs*]] ...) ,body)
  (let-values ([(tosubst newbinds)
                (partition substitutable? (map list lhs* ty* rhs*))])

  (define newsubst (append tosubst substs))
  (define (newdriver x f) (do-expr x f newsubst))

  (if (null? newbinds)
      (fallthru body newdriver)
      `(let ,newbinds ,(fallthru body newdriver))))]
```

The `partition` function takes a predicate returning true or false, and a list of elements, and splits the list into two. All elements to which application of the predicate returns true are in the first, and all those false in the second. The bindings in the transformed `let` form are just the subset of the originals for which there was no substitution; these are exactly what we call `newbinds`. If `newbinds` is empty, then all the bindings have substitutions, and we do not need a `let` form; we just replace it with the processed body.

`do-expr` is the main function containing our `match` form. We need to specify it explicitly, because we need to pass along the new substitution mapping, `newsubst` (the original being `substs`). The driver given to the core traverser only takes two arguments; we need to use this closure to pass along the mapping.

This is the core piece of code used in copy propagation. The `match` clause for variable references is simple: it just checks the mapping `substs` for a substitution, and uses it if it exists.

The nested nature of `let` forms in the intermediate representation (`let*` and `letrec` not being allowed), along with the essential function of the core traverser, have made it particularly easy to locate substitutions only within their allowed scopes.

3.2.2 Reusing Box Code

In elaboration of a WaveScript program, it is possible that multiple boxes are generated that share exactly the same code. This is not surprising – in fact, it is encouraged – given the functionality of WaveScript previously discussed. Previously, though, such code was not reused, and would become duplicated during translation to the target back-end. We describe this small optimization of reusing such identical code between boxes. Though the runtime performance improvement of this optimization is negligible (some small reduction in instruction-cache misses has been measured for some applications), there is benefit in greatly reduced compile times of the target-language code. For some applications, the reduction is drastic (around four times faster).

Two `iterate` boxes are considered identical if they have the same numbers and types of state bindings (but, of course, potentially different right-hand sides), and if the code for one can be α-converted to that of the other [Pie]. In determining the equivalence of two boxes, though, we do not attempt α-conversion of one code to the other. Instead, we reuse an existing part of the compiler: the `rename_vars` pass will, as the name implies, rename every variable in the intermediate program. Thus, it performs an α-conversion, but gives the caller no control over what the new variable names will be. The important point, though, is that `rename_vars` is deterministic, and does not depend on the names in the initial program: if programs A and B can be α-converted to each other, then running `rename_vars` on each of them will produce exactly the same S-expression.[1]

Our algorithm, then, uses a simple doubly nested loop to identify identical `iterate` boxes, and gives them the same unique *template tag* (the mechanism for tagging will be discussed in Section 3.3.1. For the sake of simplicity, it does not remove the identical code from the intermediate program; redundant code is culled only during transformation to the back-end (more on this further down). It does, though, replace the original code with that obtained by running it through `rename_vars`; this will be

[1]To be accurate, it is not *strictly* true that the output of `rename_vars` will be a program with *all* variable names different from those in the input; `rename_vars` simply uses a counter to assign some unique name to each variable it encounters. Indeed, running the output of `rename_vars` through `rename_vars` again will produce exactly the same program.

necessary during back-end transformation, as will be seen shortly.

We can employ one speed-up, though: soon after the WaveScript program is parsed, before meta-program evaluation (i.e. before the stream graph is produced), every `iterate` construct is given an *initial template tag.* When the meta-program evaluator creates the stream graph, it opaquely copies the initial template tag to the `iterate` box in the graph, from the `iterate` construct that generated it. Because duplicate `iterate` boxes occur largely as a result of distinct instantiations of the same generically written WaveScript `iterate` constructs (refer back to Figure 2-5), boxes with the same initial template tag are likely to be identical. Because of some degenerate cases (e.g. when the conditional of an `if/then/else` statement can be fully statically evaluated), boxes with the same initial template tag are not guaranteed to be identical; however, it makes the most sense to check equivalence only between boxes that do have the same initial template tag. Were two boxes to be identical, but have different initial template tags, it would imply that the WaveScript programmer wrote two separate, identical `iterate` constructs – we are not overly concerned about this case.

Code Generation for XStream

As mentioned, the back-end of focus here is XStream. Because changes to the back-end *transformer* – the module that transforms the stream graph into code in the back-end's language – are necessary, we briefly describe the structure of stream-graph programs running atop XStream.

Each box is represented as an object of type `WSBox`, with a member function `iterate()` embodying its code. Box state is kept simply as member variables of the object. Stream connections between boxes are created explicitly with the `WSQuery::connectOps()` member function.

Originally, the XStream transformer created one `WSBox` subclass per box in the stream graph, and instantiated exactly one object of each such class. State variables were transformed into protected member variables.

The new approach simply creates one `WSBox` subclass per template tag, as all boxes

Benchmark	*Compile times*		*Class reduction*
	Reuse off	*Reuse on*	
Marmot	13s	9s	26%
EEG	57s	28s	64%
Pipeline	6s	5s	25%
FilterBankNew	44s	11s	80%

Table 3.1: Reduction in compile times, and numbers of classes, with box-code reuse

sharing a template tag are known to have identical code. Initial values for the state variables, unintuitively, are not passed into the constructor. Instead, state member variables are made public, and they are initialized outside of the constructor, just after the object's creation – the reason for this is purely one of simplicity, to change the existing transformer code as little as possible.[2]

Reductions in Compile Time

In Table 3.1 we show times for compilation of each of the four benchmarks' C++ codes, generated by the WaveScript compiler, with code reuse both on and off. We also show the corresponding reductions in number of `WSBox` subclasses. Compilation was done on an Intel Pentium Dual-based PC, at 2.9 GHz, using g++ version 4.2.3.

It should be noted that, even though the number of classes for EEG is reduced immensely, much of its compile time also comes from large coefficient tables written directly in the code.

3.3 Support for Profiling

The more interesting optimizations employ profiling of the stream graph's execution. As discussed, the goal is to capture metrics about each operator in the stream graph, and use them to inform optimizations. We would like to flexibly support many pieces of profiling information, and we describe how this is accomplished within the WaveScript compiler, although for now we mainly use per-box output-data rates and

[2]It should be noted that we have little interest in generating "clean" C++ code.

per-box run times.

3.3.1 Annotations

To support a number of optimizations, especially those that use profiling, a mechanism for attaching arbitrary metadata to stream-graph boxes was required. Though this mechanism itself required (minor) changes to many disparate parts of the compiler, one of its requirements was that adding new pieces of metadata would *not* require any further change, except to the optimizations which made use of them directly.

The syntax of the intermediate representation was changed to include an *annotation list* with every box in the stream graph. An annotation list is essentially a Scheme-style association list, i.e. a list of key/value pairs (the key always being a symbol). Figure 3-2 shows the intermediate representation for a very simple box, before and after the change to add annotation lists.

The change, of course, is a fairly simple one. The (tagged) annotation list becomes the first subform of the `iterate` form. In this example, the annotation list includes a unique name for the box, as well as its initial template tag, as described in Section 3.2.2.

The change requires, though, that every `match` clause that matches against such a form be updated. A clause that attempted to match with the pattern

```
(iterate (let ,bindings ,body) ,input)
```

must now, for example, match with

```
(iterate (annotations . ,annot) (let ,bindings ,body) ,input)
```

instead (similarly for `timer` and `merge`).

Unfortunately, the compiler, and the behavior desired, are a bit more complex than this. To support some optimizations – such as code reuse – annotations need to be inserted before meta-program evaluation, i.e. before the intermediate representation that describes a stream graph is generated. Before this point, `iterate` forms may be treated similarly to other forms, all of which might be required to match against a

```
(iterate
  (let ([s_1 (Ref Int) (Mutable:ref '0)])
    (lambda (__2 VQ_3)
            (#() (VQueue Int))
            (begin
              (emit VQ_3 (deref s_1))
              (let ([tmpsmp_5 Int (+ (deref s_1) '1)])
                (set! s_1 tmpsmp_5))
              (let ([tmpsmp_7 (VQueue Int) VQ_3])
                tmpsmp_7))))
  tmpsmp_9)
```

(a)

```
(iterate
  (annotations [name . s_155]
               [initial-template-name . tmpl_1])
  (let ([s_1 (Ref Int) (Mutable:ref '0)])
    (lambda (__2 VQ_3)
            (#() (VQueue Int))
            (begin
              (emit VQ_3 (deref s_1))
              (let ([tmpsmp_5 Int (+ (deref s_1) '1)])
                (set! s_1 tmpsmp_5))
              (let ([tmpsmp_7 (VQueue Int) VQ_3])
                tmpsmp_7))))
  tmpsmp_9)
```

(b)

Figure 3-2: The original intermediate representation for a simple box is shown in (a); the same box, with annotations attached (italicized), is shown in (b).

more generic pattern, in which the first subform is a comma-unquoted symbol. Such cases must be recognized, and in some cases split into two; a fair amount of care is required.

Merging Annotation Lists

Reading a metadatum from an annotation list could not be simpler: the Scheme function `assq` is used directly. Adding a metadatum is also easy, assuming it does not already exist in the annotation list; however, we must consider the more general case of *merging* two annotation lists, whose contained metadata may overlap: as will be seen in Section 3.4, when boxes themselves are merged, their annotation lists must be merged intelligently. For each metadatum, the proper behavior depends on its name, and on the context of the merge. Consider merging two annotation lists, `L` and `R`. For each metadatum there are five choices for how to merge:

- `left`: if the metadatum exists in `L`, use it; otherwise use the one from `R`
- `right`: if the metadatum exists in `R`, use it; otherwise use the one from `L`
- `left-only`: if the metadatum exists in `L`, use it; otherwise, do *not* include the metadatum in the merged list
- `right-only`: if the metadatum exists in `R`, use it; otherwise, do *not* include the metadatum in the merged list
- `manual`: use an ad-hoc function to manually generate a new metadatum, from each in `L` and `R`

The `merge-annotations` function, in addition to taking `L` and `R`, takes an association list of *hints* indicating which of these five behaviors should be used for each metadatum (and, also, a default behavior for metadatum keys not listed among the hints).

This very general behavior is not merely theoretical: it is very important to allow such flexibility, as will be seen in the coming sections.

3.3.2 Data-Rate Profiling

The Scheme simulator, as discussed, provides a working run-time engine for execution of WaveScript stream graphs. It is not intended for performance-critical deployment, and is included for use mostly as a verification and testing tool for the programmer. It is not, therefore, sufficient for providing many types of real run-time profiling information that might be desired; however, it is absolutely capable of accurately recording the *data rates* between boxes, i.e. the number of tuples emitted per box, because it still processes streams exactly as they would be by a faster back-end.[3] Furthermore, when accompanied by knowledge of the sizes of data types in XStream, it can report these data rates in bytes-per-tuple as well.

Since the Scheme simulator is easily run as a mini-pass within the compiler, it is a very useful way of capturing this profiling information. When simulation is concluded, the data rates (of tuples and bytes) are attached as `data-rate` annotations to their respective boxes, as described in Section 3.3.1.

For reference, Figure 3-3 shows a high-level view of the flow of the WaveScript compiler, once profiling has been added. Simulator profiling is performed on the stream graph, which is created by the meta-program evaluation step. Further metrics are profiled during back-end execution, to be used during re-compilation.

3.3.3 Profiling Outside of the Compiler

Aside from data rates, most profiled information must be collected during execution on the back-end that will be used for deployment. In our case, this means profiling must be done while running the stream graph on XStream.

[3]Note that while this is usually true, it is not *strictly* true. As discussed, `merge` introduces non-determinism, which is not accounted for here. We assume that programs written in WaveScript force determinism on their streams, or are not significantly affected by non-determinism; this is the case with the benchmarks used in this thesis.

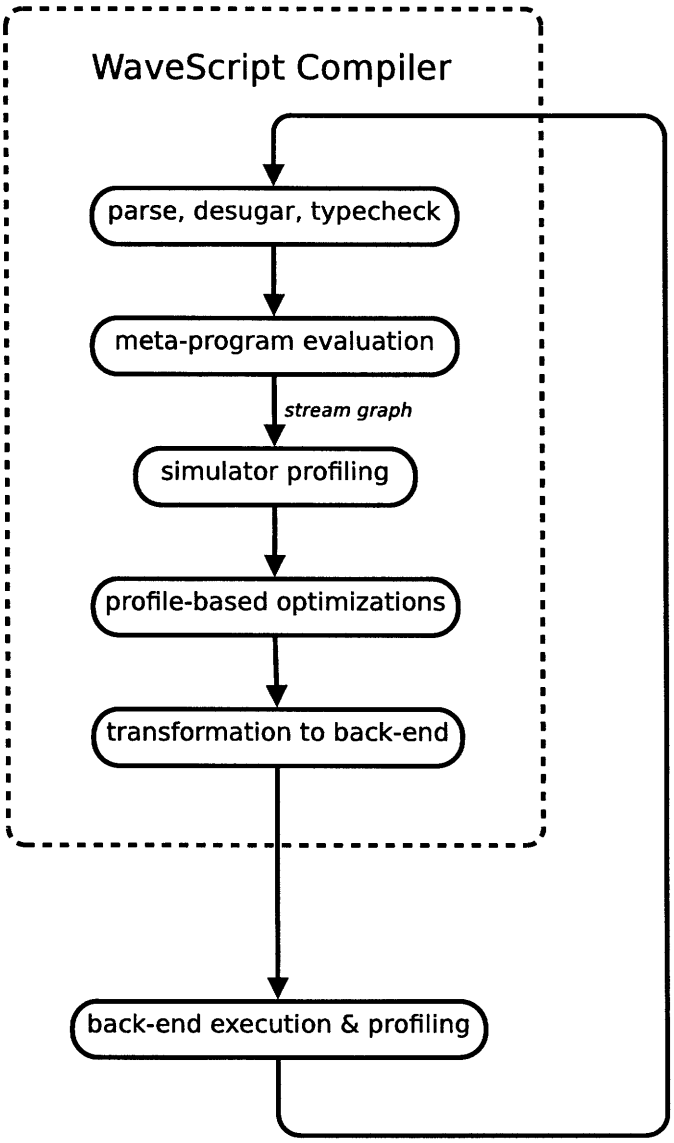

Figure 3-3: A simplified view of the major stages of the WaveScript compiler, with profiling support added

Profiling in C++

Fortunately, the Performance Application Programming Interface (PAPI) makes precise fine-grained profiling of the XStream-based code relatively easy [BDG+00]. We rely on performance counters within the CPU to track desired variables: for example, the number of clock cycles passed, or the number of cache misses that have occurred. Once our program has instructed PAPI to begin tracking a performance counter, a single function (either `PAPI_get_virt_cyc()`, or `PAPI_read()`) is used to sample that counter. Because we wish to gather performance data per box of the stream graph, the C++ code generated by the compiler must be supplemented as follows. A sample is taken at the beginning of each box-fire (i.e. `WSBox::iterate()` call), and at the end of each box-fire; the difference is added to a running counter kept for the box. Each box-fire makes calls to `emit()`, though, which may proceed to call the `iterate()` function of the downstream box. Thus, we also take samples just before, and just after, every `emit()` call a box makes, and we subtract their difference from the box's counter.

It is difficult to measure the overhead of such run-time profiling; we essentially assume that it is not significant. In any case, profiled information is consistent among identical runs. We would expect that to be true, since the values of performance counters in use are saved during context switches (including those for kernel-based threads). Profiling, though, is still safest to perform on an unloaded computer, to minimize cache pollution.

Feeding Profiled Information Into the Compiler

This method complicates the desired seamlessness of using profiling to inform optimization: a WaveScript program must be compiled once, run on XStream, then fed back into the compiler along with the new XStream-profiled information. Much fuss is avoided using annotations, though: the C++ code generated during initial compilation is modified to print the S-expression of a list of annotation lists (one per box), which is simply grepped and given to the compiler as a *compilation pa-*

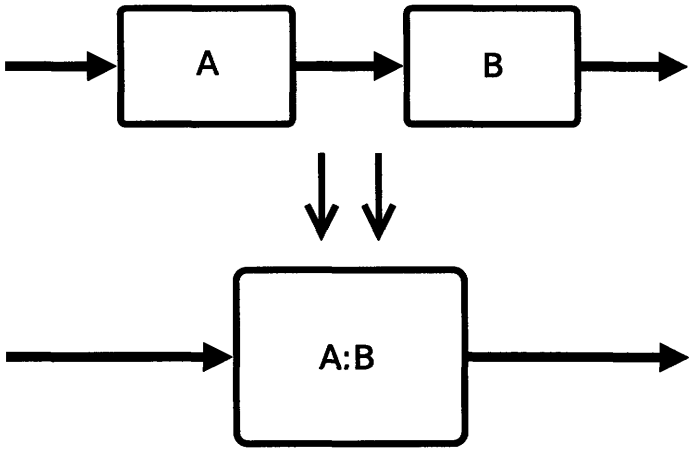

Figure 3-4: If boxes A and B form a pipeline, they can be merged into a single box.

rameters file. These annotation lists contain the just-profiled data; they are merged with those of their respective boxes at an early stage in the compiler, soon after meta-program evaluation. Note that, now, it becomes critical that the box names in the intermediate program during re-compilation are the same as those from the initial compilation. The naming procedures are, indeed, all deterministic within the compiler; but we must add to the compiler's specification that the naming procedures stay deterministic, dependant only upon the given WaveScript source code.

3.4 Using Profiles: Box Merging

Box merging is the mechanism by which two – and, by induction, many – boxes are rewritten as one. Merging the right pairs of boxes is a performance optimization: the target language's compiler has more opportunity to do its own optimization if code from one function is inlined with another; and XStream's `emit()` calls typically have significant overhead that can be eliminated. Before describing the decision behind when to perform box merging, we first describe how it is done.

3.4.1 Merging Pipelined Boxes

Two boxes in a *pipeline* configuration can be merged into one box with the same functionality. Boxes A and B are in a pipeline configuration when B's only input stream is A's output stream; A's output stream does not feed into any other box; and A has only one input stream. Figure 3-4 illustrates this. Note that B's output stream may feed into many other boxes.

The mechanism to merge is essentially as follows: replace every `emit` statement in A's code with B's code. But the value emitted in A must become the input variable for B, so every time we do this replacement, we wrap B's code with a `let` form, binding the input argument to B to the value in the `emit` statement. Formally, we separate each box into its state and its code as distinct objects: each box has its set of state – A_{state} and B_{state} – and its code – A_{code} and B_{code}; each of the latter is treated as a function of the value by which the respective box is fired. The state for the merged box is just the union of the two boxes' state sets (assuming no overlap in state-variable names); their codes are merged using the substitution

$$[B_{code}/emit]A_{code}$$

where *emit* is treated as a function representing the code's `emit` statement. In the relevant compiler code, the `rename_vars` function is applied before merging boxes, to ensure that the state-variable names in A_{state} and B_{state} do not overlap, and so that A_{code} and B_{code} do not share any free variables.

3.4.2 Identifying Pipelines

Before two pipelined boxes can be merged, they must be identified as having a pipeline configuration. To do this, we rely on an existing mini-pass, `smoosh-together`, to rewrite the intermediate program in a way that makes pipelines fairly easy to identify. Recall the pattern that `iterate` boxes match:

```
(iterate (annotations . ,annot) ,body ,input-stream)
```

`smoosh-together` rewrites the program in such a way that, if a box's output stream is used by only one other box, then it appears directly as the `input-stream` variable in the pattern above. In other words, after `smoosh-together`, all pairs of pipelined boxes will match this pattern:

```
(iterate (annotations . ,down-annot)
         ,down-body
         (iterate (annotations . ,up-annot) ,up-body ,up-input-stream))
```

Here, the outermost `iterate` construct describes the downstream box, which takes the stream from the box described by the innermost `iterate` construct as input.

3.4.3 Deciding When to Merge

The decision of when to merge pipelined boxes is made based on profiling information. The initial estimation is that two pipelined boxes are good candidates for merging if the stream between them has a very high data rate, because of the high overhead of `emit` statements in XStream. This is particularly true if the downstream box runs very quickly per firing, because the `emit` statement may become the dominant cost.

By experimentation, it has been observed that any performance enhancements from additionally merging boxes with lower data rates is usually negligible. Moreover, there is a potential reason not to merge boxes unnecessarily. Though the benchmarks here are only run on a single CPU, ultimately, multi-core and multi-host back-ends will be desired, and on these platforms, a single box firing always runs on a single core; merging boxes will potentially reduce parallelism on such systems, so it should not be done without reason.

Our strategy uses a fairly simple heuristic that has been observed to provide a performance benefit in practice. If a box is in a pipeline with its downstream neighbor, and if its output data rate (in bytes) is more than one standard deviation above the average, then it is merged with that neighbor.

Benchmark	*Merging off*	*Merging on*	*Speedup*
Marmot	86.241s	79.401s	8.61%
EEG	19.701s	19.019s	3.59%
Pipeline	24.324s	22.886s	6.28%
FilterBankNew	63.368s	62.972s	0.63%

Table 3.2: Speedup using box merging

3.4.4 Performance Benefits

Table 3.2 shows times for trial runs of the benchmark applications, with and without box merging. Execution was done on an Intel Pentium Dual-based PC running at 2.9 GHz (confined to a single core).

The marmot-detection application sees the most benefit: it is a fairly large program, with several high-rate streams around which "small" (in terms of code size and run time) boxes can be merged. The pipeline-leak detector is close behind, for similar reasons. EEG and FilterBankNew benefit somewhat less: for these benchmarks, the "heavier" streams are not much heavier than the rest, so even when their respective boxes are merged downstream, the performance improvements gained are not as great.

It should also be noted that the amount of new intra-procedural optimization that box merging enables is difficult to judge. This may be an area for more research in the future.

3.5 Using Profiles: Window/Dewindow

The "Window/Dewindow" optimization attempts to directly improve cache locality of large-enough applications by forcing certain boxes to fire multiple times, sequentially, before other boxes are allowed to fire. This optimization, then, would appear at first glance to require changes to the back-end engine, but in fact it does not; instead, it reuses existing pieces of the WaveScript compiler to modify the intermediate program.

Recall the `window` and `dewindow` operators; a fixed-size *windowing* box was described in 2.1.4. In WaveScript, `window` and `dewindow` are actually functions that

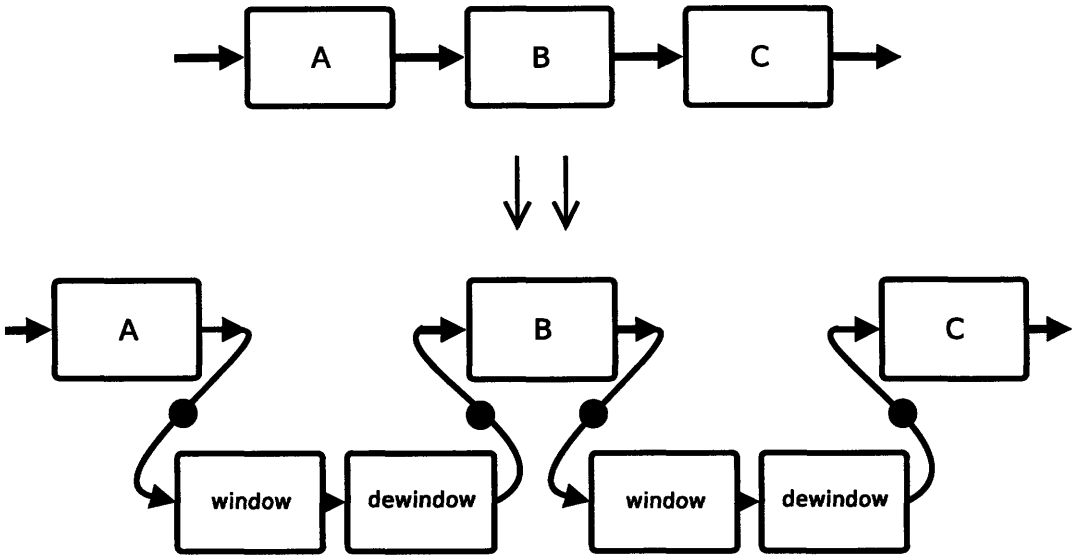

Figure 3-5: Insertion of `window/dewindow` pairs to isolate a box: here, box B is isolated to run multiple times in a row.

create such boxes, the former taking an argument indicating the concrete Sigseg size to use. `window` produces a box that fires n times, for some fixed n, and then emits a single Sigseg containing the last n input tuples it received. `dewindow` is the inverse: every time its box fires, it takes a Sigseg as input, and emits its n elements in order.

A `window/dewindow` pipeline, then, itself inserted in a pipeline in the stream graph, effectively splits the execution into two parts, forcing the upstream subgraph to produce n tuples before the downstream subgraph can consume any.

Beyond this, we can isolate a single box by placing `window/dewindow` pairs both before and after it. Figure 3-5 shows how this is accomplished: the `window/dewindow` pairs are inserted directly before and directly after box B, ensuring that it runs multiple times sequentially. In the figure, the filled circles attached to streams indicate that the respective boxes should be merged, using the box-merging mechanism previously described: `window` and `dewindow` boxes are known to be very small and fast, so there is no reason no to merge them as shown.

3.5.1 Deciding When to Window/Dewindow

Boxes with a long per-firing run time are expected to benefit less from this optimization, than boxes with very short run time. Especially if a fast-running box fires many times, the overhead of an instruction-cache miss of its code will be more significant. Thus, the decision of when to apply this optimization uses profiling information gathered from execution on XStream, as described in Section 3.3.3, to examine per-box total run times. It also uses per-box firing counts, obtained from profiling in the Scheme simulator.

Deciding on a specific heuristic is, as was the case with box merging, not easy. Once again, there is a reason for not applying the optimization to every box: every application of it puts further pressure on the data cache. Thus, we use a fairly conservative approach to identify the boxes most likely to benefit: the same standard-deviation threshold is used as described in Section 3.4.3, except that rather than using each box's output data rate, the ratio of its number of firings over its total run time is used. This simple approach shows some performance benefit, but developing better heuristics is an area of ongoing research.

3.5.2 Deciding on Window Sizes

Choosing proper window sizes is a similarly difficult decision. In terms of instruction-cache locality, bigger window sizes are generally better; however, bigger window sizes put more pressure on the data cache.

In fact, there are two window sizes to consider: that for input, and that for output. Fortunately, it is an easy enough choice to match the ratio of input window size over output window size, to the ratio of number of firings over number of tuples emitted.

Given this ratio, we keep the window sizes conservatively small: the smaller of the two is 16, unless that would force the other to be bigger than 128; in that case, the larger is 128. Experimentation with the benchmark applications has yieled no observable improvement for larger window sizes; however, this problem again could likely benefit from improved heuristics.

3.5.3 Windowing with Arrays Instead of Sigsegs

Though it is quite convenient to reuse the `window` and `dewindow` operators directly, it is wasteful to allocate a new Sigseg for every window of tuples created. It is much preferable to use a single array, allocated exactly once. This is possible, and involves only a simple rewrite of `window` and `dewindow`, but it relies on two external guarantees:

- The two boxes upstream and downstream from each created `window` box *must always run* on the same CPU. Using Figure 3-5 as a template, boxes A, B, and C must all run on the same CPU.[4] In our tests, which are all single-core, this is not a concern, but it can be guaranteed using annotations indicating what *CPU group* a box is a member of: boxes in the same CPU group must always run on the same CPU.

- The run-time scheduler must be depth-first in nature, meaning that a box, if it emits a tuple during a firing, will not fire again before the downstream box fires. Were this not the case, the upstream box might write new data into the single window-array before the downstream box has consumed original data. This is harder to guarantee automatically; it is a fact about the run-time engine that must be known to be true. In our case, the XStream scheduler we use is indeed depth-first.

Alternatively, of course, the entire new pipeline of seven boxes may be merged together, but this is avoided for several reasons. Refer again to Figure 3-5. If either A or B contains multiple `emit` statements, code expansion will occur, putting pressure on the instruction cache. Even if this is not the case, code locality may be disrupted: when the code for C – which is lodged somewhere in the middle of the new, seven-merged box – finally gets to execute (after n runs of the upstream code), it is likely to emit many tuples of its own, triggering its downstream boxes to run (with a depth-first scheduler, at least). If there are many boxes further downstream, this makes it

[4]It should be noted that arrays passed between boxes are passed *by reference*, a fact which WaveScript does not attempt to hide.

Benchmark	*Win/dewin off*	*Win/dewin on*	*Speedup*
Marmot	86.241s	80.593s	7.01%
EEG	19.701s	19.213s	2.54%
Pipeline	24.324s	23.411s	3.90%
FilterBankNew	63.368s	62.868s	0.80%

Table 3.3: Speedup using window/dewindow

less likely that the code for B will still be in the instruction cache, when it comes time for it to finish (remember that, in the seven-merged box, the code for C is essentially sandwiched somewhere in the middle of the code for B).

3.5.4 Performance Benefits

In Table 3.3 are shown trial runs of the benchmark applications, with and without the window/dewindow optimization. The same PC was used as in Section 3.4.4, again with confinement to a single core.

The marmot detector once again sees the most benefit, likely owing to its program size. The pipeline-leak detector, which is smaller, sees less improvement. EEG, though its stream graph is quite large, benefits greatly from code reuse (as seen in Section 3.2.2), so its instruction-cache locality should not be as problematic; the case is the same for FilterBankNew, whose graph is essentially structured as several identical filter pipelines.

3.6 Box Merging and Window/Dewindow

Combining the two optimizations just discussed does not in general give strictly additive performance benefit, because the sets of boxes chosen for use by them are likely to have overlap. In Table 3.4 we present results from trial runs in which the window/dewindow optimization is applied before the box-merging optimization. The same PC was used as in Sections 3.4.4 and 3.5.4.

In fact, by manual inspection, it is seen that both optimizations, with the policies specified, do often want to use the same boxes. Thus, though it has not been explored

Benchmark	*Both off*	*Both on*	*Speedup*
Marmot	86.241s	78.171s	10.32%
EEG	19.701s	18.990s	3.74%
Pipeline	24.324s	22.521s	8.01%
FilterBankNew	63.368s	62.852s	0.82%

Table 3.4: Speedup using both window/dewindow and box merging

here, a joint policy would likely be the best option.

Chapter 4

Future Work

There are many areas available for future work. As mentioned, better heuristics for the two optimizations based on profiling may lead to better performance. Though support does now exist to profile XStream applications with any of the performance counters exposed by PAPI, currently only clock-cycle counts are used. Balancing instruction- and data-cache performance by measuring their respective per-box miss rates may lead to new insights.

Currently, a new, lighter-weight back-end for WaveScript is in development by the WaveScope group. It is targeting good multi-core performance, and it is scheduler-agnostic: every box runs in its own thread. One extension of this thesis would be to support more aggressive box merging as a way of generating a static schedule, within the compiler, for this back-end. Our discussion has focused on merging boxes arranged in pipeline configurations; but if we expand the WaveScript stream-graph model within the compiler to allow multi-output boxes, and cycles within the graph, then it will be possible to merge any pair of boxes, whether they are adjacent in the graph or not. Application of the window/dewindow optimization, along with arbitrary box merging, will essentially allow the compiler to form its own static schedule. By merging down a stram graph in this manner until there are just as many boxes as cores, it is believed that an efficient program can be generated for the new back-end.

A more ambitious extension of this idea is to generate several such groups of n boxes (n being the number of cores), each being a different representation of the same

stream graph, and to extend the back-end to be able to switch between these representations at run time. The profiling currently used only aggregates measurements over an entire execution; it does not attempt to identify how data rates change over time, or whether data rates change in response to other changes. The availability of such information could justify dynamically switching representations at run time.

Chapter 5

Conclusion

Stream-graph programs allow a number of different opportunities for optimization than strictly procedural programs. Asynchronous stream graphs, in particular, are difficult to optimize because of their inherent unpredictability. Profiling – during simulation, and during real execution – can expose information to help drive some optimizations. More aggressive profiling may lead to better optimizations in the future.

XStream is a fairly robust engine for running asynchronous stream-graph programs, but its opacity has made it somewhat difficult to optimize for, from within the WaveScript compiler. It is believed that better control over scheduling decisions, and likely also over memory management decisions, would enable better optimizations by the compiler. This is partly the goal of the new, lighter-weight back-end mentioned in the previous section: more research will be done, especially in obtaining multi-core performance.

Still, as we have shown, there are techniques that can be implemented in the WaveScript compiler, targeting XStream now, to obtain some performance improvement. While the gains measured here are not especially dramatic, neither are they negligible: it is clear that using profiled information can inform optimizations for WaveScript programs. The policies used in this thesis, and the profiled metrics that those policies use, drive the optimizations to speedups in the 5-10% range for some realistic applications. As fairly simple policies have been used, and as only one metric

gathered from (non-simulator) execution has been used by them (namely, run time), there is, furthermore, reason to expect even better gains from this same foundation in the future.

Bibliography

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[BBC+04] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[BDG+00] S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[CCD+] Sirish Ch, Owen Cooper, Amol Deshp, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world +.

[GMN+08] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Xstream: A signal-oriented

data stream management system. In *ICDE*, 2008.

[IGH+] John Davis Ii, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong. -ptolemy ii-heterogeneous concurrent modeling and design in java.

[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.

[KL99] Daniel Kästner and Marc Langenbach. Code optimization by integer linear programming. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 122–136, London, UK, 1999. Springer-Verlag.

[LHJ+01] Edward A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. Sachs M. Stewart, K. Vissers, and P. Whitaker. Overview of the ptolemy project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001.

[LN04] Edward A. Lee and Stephen Neuendorffer. Classes and subclasses in actor-oriented design. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 161–168, 2004.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, August 1997.

[NGC+08] Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett. Design and evaluation of a compiler for embedded stream programs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 131–140, New York, NY, USA, 2008. ACM.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, 2000.

[Pet77] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[Pie] Benjamin C. Pierce. Foundational calculi for programming languages [to appear in the crc handbook of computer science and engineering].

[TKA] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications.

[WD] Andrew K. Wright and Bruce F. Duba. 1 pattern matching for scheme.

[Wee06] Stephen Weeks. Whole-program compilation in mlton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM.

[ZSC+03] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur C Etintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26:3–10, 2003.