LaminarIR: Compile-Time Queues for Structured Streams

Yousun Ko
Yonsei University, South Korea
yousun.ko@cs.yonsei.ac.kr

Bernd Burgstaller
Yonsei University, South Korea
bburg@cs.yonsei.ac.kr

Bernhard Scholz
University of Sydney, Australia
scholz@it.usyd.edu.au

Abstract

Stream programming languages employ FIFO (first-in, first-out) semantics to model data channels between producers and consumers. A FIFO data channel stores tokens in a buffer that is accessed indirectly via read- and write-pointers. This indirect token-access decouples a producer's write-operations from the read-operations of the consumer, thereby making dataflow implicit. For a compiler, indirect token-access obscures data-dependencies, which renders standard optimizations ineffective and impacts stream program performance negatively.

In this paper we propose a transformation for structured stream programming languages such as StreamIt that shifts FIFO buffer management from run-time to compile-time and eliminates splitters and joiners, whose task is to distribute and merge streams. To show the effectiveness of our lowering transformation, we have implemented a StreamIt to C compilation framework. We have developed our own intermediate representation (IR) called LaminarIR, which facilitates the transformation. We report on the enabling effect of the LaminarIR on LLVM's optimizations, which required the conversion of several standard StreamIt benchmarks from static to randomized input, to prevent computation of partial results at compile-time. We conducted our experimental evaluation on the Intel i7-2600K, AMD Opteron 6378, Intel Xeon Phi 3120A and ARM Cortex-A15 platforms. Our LaminarIR reduces data-communication on average by 35.9% and achieves platform-specific speedups between 3.73x and 4.98x over StreamIt. We reduce memory accesses by more than 60% and achieve energy savings of up to 93.6% on the Intel i7-2600K.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Algorithms, Performance

Keywords stream programming, synchronous data flow, program transformation, performance analysis, compiler optimization

1. Introduction

Streaming applications contain an abundance of parallelism due to independent actors that communicate via data channels. Hence, the traditional focus of stream programming compilers has been to leverage the available parallelism during compilation, auto-tuning and run-time adaptation by exploiting the underlying stream graph structure with the objective to maximize data throughput [9, 13, 15, 16]. However, the optimization opportunities with the actual data transfers between communicating actors have been neglected. Such optimization opportunities are particularly imminent in structured stream programs that re-distribute data via split-join constructs.

Actor communication is based on the notion of FIFO queues that isolate the producer from the consumer. Although the FIFO queue is an elegant conceptual model for communication, it is nevertheless obscuring the access of information between producer and consumer. Hence, optimizing compilers cannot automatically discover the information flow due to indirect memory accesses, and standard compiler optimizations including register allocation and constant propagation become ineffective across actor boundaries. I.e., FIFO queues perforate each data channel by an indirect token store operation on the producer side, plus an indirect token load operation on the consumer side. Queues as data channels are thus a dead-end performance-wise, and a valid question is whether the problem of FIFO queues could be ignored by scaling to a larger number of cores instead. Unfortunately, stateful actors, which are actors that pass state information from one invocation to the next, limit the available parallelism as reported in the literature [13, 16]. Also, high performance applications that require peak performance will not seek a compromise.

The goal of the presented work is to remedy the current situation by shifting FIFO queue-management from run-time to compile-time. A key observation is that the abstraction level of stream program representations is too high for compilers to map stream programs effectively onto von Neumann architectures.

We propose a lowering transformation that converts a stream program to *LaminarIR*, our stream program IR. Lowering proceeds in two steps: first, a *local direct access transformation* explicates the token-flow within actors if a programming language employs a push/pop semantic (as with StreamIt [24]). The second step performs a *global direct access transformation* for inter-actor communication across the whole stream graph. In LaminarIR, actor declarations must *name* the tokens on each incoming and outgoing data channel. Work functions explicitly refer to named tokens rather than relying on FIFO queue operations for data communication.

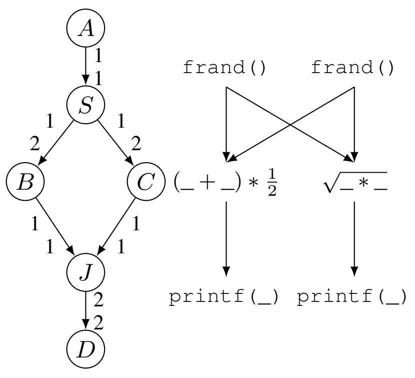
The paper makes the following contributions:

1. We identify FIFO queues as the dominating roadblock for achieving high performance with stream programming languages. This roadblock hampers the implementation of structured stream programs on von-Neumann architectures.
2. We establish the program transformation and its underlying theory to replace FIFO queues by the LaminarIR format, which manages buffers already at compile-time.
3. We evaluate the proposed transformation across four hardware platforms in terms of performance and energy consumption.

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2737994

```
1  void->void pipeline Prog{
2   add A();
3   add float->float splitjoin
4   {
5    split duplicate;
6    add B();
7    add C();
8    join roundrobin;
9   }
10  add D();
11 }
12 void->float filter A(){
13  work push 1
14   {push(my.frand());}
15 }
16 float->float filter B(){
17  work push 1 pop 2{
18   push(pop()+pop()/2);}
19 }
20 float->float filter C() {
21  work push 1 pop 2{
22   push(sqrt(pop()*pop()));}
23 }
24 float->void filter D(){
25  work pop 2{
26   println(pop());
27   println(pop());}
28 }
```

(a)



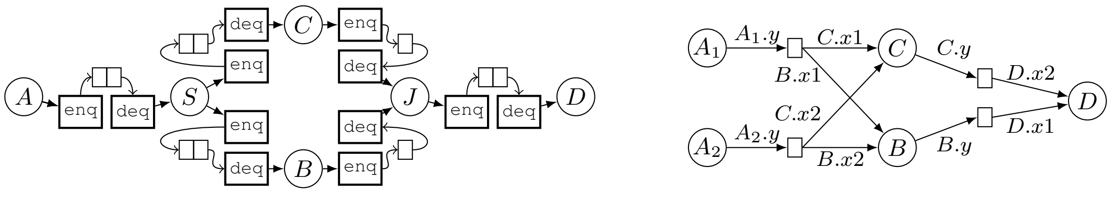
(b) (c)

```
sdf Prog {
 float A    ->S(1);
 float S(1)->B;
 float S(1)->C;
 float B    ->J(1);
 float C    ->J(1);
 float J(2)->D;
 actor A{
  firing:{y=frand();}
  output: S: y;
 }
 actor B{
  input : S: x1,x2;
  firing:{y=(x1+x2)/2;}
  output: J: y;
 }
 actor C {
  input : S: x1,x2;
  firing:{y=sqrt(x1*x2);}
  output: J: y;
 }
 actor D {
  input : J: x1,x2;
  firing:{println(x1);
          println(x2);}
 }
}
```

(d)

```
void Prog(){
 float x1;
 float x2;
 float x3;
 float x4;

 for(;;){
  // actor A
  x1=frand();
  x2=frand();

  // actor B
  x3=(x1+x2)/2;

  // actor C
  x4=sqrt(x1*x2);

  // actor D
  printf("%f\n",x3);
  printf("%f\n",x4);
 }
}
```

(e)



(f) (g)

Figure 1: (a) StreamIt source-code, (b) streamgraph, (c) dataflow-graph for schedule $2A, 2S, B, C, J, D$, (d) LaminarIR, (e) generated C code, (f) implementation based on FIFO queues, and (g) implementation employing direct variable access through indirections

Our evaluation is based on a representative set of StreamIt benchmarks.

The remainder of this paper is organized as follows. In Section 2 we present a motivating example for our framework. In Section 3 we describe the LaminarIR, the local direct access transformation with StreamIt as a case-study, and the LaminarIR code generator. Section 4 describes our global direct access transformation. Section 5 contains the experimental evaluation. We survey related work in Section 6 and draw our conclusions in Section 7.

2. Motivating Example

Let us consider the example in Fig. 1a. This example uses StreamIt, but our proposed technique applies to any stream programming model that employs static data rates. Our example computes the arithmetic and geometric means from pairs of numbers produced by the source actor A. Actor A lifts input data into the stream-graph by calling a native function `my.frand()`, which is a floating-point random number generator that employs glibc's `rand()` function under the hood. Each invocation of the source actor produces one random number that is then passed to the splitter S, which duplicates its input stream for actors B and C to compute the arithmetic and the geometric mean, respectively. The output streams of actors B and C are joined for actor D to output the results. The stream graph of this example is depicted in Figure 1b; the numbers stated with each stream-graph edge denote the number of data-items (tokens) produced and consumed by one invocation of the respective edge's producer and consumer actors. During code generation, a stream compiler will use these production and consumption rates to dimension the buffer capacity required on each stream-graph edge. These capacities will depend on the chosen actor execution schedule; To keep the example simple, we assume the sequential schedule $2A, 2S, B, C, J, D$, where actor A executes twice, followed by two executions of the splitter S, and then single invocations of actors B, C, J and D. An implementation for this schedule is depicted in Figure 1f. Therein each `push()` statement has been mapped to an `enq` operation that places a token in the corresponding actor's output queue. `deq` operations read tokens from an actor's input queue. On a shared-memory CPU, these queues will be implemented as buffers that are accessed indirectly via read- and write-pointers. Every `enq` and `deq` operation entails the overhead of maintaining the read- and write-pointers, plus the cost of indirect token-access itself.

We can specialize our program by connecting producer-side operands to the operations in the consumer that process them. For example, from the above schedule we can easily infer that the tokens produced by two invocations of actor A become the input for the arithmetic and the geometric mean computations of actors B and C in lines 18 and 22 of Figure 1a. Instead of routing these tokens through the duplicating splitter S and its associated queues, we explicate the token-flow through the stream graph as depicted by the data dependence graph in Figure 1c. There, each operation from a consumer actor (e.g., $(\_ + \_) * \frac{1}{2}$ and $\sqrt{\_ * \_}$) *directly* accesses its operands from the producer actor. Direct token-access eliminates splitters, joiners and FIFO queues altogether. Created by a compiler, the direct access format effectively shifts FIFO buffer management from run-time to compile-time.

This transformation cannot be established solely from an actor's source-code, because the amount of tokens that occur on a data channel at any given point in time is determined by the actor execution schedule. E.g., although it is true that the `push` statement in line 14 of actor A always enqueues a token at the back of the queue, the memory address of the accessed queue position depends on the state of the queue, i.e., the number of elements already in the queue. Thus, in the schedule $2A, 2S, B, C, J, D$ the first invocation of actor A will write to a different memory address than the second invocation.

As an example transformation, consider actor C in Figure 1d, which is the result of the local direct access conversion of our motivating example. The `input` section of actor C states two named tokens `x1` and `x2`, which are written in this order by splitter S. The actor stores its output in token `y`; the geometric mean computation in the `firing` section is defined in terms of the named tokens `x1`, `x2` and `y`. We refer to named tokens as *indirections*, because they divert a token access inside of an actor to an abstract storage location, where it can be aliased with the corresponding token access at the opposite endpoint of the respective data channel during the second step of our lowering transformation. This second step is a global graph transformation that employs the stream graph topology to pair producer- and consumer-side indirections. The result of this transformation on our motivating example is depicted in Figure 1g. Boxes ($\square$) denote abstract token storage locations between actors. Tokens are passed explicitly by a write-operation of the producer on an indirection which is aliased by an indirection in the consumer. E.g., the first invocation of actor A, i.e., A_1, writes its output to indirection $A_1.y$, which is aliased by indirections $C.x_1$ and $B.x_1$ of actors C and B respectively. Splitters and joiners are eliminated during the global indirect access transformation. The LaminarIR format is on a sufficiently low abstraction level for LLVM to promote all token variables to SSA form. The dataflow graph in Figure 1c depicts the dependencies that LLVM generated from the LaminarIR of our motivating example.

For the sake of readability we have restricted the size of the motivating example. Note however that the proposed technique applies to synchronous data flow (SDF) graphs in general. Our experimental evaluation in Section 5 applies this technique to entire StreamIt applications. Our proposed technique is applicable to static-rate sub-graphs of hybrid approaches like the one from Soulé et al. [21]. With such a hybrid approach, edges with dynamic data rates are kept as FIFO queues, whereas edges with static data rates are transformed to the LaminarIR format. Conversely, the LaminarIR does *not* inhibit parallelization. Rather, our technique can be applied to any stream sub-graph with static data-rates. Although outside the scope of this paper, such sub-graphs can be produced by state-of-the-art multicore parallelization techniques. Our optimization is thus orthogonal to stream parallelization techniques to enhance performance.

3. LaminarIR

The LaminarIR is a domain-specific program representation for stream programming languages that employ static data rates. Unlike StreamIt, which represents programs as hierarchies of pipelines, loops and split-join compound statements, LaminarIR employs a flat, directed graph structure which is not restricted to structured stream graphs. LaminarIR provides an actor with *direct access* to the tokens of its incoming and outgoing data channels. The LaminarIR is thus on a lower abstraction level than StreamIt, which abstracts away direct token access adapting a queue model with the `push`, `pop`, and `peek` statements.

$$\begin{array}{rcl}
\textit{program} & ::= & \textbf{sdf}\ \textit{id}\ \textbf{\{}\ \overline{\textit{edge}}\ \overline{\textit{node}}\ \textbf{\}} \\
\textit{edge} & ::= & \textit{type id .idx}?\ \textbf{(}\textit{rate}\textbf{)}?\ \textbf{->}\ \textit{id .idx}?\ \textbf{(}\textit{rate}\textbf{)}? \\
\textit{delay} & ::= & \textbf{= \{}\ \overline{\textit{val}}\ \textbf{\}} \\
\textit{node} & ::= & \textbf{actor}\ \textit{id}\ \textbf{\{}\ \textit{StateDef}?\ \textit{NodeDef}\ \textbf{\}} \\
\textit{StateDef} & ::= & \textbf{state:}\ \textit{code}\quad \textbf{init:}\ \textit{code} \\
\textit{NodeDef} & ::= & \textit{input}?\ \overline{\textit{firing output}?} \\
\textit{input} & ::= & \textbf{input:}\ \overline{\textit{channel}} \\
\textit{firing} & ::= & \textbf{firing:}\ \overline{\textit{code}} \\
\textit{output} & ::= & \textbf{output:}\ \overline{\textit{channel}} \\
\textit{channel} & ::= & \textit{id}\ \textbf{:}\ \textit{indirections} \\
\textit{indirections} & ::= & \textit{id}\ \overline{\textbf{,}\textit{id}}
\end{array}$$

Figure 2: Abstract Syntax of the LaminarIR. For brevity, constructs not relevant for buffer management have been omitted.

LaminarIR's abstract syntax is summarized in Figure 2. Each program starts with a sequence of $\overline{\textit{edge}}$ declarations that define the topology of the program's stream graph. Here, $\overline{\textit{edge}}$ denotes a list of *edge* components; we use similar notation for other lists in the LaminarIR grammar. Lines 2–7 from Figure 1d constitute the edge section of our motivating example's stream graph. An *edge* declaration consists of the token-type and the producer and consumer actor of the edge. With splitters and joiners, we additionally require the specification of input and output data-rates from which LaminarIR can deduce whether the actor is either a splitters or joiner. If an actor is not a splitter/joiner, the actor requires an explicit *node* declaration that includes data-rates for consumption and production. As an example, splitter S referred in line 2 of Figure 1d will consume one token on the edge $A{-}{>}S$ on each invocation. An actor can have several instances, for which the LaminarIR provides optional indexes (*idx*). The indexes are appended to the produced/consumer declarations of an *edge* declarations. Throughout this paper we distinguish actor instances by subscripts.

Actors other than splitters and joiners are declared in the $\overline{\textit{node}}$ section of the LaminarIR. A node declaration defines incoming channels and channel indirections in the node's *input* section, and outgoing channels and their indirections in the *output* section. The order of stated indirections (left-to-right) corresponds to their positions (front-to-back) in the associated queue. Actor computations are defined in the *firing* section of an actor declaration. Indirections are referenced like regular variables, except that input indirection are restricted to read-accesses, and output indirections to write-accesses, respectively. No constraints are imposed on the order of indirection accesses within an actor's *firing* section. An actor is stateful if an optional *state* section is defined. State variables declared in the *state* section are initialized in the *init* section, which is run once before execution of the program's steady-state.

The code generator of the LaminarIR framework, as shown in Figure 3, uses actor functions as code templates, computes a schedule and produces a single block to execute a finite periodic schedule. Hence, tokens become local and the dataflow between actor invocations for the compiler is fully exposed.

3.1 Local Direct Access Transformation

StreamIt provides a queue semantics to access input and output tokens of an actor, whereas LaminarIR has named tokens only. To lower queue operations for accessing input and output tokens, LaminarIR requires a mapping from StreamIt's `push` statements to named output tokens and from StreamIt's `pop` and `peek` operations to named input tokens. This mapping must be static, i.e., for all possible program paths in the control-flow of the actor, the conversion must be semantically correct. However, the queue operations may be dependent on data-dependent control-flow constructs. Hence, an automatic conversion may fail. To overcome this issue,

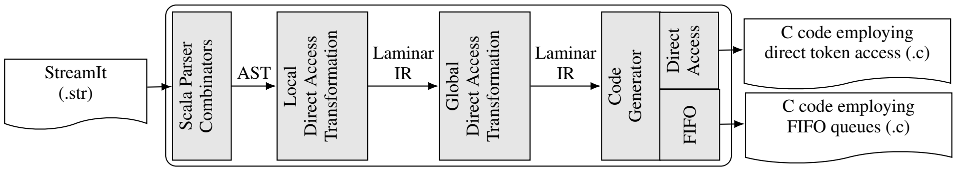

StreamIt
(.str)
Scala Parser Combinators
AST
Local Direct Access Transformation
Laminar IR
Global Direct Access Transformation
Laminar IR
Code Generator
Direct Access
FIFO
C code employing direct token access (.c)
C code employing FIFO queues (.c)

Figure 3: LaminarIR framework

we have devised a translation scheme that performs constant propagation and loop-unrolling, which covers almost all practical cases to convert `push` and `pop` operations of StreamIt to named tokens. Our local conversion covers all but 3 actors of the StreamIt benchmarks with static data-rates from [23].

If the translation fails, we introduce dynamic read/write counters for the queue position that are incremented when `push`/`pop` statements are executed. Depending on the counters, the appropriate named token is accessed via a switch/case statement. This is possible, because the number of named tokens for each channel is a constant. However, the dynamic conversion from `push` and `pop` statements imposes a run-time overhead as discussed in the experimental evaluation in Section 5.

Our StreamIt-to-C source to source transformation framework is depicted in Figure 3. Parsed StreamIt programs are lowered to LaminarIR by the local and global direct access transformations, to be followed by C code generation. The StreamIt compiler applies its own set of graph transformations, e.g., by combining multiple actors into a single aggregate actor (fusion) or by instantiating an actor multiple times (fission) and wrapping the instances by a round-robin split-join combination to achieve data parallelism [14]. To isolate the effects of the LaminarIR as the independent variable in our experimental evaluation, we have implemented our own FIFO queue backend. Unlike StreamIt, our FIFO queue backend does not change the underlying stream graph structure. (Our experimental evaluation in Section 5 compares LaminarIR to both StreamIt and our own FIFO queue backend.) The LaminarIR framework and the benchmarks that we used with the experimental evaluation are available online [1].

4. Global Direct Access Transformation

We give two types of semantics for the SDF programming model. The first semantics may be considered as a concrete semantics for SDF. The state of the data channels in the concrete semantics are modeled via an array of lists. A list in the array represents state of a data channel between two actor firings. Actors are fired according to a sequential schedule. An actor firing (1) dequeues tokens from its incoming data channels, (2) the dequeued tokens are applied to the firing function of the actor, and (3) the produced tokens of the firing function are enqueued on its outgoing data channels. The second semantics is an auxiliary semantics, which stores tokens by indirections. Instead of having data channels storing tokens, symbols with an environment are used to represent the state between actor firings. The auxiliary semantics results in a non-constructive imperative program of infinite length with infinite number of program variables. We finitize the imperative program using the finite admissible periodic schedule of the SDF program.

4.1 Background and Notation

An SDF program is a dataflow graph which statically specifies the number of data tokens produced or consumed by each actor on each invocation *a priori* [6]. An SDF program is expressed as a directed graph with a set of actors $V = \{1, \ldots, n\}$ and a set of data channels $E = \{1, \ldots, m\}$ that connect actors. When an actor i is fired, the consumption of tokens on the incoming channels is represented by function $c : V \to (E \to \mathbb{N})$. For example $c_3(1)$ denotes the consumption rate of actor 3 on data channel 1. If data channel 1 is not an incoming edge of actor 3, then the consumption is set to zero. Similarly, the production of tokens is modeled by function $p : V \to (E \to \mathbb{N})$. An SDF program has a *delay*, which represents the initial tokens in the data channel before firing actors in a schedule $\langle u_1, u_2, \ldots \rangle$. It is assumed that an SDF program has a finite admissible periodic schedule $\langle u_1, u_2, \ldots, u_k \rangle$ that consists of a finite sequence of actor invocations [17]; the periodic finite schedule is computed at compile time, invokes each actor of the SDF graph at least once, and produces no net change in the system state, i.e., the number of tokens on each edge is the same before and after executing the schedule. Thus, a periodic finite schedule can be executed ad-infinitum without exhausting memory, and we refer to the state before and after the execution of a periodic finite schedule as the *steady-state*.

For representing the state of a data channel, we resort to simple lists for which we introduce two functions. Function $\textit{head}_k : (t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_l) \mapsto (t_1, \ldots, t_k)$ extracts the first k elements from the list and function $\textit{tail}_k : (t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_l) \mapsto (t_{k+1}, \ldots, t_l)$ extracts the elements from t_{k+1} onwards. The concatenation of two lists l_1 and l_2 is denoted by $l_1 \cdot l_2$. We extend the functions for lists to array of lists, e.g., $\textit{head}_{(k_1,\ldots,k_m)}(l_1, \ldots, l_m) = (\textit{head}_{k_1}(l_1), \ldots, (\textit{head}_{k_m}(l_m))$ to represent the state of an SDF program.

4.2 Concrete SDF Semantics

We express the concrete semantics of an SDF program by means of a simple recurrence relation between two subsequent states, because there is only a single possible transition from one actor firing to another actor firing for a given infinite schedule.

Definition 1. *For a given schedule $\langle u_1, \ldots \rangle$, the states $\langle s_0, \ldots \rangle$ of the concrete semantics are given as,*

$$\begin{aligned} s_0 &= \text{delay} \\ s_i &= \text{tail}_i(s_{i-1}) \cdot f_i(\textit{head}_i(s_{i-1})) \qquad \textit{for } i > 0 \end{aligned}$$

where the initial state s_0 is referred to as delay.

A state s_i (for all $i \geq 0$) is an array of lists representing the snapshot of data tokens stored in the data channels between two actor firings. A channel is represented by a list and the lists of all channels are collated to an array. The size of the array is determined by the number of edges in the SDF program[1] whereby the list lengths are determined by the number of tokens stored in the data channels that may change if an actor either consumes from or produces tokens for the channel. An actor firing of actor u_i is expressed by the actor firing function f_i that takes the tokens to be consumed as an input and produces the tokens on the outgoing channels. The input/output behaviour of an actor function is also represented by a function whose domain and co-domain are arrays of lists. The corresponding lists of an outgoing channel contain the produced tokens whereas channels that are not outgoing channels

[1] The underlying assumption here is that the SDF graph is stable throughout the execution and the number of edges do not change.

will be empty. The input is represented by lists of input tokens for all channels. If a list is an incoming edge, the number of elements in the list is determined by the consumption rate; otherwise the list is empty.

To extract the input tokens for the current actor firing f_i, we employ the function $\mathit{head}_i(s_{i-1})$ that accesses the input tokens of the actor firing, i.e., $\mathit{head}_{(c_{u_i}(1),c_{u_i}(2),\ldots,c_{u_i}(m))}(s_{i-1})$ where $c_{u_i}(j)$ denotes the consumption rate of actor u_i on channel j. For instance, for a channel k, $c_{u_i}(k)$ tokens are retrieved and applied to actor firing function f_i of actor u_i. The new tokens that are produced by the actor function are concatenated with $\mathit{tail}_i(s_{i-1}) = \mathit{tail}_{(c_{u_i}(1),c_{u_i}(2),\ldots,c_{u_i}(m))}(s_{i-1})$, i.e., the state of the data channels after consuming the input tokens. Note that the states $\langle s_0, \ldots \rangle$ are only defined if, for all actor firings i, there are enough tokens enqueued in state s_{i-1} for firing semantic function f_i. This property holds if the schedule is a finite admissible periodic schedule, for example.

An implementation of the concrete semantics is shown in Listing 1. We assume that the schedule is a finite periodic schedule stored in `schedule` of length K. The state is an array of token lists that stores the tokens of each channel from 1 to m. The `state` is initialized with tokens in `delay` and updated by subsequent actor firings. The variable `state` outside the loop represents s_0 of the concrete semantics. For each firing of actor u, the input for u is stored in variable `input` that receives its value by extracting the input tokens from the current state s_{i-1} represented by variable `state` using function *head*. In the next step, the semantic function for u is executed with the computed input and the output is produced. The output of the firing function is concatenated with the state after removing the input tokens using the tail function. After updating the state variable `state`, the variable represents state s_i.

```
var schedule:array[1..K] of nodes;
var consumption_rate:array[1..N]:array[1..M] of
      integer;
var delay,state,input,output:array[1..M] of list;
...
state:=delay;
i:=1
loop
 u:=schedule[i];
 i:=(i mod K)+1;
 for j in {1..M} do
  input[j]:=head(state[j],consumption_rate[u,j])
 output:=fire(u,input);
 for j in {1..M} do
  state[j]:= tail(state[j],consumption_rate[u,j]) ||
                     output[j];
```

Listing 1: Implementation of Concrete Semantics

4.3 Auxiliary Semantics

We introduce an auxiliary semantics that produces a sequential program whose variables represent tokens in the data channel. The produced sequential program does not require the notion of FIFO queues. To construct the auxiliary semantics, the data channels are represented symbolically, i.e., the data channels store symbols for which there exists an environment that maps symbols back to tokens. To ensure the correctness of the auxiliary semantics, the domain of tokens is disjoint from the domain of symbols.

The auxiliary semantics stores tokens in data channels by indirection: a list of tokens $\langle t_1, \ldots, t_k \rangle$ in the concrete semantics is represented by a sequence of symbols $\langle v_1, \ldots, v_k \rangle$ for which there is an environment $e : \{v_1 \mapsto t_1, v_2 \mapsto t_2, \ldots, v_k \mapsto t_k\}$. With the environment the symbol lists $\langle v_1, \ldots, v_k \rangle$ are mapped back to the list of tokens $\langle t_1, \ldots, t_k \rangle$. We denote the mapping by $\langle t_1, \ldots, t_k \rangle = e(\langle v_1, \ldots, v_k \rangle)$ for environment e. We introduce a helper function $(\langle v_1, \ldots, v_k \rangle, e') = \mathit{fold}(\langle t_1, \ldots, t_k \rangle, e)$ where e' is the extended environment adding the mappings $\{v_1 \mapsto t_1, v_2 \mapsto t_2, \ldots, v_k \mapsto t_k\}$ to e. The newly introduced variables $\{v_1, \ldots, v_k\}$ are disjoint from the existing variables in environment e.

Lemma 1. *For all token lists $\langle t_1, \ldots, t_k \rangle$ and an arbitrary environment e, the following holds:*

$$\langle t_1, \ldots, t_k \rangle \quad = \quad \left(\lambda \left(l', e'\right) . e'(l')\right) \mathit{fold}(\langle t_1, \ldots, t_k \rangle, e).$$

The above lemma is a correspondence lemma, i.e., the fold function converts a token sequence with an environment e to a pair consisting of a symbol sequence and an environment that can be converted back to the token sequence. This is expressed by a lambda function, that binds the symbol sequence l' and environment e' from the result of the *fold* function. When the environment e' is applied to the symbol sequence we obtain the token sequence. Hence, the translation of any token list to a symbol list is reversible under the environment e provided by the *fold* operation, because only new symbols are introduced for each token in $\langle t_1, \ldots, t_k \rangle$. To operate on symbolic states, we extend the definitions of the environment application $e(\langle v_1, \ldots, v_k \rangle)$ and folding operation to arrays of symbol lists $\mathbf{v}$ for ease of readability.

Definition 2. *For a given schedule $\langle u_1, \ldots \rangle$ the symbolic states with their environments $\langle (\mathbf{v}_0, e_0), \ldots \rangle$ in the auxiliary semantics are defined as*

$$\begin{aligned}
(\mathbf{v}_0, e_0) &= \mathit{fold}(\mathit{delay}, \emptyset) \\
(\mathbf{v}_i, e_i) &= (\lambda \left(\mathbf{l}', e'\right) . (\mathrm{tail}_i(\mathbf{v}_{i-1}) \cdot \mathbf{l}', e')) \\
&\qquad \mathrm{fold}(f_i(e_{i-1}(\mathit{head}_i(\mathbf{v}_{i-1}))), e_{i-1}), \ \ \mathit{for}\ i > 0,
\end{aligned}$$

where $\mathbf{v}_i$ is an array of symbol sequences and e_i is its variable environment of the i-th step.

The definition of the auxiliary semantics goes in-line with the concrete semantics of SDF. Instead of token lists, symbol lists and their environments are used to describe the state of the data channels. Before applying the actor firing function, the symbol lists of the input are converted to concrete token lists by environment e_{i-1}. This is necessary because the semantic function f_i is not computable symbolically in general. The result of the actor firing function is mapped back to symbol lists by the fold function. Since the fold function converts the output of the actor firing to a pair $(\mathbf{l}', e')$, we use a λ-function to construct the new symbolic state by concatenating the state after consuming the tokens from the incoming edges with the output of the actor firing function.

Lemma 2. *Semantic equivalence: For a schedule $\langle u_1, \ldots \rangle$ the evaluated symbolic states of the auxiliary semantics coincide with the states of the concrete semantics, i.e., for all $i \geq 0$, $e_i(\mathbf{v}_i) = s_i$.*

The equivalence of the concrete semantics and the auxiliary semantics can be shown by structural induction in a straightforward fashion.

Instead of using a functional notion to express the computations, the auxiliary semantics guides the translation to an imperative program. The symbols of the environments become program variables and the actor firings become function calls. The underlying assumption is that the program and the program variables are unbounded. The imperative program of the auxiliary semantics is given by

$$\begin{aligned}
\mathbf{v}_0 &\leftarrow \mathit{delay} \\
\mathit{new}_1 &\leftarrow f_1(\mathit{head}_1(\mathbf{v}_0)) \\
\mathit{new}_2 &\leftarrow f_2(\mathit{head}_2(\mathbf{v}_1)) \\
&\ldots,
\end{aligned}$$

where new_i are the newly produced symbols in the i-th step by the actor firing, i.e., $\mathbf{v}_i = \mathit{tail}_i(\mathbf{v}_{i-1}) \cdot \mathit{new}_i$. Here, the array of

Table 1: Benchmark specification

Benchmarks	Parameters and values
DCT	window size (8×8), coarse
DES	number of rounds (16)
FFT	window size (16)
MatrixMult	matrix dims (10×10, 10×10)
AutoCor	vector length (32), series length (128)
Lattice	number of Stages (10)
Serpent	number of rounds (32), length of text (128)
JPEG	window size (64), fine, loops
BeamFormer	beams (4), channels (12), coarse filter tabs (64), fine filter tabs (64)
Comp. Count.	number of values to sort (16)
RadixSort	number of values to sort (16)

symbolic sequences become block assignments of program variables and blocked argument passing to the function calls. Unfortunately, the resulting imperative program is non-constructive and we seek a program that is bounded. To finitize the imperative program, we use the existence of the finite admissible[2] periodic schedule $\langle u_1, \ldots, u_k \rangle$, which have neither net-gains nor net-losses of tokens after execution. The imperative program is rewritten using an infinite loop as follows.

$$\begin{array}{rrl} & \mathbf{v}_0 & \leftarrow \textit{delay} \\ \textit{loop}: & \textit{new}_1 & \leftarrow f_1(\textit{head}_1(\mathbf{v}_0)) \\ & \textit{new}_2 & \leftarrow f_2(\textit{head}_2(\mathbf{v}_1)) \\ & & \ldots \\ & \textit{new}_k & \leftarrow f_k(\textit{head}_k(\mathbf{v}_{k-1})) \\ & \mathbf{v}_0 & \leftarrow \mathbf{v}_k \\ & \textbf{goto} & \textit{loop} \end{array}$$

In the loop the first k actors of the finite periodic schedule are fired and the symbolic state $\mathbf{v}_k$ to $\mathbf{v}_0$ is copied at the end of the loop body. The steady-state property guarantees that both symbolic states have the same cardinality. Note that this program is semantically correct only if all symbolic tokens of $\mathbf{v}_0$ have been consumed at the end of the periodic schedule. In case that the delay is too large and there exist symbols of $\mathbf{v}_0$ in $\mathbf{v}_k$, the periodic schedule $\langle u_1, \ldots, u_k, u_1, \ldots, u_k, \ldots, u_1, \ldots, u_k \rangle$ is expanded by several iterations of the periodic finite schedule until no symbols of the delay remain in the final state of the expanded periodic schedule. The expansion is necessary since a symbol can only carry a single token and not a multitude.

Note that for splitters and joiners the actor firing function redistributes the tokens and can be directly executed at a symbolic level. Hence, splitters and joiners are dissolved in the generated program, and only actor functions that do actual computational work are performed.

5. Experimental Results

We conducted our experimental evaluation on the Intel i7-2600K, AMD Opteron 6378, Intel Xeon Phi 3120A and ARM Cortex-A15 platforms. Characteristic features of the tested platforms are summarized in Table 2. Our evaluation comprised the LaminarIR direct access format and FIFO queues (C code), and the C++ code generated by StreamIt 2.1.1. All source code was compiled by Intel's ICC compiler for the Intel Xeon Phi 3120A, and the LLVM compiler infrastructure for the other processors. LLVM's optimizing middle-end provides fine-grained control over the selection of optimization passes and their application order. We employed this facility for an in-depth analysis of the profitability of each optimization on the evaluated formats.

Our experimental evaluation focuses on single thread performance only, to dissect the effect of our optimization on the full range of low-level compiler optimizations. Focusing on single thread performance avoids the experiment to be perturbed from synchronization and inter-processor communication overhead, which are orthogonal to our optimization. StreamIt's backend was thus used with the default setting that compiles for a uniprocessor. Measurement data was collected from hardware performance counters using PAPI [10]. Except with the Intel Xeon Phi 3120A that does not support CPU frequency scaling, we employed scaling governors on all platforms to lock the clock frequency to the values stated in Table 2. As a result, the coefficient of variation of each measurements is close to 0%.

Our list of representative benchmarks from the StreamIt benchmark suite [23] is stated in Table 1. Five benchmarks are from the StreamIt Core Benchmark Suite proposed by the MIT-StreamIt-group [14]. The StreamIt Core Benchmark Suite contains 12 benchmarks, and Serpent and DES are the largest benchmarks. RadixSort (single-pipeline), AutoCor (split/joins) and JPEG (two loops) were included for their distinct stream-graph features. ComparisonCounting and MatrixMult were chosen as adversary test cases for LaminarIR. ComparisonCounting features input data-dependent push/pop statements that limit direct token access, and MatrixMult is aggressively fused by StreamIt, which incurred a penalty on the LaminarIR. BeamFormer, one of the 12 StreamIt Core benchmarks, contains 28 stateful actors. Most StreamIt benchmarks use static input data; with the LaminarIR direct access format, static input enabled LLVM to compute partial results already at compile-time. We thus manually converted the benchmarks to use randomized input instead (similar to our motivating example in Figure 1a). All evaluations shown in this paper are based on randomized input. We report on the enabling effect of the LaminarIR by comparing the performance achieved with static and randomized input, in comparison to the FIFO queues and StreamIt code.

5.1 Performance

Figure 4 shows the speedups achieved by the LaminarIR direct access format over FIFO queues and StreamIt. LaminarIR's direct access format achieves average speedups of 7.25x over FIFO queues and 3.73x over StreamIt on the Intel i7-2600K, 7.43x and 4.13x on the AMD Opteron 6378, 6.75x and 4.98x on the Intel Xeon Phi 3120A and 6.74x and 4.84x on the ARM Cortex-A15.

In general, more complex benchmarks have a tendency to contain more split-joins, which are eliminated altogether with the LaminarIR direct access format, leading to larger performance improvements. The corresponding reduction of communication costs is covered in Section 5.2. DES, which shows the best performance improvement amongst all benchmarks, achieves a 36.2x speedup with the LaminarIR direct access format over FIFO queues, and a 19.2x speedup over StreamIt on the Intel i7-2600K. The DES encryption algorithm uses static keys. With direct token access, computations on static data can be partially computed already at compile time, which reduces code-size and instruction cache misses, leading to very competitive performance compared to the FIFO queues and StreamIt representations.

Two benchmarks, ComparisonCounting and JPEG, showed a speed-down on the Intel Xeon Phi 3120A. ComparisonCounting contains a temporary array variable in the StreamIt source code,

[2] The SDF program has steady-state, i.e., the number of tokens before and after the execution of the finite periodic schedule is the same on all data channels.

Table 2: Hardware configuration

CPU	Intel i7-2600K	AMD Opteron 6378	ARM Cortex-A15	Intel Xeon Phi 3120A
Clock Freq.	3.4GHz	2.4GHz	1.7GHz	1.1GHz
Inst. Cache	32kB	64kB, shared by 2 cores	32kB	32kB, unified for both instruction and data
L1 Data Cache	32kB	16kB	32kB	
OS (Kernel ver.)	Ubuntu 12.04.4 (3.2.0)	CentOS 6.5 (2.6.32)	Linaro 13.08 (3.11.0)	Centos 6.5 (2.6.32)
C Compiler	LLVM/Clang 3.5.0			ICC 14.0.3

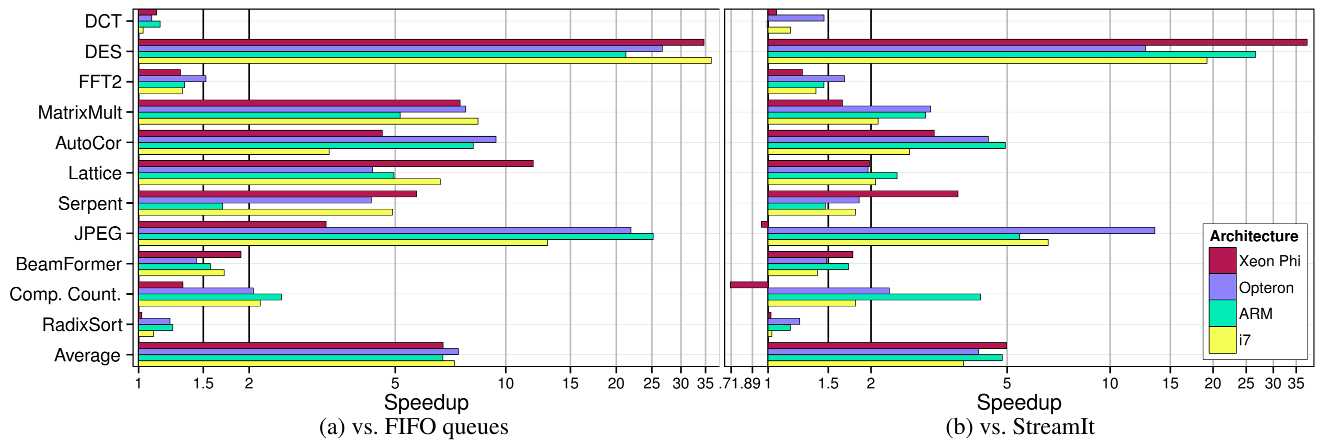


Figure 4: Speedup of LaminarIR (a) vs. FIFO queues and (b) vs. StreamIt

which our optimization does not target. We attribute the speed-down to the effect of the Intel Xeon Phi's 512bit wide SIMD registers in conjunction with this array. The JPEG benchmark contains similar programmer-provided arrays.

Table 4 compares the LaminarIR direct access format to FIFO queues and StreamIt in terms of the total number of instructions executed (columns 2 and 6), the number of memory loads (columns 3 and 7) and stores (columns 4 and 8), and the energy consumption of the CPU (columns 5 and 9). All data was collected from hardware performance counters on the Intel i7-2600K processor. Stated percentages denote LaminarIR direct access over FIFO, and LaminarIR direct access over StreamIt. E.g., LaminarIR direct access executes on average only 43.66% of instructions compared to StreamIt, and it consumes only 55.8% of the energy. The LaminarIR direct access format uses only around 40% of memory accesses on average compared to FIFO queues and StreamIt. SDF programs are data centered in general and about 40% of the total instructions executed are loads and stores on average with FIFO queues.

Figure 5 shows the effectiveness of the LaminarIR with compiler optimizations. Static input data enables compilers to compute partial results already at compile time, showing a 5.53x average speedup over randomized input when using the LaminarIR direct access format. This effect is observed to a much lesser degree with FIFO queues (1.56x average speedup) and StreamIt (1.34x average speedup). We found the amount of computations shifted to compile-time startling with some benchmarks, e.g., only 2% of instructions are left when using static input data in conjunction with the LaminarIR with the Comparison Counting benchmark. Another scenario for large improvements with the LaminarIR direct access format are programs which fit into the L1 data and instruction caches after compiler optimizations on static input have been conducted (e.g., with DES). The reduced code sizes and data accesses are beneficial especially on processors which provide smaller instruction and L1 data caches, such as the Intel Xeon Phi 3120A.

Table 3: Communication reduction from the elimination of splitters and joiners with the LaminarIR direct access format

Benchmark	Reduction	
	Abs. (byte)	Ratio to total
DCT	0	0.00%
DES	66,048	60.48%
FFT	1,024	20.00%
MatrixMult	60,800	69.72%
AutoCor	17,536	50.00%
Lattice	14,308	43.76%
Serpent	101,640	33.33%
JPEG	6,208	41.59%
BeamFormer	1,280	30.08%
Comp. Count.	1,664	52.00%
RadixSort	0	0.00%
Average		36.45%

5.2 Communication Elimination

Table 3 shows the absolute numbers of bytes that the size of data channels decreased with the LaminarIR direct access format (column "Abs. (byte)"). Column "Ratio to total" shows the proportion to the total number of bytes transferred during one steady state iteration of a benchmark. Such communication reductions are due to the elimination of splitters and joiners. No improvement is possible with the DCT and RadixSort benchmarks, because they do not contain splitters or joiners. However, the LaminarIR direct access format shows better performance than FIFO queues and StreamIt

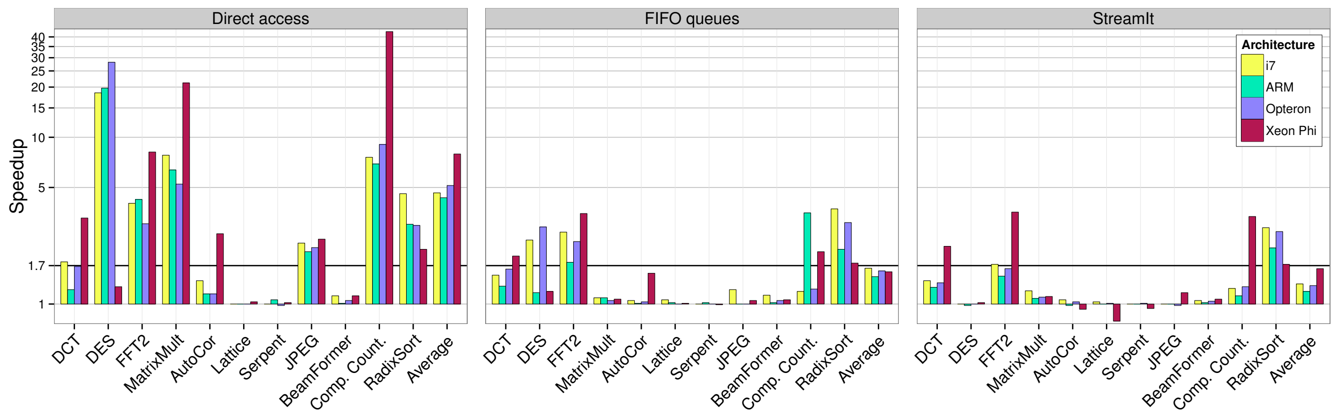


Figure 5: Effectiveness of the LaminarIR with compiler optimizations: static input data enables compilers to compute partial results already at compile-time, showing a 5.53x average speedup over dynamic input when using LaminarIR. This effect is observed to a much lesser degree with our FIFO queue implementation (1.56x average speedup) and StreamIt (1.34x average speedup).

Table 4: Improvements of the LaminarIR direct access format over FIFO queues and StreamIt on the Intel i7-2600K

Benchmarks	LaminarIR over FIFO queues				LaminarIR over StreamIt			
	Inst.	Mem. Acc.		Energy Cons.	Inst.	Mem. Acc.		Energy Cons.
		Loads	Stores			Loads	Stores	
DCT	88.29%	95.78%	99.14%	107.65%	65.11%	58.77%	97.87%	95.54%
DES	7.00%	11.85%	4.08%	3.51%	12.98%	14.09%	5.72%	6.44%
FFT	60.71%	50.25%	43.26%	73.47%	62.09%	55.49%	46.58%	72.53%
MatrixMult	14.55%	14.99%	6.34%	11.50%	29.59%	23.77%	12.13%	47.80%
AutoCor	22.26%	25.16%	9.01%	27.47%	40.30%	40.00%	26.31%	35.91%
Lattice	11.12%	12.00%	9.64%	14.26%	20.16%	15.88%	15.68%	47.38%
Serpent	22.87%	34.60%	33.37%	20.39%	45.88%	41.34%	42.94%	55.25%
JPEG	6.04%	4.85%	5.03%	8.21%	8.53%	6.30%	7.00%	16.34%
BeamFormer	59.14%	59.28%	43.73%	58.15%	59.44%	58.85%	44.22%	72.01%
Comp. Count.	47.82%	34.92%	17.15%	46.49%	57.24%	44.78%	35.08%	60.16%
RadixSort	81.91%	61.04%	79.21%	98.64%	78.95%	85.65%	77.81%	104.43%
Average	38.34%	36.79%	31.81%	42.70%	43.66%	40.45%	37.40%	55.80%

with those two benchmarks (see Figure 4 and Table 4), which implies that it is more efficient even with the same amount of data communication.

5.3 LLVM Optimization Statistics

Table 5 shows the absolute number of variables promoted to SSA form with the LaminarIR direct access format (column "Abs.") and its proportion over the number of SSA values with FIFO queues (column "vs. FIFO") and StreamIt (column "vs. StreamIt"). Because FIFO-based token access and the presence of splitters and joiners obscure the dataflow in a program, it is less likely that LLVM can connect the definition- and the use-sites of tokens in the program source-code. This problem is avoided with the LaminarIR direct access format, which uses indirections to make the token-flow between producer and consumer actors transparent. Higher numbers of promoted SSA variables indicate an improved SSA formation, which improves compiler optimizations [3]. Lower numbers of promoted SSA variables with the FIFO queues and StreamIt representations indicate the presence of array accesses which are modeled as memory accesses and thus perforate the SSA-based definition-use information that can be computed for the token-flow across a streamgraph.

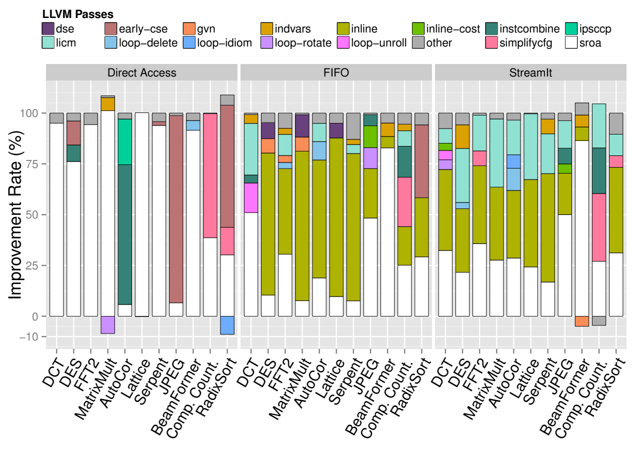


Figure 6: Contribution rate of particular LLVM optimization passes

Table 5: Enhanced SSA promotion

Benchmarks	Direct Access		
	Abs.	vs. FIFO	vs. StreamIt
DCT	138	575.00%	460.00%
DES	7,526	250.03%	302.01%
FFT2	1,240	212.33%	529.91%
MatrixMult	2,603	518.53%	2991.95%
AutoCor	418	40.50%	35.73%
Lattice	3,330	234.51%	1640.39%
Serpent	36,802	229.63%	1594.54%
JPEG	1,363	236.63%	580.00%
BeamFormer	770	168.12%	154.00%
Comp. Count.	402	219.67%	209.38%
RadixSort	250	357.14%	219.30%
Average		276.55%	792.47%

Figure 6 shows the contribution of individual LLVM optimization passes over the total performance improvement on the Intel i7-2600K. Unlike FIFO queues and StreamIt, the LaminarIR direct access format gains most by SROA (Scalar Replacement of Aggregates), which is part of LLVM's SSA formation. FIFO queues and StreamIt on the contrary gain much less from the SROA pass. Instead, they profit most from inlining of functions for buffer management.

6. Related Work

There exists a large body of work on stream programming languages and related compilation techniques, including StreamIt [24], Baker [12], Brook [11], Cg [18], CQL [4], Lime [5] and StreamFlex [22]. All except Cg implement data channels based on queues. Systems based on the streaming model such as Borealis [2], Flextream [15] and DANBI [19] employ FIFO queues for their data channels.

Because the performance overhead of FIFO queue operations is non-negligible, there has already been significant research effort to reduce the communication overhead of stream programs. Bhattacharyya et al. [7] investigated how to enhance register utilization for data transfers by tuning actor invocation schedules rather than tackling the data transfer method itself. Bier et al. [8] introduced Gabriel, a design environment for digital signal processing, which employs symbolic names for tokens across actors. Gabriel does not model global dataflow, and data is flushed to memory after each actor invocation. Sermulins et al. [20] applied scalar replacement to convert arrays into scalar variables for buffers that reside between fused actors. However, unlike LaminarIR, scalar replacement is not applied on buffers which contain delay tokens, because two adjacent actors are never fused if the downstream actor performs any peeking. This constraint implies that scalar replacement cannot be performed on stream graph cycles such as StreamIt's feedback loops. Split-joins are only fused but not eliminated. Soulé et al. [21] and Bosboom et al. [9] describe actor fusion techniques to remove inter-buffering overhead. In particular, Bosboom et al. [9] conceptually combine downstream queues of a splitter to the upstream queue of the splitter and vice versa for a joiner, but they do not remove queues as such as the communication mechanism between actors.

7. Conclusion

We introduced LaminarIR which is a low-level intermediate representation for stream programming. The LaminarIR framework enables a new compiler transformation that shifts the FIFO buffer management from run-time to compile-time. We demonstrated the effectiveness of our approach by improving the performance of StreamIt benchmarks between 3.73x and 4.98x over the original StreamIt compiler. The performance is improved between 6.64x and 7.43x for our own FIFO queue implementation. We conducted experiments on the Intel i7-2600K, AMD Opteron 6378, Intel Xeon Phi 3120A, and ARM Cortex-A15 platforms. Our approach eliminates 35.9% data-communication resulting in 60% less memory accesses on the Intel i7-2600K.

Acknowledgments

This project has been partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MEST) under grant no. 2012K2A1A9054713, the Global Ph.D Fellowship Program of the NRF (grant no. 2010-0008582), and the Australian Research Council Discovery Project DP1096445. We gratefully acknowledge access to the Intel Xeon Phi 3120A provided by the Daehan Consulting Group, Seoul, Korea.

References

[1] LaminarIR website. `http://LaminarIR.github.io`.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research*, CIDR '05, pages 277–289, Asilomar, CA, 2005.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

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

[5] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[6] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[7] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Trans. on Circuits and Systems — I: Fundamental Theory and Applications*, 42:138–150, March 1995.

[8] J. C. Bier, E. E. Goei, W. H. Ho, P. D. Lapsley, M. P. O'Reilly, G. C. Sih, and E. A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, 10(5):28–45, Sept. 1990.

[9] J. Bosboom, S. Rajadurai, W.-F. Wong, and S. Amarasinghe. StreamJIT: A commensal compiler for high-performance stream programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 177–195, New York, NY, USA, 2014. ACM.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[12] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La: Achieving high performance from compiled network

applications while enabling ease of programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 224–236, New York, NY, USA, 2005. ACM.

[13] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 357–368, New York, NY, USA, 2011. ACM.

[14] M. I. Gordon. *Compiler techniques for scalable performance of stream programs on multicore architectures*. PhD thesis, Cambridge, MA, USA, 2010.

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

[16] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 114–124, New York, NY, USA, 2008. ACM.

[17] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[18] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *ACM SIGGRAPH 2003*, SIGGRAPH '03, pages 896–907, New York, NY, USA, 2003. ACM.

[19] C. Min and Y. I. Eom. DANBI: Dynamic scheduling of irregular stream programs for many-core systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 189–200, Piscataway, NJ, USA, 2013. IEEE Press.

[20] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '05, pages 115–126, New York, NY, USA, 2005. ACM.

[21] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. 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.

[22] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput stream programming in Java. pages 211–228, 2007.

[23] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, New York, NY, USA, 2010. ACM.

[24] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.